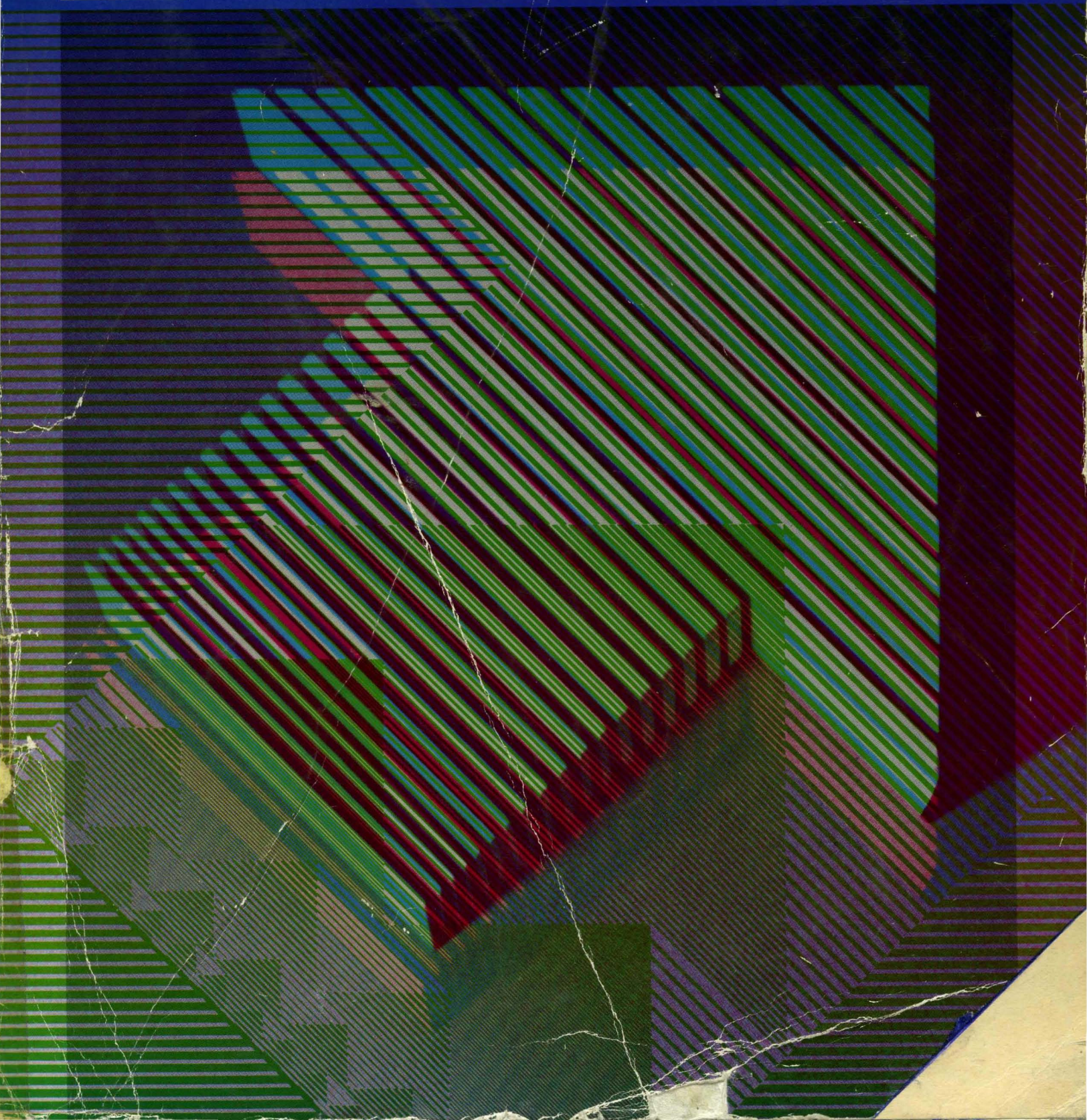
2 Reference Guide to Symbolics-Lisp

symbolics



2 Reference Guide to Symbolics-Lisp

symbolics

Cambridge, Massachusetts

Reference Guide to Symbolics-Lisp
 # 996025

March 1985

This document corresponds to Release 6.0 and later releases.

The software, data, and information contained herein are proprietary to, and comprise valuable trade secrets of, Symbolics, Inc. They are given in confidence by Symbolics pursuant to a written license agreement, and may be used, copied, transmitted, and stored only in accordance with the terms of such license.

This document may not be reproduced in whole or in part without the prior written consent of Symbolics, Inc.

Copyright © 1985, 1984, 1983, 1982, 1981, 1980 Symbolics, Inc. All Rights Reserved.
Font Library Copyright © 1984 Bitstream Inc. All Rights Reserved.

Symbolics, Symbolics 3600, Symbolics 3670, Symbolics 3640, SYMBOLICS-LISP, ZETALISP, MACSYMA, S-GEOMETRY, S-PAINT, and S-RENDER are trademarks of Symbolics, Inc.

UNIX is a trademark of Bell Laboratories, Inc. TENEX is a registered trademark of Bolt Beranek and Newman Inc. The chapter on the LOOP iteration macro is a reprint of M.I.T. Laboratory for Computer Science memo TM-169, by Glenn Burke.

Restricted Rights Legend
Use, duplication, or disclosure by the government is subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software Clause at FAR 52.227-7013.

Text written and produced on Symbolics 3600-family computers by the Documentation Group of Symbolics, Inc.

Text typography: Century Schoolbook and Helvetica produced on Symbolics 3600-family computers from Bitstream, Inc., outlines; text masters printed on Symbolics LGP-1 Laser Graphics Printers.
Cover design: Schafer/LaCasse
Cover printer: W.E. Andrews Co., Inc.
Text printer: ZBR Publications, Inc.

Printed in the USA.

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

Table of Contents

Page

I. Basic Objects 1

1. Data Types 3

1.1 The Symbol Data Type 3
1.2 The Cons Data Type 4
1.3 Numeric Data Types 4
1.4 The Compiled Function Data Type 4
1.5 The Locative Data Type 5
1.6 The Array Data Type 5
1.7 The List Data Type 5

2. Predicates 7

3. Printed Representation 13

3.1 How the Printer Works 13
3.2 Effects of Slashification on Printing 13
3.3 What the Printer Produces 14
3.3.1 Printed Representation of Integers 14
3.3.2 Printed Representation of Ratios 14
3.3.3 Printed Representation of Floating-point Numbers 14
3.3.4 Printed Representation of Complex Numbers 15
3.3.5 Printed Representation of Symbols 15
3.3.6 Printed Representation of Common Lisp Character Objects 16
3.3.7 Printed Representation of Strings 16
3.3.8 Printed Representation of Instances 16
3.3.9 Printed Representation of Arrays That Are Named Structures 16
3.3.10 Printed Representation of Arrays That Are Not Named Structures 17
3.3.11 Printed Representation of Conses 17
3.3.12 Printed Representation of Miscellaneous Data Types 18
3.4 Controlling the Printed Representation of an Object 19
3.5 How the Reader Works 19
3.6 What the Reader Recognizes 20
3.6.1 How the Reader Recognizes Integers 20
3.6.2 How the Reader Recognizes Ratios 22
3.6.3 How the Reader Recognizes Floating-point Numbers 22
3.6.4 How the Reader Recognizes Complex Numbers 24
3.6.5 How the Reader Recognizes Symbols 24
3.6.6 How the Reader Recognizes Strings 25

3.6.7 How the Reader Recognizes Conses 25
3.6.8 How the Reader Recognizes Macro Characters 26
3.7 Sharp-sign Reader Macros 27
3.8 Special Character Names 32
3.9 The Readtable 33
3.9.1 Functions That Create New Readtables 33
3.9.2 Functions That Change Character Syntax 34
3.9.3 Functions That Change Characters Into Macro Characters 35
3.9.4 Readtable Functions for Maclisp Compatibility 36

II. Lists 39

4. Manipulating List Structure 41

4.1 Conses 42
4.1.1 Composition of Cars and Cdrs 42
4.2 Basic List Operations 46
4.3 Alteration of List Structure 54
4.4 Cdr-coding 56
4.5 Tables 59
4.6 Lists as Tables 59
4.7 Association Lists 64
4.8 Property Lists 67
4.9 Hash Tables 69
4.9.1 Creating Hash Tables 71
4.9.2 Hash Table Messages 72
4.9.3 Hash Table Functions 73
4.9.4 Dumping Hash Tables to Files 75
4.9.5 Hash Tables and Loop Iteration 75
4.9.6 Hash Tables and the Garbage Collector 75
4.9.7 Hash Primitive 75
4.10 Heaps 77
4.10.1 Messages to Heaps 78
4.10.2 Heaps and Loop Iteration 79
4.11 Sorting 79

5. Locatives 83

5.1 Cells and Locatives 83
5.2 Cdr-coding and Locatives 83
5.3 Functions That Operate on Locatives 84

III. Numbers 87

6. Introduction to Numbers 89

6.1 Coercion Rules for Numbers 90

6.2 Numbers in the Compiler 90
6.3 Printed Representation of Numbers 90

7. Types of Numbers **93**

7.1 Integers 93
7.2 Rational Numbers 93
7.3 Floating-point Numbers 94
7.4 Complex Numbers 95

8. Numeric Functions **97**

8.1 Numeric Predicates 97
8.2 Numeric Comparisons 98
8.3 Arithmetic 100
8.4 Transcendental Functions 106
8.5 Numeric Type Conversions 107
8.6 Logical Operations on Numbers 113
8.7 Byte Manipulation Functions 115
8.8 Random Numbers 118
8.9 32-bit Numbers 119

IV. Evaluation **121**

9. Introduction to Evaluation **123**

10. Variables **125**

10.1 Changing the Value of a Variable 125
10.2 Binding Variables 125
10.3 Kinds of Variables 126
10.4 Special Forms for Setting Variables 128
10.5 Special Forms for Binding Variables 128
10.6 Special Forms for Defining Special Variables 134
10.7 Special Form for Declaring a Named Constant 135

11. Lexical Scoping **137**

11.1 Lexical Environment Objects and Arguments 138
11.2 Funargs and Lexical Closure Allocation 139
11.2.1 The **sys:downward-function** and **sys:downward-funarg** Declarations 140
11.3 **flet**, **labels**, and **macrolet** Special Forms 142

12. Generalized Variables **147**

13. Evaluating a Function Form **151**

13.1 Binding Parameters to Arguments 153
13.2 Examples of Simple Lambda Lists 154
13.3 Specifying Default Forms in Lambda Lists 155
13.4 Specifying a Keyword Parameter's Symbol in Lambda Lists 156
13.5 Specifying Aux-variables in Lambda Lists 157
13.6 Safety of **&rest** Arguments 157

14. Some Functions and Special Forms 159

14.1 Function for Evaluation 159
14.2 Functions for Function Invocation 159
14.3 Functions and Special Forms for Constant Values 161
14.4 Special Forms for Sequencing 164
14.5 Functions for Compatibility with Maclisp Lexprs 165

15. Multiple Values 167

15.1 Primitive for Producing Multiple Values 167
15.2 Special Forms for Receiving Multiple Values 167
15.3 Passing-back of Multiple Values 169
15.4 Interaction of Some Common Special Forms with Multiple Values 169

V. Flow of Control 173

16. Introduction to Flow of Control 175

17. Conditionals 177

18. Blocks and Exits 183

19. Transfer of Control 187

20. Iteration 189

21. Nonlocal Exits 197

22. Mapping 201

23. The loop Iteration Macro 205

23.1 Introduction to **loop** 205
23.2 Clauses 206
23.2.1 Iteration-driving Clauses 207
23.2.2 Bindings 210
23.2.3 Entrance and Exit 212
23.2.4 Side Effects 212
23.2.5 Values 212
23.2.6 Endtests 214
23.2.7 Aggregated Boolean Tests 215

23.2.8 Conditionalization 216
23.2.9 Miscellaneous Other Clauses 218
23.3 **loop** Synonyms 218
23.4 Data Types Recognized by **loop** 219
23.5 Destructuring 220
23.6 The Iteration Framework 221
23.7 Iteration Paths 222
23.7.1 **loop** Iteration Over Hash Tables or Heaps 224
23.7.2 Predefined Iteration Paths 225
23.7.3 Defining Iteration Paths 227

VI. Arrays, Characters, and Strings 233

24. Arrays 235

24.1 Array Types 235
24.1.1 **art-q** Array Type 236
24.1.2 **art-q-list** Array Type 236
24.1.3 **art-*N*b** Array Type 236
24.1.4 **art-string** Array Type 236
24.1.5 **art-fat-string** Array Type 236
24.1.6 **art-boolean** Array Type 237
24.1.7 Multidimensional Arrays 237
24.2 Array Representation Tools 237
24.3 Extra Features of Arrays 238
24.3.1 Array Leaders 238
24.3.2 Displaced Arrays 239
24.3.3 Indirect Arrays 240
24.4 Basic Array Functions 241
24.5 Accessing Multidimensional Arrays as One-dimensional 245
24.6 Getting Information About an Array 246
24.7 Changing the Size of an Array 249
24.8 Arrays Overlaid with Lists 250
24.9 Adding to the End of an Array 251
24.10 Copying an Array 252
24.11 Array Registers 255
24.11.1 Array Registers and Performance 255
24.11.2 Hints for Using Array Registers 257
24.11.3 Array Register Restrictions 258
24.12 Matrices and Systems of Linear Equations 258
24.13 Planes 260
24.14 Maclisp Array Compatibility 262

25. Characters 265

25.1 Character Objects 265
25.1.1 Character Object Details 265

25.1.2 Character Sets and Character Styles 267
25.1.3 The Device-font and Subindex Derived Fields 268
25.1.4 Two Kinds of Characters 268
25.2 Character Fields 269
25.3 Character Predicates 270
25.4 Character Comparisons 270
25.4.1 Character Comparisons Affected by Case, Style, and Bits 270
25.4.2 Character Comparisons Ignoring Case, Style, and Bits 271
25.5 Character Conversions 272
25.6 Character Names 273
25.7 Mouse Characters 273
25.8 ASCII Characters 274
25.9 Support for Nonstandard Character Sets 275

26. Strings 277

26.1 Basic String Operations 278
26.2 String Comparisons 282
26.2.1 String Comparisons Affected by Case, Style, and Bits 282
26.2.2 String Comparisons Ignoring Case, Style, and Bits 283
26.3 String Conversions 285
26.4 String Searching 286
26.4.1 String Searching Affected by Case, Style, and Bits 286
26.4.2 String Searching Ignoring Case, Style, and Bits 287
26.5 ASCII Strings 290
26.6 I/O to Strings 290
26.7 Maclisp-compatible Functions 292

VII. Functions and Dynamic Closures 295

27. Functions 297

27.1 What is a Function? 297
27.2 Function Specs 297
27.3 Simple Function Definitions 300
27.4 Operations the User Can Perform on Functions 302
27.5 Kinds of Functions 303
27.5.1 Interpreted Functions 304
27.5.2 Compiled Functions 305
27.5.3 Other Kinds of Functions 305
27.6 Function-defining Special Forms 305
27.7 Lambda-list Keywords 309
27.8 Declarations 311
27.9 How Programs Manipulate Definitions 316
27.10 How Programs Examine Functions 322
27.11 Encapsulations 325
27.11.1 Rename-within Encapsulations 329

28. Dynamic Closures 331

28.1 What is a Dynamic Closure? 331
28.2 Examples of the Use of Dynamic Closures 333
28.3 Dynamic Closure-manipulating Functions 335

VIII. Macros 337

29. Introduction to Macros 339

30. Aids for Defining Macros 343

30.1 **defmacro** 343
30.2 Backquote 345

31. Substitutable Functions 351

32. Symbol Macros 353

33. Lambda Macros 355

34. Hints to Macro Writers 357

34.1 Name Conflicts 357
34.2 **prog**-Context Conflicts 359
34.3 Macros Expanding Into Many Forms 360
34.4 Macros That Surround Code 362
34.5 Multiple and Out-of-order Evaluation 363
34.6 Nesting Macros 366
34.7 Functions Used During Expansion 368
34.8 Aid for Debugging Macros 369

35. Displacing Macros 371

36. &-Keywords Accepted by defmacro 373

37. Functions to Expand Macros 375

IX. Structure Macros 377

38. Introduction to Structure Macros 379

39. Using defstruct 383

40. Options to defstruct 385

41. Using the Constructor and Alterant Macros 395

41.1 Constructor Macros 395

41.2 By-position Constructor Macros 396
41.3 Alterant Macros 397

42. Using Byte Fields and defstruct **399**

43. Grouped Arrays **401**

44. Named Structures **403**

44.1 Introduction to Named Structures 403
44.2 Handler Functions for Named Structures 403
44.3 Functions That Operate on Named Structures 405

45. defstruct Internal Structures **407**

46. Extensions to defstruct **409**

46.1 An Example of **defstruct-define-type** 409
46.2 Options to **defstruct-define-type** 410

X. Flavors **415**

47. Introduction to the Flavor System **417**

47.1 Objects and the Flavor System 417
47.2 Modularity and Object-oriented Programming 418
47.3 Generic Operations on Objects 421
47.4 Message Passing in the Flavor System 423

48. Using the Flavor System **425**

48.1 Simple Use of Flavors 425
48.1.1 Functions for Creating Flavors 428
48.1.2 Functions for Passing Messages 430
48.2 Mixing Flavors 431

49. Flavor Functions **435**

50. defflavor Options **441**

51. Flavor Families **451**

52. Vanilla Flavor **453**

53. Method Combination **455**

54. Whoppers and Wrappers **461**

55. Copying Instances **465**

56. Implementation of Flavors **467**
56.1 Ordering Flavors, Methods, and Wrappers 467
56.2 Changing a Flavor 468

57. Zmacs Commands for Flavors **471**

58. Property List Messages **473**

59. Flavor Examiner **475**

XI. Conditions **477**

60. Introduction **479**
60.1 Overview and Definitions 479

61. How Applications Programs Treat Conditions **481**
61.1 Example of a Handler 481
61.2 Signalling 482
61.3 Condition Flavors 482

62. Creating New Conditions **485**
62.1 Creating a Set of Condition Flavors 486

63. Establishing Handlers **487**
63.1 What is a Handler? 487
63.2 Classes of Handlers 487
63.3 Reference Material 488
63.4 Application: Handlers Examining the Stack 494
63.4.1 Reference Material 495

64. Signalling Conditions **501**
64.1 Signalling Mechanism 501
64.1.1 Finding a Handler 501
64.1.2 Signalling Simple Conditions 502
64.1.3 Signalling Errors 502
64.1.4 Restriction Due to Scope 503
64.2 Reference Material 503

65. Default Handlers and Complex Modularity **509**
65.1 Reference Material 510

66. Interactive Handlers **511**

67. Restart Handlers 513

67.1 Reference Material 514
67.2 Invoking Restart Handlers Manually 516

68. Proceeding 517

68.1 Protocol for Proceeding 517
68.2 Proceed Type Messages 519
68.3 Proceeding with **condition-bind** Handlers 520
68.4 Proceed Type Names 520
68.5 Signallers 520
68.6 Reference Material 521

69. Issues for Interactive Use 523

69.1 Tracing Conditions 523
69.2 Breakpoints 523
69.3 Debugger Bug Reports 524
69.4 Debugger Special Commands 525
69.5 Special Keys 526

70. Condition Flavors Reference 529

70.1 Messages and Init Options 529
70.2 Standard Conditions 531
70.2.1 Fundamental Conditions 531
70.2.2 Lisp Errors 533
70.2.3 File-system Errors 544
70.2.4 Pathname Errors 553
70.2.5 Network Errors 554
70.2.6 Tape Errors 556

XII. Packages 557

71. The Need for Packages 559

72. Symbols 561

72.1 The Value Cell of a Symbol 561
72.2 The Function Cell of a Symbol 563
72.3 The Property List of a Symbol 564
72.4 The Print Name of a Symbol 565
72.5 The Package Cell of a Symbol 566
72.6 Creating Symbols 566

73. Sharing of Symbols Among Packages 569

73.1 External Symbols 570

73.2 Package Inheritance 571
73.3 The **global** Package 571
73.4 Home Package of a Symbol 573
73.5 Importing and Exporting Symbols 573
73.6 Shadowing Symbols 574
73.7 Introduction to Keywords 575

74. Specifying Packages in Programs 579

75. Package Names 581

75.1 Introduction to Package Names 581
75.2 Relative Package Names 582
75.3 Qualified Package Names 584
75.3.1 Introduction to Qualified Package Names 584
75.3.2 Specifying Internal and External Symbols in Packages 585
75.3.3 Qualified Package Names as Interfaces 585
75.3.4 Qualified Names of Symbols 585
75.3.5 Multilevel Qualified Package Names 586

76. Examples of Symbol Sharing Among Packages 589

77. Consistency Rules for Packages 591

78. Package Name-conflict Errors 593

78.1 Introduction to Package Name-conflict Errors 593
78.2 Checking for Package Name-conflict Errors 593
78.3 Resolving Package Name-conflict Errors 595

79. Package Functions, Special Forms, and Variables 597

79.1 The Current Package 597
79.2 Defining a Package 598
79.3 Mapping Names to Symbols 604
79.3.1 Functions That Map Names to Symbols 605
79.4 Functions That Find the Home Package of a Symbol 607
79.5 Mapping Between Names and Packages 607
79.6 Package Iteration 608
79.7 Interpackage Relations 610
79.8 Functions That Import, Export, and Shadow Symbols 611
79.9 Package "Commands" 612
79.10 System Packages 615

80. Package-related Conditions 619

81. Multipackage Programs 621

82. Compatibility with the Pre-release 5.0 Package System **625**
82.1 External-only Packages and Locking 626

XIII. Symbolics Common Lisp **629**

83. Introduction to Symbolics Common Lisp **631**

84. Using SCL **633**
84.1 SCL Packages 633
84.2 SCL and Strings 634
84.3 SCL and Symbolics Common Lisp Extensions 635
84.4 SCL and Optimization 635
84.5 SCL and Common Lisp Files 635
84.6 SCL Documentation 636

85. SCL and Common Lisp Differences **637**

Index **647**

List of Figures

Figure 1. Condition flavor hierarchy 484

PART I.

Basic Objects

1. Data Types

This section enumerates some of the various different primitive types of objects in Symbolics-Lisp. The types explained below include symbols, conses, various types of numbers, compiled functions, locatives, arrays, stack groups, and closures. Each type is given an associated symbolic name, which is returned by the function **data-type**.

1.1 The Symbol Data Type

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

- The print name is a string, which can be obtained by the function **get-pname**. This string serves as the *printed representation* of the symbol. See the section "What the Printer Produces", page 14.

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

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

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

- Symbols also have a package cell, which indicates to which *package* of names the symbol belongs. See the section "Packages", page 557.

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

1.2 The Cons Data Type

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

1.3 Numeric Data Types

There are several kinds of numbers in Symbolics-Lisp.

Fixnums represent integers in the range of -2^31 to 2^31-1. *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.

Rational numbers include both ratios and integers. *Ratios* are represented in terms of an integer numerator and denominator. The ratio is always "in lowest terms", meaning that the denominator is as small as possible.

Double-floats are double-precision floating-point numbers. *Single-floats* are single-precision floating-point numbers; they have less range and precision, and less computational overhead.

Complex numbers are pairs of noncomplex numbers, representing the real and imaginary parts of the number. The real and imaginary parts can be integer, rational, or floating-point.

Other types of numbers are likely to be added in the future. See the section "Numbers", page 87. Full details of these types and the conversions between them are discussed there.

1.4 The Compiled Function Data Type

The usual form of compiled, executable code is a Lisp object called a "compiled function". A compiled function contains the code for one function. Compiled functions are produced by the Lisp Compiler and are usually found as the definitions of symbols. The printed representation of a compiled function includes its name, so that it can be identified.

About the only useful thing to do with compiled functions is to *apply* them to arguments. However, some functions are provided for examining such objects, for user convenience. See the section "How Programs Examine Functions", page 322.

1.5 The Locative Data Type

A *locative* is a kind of a pointer to a single memory cell anywhere in the system. See the section "Locatives", page 83. The contents of this cell can be accessed by **location-contents** and updated by **(setf (location-contents ...))**.

1.6 The Array Data Type

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

1.7 The List Data Type

A *list* is not a primitive data type, but rather a data structure made up of conses and the symbol **nil**. See the section "Manipulating List Structure", page 41.

2. Predicates

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

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

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

symbolp *arg* *Function*

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

nsymbolp *arg* *Function*

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

listp *arg* *Function*

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

nlistp *arg* *Function*

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

atom *arg* *Function*

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

numberp *arg* *Function*

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

fixp *arg* *Function*

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

floatp *arg* *Function*

floatp returns **t** if its argument is a a single- or double-precision floating-point number. Otherwise it returns **nil**.

fixnump *arg* *Function*

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

bigp *arg* *Function*

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

flonump *arg* *Function*

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

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

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

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

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

complexp *x* *Function*

Returns **t** if *x* is a complex number, otherwise **nil.**

rationalp *x* *Function*

Returns **t** if *x* is a ratio. Returns **nil** if *x* is an integer. Note that in Common Lisp, **rationalp** of an integer returns **t.**

stringp *arg* *Function*

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

arrayp *arg* *Function*

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

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

functionp returns **t** if its argument is a function (essentially, something that is acceptable as the first argument to **apply**), otherwise it returns **nil.** In addition to interpreted, compiled, and built-in functions, **functionp** is true of closures, select-methods, and symbols whose function definition is **functionp.** See the section "Other Kinds of Functions", page 305. **functionp** is not true of objects that can be called as functions but are not normally thought of as functions: arrays, stack groups, entities, and instances. If *allow-special-forms* is specified and non-**nil**, then **functionp** is true of macros and special-form functions (those with quoted arguments). Normally **functionp** returns **nil** for these since they do not behave like functions. As a special case, **functionp** of a symbol whose function definition is an array returns **t**, because in this case the array is being used as a function rather than as an object.

subrp *arg* *Function*

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

closurep *arg* *Function*

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

locativep *arg* *Function*

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

errorp *object* *Function*

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

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

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

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

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

:symbol	*arg* is a symbol.
:fixnum	*arg* is a fixnum (not a bignum).
:bignum	*arg* is a bignum.
:rational	*arg* is a ratio.
:single-float	*arg* is a single-precision floating-point number.
:double-float	*arg* is a double-precision floating-point number.
:complex	*arg* is a complex number.
:list	*arg* is a cons.
:locative	*arg* is a locative pointer.
:compiled-function	*arg* is the machine code for a compiled function.
:closure	*arg* is a closure.
:select-method	*arg* is a select-method table.
:stack-group	*arg* is a stack-group.
: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.

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 (single- or double-precision).
:number	Any kind of number.
:non-complex-number	Any noncomplex number.
:instance	An instance of any flavor.
:null	**nil** is the only value that has this type.
:list-or-nil	A cons or **nil**.

See also **data-type**.

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

The following functions are some other general purpose predicates:

eq *x y* *Function*

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

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

Note that in Symbolics-Lisp equal integers are **eq**; this is not true in Maclisp. Equality does not imply **eq**ness for other types of numbers. To compare numbers, use =. See the section "Numeric Comparisons", page 98.

neq *x y* *Function*

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

eql *x y* *Function*

eql returns **t** if its arguments are **eq**, or if they are numbers of the same type with the same value, or (in Common Lisp) if they are character objects

that represent the same character. The predicate = compares the values of two numbers even if the numbers are of different types.

Examples:

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

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

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

In Symbolics-Lisp:

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

equal *x y* *Function*

The **equal** predicate returns **t** if its arguments are similar (isomorphic) objects. See the function **eq**, page 10. Two numbers are **equal** if they have the same value and type (for example, a flonum is never **equal** to an integer, even if = is true of them). For conses, **equal** is defined recursively as the two **car**s being **equal** and the two **cdr**s being equal. Two strings are **equal** if they have the same length, and the characters composing them are the same. See the function **string-equal**, page 283. Alphabetic case is ignored. All other objects are **equal** if and only if they are **eq**. Thus **equal** could have been defined by:

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

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

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

not *x* *Function*

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

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

rather than

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

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

See the function **null**, page 12.

null *x* *Function*

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

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

rather than

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

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

3. Printed Representation

3.1 How the Printer Works

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*.

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; **read** and its subfunctions are known as the *reader*. See the section "Introduction to Streams" in *Reference Guide to Streams, Files, and I/O*.

The printed representation of an object depends on its type. For descriptions of how different Lisp objects are printed:

See the section "Printed Representation of Integers", page 14.
See the section "Printed Representation of Ratios", page 14.
See the section "Printed Representation of Floating-point Numbers", page 14.
See the section "Printed Representation of Complex Numbers", page 15.
See the section "Printed Representation of Symbols", page 15.
See the section "Printed Representation of Common Lisp Character Objects", page 16.
See the section "Printed Representation of Strings", page 16.
See the section "Printed Representation of Instances", page 16.
See the section "Printed Representation of Arrays That Are Named Structures", page 16.
See the section "Printed Representation of Arrays That Are Not Named Structures", page 17.
See the section "Printed Representation of Conses", page 17.
See the section "Printed Representation of Miscellaneous Data Types", page 18.
See the section "Controlling the Printed Representation of an Object", page 19.

3.2 Effects of Slashification on Printing

Printing is done either with or without *slashification*. The unslashified version is nicer looking, but **read** cannot handle it properly. The slashified version, however, is carefully set up so that **read** is able to read it in.

The primary effects of slashification are:

- Special characters used with other than their normal meanings (for example, a parenthesis appearing in the name of a symbol) are preceded by slashes or cause the name of the symbol to be enclosed in vertical bars.

- Symbols that are not from the current package are printed out with their package prefixes. (A package prefix looks like a symbol followed by a colon).

3.3 What the Printer Produces

3.3.1 Printed Representation of Integers

If the number is negative, the printed representation begins with a minus sign ("-"). Then, the value of the variable **base** is examined, with the following results:

- If **base** is a positive integer, the number is printed out in that base (**base** defaults to 10).
- If it is a symbol with a **si:princ-function** property, the value of the property is applied to two arguments:
  - **minus** of the number to be printed
  - The stream to which to print it (this is a hook to allow output in Roman numerals and the like)

  Otherwise, the value of **base** is invalid and an error is signalled.

Finally, if **base** equals 10. and the variable ***nopoint** is **nil**, a decimal point is printed out.

Slashification does not affect the printing of numbers.

base *Variable*

The value of **base** is a number that is the radix in which integers are printed, or a symbol with a **si:princ-function** property. The initial value of **base** is 10. **base** should not be greater than 36.

***nopoint** *Variable*

If the value of ***nopoint** is **nil**, a trailing decimal point is printed when an integer 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**.

3.3.2 Printed Representation of Ratios

Ratios are printed as the numerator, followed by a backslash, followed by the denominator. Ratios print in the current **base**, not always in decimal.

3.3.3 Printed Representation of Floating-point Numbers

For a single-precision floating-point number, the printer first decides whether to use ordinary notation or exponential notation. If the magnitude of the number is so large or small that the ordinary notation would require an unreasonable number of

leading or trailing zeroes, exponential notation is used. The number is printed as follows:

- An optional leading minus sign
- One or more digits
- A decimal point
- One or more digits
- 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 single-float is lost, and no extra trailing digits are printed that do not represent information in the single-float. Feeding the printed representation of a single-float back to the reader is always supposed to produce an equal single-float. Single-floats are always printed in decimal; they are not affected by slashification nor by **base** and ***nopoint**.

The printed representation of a double-precision floating-point number is very similar to that of a single-float, except that exponential notation is always used and the exponent is delimited by "d" rather than "e".

The printed representation for floating-point infinity is as follows:

- A plus or minus sign
- The digit "1"
- The appropriate Common Lisp exponent mark character
- The exponent character, an infinity sign: ∞

3.3.4 Printed Representation of Complex Numbers

The printed representation for complex numbers is:

```
#C(realpart imagpart)
```

The real and imaginary parts of the complex number are printed in the manner appropriate to their type.

3.3.5 Printed Representation of Symbols

If slashification is off, the printed representation of a symbol is simply the successive characters of the print-name of the symbol. If slashification is on, two changes must be made.

1. The symbol might require a package prefix for **read** to work correctly, assuming that the package into which **read** reads the symbol is the one in which it is being printed. (See the section "System Packages", page 615.)
2. If the printed representation would not read in as a symbol at all (that is, if the print-name looks like a number, or contains special characters), the printed representation must have one of the following kinds of quoting for those characters.

- Slashes ("/") before each special character
- Vertical bars ("|") around the whole name

The decision whether quoting is required is made using the readtable, so it is always accurate provided that **readtable** has the same value when the output is read back in as when it was printed. See the variable **readtable**, page 33.

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

3.3.6 Printed Representation of Common Lisp Character Objects

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

3.3.7 Printed Representation of Strings

If slashification is off, the printed representation of a string is simply the successive characters of the string. If slashification is on, the string is printed between double quotes, and any characters inside the string that need to be preceded by slashes are. Normally these are just double-quote and slash. Compatibly with Maclisp, carriage return is *not* ignored inside strings and vertical bars.

3.3.8 Printed Representation of Instances

If the instance has a method for the **:print-self** message, that message is sent with three arguments: the stream to print to, the current *depth* of list structure, and whether slashification is enabled. The object should print a suitable printed representation on the stream. (See the section "Flavors", page 415. Instances are discussed there.) See the section "Printed Representation of Miscellaneous Data Types", page 18. Most such objects print as described there, except with additional information such as a name. Some objects print only their name when slashification is not in effect (when **princ**ed).

3.3.9 Printed Representation of Arrays That Are Named Structures

If the array has a named structure symbol with a **named-structure-invoke** property that is the name of a function, then that function is called on five arguments:

- The symbol **:print-self**
- The object itself
- The stream to print to
- The current *depth* of list structure
- Whether slashification is enabled

A suitable printed representation should be sent to the stream. This allows you to define your own printed representation for the array's named structures. See the section "Named Structures", page 403. If the named structure symbol does not have

a **named-structure-invoke** property, the printed representation is like that for miscellaneous 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 (">").

3.3.10 Printed Representation of Arrays That Are Not Named Structures

The printed representation of an array that is not a named structure contains the following elements, in order:

- A number sign and a less-than sign ("<")
- The "**art-**" symbol for the array type
- The dimensions of the array, separated by hyphens
- A space, the machine address of the array, and a greater-than sign (">")

3.3.11 Printed Representation of Conses

The printed representation 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 printed representations 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 printed representation 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** terminates the printed representation of the list with an ellipsis (three periods) and a close-parenthesis. If the depth of recursion exceeds the value of the variable **prinlevel**, the list is printed as "**". These two features allow a kind of abbreviated printing that is more concise and suppresses detail. Of course, neither the ellipsis nor the "**" can be interpreted by **read**, since the relevant information is lost.

prinlevel *Variable*

prinlevel can be set to the maximum number of nested lists that can be printed before the printer gives up and just prints a "**". If it is **nil**, which it is initially, any number of nested lists can be printed. Otherwise, the value of **prinlevel** must be an integer.

prinlength *Variable*

prinlength can be set to the maximum number of elements of a list that is printed before the printer gives up and print a "...". If it is **nil**, which it is initially, any length list can be printed. Otherwise, the value of **prinlength** must be an integer.

3.3.12 Printed Representation of Miscellaneous Data Types

For a miscellaneous data type, the printed representation starts with a number sign and a less-than sign, the "**dtp-**" symbol for this data type, a space, and the octal machine address of the object. Then, if the object is a microcoded function, compiled function, or stack group, its name is printed. Finally, a greater-than sign is printed.

Including the machine address in the printed representation 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 are occasionally moved, and therefore their octal machine addresses are changed. It is best to shut off garbage collection temporarily when depending on these numbers.

None of the printed representations beginning with a number sign can be read back in, nor, in general, can anything produced by instances and named structures. See the section "What the Reader Recognizes", page 20. This can be a problem if, for example, you are printing a structure into a file with the intent of reading it in later. But by setting the **si:print-readably** variable, you can make sure that what you are printing can indeed be read with the reader.

si:print-readably *Variable*

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

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

The vast majority of objects that define **:print-self** messages have much in common. This macro is provided for convenience, so that users do not have to write out that repetitious code. It is also the preferred interface to **si:print-readably**. With no keywords, **sys:printing-random-object** checks the value of **si:print-readably** and signals an error if it is not **nil**. It then prints a number sign and a less-than sign, evaluates the forms in *body*, then prints a space, the octal machine address of the object, and a greater-than sign. A typical use of this macro might look like:

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

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

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

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

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

3.4 Controlling the Printed Representation of an Object

If you want to control the printed representation of an object, usually you make the object an array that is a named structure, or an instance of a flavor. See the section "Named Structures", page 403. See the section "Flavors", page 415. Occasionally, however, you might want to get control over all printing of objects in order to change in some way how they are printed. The best way to do this is to customize the behavior of **si:print-object**, which is the main internal function of the printer. All the printing functions, such as **print** and **princ**, as well as **format**, go through this function. The way to customize it is by using the "advice" facility. See the special form **advise** in *Program Development Utilities*.

3.5 How the Reader Works

The purpose of the reader is to accept characters, interpret them as the printed representation of a Lisp object, and return a corresponding Lisp object. The reader cannot accept everything that the printer produces; for example, the printed representations of arrays (other than strings), compiled code objects, closures, stack groups, and so on cannot be read in. However, it has many features that are not seen in the printer at all, such as more flexibility, comments, and convenient abbreviations for frequently used unwieldy constructs.

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 printed representation 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 printed representation of an object has been seen, and then constructs and returns that object.

3.6 What the Reader Recognizes

3.6.1 How the Reader Recognizes Integers

The reader understands the printed representations of integers in a more general way than the printer. The syntax for a simple integer is:

- An optional plus or minus sign
- A string of digits
- An optional decimal point

A simple integer is interpreted by **read** as an integer. If the trailing decimal point is present, the digits are interpreted in decimal radix; otherwise, they are considered as a number whose radix is the value of the variable **ibase**.

ibase *Variable*

The value of **ibase** is a number that is the radix in which integers are read. The initial value of **ibase** is 10. **ibase** should not be greater than 36.

read also understands a simple integer, followed by an underscore (_) or a circumflex (^), followed by another simple integer. The two simple integers are interpreted in the usual way and the character between them indicates an operation to be performed on the two integers.

- The underscore indicates a binary "left shift"; that is, the integer to its left is doubled the number of times indicated by the integer to its right. For example, **645_6** means **64500** (in octal).
- The circumflex multiplies the integer to its left by **ibase** the number of times indicated by the integer to its right. (The second integer is not allowed to have a leading minus sign.) For example, **645^3** means **645000**.

Here are some examples of valid representations of integers to be given to **read**:

```
4
23456.
-546
+45^+6
2_11
```

The reader uses the same syntax for fixnums and bignums. 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.
```

Reading Integers in Bases Greater Than 10

The reader uses letters to represent digits greater than 10. When **ibase** is greater than 10, some tokens could be read as either integers, floating-point numbers, or symbols. The reader's action on such ambiguous tokens is determined by the value of **si:*read-extended-ibase-unsigned-number*** and **si:*read-extended-ibase-signed-number***.

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

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

nil	It is never an integer.
t	It is always an integer.
:sharpsign	It is a symbol or floating-point number at top level, but an integer after **#X** or **#*n*R**.
:single	It is a symbol or floating-point number except immediately after **#X** or **#*n*R**.

The default value is **:single**.

In the table below, the token FACE for each case could be a symbol or a hexadecimal number. **:single** makes it an integer on the second line, but a symbol on the first and third lines. **:sharpsign** makes it an integer on both the second and third lines.

	nil	**t**	**:single**	**:sharpsign**
FACE	symbol	integer	symbol	symbol
#xFACE	symbol	integer	integer	integer
#x(FACE FF 1234 5C00)	symbol	integer	symbol	integer
1d0	float	integer	float	float

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

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

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

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

The default value is **:sharpsign**.

In the table below, the token FACE for each case could be a symbol or a hexadecimal number. **:single** makes it an integer on the second line, but a symbol on the first and third lines. **:sharpsign** makes it an integer on both the second and third lines.

	nil	**t**	**:single**	**:sharpsign**
+FACE	symbol	integer	symbol	symbol
#x+FACE	symbol	integer	integer	integer
#x(+FACE +FF 1234 +5C00)	symbol	integer	symbol	integer
+1d0	float	integer	float	float

3.6.2 How the Reader Recognizes Ratios

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

3.6.3 How the Reader Recognizes Floating-point Numbers

The syntax for floating-point numbers is:

- An optional plus or minus sign
- (Optionally) some digits
- A decimal point
- One or more digits
- And an optional trailing exponent, consisting of an exponent letter, an optional minus sign, and digits representing the power of ten

If no exponent is present, the number is a single-float. If an exponent is present, the exponent letter determines the type of the number.

Floating-point Exponent Characters

The reader accepts all Common Lisp floating-point exponent characters. Following is a summary of floating-point exponent characters and the way numbers containing them are read.

Character	*Floating-point precision*
D or d	double-precision
E or e	depends on value of **cl:*read-default-float-format***
F or f	single-precision
L or l	double-precision
S or s	single-precision

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

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

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

Value	*Floating-point precision*
cl:single-float	single-precision
cl:short-float	single-precision
cl:double-float	double-precision
cl:long-float	double-precision

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

As a special case, the reader recognizes IEEE floating-point infinity. The syntax for infinity is as follows:

- A required plus or minus sign
- The digit "1"
- Any of the Common Lisp exponent mark characters
- And the exponent character, which must be an infinity sign: ∞

Here are some examples of printed representations that read as single-floats:

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

Here are some examples of printed representations that read as double-floats:

```
0d0
1.5d9
-42D3
1.d5
-1d∞
```

3.6.4 How the Reader Recognizes Complex Numbers

The reader recognizes **#C(number1 number2)** as a complex number. The numbers can be of any noncomplex type and are read according to the rules for those types. **number1** is used as the real part and **number2** is used as the imaginary part. If the types of the real and imaginary parts differ, coercion rules are applied to make them the same. If the real part is rational and the imaginary part is integer zero, the result is simply the rational real part.

3.6.5 How the Reader Recognizes Symbols

A string of letters, numbers, and "extended alphabetic" characters is recognized by the reader as a symbol, provided it cannot be interpreted as a number. Alphabetic case is ignored in symbols; lowercase letters are translated to uppercase. When the reader sees the printed representation of a symbol, it *interns* it on a *package*. See the section "Packages", page 557.

Symbols can start with digits; for example, **read** accepts one named "-345T". If you want to put strange characters (such as lowercase letters, parentheses, or reader macro characters) inside the name of a symbol, put a slash before each strange character. If you want to have a symbol whose print-name looks like a number, put a slash before some character in the name. You can also enclose the name of a symbol in vertical bars, which quotes all characters inside, except vertical bars and slashes, which must be quoted with slash.

Examples of symbols:

```
foo
bar/(baz/)
34w23
|Frob Sale|
```

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

3.6.6 How the Reader Recognizes Strings

The reader recognizes strings, which should be surrounded by double quotes. If you want to put a double quote or a slash inside a string, precede it by a slash. Examples of strings:

```
"This is a typical string."
"That is known as a /"cons cell/" in Lisp."
```

3.6.7 How the Reader Recognizes Conses

When **read** sees an open parenthesis, it knows that the printed representation 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 printed representation for a cons is only recognized if it is surrounded by delimiters (typically spaces). Thus dot can 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 Symbolics-Lisp it reads as a list of a symbol **a.b**.

Tokens that consist of more than one dot, but no other characters, are valid symbols in Zetalisp but errors in Common Lisp. For Common Lisp, the variable **si:*read-multi-dot-tokens-as-symbols*** should be set to **nil**.

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

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

If the circle-X (⊗) character is encountered, it is an octal escape, which might be useful for including unusual 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.

3.6.8 How the Reader Recognizes 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 (')	An abbreviation to make it easier to put constants in programs. *'foo* reads the same as **(quote** *foo*).
Semicolon (;)	Used to enter comments. The semicolon and everything up through the next carriage return are ignored. Thus a comment can be put at the end of any line without affecting the reader.
Backquote (`)	Makes it easier to write programs to construct lists and trees by using a template. See the section "Backquote", page 345.
Comma (,)	Part of the syntax of backquote. It is invalid if used other than inside the body of a backquote. See the section "Backquote", page 345.
Sharp sign (#)	Introduces a number of other syntax extensions. See the section "Sharp-sign Reader Macros", page 27. Unlike the preceding characters, sharp sign is not a delimiter. A sharp sign in the middle of a symbol is an ordinary character.

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

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

si:read-recursive *stream* *Function*

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

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

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

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

3.7 Sharp-sign Reader Macros

The reader's syntax includes several abbreviations introduced by sharp sign (#). These take the general form of a sharp sign, a second character that 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.

The function **set-syntax-#-macro-char** can be used to define your own sharp-sign abbreviations.

#\ or #/

#*x* (or #/*x*, which is identical) reads in as the number that is the character code for the character *x*. For example, #**a** is equivalent to **141** but clearer in its intent. This is the recommended way to include character constants in your code. Note that the slash causes this construct to be parsed correctly by the editor.

As in strings, upper- and lowercase letters are distinguished after #\. Any character works after #\, even those that are normally special to **read**, such as parentheses.

#*name* (or #/*name*) reads in as the number which is the character code for the nonprinting character symbolized by *name*. A large number of character names are recognized. See the section "Special Character Names", page 32. For example, #**return** reads in as an integer, being the character code for the Return character in the Symbolics Lisp Machine character set. In general, the names that are written on the keyboard keys are accepted. The abbreviations **cr** for **return** and **sp** for **space** are accepted and generally preferred, since these characters are used so frequently. The page separator

character is called **page**, although **form** and **clear-screen** are also accepted since the keyboard has one of those legends on the page key. The rules for reading *name* are the same as those for symbols; thus upper- and lowercase letters are not distinguished, and the name must be terminated by a delimiter such as a space, a carriage return, or a parenthesis.

When the system types out the name of a special character, it uses the same table as the #\ reader; therefore, any character name typed out is acceptable as input.

#\ (or #/) can also be used to read in the names of characters that have control and meta bits set. The syntax looks like **#\control-meta-b** to get a "B" character with the control and meta bits set. You can use any of the prefix bit names **control**, **meta**, **hyper**, and **super**. They can be in any order, and upper- and lowercase letters are not distinguished. The last hyphen can be followed by a single character, or by any of the special character names normally recognized by #\. If it is a single character, it is treated the same way the reader normally treats characters in symbols; if you want to use a lowercase character or a special character such as a parenthesis, you must precede it with a slash character. Examples: **#\Hyper-Super-A**, **\meta-hyper-roman-i**, **#\CTRL-META-/(**.

The character can also be modified with control and meta bits by inserting one or more special characters between the # and the \. This syntax is obsolete since it is not mnemonic and it generally unclear. However, it is used in some old programs, so here is how it is defined. #α\x generates Control-*x*. #β\x generates Meta-*x*. #π\x generates Super-*x*. #λ\x generates Hyper-*x*. These can be combined, for instance #πβ\& generates Super-Meta-ampersand. Also, #ε\x is an abbreviation for #αβ\x. When control bits are specified, and *x* is a lowercase alphabetic character, the character code for the uppercase version of the character is produced.

In Common Lisp, #*char* (or #/*char*) can cause *char* to read as a character object instead of an integer, depending on the readtable.

#^ #^*x* is exactly like #α/*x* if the input is being read by Symbolics-Lisp; it generates Control-*x*. In Maclisp *x* is converted to uppercase and then exclusive-or'ed with 100 (octal). Thus #^*x* always generates the character returned by **tyi** if the user holds down the control key and types *x*. (In Maclisp #α/*x* sets the bit set by the CONTROL key when the TTY is open in **fixnum** mode.)

#' #'*foo* is an abbreviation for **(function** *foo***)**. *foo* is the printed representation 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 printed representation of a Lisp form) at read time, unless the compiler is doing the reading, in which case it is arranged that *foo* be evaluated when the QFASL file is loaded. This is a way, for example, to

include in your code complex list-structure constants that cannot be written with **quote**. Note that the reader does not put **quote** around the result of the evaluation. You must do this yourself, 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 printed representation of a Lisp form) at read time, regardless of who is doing the reading.

#: *#:name* reads *name* as an uninterned symbol. It always creates a new symbol. Like all package prefixes, **#:** can be followed by any expression. Example: **#:(a b c)**.

#B *#Brational* reads *rational* (an integer or a ratio) in binary (radix 2). Examples:

```
#B1101 <=> 13.
#B1100\100 <=> 3
```

#O **#O** *number* reads *number* in octal regardless of the setting of **ibase**. Actually, any expression can be prefixed by **#O**; it is read with **ibase** bound to 8.

#X **#X** *number* reads *number* in radix 16. (hexadecimal) regardless of the setting of *ibase*. As with **#O**, any expression can be prefixed by **#X**. The *number* can contain embedded hexadecimal "digits" A through F as well as the 0 through 9. See the section "Reading Integers in Bases Greater Than 10", page 21.

#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 is read with **ibase** bound to *radix*. *radix* must consist of only digits, and it is read in decimal. *number* can consist of both numeric and alphabetic digits, depending upon *radix*.

For example, **#3R102** is another way of writing **11.** and **#11R32** is another way of writing **35.**

#Q **#Q** *foo* reads as *foo* if the input is being read by Symbolics-Lisp, otherwise it reads as nothing (whitespace).

#M **#M** *foo* reads as *foo* if the input is being read into Maclisp, otherwise it reads as nothing (whitespace).

#N **#N** *foo* reads as *foo* if the input is being read into NIL or compiled to run in NIL, otherwise it reads as nothing (whitespace). Also, during the reading of *foo*, the reader temporarily defines various NIL-compatible sharp-sign reader macros (such as **#!** and **#"**) in order to parse the form correctly, even though it is 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* can be a boolean expression composed of **and**, **or**, and **not** operators and symbols representing items which can 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 Symbolics-Lisp, and is thus equivalent to **#Q** *form*. Similarly, **#+maclisp** *form* is equivalent to **#M** *form*. **#+(or lispm nil)** *form* makes *form* exist on either Symbolics-Lisp or in NIL. Note that items can be added to the **(status features)** list by means of **(sstatus feature** *feature*), thus allowing the user to selectively interpret or compile pieces of code by parameterizing this list. See the section "**status** and **sstatus**" in *User's Guide to Symbolics Computers*.

#- **#-***feature form* is equivalent to **#+(not** *feature*) *form*.

#| **#|** begins a comment for the Lisp reader. The reader ignores everything until the next **|#**, which closes the comment. Note that if the **|#** is inside a comment that begins with a semicolon, it is *not* ignored; it closes the comment that began with the preceding **#|**. **#|** and **|#** can be on different lines, and **#|...|#** pairs can be nested.

Using **#|...|#** always works for the Lisp reader. The editor, however, does not understand the reader's interpretation of **#|...|#**. Instead, the editor retains its knowledge of Lisp expressions. Symbols can be named with vertical bars, so the editor (not the reader) behaves as if **#|...|#** is the name of a symbol surrounded by pound signs, instead of a comment.

Note: Use **#||...||#** instead of **#|...|#** to comment out Lisp code.

The reader views **#||...||#** as a comment: the comment prologue is **#|**, the comment body is **|...|**, and the comment epilogue is **|#**. The editor, however, interprets **#||...||#** as a pound sign (#), a symbol with a zero length print name (||), lisp code (...), another symbol with a zero length print name (||), and a stray pound sign (#). Therefore, inside a **#||...||#**, the editor commands that operate on Lisp code, such as balancing parentheses and indenting code, work correctly.

#< This is not valid reader syntax. It is used in the printed representation of objects that cannot be read back in. Attempting to read a **#<** causes an error.

#◆ **#◆** turns infix expression syntax into regular Lisp code. It is intended for people who like to use traditional arithmetic expressions in Lisp code. It is not intended to be extensible or to be a full programming language. We do not intend to extend it into one.

```
(defun my-add (a b)
  #◆a+b◆)
```

The quoting character is backslash. It is necessary for including special symbols (such as -) in variable names.

! reads one Lisp expression, which can use this reader-macro inside itself.

#◊ supports the following syntax:

Delimiters — Begin the reader macro with #◊, complete it with ◊.

```
#◊a+b-c◊
```

Escape characters — Special characters in symbol names must be preceded with backslash (\). You can escape to normal Lisp in an infix expression; precede the Lisp form with exclamation point (!).

Symbols — Start symbols with a letter. They can contain digits and underscore characters. Any other characters need to be quoted with \.

Operators — It accepts the following classes of operators. Arithmetic operator precedence is like that in FORTRAN and PL/I.

Operator	*Infix*	*Lisp Equivalent*
Assignment	x : y	(setf x y)
Functions	f(x,y)	(f x y) -- also works for defstruct accessors, and so on.
Array ref	a[i,j]	(aref a i j)
Unary ops	+ - not	*same*
Binary ops	+ - * / ^ = ≠ < ≤ > ≥ and or	*same*
Conditional	if p then c	(if p c)
	if p then c else a	(if p c a)
Grouping:	(a, b, c)	(progn a b c) -- even works for (1+2)/3

The following example shows matrix multiplication using an infix expression.

```
(defun matrix-multiply (a b)
  (let ((n (array-dimension-n 2 a)))
    (unless (= n (array-dimension-n 1 b))
      (ferror "Matrices ~S and ~S do not have compatible dimensions") a b)
    (let ((d1 (array-dimension-n 1 a))
          (d2 (array-dimension-n 2 b)))
      (let ((c #◊ make\-array(list(d1, d2), !:type, art\-float)◊ ))
        (dotimes (i d1)
          (dotimes (j d2)
            #◊ c[i,j] : !(loop for k below n sum #◊ a[i,k]*b[k,j] ◊)◊))
        c))))
```

The line containing the infix expression could also have been written like this:

```
(let ((sum 0))
  (dotimes (k n) #◊ sum:sum+a[i,k]*b[k,j] ◊)
  #◊ c[i,j]:sum ◊)
```

3.8 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.

The special characters are:

```
Abort                  Break                Call                 Clear-Input=Clear
Clear-Screen           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=Form            Quote
Resume                 Return=CR            Rubout               Space=SP
Status                 Stop-Output          System               Tab
Terminal=ESC
```

These are printing characters that also have special names because they can be hard to type on a PDP-10.

```
Altmode          Circle-Plus     Delta          Gamma
Integral         Lambda          Plus-Minus     Up-Arrow
```

The following are special characters sometimes used to represent single and double mouse clicks. The buttons can be called either **l**, **m**, **r** or **1**, **2**, **3** depending on stylistic preference. These characters all contain the **%%kbd-mouse** bit.

Mouse-L-1=Mouse-1-1 Mouse-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

3.9 The Readtable

A data structure called the *readtable* is used to control the reader. It contains information about the syntax of each character. Initially it is set up to give the standard Lisp meanings to all the characters, but you can change the meanings of characters to alter and customize the syntax of characters. It is also possible to have several readtables describing different syntaxes and to switch from one to another by binding the symbol **readtable**.

readtable *Variable*

The value of **readtable** is the current readtable. This starts out as a copy of **si:initial-readtable**. You can bind this variable to temporarily change the readtable being used.

si:initial-readtable *Variable*

The value of **si:initial-readtable** is the initial standard readtable. You should never change the contents of either this readtable or **si:initial-readtable**; only examine it, by using it as the *from-readtable* argument to **copy-readtable** or **set-syntax-from-char**. Change **readtable** instead.

You can program the reader by changing the readtable in any of three ways.

- You can create a completely new readtable, using the readtable compiler (sys:io;rtc) to define new kinds of syntax and to assign syntax classes to characters. Use of the readtable compiler is not documented here.
- The syntax of a character can be set to one of several predefined possibilities.
- A character can be made into a *macro character*, whose interpretation is controlled by a user-supplied function that is called when the character is read.

3.9.1 Functions That Create New Readtables

copy-readtable &optional *from-readtable to-readtable* *Function*

from-readtable, which defaults to the current readtable, is copied. If *to-readtable* is unsupplied or **nil**, a fresh copy is made. Otherwise *to-readtable* is clobbered with the copy. Use **copy-readtable** to get a private readtable before using the other readtable functions to change the syntax of characters in it. The value of **readtable** at the start of a Symbolics Lisp Machine session is the initial standard readtable, which usually should not be modified.

3.9.2 Functions That Change Character Syntax

set-syntax-from-char *to-char from-char* &optional *to-readtable from-readtable* *Function*

Makes the syntax of *to-char* in *to-readtable* be the same as the syntax of *from-char* in *from-readtable*. *to-readtable* defaults to the current readtable, and *from-readtable* defaults to the initial standard readtable.

set-character-translation *from-char to-char* &optional *readtable* *Function*

Changes *readtable* so that *from-char* is translated to *to-char* upon read-in, when *readtable* is the current readtable. This is normally used only for translating lowercase letters to uppercase. Character translations are turned off by slash, string quotes, and vertical bars. *readtable* defaults to the current readtable.

set-syntax-from-description *char description* &optional *readtable* *Function*

Sets the syntax of *char* in *readtable* to be that described by the symbol *description*. The following descriptions are defined in the standard readtable:

si:alphabetic	An ordinary character such as "**A**".
si:break	A token separator such as "(". (Obviously left parenthesis has other properties besides being a break.)
si:whitespace	A token separator that can be ignored, such as "@".
si:single	A self-delimiting single-character symbol. The initial readtable does not contain any of these.
si:slash	The character quoter. In the initial readtable this is "/".
si:verticalbar	The symbol print-name quoter. In the initial readtable this is "|".
si:doublequote	The string quoter. In the initial readtable this is '"'.
si:macro	A macro character. Do not use this; use **set-syntax-macro-char**.
si:circlecross	The octal escape for special characters. In the initial readtable this is "⊕".
si:bitscale	A character that causes the integer to its left to be doubled the number of times indicated by the integer to its right. In the initial readtable this is "_". See the section "What the Reader Recognizes", page 20.
si:digitscale	A character that causes the integer to its left to be multiplied by **ibase** the number of times indicated by the integer to its right. In the initial readtable this is "^". See the section "What the Reader Recognizes", page 20.

si:non-terminating-macro
A macro character that is not a token separator. This is a macro character if seen alone but is just a symbol constituent inside a symbol. You can use it as a character of a symbol other than the first without slashing it. (# would be one of these if it were not built into the reader.)

readtable defaults to the current readtable.

3.9.3 Functions That Change Characters Into Macro Characters

set-syntax-macro-char *char function* &optional *readtable non-terminating-p* *Function*

Changes *readtable* so that *char* is a macro character. When *char* is read, *function* is called. *readtable* defaults to the current readtable.

function is called with two arguments: *list-so-far* and the input stream. When a list is being read, *list-so-far* is that list (**nil** if this is the first element). At the "top level" of **read**, *list-so-far* is the symbol **:toplevel**. After a dotted-pair dot, *list-so-far* is the symbol **:after-dot**. *function* can 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 is *list-so-far* with something else **nconc**'ed onto the end. At top level and after a dot if *splice-p* is non-**nil** the *thing* is ignored and the macro character does not contribute anything to the result of **read**. *type* is a historical artifact and is not really used; **nil** is a safe value. Most macro character functions return just one value and let the other two default to **nil**.

function should not have any side effects other than on the stream and *list-so-far*. Because of the way the input editor works, *function* can be called several times during the reading of a single expression in which the macro character only appears once.

char is given the same syntax that single-quote, backquote, and comma have in the initial readtable (it is called **:macro** syntax).

If *non-terminating-p* is **nil** (the default), **set-syntax-macro-char** makes a normal macro character. If it is **t**, **set-syntax-macro-char** makes a nonterminating macro character. A nonterminating macro character is a character that acts as a reader macro if seen between tokens, but if seen inside a token it acts as an ordinary letter; it does not terminate the token.

set-syntax-#-macro-char *char function* &optional *readtable* *Function*

Causes *function* to be called when *#char* is read. *readtable* defaults to the current readtable. The function's arguments and return values are the same as for normal macro characters. When *function* is called, the special variable **si:xr-sharp-argument** contains **nil** or a number that is the number or special bits between the # and *char*.

3.9.4 Readtable Functions for Maclisp Compatibility

setsyntax *character arg2 arg3* *Function*

This exists only for Maclisp compatibility. The other readtable functions are preferred in new programs. The syntax of *character* is altered in the current readtable, according to *arg2* and *arg3*. *character* can be an integer, a symbol, or a string, that is, anything acceptable to the **character** function. *arg2* is usually a keyword; it can be in any package since this is a Maclisp compatibility function. The following values are allowed for *arg2*:

:macro	The character becomes a macro character. *arg3* is the name of a function to be invoked when this character is read. The function takes no arguments, can **tyi** or **read** from **standard-input** (that is, can call **tyi** or **read** without specifying a stream), and returns an object that is taken as the result of the read.
:splicing	Like **:macro**, but the object returned by the macro function is a list that is **nconc**ed into the list being read. If the character is read not inside a list (at top level or after a dotted-pair dot), then it can 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 an integer, the character is translated to that character.
nil	The syntax of the character is not changed, but if *arg3* is an integer, 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 an integer, the character is translated to that character.

setsyntax-sharp-macro *character type function* &optional *readtable* *Function*

This exists only for Maclisp compatibility. **set-syntax-#-macro-char** is preferred. If *function* is **nil**, *#character* is turned off, otherwise it becomes a macro that calls *function*. *type* can be **:macro**, **:peek-macro**, **:splicing**, or **:peek-splicing**. The splicing part controls whether *function* returns a single

object or a list of objects. Specifying peek causes *character* to remain in the input stream when *function* is called; this is useful if *character* is something like a left parenthesis. *function* gets one argument, which is **nil** or the number between the # and the *character*.

PART II.

Lists

4. Manipulating List Structure

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

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

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

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

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

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

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

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

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

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

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

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

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

A circular list is like a list except that the cdr of the last cons, instead of being **nil**,

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

The Symbolics Lisp Machine internally uses a storage scheme called "cdr coding" to represent conses. This scheme is intended to reduce the amount of storage used in lists. The use of cdr-coding is invisible to programs except in terms of storage efficiency; programs work the same way whether or not lists are cdr-coded. 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 might want to learn about the cdr-coded representation and how to control it. See the section "Cdr-coding", page 56.

4.1 Conses

car *x* *Function*

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

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

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

cdr *x* *Function*

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

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

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

4.1.1 Composition of Cars and Cdrs

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

caar *x* *Function*

`(caar x)` is the same as `(car (car x))`

cadr *x* *Function*

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

cdar *x* *Function*

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

cddr *x* *Function*

`(cddr x)` is the same as `(cdr (cdr x))`

caaar *x* *Function*

`(caaar x)` is the same as `(car (car (car x)))`

caadr *x* *Function*

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

cadar *x* *Function*

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

caddr *x* *Function*

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

cdaar *x* *Function*

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

cdadr *x* *Function*

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

cddar *x* *Function*

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

cdddr *x* *Function*

`(cdddr x)` is the same as `(cdr (cdr (cdr x)))`

caaaar *x* *Function*

`(caaaar x)` is the same as `(car (car (car (car x))))`

caaadr *x* *Function*

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

caadar *x* *Function*

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

caaddr *x* *Function*

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

cadaar *x* *Function*

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

cadadr *x* *Function*

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

caddar *x* *Function*

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

cadddr *x* *Function*

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

cdaaar *x* *Function*

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

cdaadr *x* *Function*

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

cdadar *x* *Function*

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

cdaddr *x* *Function*

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

cddaar *x* *Function*

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

cddadr *x* *Function*

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

cdddar *x* *Function*

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

cddddr *x* *Function*

`(cddddr x)` is the same as `(cdr (cdr (cdr (cdr x))))`

cons *x y* *Function*

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

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

ncons *x* *Function*

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

xcons *x y* *Function*

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

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

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

This function creates a *cons* in a specific *area*. (Areas are an advanced feature of storage management.) See the section "Areas" in *Internals, Processes, and Storage Management*. The first two arguments are the same as the two arguments to **cons**, and the third is the number of the area in which to create the *cons*. Example:

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

ncons-in-area *x area-number* *Function*

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

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

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

The backquote reader macro facility is also generally useful for creating list structure, especially mostly constant list structure, or forms constructed by plugging variables into a template. See the section "Backquote", page 345.

car-location *cons* *Function*

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

Note: there is no **cdr-location** function; it is difficult because of the cdr-coding scheme. See the section "Cdr-coding", page 56.

4.2 Basic List Operations

length *list* *Function*

length returns the length of *list*. The length of a list is the number of elements in it. Examples:

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

length could have been defined by:

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

or by:

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

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

first *list* *Function*

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

second *list* *Function*

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

third *list* *Function*

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

fourth *list* *Function*

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

fifth *list* *Function*

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

sixth *list* *Function*

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

seventh *list* *Function*

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

rest1 *list* *Function*

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

rest2 *list* *Function*

rest2 returns the rest of the elements of a list, starting with element 2 (counting the first element as the zeroth). Thus **rest2** is identical to **cddr**. The reason these names are provided is that they make more sense when you are thinking of the argument as a list rather than just as a cons.

rest3 *list* *Function*

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

rest4 *list* *Function*

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

nth *n list* *Function*

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

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

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

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

nth could have been defined by:

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

nthcdr *n list* *Function*

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

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

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

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

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

last *list* *Function*

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

```
(setq x '(a b c d))
(last x) => (d)
(rplacd (last x) '(e f))
x => '(a b c d e f)
```

last could have been defined by:

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

list &rest *args* *Function*

list constructs and returns a list of its arguments. Example:

```
(list 3 4 'a (car '(b . c)) (+ 6 -2)) => (3 4 a b 4)
```

list could have been defined by:

```
(defun list (&rest args)
    (let ((list (make-list (length args))))
      (do ((l list (cdr l))
           (a args (cdr a)))
          ((null a) list)
        (rplaca l (car a)))))
```

list* &rest *args* *Function*

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

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

This is like

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

More examples:

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

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

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

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

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

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

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

:area The value specifies in which area the list should be created. See the section "Areas" in *Internals, Processes, and Storage Management*. It should be either an area number (an integer), or **nil** to mean the default area.

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

make-list always creates a *cdr-coded* list. See the section "Cdr-coding", page 56. Examples:

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

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

circular-list &rest *args* *Function*

circular-list constructs a circular list whose elements are **args**, repeated infinitely. **circular-list** is the same as **list** except that the list itself is used as the last cdr, instead of **nil**. **circular-list** is especially useful with **mapcar**, as in the expression:

```
(mapcar (function +) foo (circular-list 5))
```

which adds each element of **foo** to 5. **circular-list** could have been defined by:

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

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

Returns a list that is **equal** to *list*, but not **eq**. **copylist** does not copy any elements of the list: only the conses of the list itself. The returned list is fully cdr-coded to minimize storage. See the section "Cdr-coding", page 56. If the list is "dotted", that is, **(cdr (last** *list*)) is a non-**nil** atom, this is true of the returned list also. You can optionally specify the area in which to create the new copy.

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

This is the same as **copylist** except that the last cons of the resulting list is never cdr-coded. See the section "Cdr-coding", page 56. This makes for increased efficiency if you **nconc** something onto the list later.

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

copyalist is for copying association lists. See the section "Tables", page 59. The *list* is copied, as in **copylist**. In addition, each element of *list* that is a cons is replaced in the copy by a new cons with the same car and cdr. You can optionally specify the area in which to create the new copy.

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

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

copytree-share *tree* &optional *area* (*hash* **(make-equal-hash-table)**) *Function*

copytree-share is similar to **copytree**; it makes a copy of an arbitrary structure of conses, copying at all levels, and optimally cdr-coding. However, it also assures that all lists or tails of lists are optimally shared when **equal**.

copytree-share takes as arguments the tree to be copied, and optionally a storage area and an externally created hash table to be used for the equality testing.

Note: **copytree-share** might be very slow in the general case, for long lists. However, applying it at the appropriate level of a specific structure-copying routine (furnishing a common externally created hash table) is likely to yield all the sharing possible, at a much lower computational cost. For example, **copytree-share** could be applied only to the branches of a long alist.

Example:

```
(copytree-share '((1 2 3) (1 2 3) (0 1 2 3) (0 2 3)))
```

If `x = '(1 2 3)`, the above returns (roughly):

```
'(,x ,x (0 . ,x) (0 . ,(cdr x)))
```

reverse *list* *Function*

reverse creates a new list whose elements are the elements of *list* taken in reverse order. **reverse** does not modify its argument, unlike **nreverse**, which is faster but does modify its argument. The list created by **reverse** is not cdr-coded. Example:

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

reverse could have been defined by:

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

nreverse *list* *Function*

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

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

nreverse could have been defined by:

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

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

nreverse does something inefficient with cdr-coded lists, because it just uses **rplacd** in the straightforward way. See the section "Cdr-coding", page 56. Using **reverse** might be preferable in some cases.

append &rest *lists* *Function*

The arguments to **append** are lists. The result is a list that is the

concatenation of the arguments. The arguments are not changed (see **nconc**). Example:

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

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

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

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

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

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

These definitions do not express the full functionality of **append**; the real definition minimizes storage utilization by cdr-coding the list it produces, using *cdr-next* except at the end where a full node is used to link to the last argument, unless the last argument is **nil** in which case *cdr-nil* is used. See the section "Cdr-coding", page 56.

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

nconc &rest *lists* *Function*

nconc takes lists as arguments. It returns a list that is the arguments concatenated together. The arguments are changed, rather than copied. See the function **append**, page 51. Example:

```
(setq x '(a b c))
(setq y '(d e f))
(nconc x y) => (a b c d e f)
x => (a b c d e f)
```

Note that the value of **x** is now different, since its last cons has been **rplacd**d to the value of **y**. If the nconc form is evaluated again, it would yield a piece of "circular" list structure, whose printed representation would be **(a b c d e f d e f d e f ...)**, repeating forever.

nconc could have been defined by:

```
(defun nconc (x y)                ;for simplicity, this definition
    (cond ((null x) y)            ;only works for 2 arguments.
          (t (rplacd (last x) y) ;hook y onto x
             x)))                 ;and return the modified x.
```

nreconc *x y* *Function*

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

nreconc could have been defined by:

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

using the same **nreverse1** as above.

butlast *list* *Function*

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

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

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

nbutlast *list* *Function*

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

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

firstn *n list* *Function*

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

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

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

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

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

ldiff *list sublist* *Function*

list should be a list, and *sublist* should be one of the conses that make up *list*. **ldiff** (meaning "list difference") returns 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.

4.3 Alteration of List Structure

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

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

rplaca *x y* *Function*

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

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

rplacd *x y* *Function*

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

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

subst *new old tree* *Function*

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

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

subst could have been defined by:

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

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

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

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

nsubst *new old tree* *Function*

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

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

sublis *alist tree* *Function*

sublis makes substitutions for symbols in a tree. The first argument to **sublis** is an association list. See the section "Tables", page 59. The second argument is the tree in which substitutions are to be made. **sublis** looks at all symbols in the fringe of the tree; if a symbol appears in the association list, occurrences of it are replaced by the object with which it is associated. The argument is not modified; new conses are created where necessary and only where necessary, so the newly created tree shares as much of its substructure as possible with the old. For example, if no substitutions are made, the result is just the old tree. Example:

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

sublis could have been defined by:

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

nsublis *alist tree* *Function*

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

nsublis could have been defined by:

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

4.4 Cdr-coding

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

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

Very often, conses are used to store lists. If the above representation is used, a list of n elements requires two times n words of memory: n to hold the pointers to the

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

The way it works is that there is an extra two-bit field in every word of memory, called the "cdr-code" field. There are three meaningful values that this field can have, which are called cdr-normal, cdr-next, and cdr-nil. The regular, noncompact way to store a cons is by two contiguous words, the first of which holds the car and the second of which holds the cdr. In this case, the cdr code of the first word is cdr-normal. (The cdr code of the second word does not matter; 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 find that they work fine even if you mix the two kinds of storage representation within the same list. There is no problem with doing that.

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

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

writes to a location, it first reads the location, and if it contains an invisible pointer then it writes to the location addressed by the invisible pointer instead. (This is a somewhat simplified explanation; actually there are several kinds of invisible pointer that are interpreted in different ways at different times, used for things other than the cdr coding scheme.)

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

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

Why is any of this important to users? In fact, it is all invisible to you; everything works the same way whether or not compact representation is used, from the point of view of the semantics of the language. That is, the only difference that any of this makes is in efficiency. The compact representation is more efficient in most cases. However, if the conses are going to get **rplacd**'ed, then invisible pointers are created, extra memory is allocated, and the compact representation is 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 are subject to **rplacd** operations, including **nconc** and **nreverse**, and the compact representation for other structures. The functions **cons**, **xcons**, **ncons**, and their area variants make conses in the normal representation. The functions **list**, **list***, **list-in-area**, **make-list**, and **append** use the compact representation. The other list-creating functions, including **read**, currently make normal lists, although this might get changed. Some functions, such as **sort**, take special care to operate efficiently on compact lists (**sort** effectively treats them as arrays). **nreverse** is rather slow on compact lists, since it simply uses **rplacd**.

(copylist *x***)** is a suitable way to copy a list, converting it into compact form. See the function **copylist**, page 50.

4.5 Tables

Symbolics-Lisp includes functions that simplify the maintenance of tabular data structures of several varieties. The simplest is a plain list of items, which models (approximately) the concept of a *set*. There are functions to add (**cons**), remove (**delete, delq, del, del-if, del-if-not, remove, remq, rem, rem-if, rem-if-not**), and search for (**member, memq, mem**) items in a list. Set union, intersection, and difference functions can be easily written using these.

Association lists are very commonly used. An association list is a list of conses. The car of each cons is a "key" and the cdr is a "datum", or a list of associated data. The functions **assoc, assq, ass, memass**, and **rassoc** can be used to retrieve the data, given the key. For example:

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

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

Structured records can be stored as association lists or as stereotyped cons-structures where each element of the structure has a certain car-cdr path associated with it. However, these are better implemented using structure macros. See the section "Structure Macros", page 377.

Simple list-structure is very convenient, but might not be efficient enough for large data bases because it takes a long time to search a long list. Symbolics-Lisp includes hash table facilities for more efficient but more complex tables, and a hashing function (**sxhash**) to aid you in constructing your own facilities. See the section "Hash Tables", page 69.

4.6 Lists as Tables

memq *item list* *Function*

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

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

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

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

memq could have been defined by:

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

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

member *item list* *Function*

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

member could have been defined by:

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

mem *predicate item list* *Function*

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

mem is usually used with equality predicates other than **eq** and **equal**, such as =, **char-equal** or **string-equal**. It can also be used with noncommutative predicates. The predicate is called with *item* as its first argument and the element of *list* as its second argument, so:

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

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

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

find-position-in-list looks down *list* for an element that is **eq** to *item*, like **memq**. However, it returns the numeric index in the list at which it found the first occurrence of *item*, or **nil** if it did not find it at all. This function is sort of the complement of **nth**; like **nth**, it is zero-based. See the function **nth**, page 47. Examples:

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

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

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

tailp *sublist list* *Function*

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

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

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

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

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

rather than:

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

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

(**delq** *item list n*) is like (**delq** *item list*) except only the first *n* instances of *item* are deleted. *n* is allowed to be zero. If *n* is greater than or equal to the number of occurrences of *item* in the list, all occurrences of *item* in the list are deleted. Example:

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

delq could have been defined by:

```
(defun delq (item list &optional (n -1))
  (cond ((or (atom list) (zerop n)) list)
        ((eq item (car list))
         (delq item (cdr list) (1- n)))
        (t (rplacd list (delq item (cdr list) n)))))
```

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

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

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

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

del is the same as **delq** except that it takes an extra argument that should be a predicate of two arguments, which is used for the comparison instead of **eq**. **(del 'eq a b)** is the same as **(delq a b)**. See the function **mem**, page 60.

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

remq is similar to **delq**, except that the list is not altered; rather, a new list is returned. Examples:

```
(setq x '(a b c d e f))
(remq 'b x) => (a c d e f)
x => (a b c d e f)
(remq 'b '(a b c b a b) 2) => (a c a b)
```

remove *item list* &optional *n* *Function*

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

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

rem is the same as **remq** except that it takes an extra argument that should be a predicate of two arguments, which is used for the comparison instead of **eq**. **(rem 'eq a b)** is the same as **(remq a b)**. See the function **mem**, page 60.

union &rest *lists* *Function*

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

intersection &rest *lists* *Function*

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

nunion &rest *lists* *Function*

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

nintersection &rest *lists* *Function*

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

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

subset and **rem-if-not** do the same thing, but they are used in different contexts. **rem-if-not** means "remove if this condition is not true"; that is, it keeps the elements for which *predicate* is true. **subset** refers to the function's action if *list* is considered to represent a mathematical set.

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**.

If *extra-lists* is present, each element of *extra-lists* (that is, each further argument to **subset** or **rem-if-not**) is a list of objects to be passed to *predicate* as *predicate*'s second argument, third argument, and so on. The reason for this is that *predicate* might be a function of many arguments; *extra-lists* lets you control what values are passed as additional arguments to *predicate*. However, the list returned by **subset** or **rem-if-not** is still a "subset" of those values that were passed as the *first* argument in the various calls to *predicate*.

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

subset and **rem-if-not** do the same thing, but they are used in different contexts. **rem-if-not** means "remove if this condition is not true"; that is, it keeps the elements for which *predicate* is true. **subset** refers to the function's action if *list* is considered to represent a mathematical set.

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**.

If *extra-lists* is present, each element of *extra-lists* (that is, each further argument to **subset** or **rem-if-not**) is a list of objects to be passed to *predicate* as *predicate*'s second argument, third argument, and so on. The reason for this is that *predicate* might be a function of many arguments; *extra-lists* lets you control what values are passed as additional arguments to *predicate*. However, the list returned by **subset** or **rem-if-not** is still a "subset" of those values that were passed as the *first* argument in the various calls to *predicate*.

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

subset-not and **rem-if** do the same thing, but they are used in different contexts. **rem-if** means "remove if this condition is true". **subset-not** refers to the function's action if *list* is considered to represent a mathematical set.

predicate should be a function of one argument. A new list is made by applying *predicate* to all the elements of *list* and removing the ones for which the predicate returns non-**nil**. The meaning of *extra-lists* is the same as for **subset** and **rem-if-not**.

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

subset-not and **rem-if** do the same thing, but they are used in different contexts. **rem-if** means "remove if this condition is true". **subset-not** refers to the function's action if *list* is considered to represent a mathematical set.

predicate should be a function of one argument. A new list is made by applying *predicate* to all the elements of *list* and removing the ones for which the predicate returns non-**nil**. The meaning of *extra-lists* is the same as for **subset** and **rem-if-not**.

del-if *predicate list* *Function*

del-if is just like **rem-if** except that it modifies *list* rather than creating a new list and it does not take an *extra-lists* **&rest** argument.

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

del-if-not is just like **rem-if-not** except that it modifies *list* rather than creating a new list and it does not take an *extra-lists* **&rest** argument.

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

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

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

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

4.7 Association Lists

assq *item alist* *Function*

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

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

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

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

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

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

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

assq could have been defined by:

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

assoc *item alist* *Function*

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

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

assoc could have been defined by:

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

ass *predicate item alist* *Function*

ass is the same as **assq** except that it takes an extra argument that should be a predicate of two arguments, which is used for the comparison instead of **eq**. **(ass 'eq a b)** is the same as **(assq a b)**. See the function **mem**, page 60. As with **mem**, you may use noncommutative predicates; the first argument to the predicate is *item* and the second is the key of the element of *alist*.

memass *predicate item alist* *Function*

memass searches *alist* just like **ass**, but returns the portion of the list beginning with the pair containing *item*, rather than the pair itself. **(car (memass *x y z*)) = (ass *x y z*)**. See the function **mem**, page 60. As with **mem**, you can use noncommutative predicates; the first argument to the predicate is *item* and the second is the key of the element of *alist*.

rassq *item alist* *Function*

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

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

rassoc *item alist* *Function*

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

rass *predicate item alist* *Function*

rass is to **rassq** as **ass** is to **assq**. That is, it takes a predicate to be used instead of **eq**. See the function **mem**, page 60. As with **mem**, you can use noncommutative predicates; the first argument to the predicate is *item* and the second is the cdr of the element of *alist*.

sassq *item alist function* *Function*

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

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

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

sassoc *item alist function* *Function*

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

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

pairlis *cars cdrs* *Function*

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

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

4.8 Property Lists

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

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

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

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

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

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

2. The system associates a property list with every symbol. See the section "The Property List of a Symbol", page 564. A symbol can be used where a property list is expected; the property-list primitives automatically find the symbol's property list and use it.

3. A property list can be a memory cell in the middle of some data structure, such as a list, an array, an instance, or a defstruct. An arbitrary memory cell of this kind is named by a locative. See the section "Locatives", page 83. Such locatives are typically created with the **locf** special form. See the macro **locf**, page 148.

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

Here is an example of the list of entries inside the property list of a symbol named **b1** that is being used by a program that deals with blocks:

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

There are three properties, and so the list has six elements. The first property's indicator is the symbol **color**, and its value is the symbol **blue**. We say that "the value of **b1**'s **color** property is **blue**", or, informally, that "**b1**'s **color** property is **blue**." The program is probably representing the information that the block represented by **b1** is painted blue. Similarly, it is probably representing in the rest of the property list that block **b1** is on top of block **b6**, and that **b1** is associated with blocks **b2**, **b3**, and **b4**.

get *plist indicator* *Function*

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

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

getl *plist indicator-list* *Function*

getl is like **get**, except that the second argument is a list of indicators. **getl** searches down *plist* for any of the indicators in *indicator-list* until it finds a property whose indicator is one of the elements of *indicator-list*. If *plist* is a symbol, the symbol's associated property list is used. **getl** returns the portion of the list inside *plist* beginning with the first such property that it found. So the **car** of the returned list is an indicator, and the **cadr** is the property value. If none of the indicators on *indicator-list* are on the property list, **getl** returns **nil**. For example, if the property list of **foo** were:

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

then:

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

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

putprop *plist x indicator* *Function*

This gives *plist* an *indicator*-property of *x*. After this is done, **(get** *plist indicator***)** returns *x*. If *plist* is a symbol, the symbol's associated property list is used. **putprop** returns its second argument.

Example:

```
(putprop 'Nixon 'not 'crook) => NOT
```

defprop *symbol x indicator* *Special Form*

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

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

is the same as:

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

remprop *plist indicator* *Function*

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

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

then:

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

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

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

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

There is a mixin flavor, called **si:property-list-mixin**, that provides messages that do things analogous to what the above functions do.

4.9 Hash Tables

A hash table is a Lisp object that works something like a property list. Each hash table has a set of *entries*, each of which associates a particular *key* with a particular *value*. The basic functions that deal with hash tables can create entries, delete entries, and find the value that is associated with a given key. Finding the value is very fast even if there are many entries, because hashing is used; this is an important advantage of hash tables over property lists. See the section "Hash Primitive", page 75.

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

Hash tables come in two kinds, the difference being whether the keys are compared using **eq** or using **equal**. The following discussion refers to the **eq** kind of hash table; the other kind is described in another section. See the section "Creating Hash Tables", page 71.

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

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

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

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

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

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

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

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

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

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

Hash tables are implemented as instances of flavors. The two flavors for the two kinds of hash tables are **si:eq-hash-table** and **si:equal-hash-table**. See the section "Hash Table Messages", page 72.

4.9.1 Creating Hash Tables

A new hash table using **eq** for comparisons of the key is created by making an instance of the **si:eq-hash-table** flavor. (See the function **make-instance**, page 430.) The function **make-hash-table** also creates an **eq** hash table, and takes the init options to pass on to **make-instance** as arguments.

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

si:eq-hash-table *Flavor*

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

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

:area Specifies the area in which the hash table should be created. This is just like the **:area** option to **make-array**. See the function **make-array**, page 241. The default is **working-storage-area**.

:growth-factor Specifies how much to increase the size of the hash table when it becomes full. If it is an integer, the hash table is increased by that number. If it is a floating-point number greater than one, the new size of the hash table is the old size multiplied by that number.

:rehash-before-cold

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

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

Normally a hash table is automatically rehashed "on demand" the first time it is used after the hashing has

become invalidated. This first **:get-hash** operation is therefore much slower than normal.

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

si:equal-hash-table *Flavor*

A table of this flavor uses the **equal** function for comparison of the hash keys. It accepts the following init option as well as those described for **eq** hash tables. See the flavor **si:eq-hash-table**, page 71.

:rehash-threshold

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

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

This creates a new hash table using the **eq** function for comparison of the keys. This function just calls **make-instance** using the **si:eq-hash-table** flavor, passing *options* to **make-instance** as init options. See the flavor **si:eq-hash-table**, page 71.

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

This creates a new hash table using the **equal** function for comparison of the keys. This function just calls **make-instance** using the **si:equal-hash-table** flavor, passing *options* to **make-instance** as init options. See the flavor **si:equal-hash-table**, page 72.

4.9.2 Hash Table Messages

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

:get-hash *key* *Message*

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

:put-hash *key value* *Message*

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

:rem-hash *key* *Message*

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

:swap-hash *key value* *Message*

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

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

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

:clear-hash *Message*

Remove all the entries from the hash table.

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

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

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

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

:size *Message*

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

:filled-elements *Message*

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

4.9.3 Hash Table Functions

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

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

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

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

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

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

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

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

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

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

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

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

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

4.9.4 Dumping Hash Tables to Files

Instances of hash tables can be dumped to files by using any of the dump functions. See the function **sys:dump-forms-to-file** in *Reference Guide to Streams, Files, and I/O*. The hash table flavors have the **:fasd-form** methods required to support dumping of their data to a fasd file.

4.9.5 Hash Tables and Loop Iteration

loop provides an iteration path for iterating over every entry in a hash table. See the section "**loop** Iteration Over Hash Tables or Heaps", page 224.

4.9.6 Hash Tables and the Garbage Collector

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

4.9.7 Hash Primitive

Hashing is a technique used in algorithms to provide fast retrieval of data in large tables. A function, known as a "hash function", is created, which takes an object that might be used as a key, and produces a number associated with that key. This number, or some function of it, can be used to specify where in a table to look for the datum associated with the key. It is always possible for two different objects to "hash to the same value"; that is, for the hash function to return the same number for two distinct objects. Good hash functions are designed to minimize this by evenly distributing their results over the range of possible numbers. However, hash table algorithms must still deal with this problem by providing a secondary search, sometimes known as a *rehash*. For more information, consult a textbook on computer algorithms.

si:equal-hash *object* *Function*

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

si:equal-hash uses **%pointer** to define the hash key for data types such as arrays, stack groups, or closures. This means that some of the hash keys in **equal** hash tables are based on a virtual memory address. Hash tables that are at all dependent on memory addresses are rehashed when the garbage collector flips.

si:equal-hash returns a second value (**t** or **nil**), if it has used **%pointer** to define the hash key. It returns **t** if the first value returned by **si:equal-hash** depends on the GC generation. That is, if **t** is returned, future calls to **si:equal-hash** (for the same object) might not return the same number if an intervening GC occurs.

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

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

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

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

sxhash *object* *Function*

sxhash computes a hash code of an object, 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 nonnegative 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, always changes the hash code.

sxhash is the same as **si:equal-hash**, except that **sxhash** returns 0 as the hash value for objects with data types like arrays, stack groups, or closures. As a result, hashing such structures could degenerate to the case of linear search.

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

Therefore, **si:equal-hash** and **sxhash** are useful for retrieving data when two keys that are not the same object but are **equal**, are considered the same. If you consider two such keys to be different, then you need "hashing on **eq**", where two different objects are always considered different. In some Lisp implementations, there is an easy way to create a hash function that hashes on **eq**, namely, by returning the virtual address of the storage associated with the object. But in other implementations, including Symbolics-Lisp, this does not work, because the address associated with an object can be changed by the relocating garbage collector. The hash tables discussed here deal with this problem by using the appropriate subprimitives so that they interface correctly with the garbage collector. Hash tables that hash on **eq** are also provided. See the flavor **si:eq-hash-table**, page 71.

4.10 Heaps

A heap is a data structure in which each item is ordered by some predicate (for example, less-than) on its associated key. You can add an item to the heap, delete an item from it, or look at the top item. The "top" operation is guaranteed to return the first (that is, smallest) item in the heap. Heaps are useful in maintaining priority queues.

make-heap &key (*size* **100.**) (*predicate* **#'<**) (*growth-factor* **1.5**) *Function*
interlocking

make-heap creates a new heap. **:predicate**, **:size**, and **:growth-factor** are passed as init options to **make-instance** when the heap is created.

make-heap takes the following keyword arguments:

:predicate An ordering predicate that is applied to each key. The default is **lessp**.

:size The default is 100.

:growth-factor A number or **nil**. If it is an integer, the heap is increased by that number. If it is a floating-point number greater than one, the new size of the heap is the old size multiplied by that number. If it is **nil**, the condition **si:heap-overflow** is signalled instead of growing the heap.

:interlocking

t This causes **make-heap** to create a kind of heap that can be interlocked for use by multiple processes, using **process-lock** to perform the interlocking.

:without-interrupts
This causes **make-heap** to create a kind of heap that can be interlocked for use by multiple processes, using **without-interrupts** to perform the interlocking.

4.10.1 Messages to Heaps

The following are defined messages to heaps:

:clear of **si:heap** *Method*
Clears the heap.

:delete-by-item *item* &optional *equal-predicate* of **si:heap** *Method*
Finds the first item that satisfies *equal-predicate*, and deletes it, returning the item and key if it was found, otherwise it signals **si:heap-item-not-found**. *equal-predicate* should be a function that takes two arguments. The first argument is the item from the heap and the second argument is *item*.

:delete-by-key *key* &optional *equal-predicate* of **si:heap** *Method*
Finds the first item whose key satisfies *equal-predicate* and deletes it, returning the item and key if it was found; otherwise it signals **si:heap-item-not-found**. *equal-predicate* should be a function that takes two arguments. The first argument is the key from the heap and the second argument is *key*.

:describe *stream* of **si:heap** *Method*
Describes the heap.

:empty-p of **si:heap** *Method*
Returns **t** if the heap is empty, otherwise returns **nil**.

:find-by-item *item* &optional *equal-predicate* of **si:heap** *Method*
Finds the first item that satisfies *equal-predicate* and returns the item and key if it was found; otherwise it signals **si:heap-item-not-found**. *equal-predicate* should be a function that takes two arguments. The first argument is the item from the heap and the second argument is *item*.

:find-by-key *key* &optional *equal-predicate* of **si:heap** *Method*
Finds the first item whose key satisfies *equal-predicate* and returns the item and key if it was found; otherwise it signals **si:heap-item-not-found**. *equal-predicate* should be a function that takes two arguments. The first argument is the key from the heap and the second argument is *key*.

:insert *item key* of **si:heap** *Method*

Inserts *item* into the heap based on its key *key*, and returns *item* and *key*.

:remove of **si:heap** *Method*

Removes the top item from the heap and returns it and its key as values. The third value is **nil** if the heap was empty; otherwise it is **t**.

:top of **si:heap** *Method*

Returns the value and key of the top item on the heap. The third value is **nil** if the heap was empty; otherwise it is **t**.

4.10.2 Heaps and Loop Iteration

loop provides an iteration path for iterating over every entry in a heap. See the section "**loop** Iteration Over Hash Tables or Heaps", page 224.

4.11 Sorting

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

After sorting, the argument (either list or array) has been rearranged internally to be completely ordered. In the case of an array argument, this is accomplished by permuting the elements of the array, while in the list case, the list is reordered by **rplacd**s in the same manner as **nreverse**. Thus, if the argument should not be clobbered, you must sort a copy of the argument, obtainable by **fillarray** or **copylist**, as appropriate. Furthermore, **sort** of a list is like **delq** in that it should not be used for effect; the result is conceptually the same as the argument but in fact is a different Lisp object.

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

The sorting package is smart about compact lists; it sorts compact sublists as if they were arrays. See the section "Cdr-coding", page 56. An explanation of compact lists is in that section.

sort *table predicate* *Function*

The first argument to **sort** is an array or a list. The second is a predicate, which must be applicable to all the objects in the array or list. The predicate

should take two arguments, and return non-**nil** if and only if the first argument is strictly less than the second (in some appropriate sense). The predicate should return **nil** if its arguments are equal. For example, to sort in the opposite direction from <, use >, not ≥. This is because the quicksort algorithm used to sort arrays and cdr-coded lists becomes very much slower when the predicate returns non-**nil** for equal elements while sorting many of them.

The **sort** function proceeds to sort the contents of the array or list under the ordering imposed by the predicate, and returns the array or list modified into sorted order. Note that since sorting requires many comparisons, and thus many calls to the predicate, sorting is 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 can change the order of the conses of the list (using **rplacd**), and so it cannot be used merely for side effect; only the *returned value* of **sort** is the sorted list. This changes the original list; if you need both the original list and the sorted list, you must copy the original and sort the copy. See the function **copylist**, page 50.

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

If the argument to **sort** is an array with a fill pointer, note that, like most functions, **sort** considers the active length of the array to be the length, and so only the active part of the array is sorted. See the function **array-active-length**, page 247.

sortcar *x predicate* *Function*

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

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

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

stable-sort *x predicate* *Function*

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

stable-sortcar *x predicate* *Function*

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

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

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

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

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

5. Locatives

5.1 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 never need them.

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

There are a set of functions that create locatives to cells; the functions are documented with the kind of object to which they create a pointer. See the function **ap-1**, page 244. See the function **ap-leader**, page 245. See the function **car-location**, page 45. See the function **value-cell-location**, page 563. The macro **locf** can be used to convert a form that accesses a cell to one that creates a locative pointer to that cell.

For example:

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

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

5.2 Cdr-coding and Locatives

Either of the functions **car** and **cdr** can 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)
```

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

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

For example:

```
(rplaca (value-cell-location x) y)
```

is the same as:

```
(set x y)
```

The following example takes advantage of **locf** *symbol* to cons up a list in forward order without special-case code. The first time through the loop, the **(setf (cdr location) ...)** is equivalent to **(setq result ...)**; on later times through the loop the **setf** tacks an additional cons onto the end of the list.

```
(defun simplified-version-of-mapcar (fcn list)
  (do ((list list (cdr list))
       (result nil)
       (location (locf result)))
      ((null list) result)
    (let ((new-cons (ncons (funcall fcn (car list)))))
      ;; tack it onto the end
      (setf (cdr location) new-cons)
      ;; get a pointer to the new tail
      (setq location (locf (cdr new-cons))))))
```

5.3 Functions That Operate on Locatives

location-contents *locative* *Function*

Returns the contents of the cell at which *locative* points. For example:

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

is the same as:

```
(symeval x)
```

To store objects into the cell at which a locative points, you should use **(setf (location-contents x) y)** as shown in the following example:

```
(setf (location-contents (value-cell-location x)) y)
```

This is the same as:

```
(set x y)
```

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

location-makunbound is a version of **makunbound** that can be used on any cell in the Symbolics Lisp Machine. It takes a locative pointer to designate the cell rather than a symbol. (**makunbound** is restricted to use with symbols.)

location-makunbound takes a symbol as an optional second argument: *variable-name* of the location that is being made unbound. It uses *variable-name* to label the null pointer it stores so that the Debugger knows the name of the unbound location if it is referenced. This is particularly appropriate when the location being made unbound is really a variable value cell of one sort or another, for example, closure or instance.

location-boundp *location* *Function*

location-boundp is a version of **boundp** that can be used on any cell in the Symbolics Lisp Machine. It takes a locative pointer to designate the cell rather than a symbol. The following two calls are equivalent:

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

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

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

PART III.

Numbers

6. Introduction to Numbers

Symbolics-Lisp includes several types of numbers, with different characteristics. Most numeric functions accept any type of numbers as arguments and do the right thing. That is to say, they are *generic*. Maclisp contains both generic numeric functions (like **plus**) and specific numeric functions (like +), which only operate on a certain type, and are much more efficient. In Symbolics-Lisp, this distinction does not exist; both function names exist for compatibility but they are identical. The microprogrammed structure of the machine makes it possible to have only the generic functions without loss of efficiency.

Symbolics-Lisp Number Types

The types of numbers in Symbolics-Lisp are:

fixnum	Fixnums are 32-bit 2's complement binary integers. These are the "preferred, most efficient" type of number.
bignum	Bignums are arbitrary-precision binary integers.
ratio	A ratio is a pair of integers, representing the numerator and denominator of the number.
single-float	Single-precision floating-point numbers have a precision of 24 bits, or about 7 decimal digits. Their range is from 1.1754944e-38 to 3.4028235e38.
double-float	Double-precision floating-point numbers have a precision of 53 bits, or about 16 decimal digits. Their range is from 2.2250738585072014d-308 to 1.7976931348623157d308.
complex number	A complex number is a pair of noncomplex numbers, representing the real and imaginary parts of the number.

Generally, Lisp objects have a unique identity; each exists, independent of any other, and you can use the **eq** predicate to determine whether two references are to the same object or not. Numbers are the exception to this rule; they do not work this way.

The following function can return either **t** or **nil**.

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

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

Fixnums and single-floats 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.

6.1 Coercion Rules for Numbers

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 single-float or a double-float, the result is a single-float or a double-float (respectively). When a single-float meets a double-float, the result is a double-float. When a single-precision floating-point number meets a double-precision floating-point number, the result is a double-float.

Thus, if the constants in a numerical algorithm are written as single-floats (assuming this provides adequate precision), and if the input is a single-float, the computation is done in single-float mode and the result is a single-float. If the input is a double-float the computations are done in double precision and the result is a double-float, although the constants still have only single precision. For most algorithms, it is desirable to have two separate sets of constants to maintain accuracy for double precision and speed for single precision.

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

6.2 Numbers in the Compiler

Unlike Maclisp, Symbolics-Lisp does not have number declarations in the compiler. Note that because fixnums and single-floats require no associated storage they are as efficient as declared numbers in Maclisp.

6.3 Printed Representation of Numbers

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 single-precision floating-point number. If a number has an

exponent separated by "d", it is a double-precision floating-point number. Fixnums and bignums have similar printed representations since there is no numerical value that has a choice of whether to be a fixnum or a bignum; an integer is a bignum if and only if its magnitude is too big for a fixnum. See the section "What the Reader Recognizes", page 20.

7. Types of Numbers

7.1 Integers

Symbolics-Lisp has two primitive types of integers, fixnums and bignums. The distinction between fixnums and bignums is largely transparent to the user. You simply compute with integers, and the system represents some as fixnums and the rest (less efficiently) as bignums. 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 that only work on fixnums; this fact is noted in their documentation. Also when using **eq** on numbers you should be aware of the fixnum/bignum distinction.

Integer 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 yields 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. See the section "Numeric Type Conversions", page 107.

7.2 Rational Numbers

Rational numbers include both ratios and integers. Ratios are represented in terms of an integer numerator and denominator. The ratio is always "in lowest terms", meaning that the denominator is as small as possible. If the denominator is 1, the rational number is represented as an integer. The denominator is always positive; the sign of the number is carried by the numerator. See the section "Numeric Type Conversions", page 107.

7.3 Floating-point Numbers

IEEE Floating-point Representation

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

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

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

See the IEEE standard: Microprocessor Standards Committee, IEEE Computer Society, "A Proposed Standard for Binary Floating-Point Arithmetic: Draft 8.0 of IEEE Task P754," *Computer*, March 1981, pp. 51-62. See the section "Numeric Type Conversions", page 107.

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, unless the computation is inside the body of a **without-floating-underflow-traps**.

without-floating-underflow-traps *body...* *Special Form*

Inhibits trapping of floating-point exponent underflow traps within the body of the form. The result of a computation which would otherwise underflow is a denormalized number or zero, whichever is closest to the mathematical result.

Example:

```
(describe (without-floating-underflow-traps (expt .1 40))) =>
1.0e-40 is a single-precision floating-point number.
  Sign 0, exponent 0, 23-bit fraction 213302 (denormalized)
1.0e-40
```

7.4 Complex Numbers

A complex number is a pair of noncomplex numbers, representing the real and imaginary parts of the number. The types of the real and imaginary parts are always the same. No Symbolics-Lisp complex number has a rational real part and an imaginary part of integer zero. Such a number is always represented simply by the rational real part. See the section "Numeric Type Conversions", page 107.

8. Numeric Functions

8.1 Numeric Predicates

zerop *x* *Function*

Returns **t** if *x* is zero. Otherwise it returns **nil**. If *x* is not a number, **zerop** causes an error. For floating-point numbers, this only returns **t** for exactly **0.0**, **-0.0**, **0.0d0** or **-0.0d0**; there is no "fuzz".

plusp *x* *Function*

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

minusp *x* *Function*

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

oddp *number* *Function*

Returns **t** if *number* is odd, otherwise **nil**. If *number* is not an integer, **oddp** causes an error.

evenp *number* *Function*

Returns **t** if *number* is even, otherwise **nil**. If *number* is not an integer, **evenp** causes an error.

signp *test x* *Special Form*

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

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

Examples:

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

See the section "Predicates", page 7.

8.2 Numeric Comparisons

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

= *x y* *Function*

Returns **t** if *x* and *y* are numerically equal. An integer can be = to a floating-point number.

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

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

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

The following function is a synonym of **greaterp**:

>

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

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

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

The following function is a synonym of **>** :

greaterp

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

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

The following function is a synonym of >= :

≥

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

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

The following function is a synonym of ≥ :

>=

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

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

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

The following function is a synonym of **lessp**:

<

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

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

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

The following function is a synonym of < :

lessp

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

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

The following function is a synonym of <= :

≤

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

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

The following function is a synonym of ≤ :

<=

≠ *x y* *Function*

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

max &rest *args* *Function*

max returns the largest of its arguments. **max** requires at least one argument. Example:

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

min &rest *args* *Function*

min returns the smallest of its arguments. **min** requires at least one argument. Example:

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

8.3 Arithmetic

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

plus &rest *args* *Function*

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

The following functions are synonyms of **plus**:

+
+$

+ &rest *args* *Function*

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

The following functions are synonyms of + :

plus
+$

+$ &rest *args* *Function*

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

The following functions are synonyms of +$:

plus
+

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

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

minus *x* *Function*

Returns the negative of *x*. Examples:

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

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

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

The following function is a synonym of - :

-$

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

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

The following function is a synonym of -$:

abs *x* *Function*

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

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

For complex numbers, **abs** could have been defined by:

```
(defun abs (x)
  (sqrt (+ (^ (realpart x) 2) (^ (imagpart x) 2))))
```

conjugate *x* *Function*

Returns the complex conjugate of *x*. The conjugate of a noncomplex number is itself. **conjugate** could have been defined by:

```
(defun conjugate (x)
  (complex (realpart x) (- (imagpart x))))
```

times &rest *args* *Function*

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

The following functions are synonyms of **times**:

*****
***$**

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

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

The following functions are synonyms of ***** :

times
***$**

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

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

The following functions are synonyms of ***$** :

times
*****

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

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

// arg &rest *args* *Function*

The name of this function is written // rather than / because / is the quoting character in Lisp syntax and must be doubled. With more than one argument, // is the same as **quotient**; it returns the first argument divided by all of the rest of its arguments. With only one argument, (// *x*) is the same as (// 1 *x*). The exact rules for the meaning of the quotient and remainder of two integers are in another section. See the section "Integers", page 93. That section explains why the rules used for integer division are not correct for all applications. Examples:

```
(// 3 2) => 1        ;Integer division truncates.
(// 3 -2) => -1
(// -3 2) => -1
(// -3 -2) => 1
(// 3 2.0) => 1.5
(// 3 2.0d0) => 1.5d0
(// 4 2) => 2
(// 12. 2. 3.) => 2
(// 4.0) => .25
```

The following function is a synonym of // :

//$

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

The name of this function is written //$ rather than / because / is the quoting character in Lisp syntax and must be doubled. With more than one argument, //$ is the same as **quotient**; it returns the first argument divided by all of the rest of its arguments. With only one argument, (//$ *x*) is the same as (//$ **1** *x*). The exact rules for the meaning of the quotient and remainder of two integers are in another section. See the section "Integers", page 93. That section explains why the rules used for integer division are not correct for all applications. Examples:

```
(//$ 3 2) => 1          ;Integer division truncates.
(//$ 3 -2) => -1
(//$ -3 2) => -1
(//$ -3 -2) => 1
(//$ 3 2.0) => 1.5
(//$ 3 2.0d0) => 1.5d0
(//$ 4 2) => 2
(//$ 12. 2. 3.) => 2
(//$ 4.0) => .25
```

The following function is a synonym of //$:

//

remainder *x y* *Function*

Returns the remainder of *x* divided by *y*. *x* and *y* must be integers (fixnums or bignums). The exact rules for the meaning of the quotient and remainder of two integers are given in another section. See the section "Integers", page 93.

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

The following function is a synonym for **remainder**:

\

\ x y *Function*

Returns the remainder of *x* divided by *y*. *x* and *y* must be integers (fixnums or bignums). The exact rules for the meaning of the quotient and remainder of two integers are given in another section. See the section "Integers", page 93.

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

The following function is a synonym for \ :

remainder

mod *x y* *Function*

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

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

add1 *x* *Function*

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

The following functions are synonyms of **add1**:

1+
1+$

1+ *x* *Function*

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

The following functions are synonyms of **1+** :

add1
1+$

1+$ *x* *Function*

(1+$ *x***)** is the same as **(plus** *x* **1)**.

The following functions are synonyms of **1+$** :

add1
1+

sub1 *x* *Function*

(sub1 *x***)** is the same as **(difference** *x* **1)**.

The following functions are synonyms of **sub1**:

1-
1-$

1- *x* *Function*

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

The following functions are synonyms of **1-** :

sub1
1-$

1-$ *x* *Function*

(1-$ *x***)** is the same as **(difference** *x* **1)**.
The following functions are synonyms of **1-$** :

sub1
1-

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

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

The following function is a synonym of **gcd**:

\\

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

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

The following function is a synonym of \\ :

gcd

expt *x y* *Function*

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

The following functions are synonyms of **expt**:

^

^$

^ *x y* *Function*

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

The following functions are synonyms of ^ :

expt
^$

^$ *x y* *Function*

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

algorithm is used, while if the exponent is floating the result is **(exp (* *y* (log *x*)))**.

The following functions are synonyms of **^$** :

expt

sqrt *x* *Function*

Returns the square root of *x*.

isqrt *x* *Function*

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

signum *value* *Function*

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

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

The definition is compatible with the current Common Lisp design.

8.4 Transcendental Functions

These functions are only for floating-point arguments; if given an integer they convert it to a single-float. If given a double-float, they return a double-float.

exp *x* *Function*

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

log *x* *Function*

Returns the natural logarithm of *x*.

sin *x* *Function*

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

cos *x* *Function*

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

tan *x* *Function*

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

sind *x* *Function*

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

cosd *x* *Function*

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

tand *x* *Function*

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

cis *x* *Function*

x must be a noncomplex number. **cis** could have been defined by:

```
(defun cis (x)
  (complex (cos x) (sin x)))
```

Mathematically, this is equivalent to e^{ix}.

atan *y x* *Function*

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

atan2 *y x* *Function*

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

phase *x* *Function*

The phase of a number is the angle part of its polar representation as a complex number. The phase of zero is arbitrarily defined to be zero. **phase** could have been defined as:

```
(defun phase (x)
  (atan2 (imagpart x) (realpart x)))
```

sinh *x* *Function*

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

cosh *x* *Function*

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

tanh *x* *Function*

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

8.5 Numeric Type Conversions

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

fix *x* *Function*

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

fixr *x* *Function*

Converts *x* from a floating-point number 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))))
```

rational *x* *Function*

Converts any noncomplex number to an equivalent rational number. If *x* is a floating-point number, **rational** returns the rational number of least denominator, which when converted back to the same floating-point precision, is equal to *x*.

numerator *x* *Function*

If *x* is a ratio, **numerator** returns the numerator of *x*. If *x* is an integer, **numerator** returns *x*.

denominator *x* *Function*

If *x* is a ratio, **denominator** returns the denominator of *x*. If *x* is an integer, **denominator** returns 1.

float *x* *Function*

Converts any kind of number to a single-precision floating-point number. Note that **float** reduces a double-precision argument to single precision.

dfloat *x* *Function*

Converts any kind of number to a double-precision floating-point number.

complex *real* &optional *imag* *Function*

Constructs a complex number from real and imaginary noncomplex parts. If the types of the real and imaginary parts are different, the coercion rules are applied to make them the same. If *imag* is not specified, a zero of the same type as *real* is used. If *real* is an integer or a ratio, and *imag* is 0, the result is *real*.

realpart *x* *Function*

If *x* is a complex number, **realpart** returns the real part of *x*. If *x* is a noncomplex number, **realpart** returns *x*.

imagpart *x* *Function*

If *x* is a complex number, **imagpart** returns the imaginary part of *x*. If *x* is a noncomplex number, **imagpart** returns a zero of the same type as *x*.

floor *number* &optional (*divisor* **1**) *Function*

Divides *number* by *divisor*, and truncates the result toward negative infinity. The truncated result and the remainder are the returned values.

number and *divisor* must each be a noncomplex number. Not specifying a divisor is exactly the same as specifying a divisor of 1.

If the two returned values are Q and R, then **(+ (* Q** *divisor***) R)** equals *number*. If *divisor* is 1, then Q and R add up to *number*. If *divisor* is 1 and *number* is an integer, then the returned values are *number* and 0.

The first returned value is always an integer. The second returned value is integral if both arguments are integers, is rational if both arguments are rational, and is floating-point if either argument is floating-point. If only one argument is specified, then the second returned value is always a number of the same type as the argument.

Examples:

```
(floor 5)      --> 5 0        (floor -5)      --> -5 0
(floor 5.2)    --> 5 0.2      (floor -5.2)    --> -6 0.8
(floor 5.8)    --> 5 0.8      (floor -5.8)    --> -6 0.2
(floor 5 3)    --> 1 2        (floor -5 3)    --> -2 1
(floor 5 4)    --> 1 1        (floor -5 4)    --> -2 3
(floor 5.2 3)  --> 1 2.2      (floor -5.2 3)  --> -2 0.8
(floor 5.2 4)  --> 1 1.2      (floor -5.2 4)  --> -2 2.8
(floor 5.8 3)  --> 1 2.8      (floor -5.8 3)  --> -2 0.2
(floor 5.8 4)  --> 1 1.8      (floor -5.8 4)  --> -2 2.2
```

Using **floor** with one argument is the same as the **fix** function. Use **floor** instead.

Related Topics:

ceiling
truncate
round

ceiling *number* &optional (*divisor* **1**) *Function*

Divides *number* by *divisor*, and truncates the result toward positive infinity. The truncated result and the remainder are the returned values.

number and *divisor* must each be a noncomplex number. Not specifying a divisor is exactly the same as specifying a divisor of 1.

If the two returned values are Q and R, then **(+ (* Q** *divisor***) R)** equals *number*. If *divisor* is 1, then Q and R add up to *number*. If *divisor* is 1 and *number* is an integer, then the returned values are *number* and 0.

The first returned value is always an integer. The second returned value is integral if both arguments are integers, is rational if both arguments are rational, and is floating-point if either argument is floating-point. If only one argument is specified, then the second returned value is always a number of the same type as the argument.

Examples:

```
(ceiling 5)      --> 5 0        (ceiling -5)      --> -5 0
(ceiling 5.2)    --> 6 -0.8     (ceiling -5.2)    --> -5 -0.2
(ceiling 5.8)    --> 6 -0.2     (ceiling -5.8)    --> -5 -0.8
(ceiling 5 3)    --> 2 -1       (ceiling -5 3)    --> -1 -2
(ceiling 5 4)    --> 2 -3       (ceiling -5 4)    --> -1 -1
(ceiling 5.2 3)  --> 2 -0.8     (ceiling -5.2 3)  --> -1 -2.2
(ceiling 5.2 4)  --> 2 -2.8     (ceiling -5.2 4)  --> -1 -1.2
(ceiling 5.8 3)  --> 2 -0.2     (ceiling -5.8 3)  --> -1 -2.8
(ceiling 5.8 4)  --> 2 -2.2     (ceiling -5.8 4)  --> -1 -1.8
```

Related Topics:

floor
truncate
round

truncate *number* &optional (*divisor* **1**) *Function*

Divides *number* by *divisor*, and truncates the result toward zero. The truncated result and the remainder are the returned values.

number and *divisor* must each be a noncomplex number. Not specifying a divisor is exactly the same as specifying a divisor of 1.

If the two returned values are Q and R, then **(+ (* Q** *divisor***) R)** equals *number*. If *divisor* is 1, then Q and R add up to *number*. If *divisor* is 1 and *number* is an integer, then the returned values are *number* and 0.

The first returned value is always an integer. The second returned value is integral if both arguments are integers, is rational if both arguments are rational, and is floating-point if either argument is floating-point. If only one argument is specified, then the second returned value is always a number of the same type as the argument.

Examples:

```
(truncate 5)     --> 5 0        (truncate -5)     --> -5 0
(truncate 5.2)   --> 5 0.2      (truncate -5.2)   --> -5 -0.2
(truncate 5.8)   --> 5 0.8      (truncate -5.8)   --> -5 -0.8
(truncate 5 3)   --> 1 2        (truncate -5 3)   --> -1 -2
(truncate 5 4)   --> 1 1        (truncate -5 4)   --> -1 -1
(truncate 5.2 3) --> 1 2.2      (truncate -5.2 3) --> -1 -2.2
(truncate 5.2 4) --> 1 1.2      (truncate -5.2 4) --> -1 -1.2
(truncate 5.8 3) --> 1 2.8      (truncate -5.8 3) --> -1 -2.8
(truncate 5.8 4) --> 1 1.8      (truncate -5.8 4) --> -1 -1.8
```

Related Topics:

floor
ceiling
round

round *number* &optional (*divisor* **1**) *Function*

Divides *number* by *divisor*, and rounds the result toward the nearest integer. If the result is exactly halfway between two integers, then it is rounded to the one that is even. The rounded result and the remainder are the returned values.

number and *divisor* must each be a noncomplex number. Not specifying a divisor is exactly the same as specifying a divisor of 1.

If the two returned values are Q and R, then **(+ (* Q** *divisor***) R)** equals *number*. If *divisor* is 1, then Q and R add up to *number*. If *divisor* is 1 and *number* is an integer, then the returned values are *number* and 0.

The first returned value is always an integer. The second returned value is integral if both arguments are integers, is rational if both arguments are rational, and is floating-point if either argument is floating-point. If only one argument is specified, then the second returned value is always a number of the same type as the argument.

Examples:

```
(round 5)     --> 5 0        (round -5)     --> -5 0
(round 5.2)   --> 5 0.2      (round -5.2)   --> -5 -0.2
(round 5.8)   --> 6 -0.2     (round -5.8)   --> -6 0.2
(round 5 3)   --> 2 -1       (round -5 3)   --> -2 1
(round 5 4)   --> 1 1        (round -5 4)   --> -1 -1
(round 5.2 3) --> 2 -0.8     (round -5.2 3) --> -2 0.8
(round 5.2 4) --> 1 1.2      (round -5.2 4) --> -1 -1.2
(round 5.8 3) --> 2 -0.2     (round -5.8 3) --> -2 0.2
(round 5.8 4) --> 1 1.8      (round -5.8 4) --> -1 -1.8
```

Using **round** with one argument is the same as the **fixr** function. Use **round** instead.

Related Topics:

floor
ceiling
truncate

sys:ffloor *number* &optional (*divisor* **1**) *Function*

This is just like **floor**, except that the first returned value is always a floating-point number instead of an integer. The second returned value is the remainder. If *number* is a floating-point number and *divisor* is not a floating-point number of longer format, then the first returned value is a floating-point number of the same type as *number*.

Examples:

```
(sys:ffloor 5)      --> 5.0 0        (sys:ffloor -5)      --> -5.0 0
(sys:ffloor 5.2)    --> 5.0 0.2      (sys:ffloor -5.2)    --> -6.0 0.8
(sys:ffloor 5 3)    --> 1.0 2        (sys:ffloor -5 3)    --> -2.0 1
(sys:ffloor 5.2 4)  --> 1.0 1.2      (sys:ffloor -5.2 4)  --> -2.0 2.8
(sys:ffloor 4.2d0)  --> 4.0d0 0.2    (sys:ffloor -4.2d0)  --> -5.0d0 0.8
```

Related Topics:

sys:fceiling
sys:ftruncate
sys:fround

sys:fceiling *number* &optional (*divisor* **1**) *Function*

This is just like **ceiling**, except that the first returned value is always a floating-point number instead of an integer. The second returned value is the remainder. If *number* is a floating-point number and *divisor* is not a floating-point number of longer format, then the first returned value is a floating-point number of the same type as *number*.

Examples:

```
(sys:fceiling 5)      --> 5.0 0        (sys:fceiling -5)      --> -5.0 0
(sys:fceiling 5.2)    --> 6.0 -0.8     (sys:fceiling -5.2)    --> -5.0 -0.2
(sys:fceiling 5 3)    --> 2.0 -1       (sys:fceiling -5 3)    --> -1.0 -2
(sys:fceiling 5.2 4)  --> 2.0 -2.8     (sys:fceiling -5.2 4)  --> -1.0 -1.2
(sys:fceiling 4.2d0)  --> 5.0d0 -0.8   (sys:fceiling -4.2d0)  --> -4.0d0 -0.2
```

Related Topics:

sys:ffloor
sys:ftruncate
sys:fround

sys:ftruncate *number* &optional (*divisor* **1**) *Function*

This is just like **truncate**, except that the first returned value is always a floating-point number instead of an integer. The second returned value is the remainder. If *number* is a floating-point number and *divisor* is not a floating-point number of longer format, then the first returned value is a floating-point number of the same type as *number*.

Examples:

```
(sys:ftruncate 5)      --> 5.0 0        (sys:ftruncate -5)      --> -5.0 0
(sys:ftruncate 5.2)    --> 5.0 0.2      (sys:ftruncate -5.2)    --> -5.0 -0.2
(sys:ftruncate 5 3)    --> 1.0 2        (sys:ftruncate -5 3)    --> -1.0 -2
(sys:ftruncate 5.2 4)  --> 1.0 1.2      (sys:ftruncate -5.2 4)  --> -1.0 -1.2
(sys:ftruncate 4.2d0)  --> 4.0d0 0.2    (sys:ftruncate -4.2d0)  --> -4.0d0 -0.2
```

Related Topics:

sys:ffloor
sys:fceiling
sys:fround

sys:fround *number* &optional (*divisor* **1**) *Function*

This is just like **round**, except that the first returned value is always a floating-point number instead of an integer. The second returned value is the remainder. If *number* is a floating-point number and *divisor* is not a floating-point number of longer format, then the first returned value is a floating-point number of the same type as *number*.

Examples:

```
(sys:fround 5)      --> 5.0 0          (sys:fround -5)      --> -5.0 0
(sys:fround 5.2)    --> 5.0 0.2        (sys:fround -5.2)    --> -5.0 -0.2
(sys:fround 5 3)    --> 2.0 -1         (sys:fround -5 3)    --> -2.0 1
(sys:fround 5.2 4)  --> 1.0 1.2        (sys:fround -5.2 4)  --> -1.0 -1.2
(sys:fround 4.2d0)  --> 4.0d0 0.2      (sys:fround -4.2d0)  --> -4.0d0 -0.2
```

Related Topics:

sys:ffloor
sys:fceiling
sys:ftruncate

8.6 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 32-bit fixnums. Negative numbers are operated on in their 2's-complement representation.

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

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

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

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

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

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

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

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

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

lognot *number* *Function*

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

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

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

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

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

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

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

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

Examples:

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

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

bit-test *x y* *Function*

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

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

lsh *x y* *Function*

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

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

ash *x y* *Function*

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

rot *x y* *Function*

Returns *x* rotated left *y* bits if *y* is positive or zero, or *x* rotated right |*y*| bits if *y* is negative. The rotation considers *x* as a 32-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) => 10000000000
(rot -1 7) => -1
(rot 15 32.) => 15
```

haulong *x* *Function*

This returns the number of significant bits in |*x*|. *x* must be an integer. Its sign is ignored. The result is the least integer strictly greater than the base-2 logarithm of |*x*|. Examples:

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

haipart *x n* *Function*

Returns the high *n* bits of the binary representation of |*x*|, or the low -*n* bits if *n* is negative. *x* must be an integer; 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))))
```

8.7 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 *byte*

specifiers to designate a specific byte position within any word. A byte specifier consists of the *size* (in bits) and *position* of the byte within the number, counting from the right in bits. A position of zero means that the byte is at the right end of the number. Byte specifiers are built using the **byte** function.

For example, the byte specifier **(byte 8 0)** refers to the lowest eight bits of a word, and the byte specifier **(byte 8 8)** refers to the next eight bits.

Bytes are extracted from and deposited into 2's complement signed integers. Treating the integers as signed means that negative numbers have an arbitrary number of "1" bits on the left. Bytes, being a specified number of bits, are never negative.

The actual format of byte specifiers is a fixnum where the low 6 bits specify the size and the rest of the bits specify the position. These byte specifiers are stylized as *ppss*. Since byte specifiers only have 6 bits to store the field width, there is a maximum byte field of 63 bits.

byte *size position* *Function*

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

Example:

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

byte-size *byte-specifier* *Function*

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

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

byte-position *byte-specifier* *Function*

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

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

ldb *ppss num* *Function*

ppss specifies a byte of *num* to be extracted. The *ss* bits of the byte starting at bit *pp* are the lowest *ss* bits in the returned value, and the rest of the bits in the returned value are zero. The name of the function, **ldb**, means "load byte". *num* must be an integer. **ldb** always returns a nonnegative number. Example:

```
(ldb (byte 1 2) 5) => 1
(ldb (byte 32. 0) -1) => (1- 1_32.)  ;;a positive bignum
(ldb (byte 16. 24.) -1_31.) => #o177600
```

```
(ldb #o0306 #o4567) => #o56
```

load-byte *num position size* *Function*

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

ldb-test *ppss y* *Function*

ldb-test is a predicate that returns **t** if any of the bits designated by the byte specifier *ppss* are 1's in *y*. That is, it returns **t** if the designated field is nonzero. **ldb-test** is implemented as a macro which expands as follows:

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

mask-field *ppss num* *Function*

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

```
(mask-field (byte 6 3) #o4567) => #o560
```

dpb *byte ppss num* *Function*

Returns a number that is the same as *num* except in the bits specified by *ppss*. The low *ss* bits of *byte* are placed in those bits. *byte* is interpreted as being right-justified, as if it were the result of **ldb**. *num* must be an integer. The name means "deposit byte". Example:

```
(dpb 1 (byte 1 2) 1) => 5
(dpb 0 (byte 1 31.) -1_31.) => -1_32.  ;;a bignum
(dpb -1 (byte 40. 0) -1_32.) => -1

(dpb #o23 #o0306 #o4567) => #o4237
```

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

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

deposit-field *byte ppss num* *Function*

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

```
(deposit-field #o230 (byte 6 3) #o4567) => #o4237
```

%logldb *ppss fixnum* *Function*

%logldb is like **ldb** except that it only loads out of fixnums and allows a byte size of 32. bits of the fixnum including the sign bit.

The behavior of**%logldb** depends on the size of fixnums, so functions using it might not work the same way on future implementations of Symbolics-Lisp. Its name starts with "%" because it is more like machine-level subprimitives than other byte manipulation functions.

%logdpb *byte ppss fixnum* *Function*

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

The behavior of**%logdpb** depends on the size of fixnums, so functions using it might not work the same way on future implementations of Symbolics-Lisp. Its name starts with "%" because it is more like machine-level subprimitives than other byte manipulation functions.

8.8 Random Numbers

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

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

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

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

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

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

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

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

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

array must be a random-array, such as is created by **si:random-create-array**. If *new-seed* is provided, it should be an integer, 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).

8.9 32-bit Numbers

Sometimes it is desirable to have a form of arithmetic that has no overflow checking (which would produce bignums), and truncates results to the word size of the machine. In Symbolics-Lisp, this is provided by the following set of functions. Their answers are only correct modulo 2^32.

These functions should *not* be used for "efficiency"; they are probably less efficient than the functions which *do* check for overflow. They are intended for algorithms which require this sort of arithmetic, such as hash functions and pseudorandom number generation.

%32-bit-plus *x y* *Function*

Returns the sum of *x* and *y* in 32-bit wraparound arithmetic. Both arguments must be fixnums. The result is a fixnum.

%32-bit-difference *x y* *Function*

Returns the difference of *x* and *y* in 32-bit wraparound arithmetic. Both arguments must be fixnums. The result is a fixnum.

PART IV.

Evaluation

9. Introduction to Evaluation

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

form	*Result*
A symbol	The binding of *form*. If *form* is unbound, an error is signalled. See the section "Variables", page 125. Some symbols can also be constants, for example: **t**, **nil**, keywords, and **defconstant**.
Not a symbol or list	*form*
A list	The evaluator examines the car of the list to figure out what to do next. There are three possibilities: the form can be a *special form*, a *macro form*, or a *function form*. Conceptually, the evaluator knows specially about all the symbols whose appearance in the car of a form make that form a special form, but the way the evaluator actually works is as follows. If the car of the form is a symbol, the evaluator finds the function definition of the symbol in the local lexical environment. If no definition exists there, the evaluator finds it in the global environment, which is in the function cell of the symbol. In either case, the evaluator starts all over as if that object had been the car of the list. (See the section "Symbols", page 561.) If the car is not a symbol, but a list whose car is the symbol **special**, this is a macro form or a special form. If it is a "special function", this is a special form. See the section "Kinds of Functions", page 303. Otherwise, it should be a regular function, and this is a function form.
A special form	It is handled accordingly; each special form works differently. See the section "Kinds of Functions", page 303. The internal workings of special forms are explained in more detail in that section, but this hardly ever affects you.
A macro form	The macro is expanded and the result is evaluated in place of *form*. See the section "Macros", page 337.
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*.

See the section "Variables", page 125. The way variables work and the ways in which they are manipulated, including the binding of arguments, is explained in that section. See the section "Evaluating a Function Form", page 151. That section contains a basic explanation of functions. See the section "Multiple Values", page 167. The way functions can return more than one value is explained there. See the section "Functions", page 297. The description of all of the kinds of functions, and the means by which they are manipulated, is there. The **evalhook** facility lets you do something arbitrary whenever the evaluator is invoked. See the section **"evalhook"** in *Program Development Utilities*. Special forms are described throughout the documentation.

10. Variables

In Symbolics-Lisp, 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. If it is a special variable, it uses the value cell to hold the value of the variable. It it is not special, it looks it up in the local lexical environment. If you evaluate a symbol, you get back the contents of the symbol's value cell.

10.1 Changing the Value of a Variable

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. This is enforced by having binding done only by special forms that are defined to bind some variables, then evaluate some subforms, and then unbind those variables. So the variables are all unbound when the form is finished. This means that the evaluation of the form does not disturb the values of the variables that are bound; their old value, before the evaluation of the form, is restored when the evaluation of the form is completed. If such a form is exited by a nonlocal exit of any kind, such as **throw** or **return**, the bindings are undone whenever the form is exited.

10.2 Binding Variables

The simplest construct for binding variables is the **let** special form. The **do** and **prog** special forms can also bind variables, in the same way **let** does, but they also control the flow of the program and so are explained elsewhere. See the section "Iteration", page 189. **let*** is just a sequential version of **let**.

Binding is an important part of the process of applying functions to arguments. See the section "Evaluating a Function Form", page 151.

10.3 Kinds of Variables

In Symbolics Lisp, there are three kinds of variables: *local*, *special*, and *instance*. A special variable has dynamic scope: any Lisp expression can access it simply by mentioning its name. A local variable has lexical scope: only Lisp expressions lexically contained in the special form that binds the local variable can access it. An instance variable has a different kind of lexical scope: only Lisp expressions lexically contained in methods of the appropriate flavor can access it. Instance variables are explained in another section. See the section "Modularity and Object-oriented Programming", page 418.

Variables are assumed to be local unless they have been declared to be special or they have been implicitly declared to be instance variables by **defmethod**. Variables can be declared special by the special forms **defvar** and **defconst**, or by explicit declarations. See the section "Declarations", page 311. 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. Named constants are considered to be a kind of special variable whose value is never changed.

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 nonspecial variables. Rather, the compiler converts the references to such variables within the program into more efficient references that do not involve symbols at all. The interpreter stores the values of variables in the same places as the compiler, but uses less specialized and efficient mechanisms to access them.

The value of a special variable is stored in the value cell of the associated symbol. Binding a special variable saves the old value away and then uses the value cell of the symbol to hold the new value.

When a local variable is bound, a memory cell is allocated in a hidden, internal place (the Lisp control stack) and the value of the variable is stored in this cell. You cannot use a local variable without first binding it; you can only use a local variable inside a special form that binds that variable. Local variables do not have any "top-level" value; they do not even exist outside the form that binds them.

The value of an instance variable is stored in an instance of the appropriate flavor. Each instance has its own copy of the instance variable. It is impermissible to bind an instance variable.

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. Binding a local variable, however, does not do anything to the symbol. In fact, it creates a new memory cell to hold the value, that is, a new local variable.

A reference to a variable that you did not bind yourself is called a *free reference*.

When one function definition is nested inside another function definition, using **lambda**, **flet**, or **labels**, the inner function has access to the local variables bound by the outer function. An access by the inner function to a local variable of the outer function looks like a free reference when only the inner function is considered. However, when the entire surrounding context is considered, it is a bound reference. We call this a *captured free reference.* When a function definition is nested inside a method, it can refer to instance variables just as the method can.

You cannot use a local variable without first binding it. Another way to say this is that you cannot ever have an uncaptured free reference to a local variable. If you try to do so, the compiler complains and assumes that the variable is special, but was accidentally not declared. The interpreter also assumes that the variable is special, but does not print a warning message.

Here is an example of how the compiler and the interpreter produce the same results, but the compiler prints more warning messages.

```
(setq a 2)          ;Set the special variable a to the value 2.
                     ;But don't declare a special.

(defun foo ()       ;Define a function named foo.
  (let ((a 5))      ;Bind the local variable a to the value 5.
    (bar)))         ;Call the function bar.

(defun bar ()       ;Define a function named bar.
  a)               ;It makes a free reference to the special variable a.

(foo) => 2          ;Calling foo returns 2.

(compile 'foo)      ;Now compile foo.
                     ;This warns that the local variable a was bound,
                     ;but was never used.

(foo) => 2          ;Calling foo still returns 2.

(compile 'bar)      ;This warns about the free reference to a.

(foo) => 2          ;Calling foo still returns 2.
```

When **bar** was compiled, the compiler saw the free reference and printed a warning message: `Warning: a declared special`. It 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, such as **foo** in the example, 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 produces 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 **defvar** rather than **declare** inside the body of a form, the declaration is "global"; that is, it applies wherever that variable name is seen. After **fuzz** has been declared special using **defvar**, all following uses of **fuzz** are treated 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 editor can find them easily.

10.4 Special Forms for Setting Variables

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

Used to set the value of one or more variables. The first *value* is evaluated, and the first *variable* is set to the result. Then the second *value* is evaluated, the second *variable* is set to the result, and so on for all the variable/value pairs. **setq** returns the last value, that is, the result of the evaluation of its last subform. Example:

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

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

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

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

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

10.5 Special Forms for Binding Variables

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

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

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

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

You can omit the *vform* from a **let** clause, in which case it is as if the *vform* were **nil**: the variable is bound to **nil**. Furthermore, you can replace the entire clause (the list of the variable and form) with just the variable, which also means that the variable gets bound to **nil**. It is customary to write just a variable, rather than a clause, to indicate that the value to which the variable is bound does not matter, because the variable is **setq**'ed before its first use. 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*

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**.

compiler-let *bindlist body...* *Special Form*

When interpreted, a **compiler-let** form is equivalent to **let** with all variable bindings declared special. When the compiler encounters a **compiler-let**, however, it performs the bindings specified by the form (no compiled code is generated for the bindings) and then compiles the body of the **compiler-let** with all those bindings in effect. In particular, macros within the body of the **compiler-let** form are expanded in an environment with the indicated bindings. See the section "Nesting Macros", page 366.

compiler-let allows compiler switches to be bound locally at compile time, during the processing of the *body* forms. Value forms are evaluated at compile time. See the section "Compiler Switches". In the following example the use of **compiler-let** prevents the compiler from open-coding the **map**.

```
(compiler-let ((open-code-map-switch nil))
  (map (function (lambda (x) ...)) foo))
```

letf *places-and-values body...* *Special Form*

Just like **let**, except that it can bind any storage cells rather than just variables. The cell to be bound is specified by an access form that must be acceptable to **locf**. For example, **letf** can be used to bind slots in a structure. **letf** does parallel binding.

Given the following structure, **letf** calls **do-something-to** with **ship**'s x position bound to zero.

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

(letf (((position-x ship) 0))
  (do-something-to ship))
```

It is preferable to use **letf** instead of the **bind** subprimitive.

letf* *places-and-values body...* *Special Form*

Just like **let***, except that it can bind any storage cells rather than just variables. The cell to be bound is specified by an access form that must be acceptable to **locf**. For example, **letf*** can be used to bind slots in a structure. **letf*** does sequential binding.

Given the following structure, **letf*** calls **do-something-to** with **ship**'s x position bound to 0 and y position bound to 5.

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

(letf* (((position-x ship) 0)
        ((position-y ship) (+ (position-x ship) 5)))
  (do-something-to ship))
```

It is preferable to use **letf*** instead of the **bind** subprimitive.

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

A variant of **let** in which the binding of variables is conditional. The variables must all be special variables. The **let-if** special form, typically written as:

```
(let-if cond
        ((var-1 val-1) (var-2 val-2)...)
  body-form1 body-form2...)
```

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

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

Similar in form to **let**. The difference is that **let-globally** does not *bind* the variables; instead, it saves the old values and *sets* the variables, and sets up an **unwind-protect** to set them back. The important difference between **let-globally** and **let** is that when the current stack group calls some other stack group, the old values of the variables are *not* restored. Thus, **let-globally** makes the new values visible in all stack groups and processes that do not bind the variables themselves, not just the current stack group.

let-globally-if *predicate varlist body...* *Special Form*

let-globally-if is like **let-globally**. It takes a predicate form as its first argument. It binds the variables only if *predicate* evaluates to something other than **nil**. *body* is evaluated in either case.

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

Provides 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*

A somewhat modified kind of **progv**; like **progv**, it only works for special variables. First, *vars-and-vals-form* is evaluated. Its value should be a list that looks like the first subform of a **let**:

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

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

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

destructuring-bind *variable-pattern data body...* *Special Form*

Binds variables to values, using **defmacro**'s destructuring facilities, and evaluates the body forms in the context of those bindings.

First *data* is evaluated. If *variable-pattern* is a symbol, it is bound to the result of evaluating *data*. If *variable-pattern* is a tree, the result of evaluating *data* should be a tree of the same shape. The trees are disassembled, and each variable that is a component of *variable-pattern* is bound to the value that is the corresponding element of the tree that results from evaluating *data*. If not enough values are supplied, the remaining variables are bound to **nil**. If too many values are supplied, the excess values are ignored. Finally, the body forms are evaluated sequentially, the old values of the variables are restored, and the result of the last body form is returned.

As with the pattern in a **defmacro** form, *variable-pattern* actually resembles the **lambda**-list of a function; it can have **&**-keywords. See the section "**&**-Keywords Accepted by **defmacro**", page 373.

Example:

```
(destructuring-bind (a (b) &optional (c 'd))
                    '((x y) (z))
  (values a b c))
```

returns **(x y)**, **z**, and **d**.

scl:destructuring-bind also exists. It is the same as **destructuring-bind** except that it signals an error if the trees *data* and *variable-pattern* do not match.

desetq *{variable-pattern value-pattern}...* *Special Form*

Lets you assign values to variables through destructuring patterns. In place of a variable to be assigned, you can provide a tree of variables. The value to be assigned must be a tree of the same shape. The trees are destructured into their component parts, and each variable is assigned to the corresponding part of the value tree.

The first *value-pattern* is evaluated. If *variable-pattern* is a symbol, it is set to the result of evaluating *value-pattern*. If *variable-pattern* is a tree, the result of evaluating *value-pattern* should be a tree of the same shape. The trees are destructured, and each variable that is a component of *variable-pattern* is set to the value that is the corresponding element of the tree that results from evaluating *value-pattern*. This process is repeated for each pair of *variable-pattern* and *value-pattern*. **desetq** returns the last value. Example:

```
(desetq (a b) '((x y) z) c b)
```

a is set to **(x y)**, **b** is set to **z**, and **c** is set to **z**. The form returns the value of the last form, which is the symbol **z**.

dlet *((variable-pattern value-pattern)...) body...* *Special Form*

Binds variables to values, using destructuring, and evaluates the body forms in the context of those bindings. In place of a variable to be assigned, you can provide a tree of variables. The value to be assigned must be a tree of the same shape. The trees are destructured into their component parts, and each variable is assigned to the corresponding part of the value tree.

First the *value-patterns* are evaluated. If a *variable-pattern* is a symbol, it is bound to the result of evaluating the corresponding *value-pattern*. If *variable-pattern* is a tree, the result of evaluating *value-pattern* should be a tree of the same shape. The trees are destructured, and each variable that is a component of *variable-pattern* is bound to the value that is the corresponding element of the tree that results from evaluating *value-pattern*. The bindings happen in parallel; all the *value-patterns* are evaluated before any variables are bound. Finally, the body forms are evaluated sequentially, the old values of the variables are restored, and the result of the last body form is returned. Example:

```
(dlet (((a b) '((x y) z))
       (c 'd))
       (values a b c))
```

returns **(x y)**, **z**, and **d**.

dlet* *((variable-pattern value-pattern)...) body...* *Special Form*

Binds variables to values, using destructuring, and evaluates the body forms in the context of those bindings. In place of a variable to be assigned, you can provide a tree of variables. The value to be assigned must be a tree of

the same shape. The trees are destructured into their component parts, and each variable is assigned to the corresponding part of the value tree.

The first *value-pattern* is evaluated. If *variable-pattern* is a symbol, it is bound to the result of evaluating *value-pattern*. If *variable-pattern* is a tree, the result of evaluating *value-pattern* should be a tree of the same shape. The trees are destructured, and each variable that is a component of *variable-pattern* is bound to the value that is the corresponding element of the tree that results from evaluating *value-pattern*. The process is repeated for each pair of *variable-pattern* and *value-pattern*. The bindings happen sequentially; the variables in each *variable-pattern* are bound before the next *value-pattern* is evaluated. Finally, the body forms are evaluated sequentially, the old values of the variables are restored, and the result of the last body form is returned. Example:

```
(dlet* (((a b) '((x y) z)) (c b)) (values a b c))
```

returns **(x y)**, **z**, and **z**.

10.6 Special Forms for Defining Special Variables

defvar *variable* &optional *initial-value documentation* *Special Form*

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 and records its location for the sake of the editor so that you can ask to see where the variable is defined. If a second subform is supplied,

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

variable is initialized to the result of evaluating the form *initial-value* unless it already has a value, in which case it keeps that value. *initial-value* is not evaluated unless it is used; this is useful if it does something expensive like creating a large data structure.

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

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

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

allows you to include a documentation string that describes what the variable is for or how it is to be used. Using such a documentation string is even

better than commenting the use of the variable, because the documentation string is accessible to system programs that can show the documentation to you while you are using the machine.

If **defvar** is used in a patch file or is a single form (not a region) evaluated with the editor's compile/evaluate from buffer commands, if there is an initial-value the variable is always set to it regardless of whether it is already bound. See the section "Patch Facility" in *Program Development Utilities*.

defconst *variable initial-value* &optional *documentation* *Special Form*

The same as **defvar**, except that *variable* is always set to *initial-value* regardless of whether *variable* is already bound. The rationale for this is that **defvar** declares a global variable, whose value is initialized to something but is then changed by the functions that use it to maintain some state. On the other hand, **defconst** declares a constant, whose value is never 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 you then evaluate the **defconst** form again, the variable gets the new value. It is not the intent of **defconst** to declare that the value of *variable* never changes; for example, **defconst** is not license to the compiler to build assumptions about the value of *variable* into programs being compiled. See **defconstant** for that.

10.7 Special Form for Declaring a Named Constant

defconstant *variable initial-value* &optional *documentation* *Special Form*

Declares the use of a named constant in a program. *initial-value* is evaluated and *variable* set to the result. The value of *variable* is then fixed. It is an error if *variable* has any special bindings at the time the **defconstant** form is executed. Once a special variable has been declared constant by **defconstant**, any further assignment to or binding of that variable is an error.

The compiler is free to build assumptions about the value of the variable into programs being compiled. If the compiler does replace references to the name of the constant by the value of the constant in code to be compiled, the compiler takes care that such "copies" appear to be **eql** to the object that is the actual value of the constant. For example, the compiler can freely make copies of numbers, but it exercises care when the value is a list.

In Symbolics-Lisp, **defconstant** and **defconst** are essentially the same if the value is other than a number, a character, or an interned symbol. However, if the variable being declared already has a value, **defconst** freely changes the value, whereas **defconstant** queries before changing the value. **defconstant**'s query offers three choices: Y, N, and P.

- The Y option changes the value.
- The N option does not change the value.
- The P option changes the value and when you change any future value, it prints a warning rather than a query.

The P option sets **inhibit-fdefine-warnings** to **:just-warn**. **defconstant** obeys that variable, just as **query-about-redefinition** does. Use **(setq inhibit-fdefine-warnings nil)** to revert to the querying mode.

When the value of a constant is changed by a patch file, a warning is printed.

defconstant assumes that changing the value is dangerous because the old value might have been incorporated into compiled code, which is out of date if the value changed.

In general, you should use **defconstant** to declare constants whose value is a number, character, or interned symbol and is guaranteed not to change. An example is π. The compiler can optimize expressions that contain references to these constants. If the value is another type of Lisp object or if it might change, you should use **defconst** instead.

documentation, if provided, should be a string. It is accessible to the **documentation** function.

11. Lexical Scoping

Symbolics-Lisp has a lexically scoped interpreter and compiler. The compiler and interpreter implement the same language.

Consider the following example:

```
(defun fun1 (x)
  (fun2 3 x)
  (fun3 #'(lambda (y) (+ x y)) x 4))
```

This function passes an *internal lambda* to **fun3**. Observe that the internal lambda references the variable **x**, which is neither a lambda variable nor a local variable of this lambda. Rather, it is a variable local to the lambda's *lexical parent*, **fun1**. **fun3** receives as an argument a *lexical closure*, that is, a presentation of the internal lambda in an environment where the variable **x** can be accessed. **x** is a *free lexical variable* of the internal lambda; the closure is said to be a closure of the free lexical variables, specifically in this case, **x**.

Lexical closures, created by reference to internal functions, are to be distinguished from *dynamic closures*, which are created by the **closure** function and the **let-closed** special form. Dynamic closures are closures over *special* variables, while lexical closures are closures over *lexical, local* variables. Invocation of a dynamic closure, as a function, causes special variables to be bound. Invocation of a lexical closure simply provides the necessary data linkage for a function to run in the environment in which the closure was made.

Both the compiler and the interpreter support the accessing of lexical variables. The compiler and interpreter also support, in Zetalisp as well as Symbolics Common Lisp, the Common Lisp lexical function and macro definition special forms, **flet, labels,** and **macrolet.**

Note that access to lexical variables is true access to the instantiation of the variable and is not limited to the access of values. Thus, assuming that **map-over-list** maps a function over a list in some complex way, the following function works as it appears to, and finds the maximum element of the list.

```
(defun find-max (list)
  (let ((max nil))
    (map-over-list
      #'(lambda (element)
          (when (or (null max)
                    (> element max))
            (setq max element)))
      list)
    max))
```

11.1 Lexical Environment Objects and Arguments

Macro-expander functions, the actual functions defined by **defmacro**, **macro**, and **macrolet**, are called with two arguments — *form* and *environment*. Special form implementations used by the interpreter are also passed these two arguments. Macro-expander functions defined by files created prior to the implementation of lexical scoping are passed only a *form* argument, for compatibility.

The *environment* argument allows evaluations and expansions performed by the macro-expander function or the special form interpreter function to be performed in the proper lexical context. The *environment* argument is utilized by the macro-expander function in certain unusual circumstances:

- A macro-expander function explicitly calls **macroexpand** or **macroexpand-1** to expand some code appearing in the form which invoked it. In this case, the environment argument must be passed as a second argument to either of these functions. This is quite uncommon. Most macro-expander functions do not explictly expand code contained in their calls: **setf** is an example of a macro that does this kind of expansion.

- A macro-expander function explicitly calls **eval** to evaluate, at macro time, an expression appearing in the code which invoked it. In that case, the environment argument must be passed as a second argument to **eval**. This explicit evaluation is even more unusual: almost any use of **eval** by a macro is guaranteed to be wrong, and does not work or do what is intended in certain circumstances. The only known legitimate uses are:

    - A macro determines that some expression is in fact a constant, and computable at macro expand time, and evaluates it. Here, there are no variables involved, so the environment issue is moot.

    - A macro is called with some template code, expressed via backquote, and is expected to produce an instantiation of that template with substitutions performed. Evaluation is the way to instantiate backquoted templates.

The format of lexical environments is an internal artifact of the system. They cannot be constructed or analyzed by user code. It is, however, specified that **nil** represents a null lexical environment.

A macro defined with **defmacro** or **macrolet** can accept its expansion lexical environment (if it needs it for either of the above purposes) as a variable introduced by the lambda-list keyword **&environment** in its argument list.

A macro defined with **macro** receives its lexical environment as its second argument.

11.2 Funargs and Lexical Closure Allocation

A *funarg* is a function, usually a lambda, passed as an argument, stored into data structure, or otherwise manipulated as data. Normally, functions are simply called, not manipulated as data. The term funarg is an acronym for *functional argument*. In the following form, two functions are referred to, **sort** and <.

```
(defun number-sort (numbers)
  (sort numbers #'<))
```

sort is being called as a function, but < (more exactly, the function object implementing the < function) is being passed as a funarg.

The major feature of the lexical compiler and interpreter can be described as the support of funargs that reference free lexical variables. Funargs that do not reference free lexical variables also work. For example,

```
(defun data-sort (data)
  (sort data #'(lambda (x y) (< (fun x) (fun y)))))
```

The internal lambda above makes no free lexical references. **data-sort** would have worked prior to lexical scoping, and continues to work.

The remainder of this discussion is concerned only with funargs that make free lexical references.

The act of evaluating a form such as

```
#'(lambda (x) (+ x y))
```

produces a lexical closure. (Of course, if we are talking about compiled code, the form is never evaluated. In that case, we are talking about the time in the execution of the compiled function that corresponds to the time that the form would be evaluated.) It is that closure that represents the funarg that is passed around.

Funarg closures can be further classified by usage as *downward funargs* and *upward funargs*. A *downward funarg* is one that does not survive the function call that created the closure. For example:

```
(defun magic-sort (data parameter)
  (sort data #'(lambda (x y) (< (funk x parameter)
                                (funk y parameter)))))
```

In this example, **sort** is passed a lexical closure of the internal lambda. **sort** calls this closure many times to do comparisons. When **magic-sort** returns its value, no function or data structure is referencing that closure in any way. That closure is being used as a *downward* funarg; it does not survive the call to **magic-sort**.

In this example,

```
(defun make-adder (x)
  #'(lambda (y) (+ x y)))
```

the closure of the internal lambda is returned from the activation of **make-adder**, and survives that activation. The closure is being used as an *upward funarg*.

The creation of lexical closures involves the allocation of storage to represent them. This storage can either be allocated on the stack or in the heap. Storage allocated in the heap remains allocated until all references to it are discarded and it is garbage collected. Storage allocated on the stack is transient, and is deallocated when the stack frame in which it is allocated is abandoned. Stack-allocated closures are more efficient, and thus to be desired. Stack-allocated closures can only be used when a funarg is used as a downward funarg. Closures of upward funargs must be allocated in the heap.

Funargs can be passed to any functions. These functions might well store them in permanent data structure, or return them nonlocally, or cause other upward use. Therefore, the compiler and interpreter, in general, must and do assume potential upward use of all funargs. Thus, they cause their closures to be allocated in the heap unless special measures are taken to convey the guarantee of downward-only use. Note that the more general (heap-allocated) closure is guaranteed to work in all cases.

The ephemeral garbage collector substantially reduces the overhead of heap allocation of short-lived objects. Thus, you might be able to ignore these issues entirely, and let the system do as well as it can without additional help.

11.2.1 The sys:downward-function and sys:downward-funarg Declarations

There are two ways to convey the guarantee of downward-only use of a funarg. These are the **sys:downward-function** and **sys:downward-funarg** declarations.

sys:downward-function Declaration

The declaration **sys:downward-function**, in the body of an internal lambda, guarantees to the system that lexical closures of the lambda in which it appears are only used as downward funargs, and never survive the calls to the procedure that produced them. This allows the system to allocate these closures on the stack.

```
(defun special-search-table (item)
  (block search
    (send *hash-table* :map-hash
          #'(lambda (key object)
              (declare (sys:downward-function))
              (when (magic-function key object item)
                (return-from search object))))))
```

Here, the **:map-hash** message to the hash table calls the closure of the internal lambda many times, but does not store it into permanent variables or data structure, or return it "around" **special-search-table**. Therefore, it is guaranteed that the closure does not survive the call to **special-search-table**. It is thus safe to allow the system to allocate that closure on the stack.

Stack-allocated closures have the same lifetime (*extent*) as **&rest** arguments and lists created by **with-stack-list** and **with-stack-list***, and require the same precautions. See the section "Lambda-list Keywords", page 309.

sys:downward-funarg Declaration

The **sys:downward-funarg** declaration (not to be confused with **sys:downward-function**) permits a procedure to declare its intent to use one or more of its arguments in a downward manner. For instance, **sort**'s second argument is a funarg, which is only used in a downward manner, and is declared this way. The second argument to **process-run-function** is a good example of a funarg that is not downward. Here is an example of a function that uses and declares its argument as a downward funarg.

```
(defun search-alist-by-predicate (alist predicate)
  (declare (sys:downward-funarg predicate))
  ;; Traditional "recursive" style, for variety.
  (if (null alist)
      nil
      (let ((element (car list))
            (rest (cdr list))
        (if (funcall predicate (car element))
            (cdr element)
            (search-alist-by-predicate rest predicate))))))
```

This function only calls the funarg passed as the value of **predicate**. It does not store it into permanent structure, return it, or throw it around **search-alist-by-predicate**'s activation.

The reason you so declare the use of an argument is to allow the system to deduce guaranteed downward use of a funarg without need for the **sys:downward-function** declaration. For instance, if **search-alist-by-predicate** were coded as above, we could write

```
(defun look-for-element-in-tolerance (alist required-value tolerance)
  (search-alist-by-predicate alist
    #'(lambda (key)
        (< (abs (- key required-value)) tolerance))))
```

to search the keys of the list for a number within a certain tolerance of a required value. The lexical closure of the internal lambda is automatically allocated by the system on the stack because the system has been told that any funarg used as the first argument to **search-alist-by-predicate** is used only in a downward manner. No declaration in the body of the lambda is required.

All appropriate parameters to system functions have been declared in this way.

There are two possible forms of the **downward-funarg** declaration:

(declare (sys:downward-funarg *var1 var2 ...* **)**
Declares the named variables, which must be parameters (formal arguments) of the function in which this declaration appears, to have their values used only in a downward fashion. This affects the generation of closures as functional arguments to the function in which this declaration appears: it does not directly affect the function itself. Due to an implementation restriction, *var-i* cannot be a keyword argument.

(declare (sys:downward-funarg *))
Declares guaranteed downward use of all functional arguments to this function. This is to cover closures of functions passed as elements of **&rest** arguments and keyword arguments.

Notes:

The special forms **flet** and **labels** (additions to Zetalisp from Common Lisp) generate lexical closures if necessary. The compiler decides how to allocate a closure generated by **flet** or **labels** after analysis of the use of the function defined by the use of **flet** or **labels**.

It is occasionally appropriate to introduce the **sys:downward-funarg** and **sys:downward-function** (as well as other) declarations into the bodies of functions defined by **flet** and **labels**.

There is no easy way to see if code allocates lexical closures on the heap or on the stack; if disassembly of a compiled function reveals a call to **sys:make-lexical-closure**, heap allocation is indicated.

11.3 flet, labels, and macrolet Special Forms

flet *((name args function-body...) ...) body...* *Special Form*
Defines named internal functions. **flet** (function **let**) defines a lexical scope, *body*, in which these names can be used to refer to these functions. *((name args function-body...) ...)* is a list of clauses, each of which defines one function. Each clause of the **flet** is identical to the cdr of a **defun** special form; it is a function name to be defined, followed by an argument list, possibly declarations, and function body forms. **flet** is a mechanism for defining internal subroutines whose names are known only within some local scope.

Functions defined by the clauses of a single **flet** are defined "in parallel", similar to **let**. The names of the functions being defined are not defined and not accessible from the bodies of the functions being defined. The **labels** special form is used to meet those requirements. See the special form **labels**, page 144.

Here is an example of the use of **flet**:

```
(defun triangle-perimeter (p1 p2 p3)
  (flet ((squared (x) (* x x)))
    (flet ((distance (point1 point2)
             (sqrt (+ (squared (- (point-x point1)
                                  (point-x point2)))
                      (squared (- (point-y point1)
                                  (point-y point2)))))))
      (+ (distance p1 p2)
         (distance p2 p3)
         (distance p1 p3)))))
```

flet is used twice here, first to define a subroutine **squared** of **triangle-perimeter**, and then to define another subroutine, **distance**. Note that since **distance** is defined within the scope of the first **flet**, it can use **squared**. **distance** is then called three times in the body of the second **flet**. The names **squared** and **distance** are not meaningful as function names except within the bodies of these **flet**s.

Note that functions defined by **flet** are internal, lexical functions of their containing environment. They have the same properties with respect to lexical scoping and references as internal lambdas. They can make free lexical references to variables of that environment and they can be passed as *funargs* to other procedures. Functions defined by **flet**, when passed as funargs, generate closures. The allocation of these closures, that is, whether they appear on the stack or in the heap, is controlled in the same way as for internal lambdas. See the section "Funargs and Lexical Closure Allocation", page 139.

Here is an example of the use, as a funarg, of a closure of a function defined by **flet**.

```
(defun sort-by-closeness-to-goal (list goal)
  (flet ((closer-to-goal (x y)
           (< (abs (- x goal)) (abs (- y goal)))))
    (sort list #'closer-to-goal)))
```

This function sorts a list, where the sort predicate of the (numeric) elements of the list is their absolute distance from the value of the parameter **goal**. That predicate is defined locally by **flet**, and passed to **sort** as a funarg.

Note that **flet** (as well as **labels**) defines the use of a name as a function, not as a variable. Function values are accessed by using a name as the car of a form or by use of the **function** special form (usually expressed by the reader macro #').

Within its lexical scope, **flet** can be used to redefine names that refer to globally defined functions, such as **sort** or **cdar**, though this is not recommended for stylistic reasons. This feature does, however, allow you to bind names with **flet** in an unrestricted fashion, without binding the name of some other function that you might not know about (such as

number-into-array), and thereby causing other functions to malfunction. This occurs because **flet** always creates a lexical binding, not a dynamic binding. Contrast this with **let**, which usually creates a lexical binding, unless the variable being bound is declared special, in which case it creates a dynamic binding.

flet can also be used to redefine function names defined by enclosing uses of **flet** or **labels**.

labels *((name args function-body...) ...) body...* *Special Form*

Identical to **flet** in structure and purpose, but has slightly different scoping rules. It, too, defines one or more functions whose names are made available within its body. In **labels**, unlike **flet**, however, the functions being defined can refer to each other mutually, and to themselves, recursively. Any of the functions defined by a single use of **labels** can call itself or any other; there is no order dependence. Although **flet** is analogous to **let** in its parallel binding, **labels** is not analogous to **let***.

labels is in all other ways identical to **flet**. It defines internal functions that can be called, re-redefined, passed as funargs, and so on.

Functions defined by **labels**, when passed as funargs, generate closures. The allocation of these closures, that is, whether they appear on the stack or in the heap, is controlled in the same way as for internal lambdas. See the section "Funargs and Lexical Closure Allocation", page 139.

Here is an example of the use of **labels**:

```
(defun combinations (total-things at-a-time)
  ;; This function computes the number of combinations of
  ;; total-things things taken at-a-time at a time.
  ;; There are more efficient ways, but this is illustrative.
  (labels ((factorial (x)
             (permutations x x))
           (permutations (x n)                ;x things n at a time
             (if (= n 1)
                 x
                 (* x (permutations (1- x) (1- n))))))
    (// (permutations total-things at-a-time)
        (factorial at-a-time))))
```

macrolet *((name args macro-body...) ...) body...* *Special Form*

Defines, within its scope, a macro. It establishes a symbol as a name denoting a macro, and defines the expander function for that macro. **defmacro** does this globally; **macrolet** does it only within the (lexical) scope of its body. A macro so defined can be used as the car of a form within this scope. Such forms are expanded according to the definition supplied when interpreted or compiled.

The syntax of **macrolet** is identical to that of **flet** or **labels**: it consists of clauses defining local, lexical macros, and a body in which the names so defined can be used. *((name args macro-body...) ...) body...* is a list of clauses each of which defines one macro. Each clause is identical to the cdr of a **defmacro** form: it has a name being defined (a symbol), a macro pseudo-argument list, and an expander function body.

The pseudo-argument list is identical to that used by **defmacro**. It is a pattern, and can use appropriate lambda-list keywords for macros, including **&environment**. See the section "Lexical Environment Objects and Arguments", page 138.

Here is an example of the use of **macrolet**:

```
(defun check-value (z)
  (block check-value
    (macrolet ((succeed () (return-from check-value t))
               (fail () (return-from check-value nil)))
      (cond ((test-1 z) (fail))
            ((test-2 z) (succeed))
            (t (fail))))))
```

It is important to realize that macros defined by **macrolet** are run (when the compiler is used), at compile time, not run-time. The expander functions for such macros, that is, the actual code in the body of each **macrolet** clause, cannot attempt to access or set the values of variables of the function containing the use of **macrolet**. Nor can it invoke run-time functions, including local functions defined in the lexical scope of the **macrolet** by use of **flet** or **labels**. The expander function can freely generate code that uses those variables and/or functions, as well as other macros defined in its scope, including itself.

There is an extreme subtlety with respect to expansion-time environments of **macrolet**. It should not affect most uses. The macro-expander functions are closed in the global environment; that is, no variable or function bindings are inherited from any environment. This also means that macros defined by **macrolet** cannot be used in the expander functions of other macros defined by **macrolet** within the scope of the outer **macrolet**. This does not prohibit either of the following:

- Generation of code by the inner macro that refers to the outer one.
- Explicit expansion (by **macroexpand** or **macroexpand-1**), by the inner macro, of code containing calls to the outer macro. Note that explicit environment management must be utilized if this is done. See the section "Lexical Environment Objects and Arguments", page 138.

12. Generalized Variables

In Lisp, a variable is something that can remember one piece of data. The main operations on a variable are to recover that piece of data, and to change it. These might be called *access* and *update*. The concept of variables named by symbols can be generalized to any storage location that can remember one piece of data, no matter how that location is named. See the section "Variables", page 125.

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

setf is particularly useful in combination with structure accessors, such as those created with **defstruct**, because the knowledge of the representation of the structure is embedded inside the accessor, and you should not have to know what it is in order to alter an element of the structure.

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

setf *access-form value* *Macro*

Takes a form that *accesses* something, and "inverts" it to produce a corresponding form to *update* the thing. A **setf** expands into an update form, which stores the result of evaluating the form *value* into the place referenced by the *access-form*. Examples:

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

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

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))
```

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

Furthermore, the value produced by **setf** depends on the structure type and is not guaranteed; **setf** should be used for side effect only. If you want well-defined semantics, you can use **cl:setf** in your Symbolics-Lisp programs.

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 returns the address of that cell as a locative pointer. See the section "Locatives", page 83. This locative pointer is a kind of name for the variable that is a first-class Lisp data object. It can be passed as an argument to a function that operates on any kind of variable, regardless of how it is named. It can be used to *bind* the variable, using the **bind** subprimitive.

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

locf *access-form* *Macro*

Takes a form that *accesses* some cell and produces a corresponding form to create a locative pointer to that cell. Examples:

```
(locf (array-leader foo 3)) ==> (ap-leader foo 3)
(locf a) ==> (variable-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, **locf** expands the *access-form* and starts over again. This lets you use **locf** together with **defstruct** accessors.

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

Both **setf** and **locf** work by means of property lists. When the form **(setf (aref q 2) 56)** is expanded, **setf** looks for the **setf** property of the symbol **aref**. The value of the **setf** property of a symbol should be a cons whose car is a pattern to be matched with the *access-form*, and whose cdr is the corresponding *update-form*, with the symbol **si:val** in place of the value to be stored. The **setf** property of **aref** is a cons whose car is **(aref array . subscripts)** and whose cdr is **(aset si:val array . subscripts)**. If the transformation that **setf** is to do cannot

be expressed as a simple pattern, an arbitrary function can 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 is applied to two arguments, **(foo bar)** and **baz**, and the result is 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* &optional *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 might evaluate parts of *ref* more than once. (**cl:incf** does not evaluate any part of *ref* more than once.)

Example:

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

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

decf *access-form* &optional *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.

You must take great care with **decf** because it might evaluate parts of *ref* more than once. (**cl:decf** does not evaluate any part of *ref* more than once.)

swapf *a b* *Macro*

Exchanges the value of one generalized variable with that of another. *a* and *b* are access-forms suitable for **setf**. The returned value is not defined. All the caveats that apply to **incf** apply to **swapf** as well: Forms within *a* and *b* can be evaluated more than once. (**cl:rotatef** does not evaluate any form within *a* and *b* more than once.)

Examples:

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

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

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

push *item access-form* *Macro*

Adds an item to the front of a list that 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* might be evaluated more than once. (**cl:push** does not evaluate any part of *ref* more than once.) The returned value of **push** is not defined.

push-in-area *item access-form area* *Macro*

Adds an item to the front of a list that is stored in a generalized variable. **(push-in-area** *item ref area***)** creates a new cons in *area* whose car is the result of evaluating *item* and whose cdr is the contents of *ref*, and stores the new cons into *ref*. See the section "Areas" in *Internals, Processes, and Storage Management*.

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* might be evaluated more than once. (**cl:pop** does not evaluate any part of *ref* more than once.)

13. Evaluating a Function Form

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? Symbolics-Lisp contains many kinds of functions, and applying them can do many different kinds of things. This section explains the most basic kinds of functions and how they work, and in particular, *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.

1. The number of arguments and the number of variables in the lambda list must be the same, or else an error is signalled.

2. Each variable is bound to the corresponding argument value.

3. The forms of the body are evaluated sequentially.

4. The bindings are all undone and the value of the last form in the body is returned.

This might sound something like the description of **let**. The most important difference is that the lambda-expression is 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 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. Additional features are accessible by using certain keywords (which start with **&**) and/or lists as elements of the lambda list.

The principal weakness of the simple lambda lists is that any function written with one must only take a certain fixed number of arguments. As we know, many very useful functions, such as **list**, **append**, +, and so on, accept a varying number of

arguments. Maclisp solved this problem by the use of *lexprs* and *lsubrs*, which were somewhat inelegant since the parameters had to be referred to by numbers instead of names (for example, **(arg 3)**). (For compatibility reasons, Symbolics-Lisp supports *lexprs*, but they should not be used in new programs). Simple lambda lists also require that arguments be matched with parameters by their position in the sequence. This makes calls hard to read when there are a great many arguments. Keyword parameters enable the use of other styles of call which are more readable.

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

Keyword parameters are always optional, regardless of whether the lambda list contains **&optional**. Any **&optional** appearing after the first keyword argument has no effect. **&key** and **&rest** are independent. They can both appear and they both use the same arguments from the argument list. The only rule is that **&rest** must appear before **&key** in the lambda list.

This is the ordering rule for lambda-list keywords. The following keywords must appear in this order, any or all of them can be omitted, and they cannot appear multiple times:

```
&optional &rest &key &allow-other-keys &aux
```

There are some other keywords in addition to those mentioned here. See the section "Lambda-list Keywords", page 309.

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

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

13.1 Binding Parameters to Arguments

When **apply** (the primitive function that applies functions to arguments) matches up the arguments with the parameters, it follows this algorithm:

1. The positional parameters are dealt with first.

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

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

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

    a. First, if there is a rest parameter, it is bound to a list of all the remaining arguments. If there are no remaining arguments, it gets bound to **nil**.

    b. If there are keyword parameters, the same remaining arguments are used to bind them.

5. The arguments for the keyword parameters are treated as a list of alternating keyword symbols and associated values. Each symbol is matched with the keyword parameter names, and the matching keyword parameter is bound to the value that follows the symbol. All the remaining arguments are treated in this way. Since the arguments are usually obtained by evaluation, those arguments that are keyword symbols are typically quoted in the call; however, they do not have to be. The keyword symbols are compared by means of **eq**, which means they must be specified in the correct package. The keyword symbol for a parameter has the same print name as the parameter, but resides in the keyword package regardless of what package the parameter name itself resides in. (You can specify the keyword symbol explicitly in the lambda list if you must.)

If any keyword parameter has not received a value when all the arguments have been processed, the default-form for the parameter is evaluated and the parameter is bound to its value. The default form can depend on parameters to its left in the lambda-list.

There might be a keyword symbol among the arguments that does not match any keyword parameter name. An error is signalled unless **&allow-other-keys** is present in the lambda list, or there is a keyword argument pair whose keyword is **:allow-other-keys** and whose value is not **nil**. If an error is not signalled, then the nonmatching symbols and their associated values are ignored. The function can access these symbols and values through the rest parameter, if there is one. It is common for a function to check only for certain keywords, and pass its rest parameter to another function using **lexpr-funcall**; then that function checks for the keywords that concern it.

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

13.2 Examples of Simple Lambda Lists

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

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

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

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

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

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

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

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

This illustrates that the parameters can be matched in either order.

`(x &optional y &rest z &key a b)`

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

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

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

`(&rest z &key a b c &allow-other-keys)`

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

13.3 Specifying Default Forms in Lambda Lists

If not specified, the *default-form* for each optional or keyword 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 and keyword parameters can have default forms; required parameters are never defaulted, and rest parameters always default to **nil**. For example:

`(a &optional (b 3))`

The default-form for **b** is **3**. **a** is a required parameter, and so it doesn't have a default form.

`(&optional (a 'foo) &rest d &key b (c (symeval a)))`

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

Occasionally it is important to know whether or not a certain optional or keyword parameter was defaulted. You cannot tell from just examining its value, since if the

value is the default value, there is no way to tell whether the caller passed that value explicitly, or whether the caller did not pass any value and the parameter was defaulted. The way to tell for sure is to put a third element into the list: the third element should be a variable (a symbol), and that variable is bound to **nil** if the parameter was not passed by the caller (and so was defaulted), or **t** if the parameter was passed. The new variable is called a *supplied-p* variable; it is bound to **t** if the parameter is supplied.

For example:

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

`(&key a (b (1+ a) c))`
: This is the same as the example above, except that it demonstrates use of a supplied-p variable for a keyword parameter. This example also shows the default value of one keyword parameter depending on a previous keyword parameter.

13.4 Specifying a Keyword Parameter's Symbol in Lambda Lists

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

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

`(&key ((:base base-value)))`
: This allows a keyword that the caller knows under the name **:base**, without making the parameter shadow the value of **base**, which is used for printing numbers.

`(&key ((:base base-value) 10 base-supplied))`
: When the **base** keyword is supplied, the default value of 10 is ignored and **base-supplied** is bound to **t**. If the keyword is not supplied, **base-value** is bound to 10 and **base-supplied** is bound to **nil**.

13.5 Specifying Aux-variables in Lambda Lists

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

To include such symbols, put them after any parameters, preceded by the &-keyword **&aux**. For example:

```
(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.

13.6 Safety of &rest Arguments

It is important to realize that the list of arguments to which a rest-parameter is bound is set up in whatever way is most efficiently implemented, rather than in the way that is most convenient for the function receiving the arguments. It is not guaranteed to be a "real" list. Sometimes the rest-args list is stored in the function-calling stack, and loses its validity when the function returns. If a rest-argument is to be returned or made part of permanent list-structure, it must first be copied, as you must always assume that it is one of these special lists. See the function **copylist**, page 50.

The system does not detect the error of omitting to copy a rest-argument; you simply find that you have a value that seems to change behind your back. At other times the rest-args list is an argument that was given to **apply**; therefore it is not safe to **rplaca** this list, as you might modify permanent data structure. An attempt to **rplacd** a rest-args list is unsafe in this case, while in the first case it signals an error, since lists in the stack are impossible to **rplacd**.

14. Some Functions and Special Forms

14.1 Function for Evaluation

eval *form* &optional *env* *Function*

Evaluates *form*, and returns the result. Example:

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

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

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**.

The actual name of the compiled code for **eval** is **"si:*eval"** because use of the **evalhook** feature binds the function cell of **eval**.

env defaults to the null lexical environment.

14.2 Functions for Function Invocation

apply *f arglist* *Function*

Applies the function *f* to the list of arguments *arglist*. *arglist* should be a list; *f* can be any function, but it cannot be a special form or a macro.

Examples:

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

Of course, *arglist* can be **nil**. Note: Unlike Maclisp, **apply** never takes a third argument; there are no "binding context pointers" in Symbolics-Lisp.

See the function **funcall**, page 160.

funcall *f* &rest *args* *Function*

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

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

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

Note: The Maclisp functions **subrcall**, **lsubrcall**, and **arraycall** are not needed on the Symbolics 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.

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

This is similar to a cross between **apply** and **funcall**.

(lexpr-funcall *f a1 a2 ... an l***)** applies the function *f* to the arguments *a1* through *an* followed by the elements of the list *l*. Note that since it treats its last argument specially, **lexpr-funcall** requires at least two arguments. Examples:

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

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

lexpr-funcall with two arguments does the same thing as **apply**.

send *object message-name* &rest *arguments* *Function*

Sends the message named *message-name* to the *object*. *arguments* are the arguments passed. **send** does exactly the same thing as **funcall**. For stylistic reasons, it is preferable to use **send** instead of **funcall** when sending messages because **send** clarifies the programmer's intent.

lexpr-send *object message-name* &rest *arguments* *Function*

Sends the message named *message-name* to the *object*. *arguments* are the arguments passed, except that the last element of *arguments* should be a list, and all the elements of that list are passed as arguments. Example:

```
(send some-window :set-edges 10 10 40 40)
```

does the same thing as

```
(setq new-edges '(10 10 40 40))
(lexpr-send some-window :set-edges new-edges)
```

lexpr-send is to **send** as **lexpr-funcall** is to **funcall.**

call *function* &rest *argument-specifications* *Function*

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

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

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

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

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

14.3 Functions and Special Forms for Constant Values

quote *object* *Special Form*

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

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

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

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

is converted by **read** into

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

function *f* *Special Form*

This means different things depending on whether *f* is a function or the name of a function. (Note that in neither case is *f* evaluated.) The name of a function is a symbol or a function-spec list. See the section "Function Specs", page 297. A function is typically a list whose car is the symbol **lambda**; however there are several other kinds of functions available. See the section "Kinds of Functions", page 303.

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)
```

The compiler and interpreter cannot tell that the first argument is going to be used as a function; for all they know, **mapc** treats its first argument as a piece of list structure, asking for its **car** and **cdr** and so forth. The compiler cannot compile the function; it must pass the lambda-expression unmodified. This means that the function does not get compiled, which makes it execute more slowly than it might otherwise. The interpreter cannot make references to free lexical variables work by making a lexical closure; it must pass the lambda-expression unmodified.

The **function** special form is the way to say that a lambda-expression represents a function rather than a piece of list structure. You just use the symbol **function** instead of **quote**:

```
(mapc (function (lambda (x) (car x))) some-list)
```

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

```
(mapc #'(lambda (x) (car x)) some-list)
```

If *f* is not a function but the name of a function (typically a symbol, but in general any kind of function spec), then **function** returns the definition of *f*; it is like **fdefinition** except that it is a special form instead of a function, and so

```
(function fred)
```

is like

```
(fdefinition 'fred)
```

which is like

```
(fsymeval 'fred)
```

since **fred** is a symbol.

If *f* is the name of a local function defined with **flet** or **labels**, then **(function** *f*) produces a lexical closure of *f*, just like **(function (lambda...))**.

Another way of explaining **function** is that it causes *f* to be treated the same way as it would as the car of a form. Evaluating the form (*f arg1 arg2...*) uses the function definition of *f* if it is a symbol, and otherwise expects *f* to be a list that is a lambda-expression. Note that the car of a form cannot be a nonsymbol function spec, to avoid difficult-to-read code. This can be written as:

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

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

```
(setq x 'car)
or
(setq x #'car)
```

The former causes the value of **x** to be the symbol **car**, whereas the latter causes the value of **x** to be the function object found in the function cell of **car**. When the time comes to call the function (the program does **(funcall x ...)**), either of these two work because if you use a symbol as a function, the contents of the symbol's function cell is used as the function. The former case is a bit slower, because the function call has to indirect through the symbol, but it allows the function to be redefined, traced, or advised. (See the special form **trace** in *Program Development Utilities*. See the special form **advise** in *Program Development Utilities*.) The latter case, while faster, picks up the function definition out of the symbol **car** and does not see any later changes to it.

lambda *lambda-list body...* *Special Form*

Provided, as a convenience, to obviate the need for using the **function** special form when the latter is used to name an anonymous (lambda) function. When **lambda** is used as a special form, it is treated by the evaluator and compiler identically to the way it would have been treated if it appeared as the operand of a **function** special form. For example, the following two forms are equivalent:

```
(my-mapping-function (lambda (x) (+ x 2)) list)

(my-mapping-function (function (lambda (x) (+ x 2))) list)
```

Note that the form immediately above is usually written as:

```
(my-mapping-function #'(lambda (x) (+ x 2)) list)
```

The first form uses **lambda** as a special form; the latter two do not use the **lambda** special form, but rather, use **lambda** to name an anonymous function.

Using **lambda** as a special form is incompatible with Common Lisp.

false *Function*

Takes no arguments and returns **nil**.

true *Function*

Takes no arguments and returns **t**.

ignore &rest *ignore* *Function*

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

ignore is also used to suppress compiler warnings for ignored arguments. For example:

```
(defun foo (x y)
  (ignore y)
  (sin x))
```

comment *Special Form*

Ignores its form and returns the symbol **comment**. Example:

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

Usually it is preferable to comment code using the semicolon-macro feature of the standard input syntax. This allows you to add comments to your code that are ignored by the Lisp reader. Example:

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

A problem with such comments is that they are discarded when the form is read into Lisp. If the function is read into Lisp, modified, and printed out again, the comment is 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.

14.4 Special Forms for Sequencing

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*

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 that must be computed *before* the side effects happen. Example:

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

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

prog1 never returns multiple values. See the special form **multiple-value-prog1**, page 169.

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.

14.5 Functions for Compatibility with Maclisp Lexprs

arg *x* *Function*

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

(arg *i***)**, when evaluated during the application of a lexpr, gives the value of the *i*'th argument to the lexpr. *i* must be an integer 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.
```

arg exists only for compatibility with Maclisp lexprs. To write functions that can accept variable numbers of arguments, use the **&optional** and **&rest** keywords. See the section "Evaluating a Function Form", page 151.

setarg *i x* *Function*

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

setarg exists only for compatibility with Maclisp lexprs. To write functions that can accept variable numbers of arguments, use the **&optional** and **&rest** keywords. See the section "Evaluating a Function Form", page 151.

listify *n* *Function*

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) ))
```

listify exists only for compatibility with Maclisp lexprs. To write functions that can accept variable numbers of arguments, use the **&optional** and **&rest** keywords. See the section "Evaluating a Function Form", page 151.

15. Multiple Values

The Symbolics 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.

15.1 Primitive for Producing Multiple Values

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

values &rest *args* *Function*

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

values-list *list* *Function*

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

15.2 Special Forms for Receiving Multiple Values

The special forms for receiving multiple values are **multiple-value**, **multiple-value-bind**, **multiple-value-list**, **multiple-value-call**, and **multiple-value-prog1**. These consist of a form and an indication of where to put the values returned by that form. With the first two of these, the caller requests a certain number of returned values. If fewer values are returned than the number requested, then it is exactly as if the rest of the values were present and had the value **nil**. If too many values are returned, the rest of the values are ignored. This has the advantage that you do not have to pay attention to extra values if you don't care about them, but it has the disadvantage that error-checking similar to that done for function calling is not present.

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

Used for calling a function that 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** is set to **t**, otherwise it is set to **nil**.

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

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

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

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

Evaluates *form* and returns a list of the values it returned. This is useful for when you do not know how many values to expect. Example:

```
(setq a (multiple-value-list (intern "goo")))
a => (goo nil)
```

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

multiple-value-call *function body...* *Special Form*

First evaluates *function* to obtain a function. It then evaluates all the forms in *body*, gathering together all the values of the forms (not just one value from each). It gives these values as arguments to the function and returns whatever the function returns.

For example, suppose the function **frob** returns the first two elements of a list of numbers:

```
(multiple-value-call #'+ (frob '(1 2 3)) (frob '(4 5 6)))
   <=> (+ 1 2 4 5) => 12.
```

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

Like **prog1**, except that if its first form returns multiple values, **multiple-value-prog1** returns those values. In certain cases, **prog1** is more efficient than **multiple-value-prog1**, which is why both special forms exist.

15.3 Passing-back of Multiple Values

Due to the syntactic structure of Lisp, it is often the case that the value of a certain form is the value of a subform of it. For example, the value of a **cond** is the value of the last form in the selected clause. In most such cases, if the subform produces multiple values, the original form also produces all of those values. This *passing-back* of multiple values of course has no effect unless eventually one of the special forms for receiving multiple values is reached. The exact rule governing passing-back of multiple values is as follows:

If *X* is a form, and *Y* is a subform of *X*, then if the value of *Y* is unconditionally returned as the value of *X*, with no intervening computation, then all the multiple values returned by *Y* are returned by *X*. In all other cases, multiple values or only single values can be returned at the discretion of the implementation; users should not depend on whatever way it happens to work, as it might change in the future or in other implementations. The reason we do not guarantee nontransmission of multiple values is because such a guarantee is not very useful and the efficiency cost of enforcing it is 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 that realized that the **setq**ing of the variable was unnecessary.

Note that use of a form as an argument to a function never receives multiple values from that form. That is, if the form **(foo (bar))** is evaluated and the call to **bar** returns many values, **foo** is still only called on one argument (namely, the first value returned), rather than called on all the values returned. We choose not to generate several separate arguments from the several values, because this makes the source code obscure; it is not 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 special forms discussed in another section. See the section "Special Forms for Receiving Multiple Values", page 167.

15.4 Interaction of Some Common Special Forms with Multiple Values

The interaction of special forms with multiple values can be deduced from the rule mentioned in another section: See the section "Passing-back of Multiple Values", page 169. Note well that when it says that multiple values are not returned, it

really means that they might or might not be returned, and you should not write any programs that depend on which way it works.

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

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

- **progn** passes back multiple values from its last form. **progv** and **progw** do so also. **prog1** and **prog2**, however, do not pass back multiple values (though **multiple-value-prog1** does).

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

- **cond** passes back multiple values from the last form in the selected clause, but not if the clause is only one long (that is, the returned value is the value of the predicate) since the return is conditional. This rule applies even to the last clause, where the return is not really conditional (the implementation is allowed to pass or not to pass multiple values in this case, and so you should not depend on what it does). **t** should be used as the predicate of the last clause if multiple values are desired, to make it clear to the compiler (and any human readers of the code!) that the return is not conditional.

- The variants of **cond** such as **if**, **when**, **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**. **prog** returns all of the values produced by the subform of **return**. (If a **prog** drops off the end it just returns a single **nil**.)

- **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** passes back multiple values from the last form in its body when it exits normally.

- The obsolete special form ***catch** does not pass back multiple values from the last form in its body, because it is defined to return its own second value to tell you whether the ***catch** form was exited normally or abnormally. This is sometimes inconvenient when you want to propagate back multiple values but you also want to wrap a ***catch** around some forms. Usually people get around this problem by using **catch** or by enclosing the ***catch** in a **prog** and using **return** to pass out the multiple values, **return**ing through the ***catch**.

PART V.

Flow of Control

16. Introduction to 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 can 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 section. See the section "The **loop** Iteration Macro", page 205.) There is also a very general construct to allow the traditional "goto" control structure, called **prog**.

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

There are some *nonlocal exit* control structures, analogous to the *leave*, *exit*, and *escape* constructs in many modern languages.

The general ones are **catch** and **throw**; there is also **return** and its variants, used for exiting the iteration constructs **do**, **loop**, and **prog**.

Symbolics-Lisp also provides a coroutine capability and a multiple-process facility. See the section "Processes" in *Internals, Processes, and Storage Management*. There is also a facility for generic function calling using message passing. See the section "Flavors", page 415.

17. Conditionals

if *Special Form*

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**.

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 ...)
      ... )
```

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

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

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

cond-every *Special Form*

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*

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 is not 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*

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

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

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

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

when *test body...* *Macro*

The forms in *body* are evaluated when *test* returns non-**null**. In that case, it returns the value(s) of the last form evaluated. When *test* returns **nil**, **when** returns **nil**.

```
(when (eq 1 1) (setq a b) "foo") =>
"foo"
(when (eq 1 2) (setq a b) "foo") =>
NIL
```

When *body* is empty, **when** always returns **nil**.

unless *test body...* *Macro*

The forms in *body* are evaluated when *test* returns **nil**. It returns the value of the last form evaluated. When *test* returns something other than **nil**, **unless** returns **nil**.

```
(unless (eq 1 1) (setq a b) "foo") =>
NIL
(unless (eq 1 2) (setq a b) "foo") =>
"foo"
```

When *body* is empty, **unless** always returns **nil**.

selectq *Special Form*

A conditional that chooses one of its clauses to execute by comparing the value of a form against various constants, which are typically keyword symbols. Its form is as follows:

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

The first thing **selectq** does is to evaluate *key-form*; call the resulting value *key*. Then **selectq** considers each of the clauses in turn. If *key* matches the

clause's *test*, the consequents of this clause are evaluated, and **selectq** returns the value of the last consequent. If there are no matches, **selectq** returns **nil**.

A *test* can be any of the following:

A symbol	If the *key* is **eq** to the symbol, it matches.
A number	If the *key* is **eq** to the number, it matches. Only small numbers (*integers*) work.
A list	If the *key* is **eq** to one of the elements of the list, then it matches. The elements of the list should be symbols or integers.
t or **otherwise**	The symbols **t** and **otherwise** are special keywords that match anything. Either symbol can be used; **t** is mainly for compatibility with Maclisp's **caseq** construct. To be useful, this should be the last clause in the **selectq**.

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

```
(selectq x
  (foo (do-this))
  (bar (do-that))
  ((baz quux mum) (do-the-other-thing))
  (otherwise (ferror nil "Never heard of ~S" x)))
```

is equivalent to:

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

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

select *Special Form*

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*

The same as **select**, except that you get to specify the function used for the comparison instead of **eq**. For example:

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

is equivalent to:

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

typecase *form clauses...* *Special Form*

Selects various forms to be evaluated depending on the type of some object. It is something like **select**. A **typecase** form looks like:

```
(typecase form
   (types consequent consequent ...)
   (types consequent consequent ...)
   ...
   )
```

form is evaluated, producing an object. **typecase** examines each clause in sequence. *types* in each clause is either a single type (if it is a symbol) or a list of types. If the object is of that type, or of one of those types, then the consequents are evaluated and the result of the last one is returned. Otherwise, **typecase** moves on to the next clause. As a special case, *types* can be **otherwise**; in this case, the clause is always executed, so this should be used only in the last clause. For an object to be of a given type means that if **typep** is applied to the object and the type, it returns **t**. That is, a type is something meaningful as a second argument to **typep**. Example:

```
(defun tell-about-car (x)
  (typecase (car x)
     (:fixnum "The car is a number.")
     ((:string :symbol) "The car is a name.")
     (otherwise "I don't know.")))

(tell-about-car '(1 a)) => "The car is a number."
(tell-about-car '(a 1)) => "The car is a name."
(tell-about-car '("word" "more")) => "The car is a name."
(tell-about-car '(1.0)) =>
"I don't know."
```

dispatch *Special Form*

(dispatch *byte-specifier number clauses...***)** is the same as **select** (not **selectq**), but the key is obtained by evaluating **(ldb** *byte-specifier number***)**. *byte-specifier* and *number* are both evaluated. See the section "Byte Manipulation Functions", page 115. Byte specifiers and **ldb** are explained in that section. Example:

```
(princ (dispatch 0202 cat-type
          (0 "Siamese.")
          (1 "Persian.")
          (2 "Alley.")
          (3 (ferror nil
                     "~S is not a known cat type."
                     cat-type))))
```

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

selectq-every *Special Form*

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*

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** work correctly as long as they do not use the symbol **otherwise** as the key.

18. Blocks and Exits

block and **return-from** are the primitive special forms for premature exit from a piece of code. **block** defines a place that can be exited, and **return-from** transfers control to such an exit.

block and **return-from** differ from **catch** and **throw** in their scoping rules. **block** and **return-from** have lexical scope; **catch** and **throw** have dynamic scope.

See the section "Nonlocal Exits", page 197.

block *name form...* *Special Form*

Evaluates each *form* in sequence and normally returns the (possibly multiple) values of the last *form*. However, **(return-from** *name value***)** or one of its variants (a **return** or **return-list** form) might be evaluated during the evaluation of some *form*. In that case, the (possibly multiple) values that result from evaluating *value* are immediately returned from the innermost block that has the same name and that lexically contains the **return-from** form. Any remaining forms in that block are not evaluated.

name is not evaluated. It must be a symbol.

The scope of *name* is lexical. That is, the **return-from** form must be inside the block itself (or inside a block that that block lexically contains), not inside a function called from the block.

do, **prog**, and their variants establish implicit blocks around their bodies; you can use **return-from** to exit from them. These blocks are named **nil** unless you specify a name explicitly.

For example, the following two forms are equivalent:

```
(cond ((predicate x)
       (do-one-thing))
      (t
       (format t "The value of X is ~S~%" x)
       (do-the-other-thing)
       (do-something-else-too)))

(block deal-with-x
  (when (predicate x)
    (return-from deal-with-x (do-one-thing)))
  (format t "The value of X is ~S~%" x)
  (do-the-other-thing)
  (do-something-else-too))
```

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

Exits from a **block** or a construct like **do** or **prog** that establishes an implicit block around its body.

The *value* subforms are optional. Any *value* forms are evaluated, and the resulting values (possibly multiple, possibly none) are returned from the innermost block that has the same name and that lexically contains the **return-from** form. The returned values depend on how many *value* subforms are provided:

value subforms	*Values returned from block*
None	None
1	All values that result from evaluating the *value* subform
>1	One value from each *value* subform

This means that the following two forms are equivalent:

(return-from *name form1 form2 form3***)**

(return-from *name* **(values** *form1 form2 form3***))**

The latter form is the preferred way to return multiple values, for the sake of both clarity and compatibility with Common Lisp.

name is not evaluated. It must be a symbol.

The scope of *name* is lexical. That is, the **return-from** form must be inside the block itself (or inside a block that that block lexically contains), not inside a function called from the block.

When a construct like **do** or an unnamed **prog** establishes an implicit block, its name is **nil**. You can use either **(return-from nil** *value...***)** or the equivalent **(return** *value...***)** to exit from such a construct.

The **return-from** form is unusual: It never returns a value itself, in the conventional sense. It is not useful to write **(setq a (return-from** *name* **3))**, because when the **return-from** form is evaluated, the containing block is immediately exited, and the **setq** never happens.

For an explanation of named **do**s and **prog**s: See the special form **do-named**, page 192.

Following is an example, returning a single value from an implicit block named **nil**:

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

Following is another example, returning multiple values. The function below is like **assq**, but it returns an additional value, the index in the table of the entry it found:

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

return *value...* *Special Form*

Can be used to exit from a construct like **do** or an unnamed **prog** that establishes an implicit block around its body. In this case the name of the block is **nil**, and **(return** *value...***)** is the same as **(return-from nil** *value...***)**. See the special form **return-from**, page 184.

In addition, **break** recognizes the typed-in form **(return** *value***)** specially. If this form is typed at a **break**, *value* is evaluated and returned as the value of **break**. Only the result of the first *value* form is returned, but if this form itself returns multiple values, they are all returned as the value of **break**. That is, **(return 'foo 'bar)** returns only **foo**, but **(return (values 'foo 'bar))** returns both **foo** and **bar**. See the special form **break** in *User's Guide to Symbolics Computers*.

It is valid to write simply **(return)**, which exits from the block without returning any values. **(return)** inside a break loop causes **break** to return **nil**.

If not specially recognized by **break** and not inside a block, **return** signals an error.

return-list *list* *Function*

An obsolete function supported for compatibility with earlier releases. It is like **return** except that the block returns all of the elements of *list* as multiple values. This means that the following two forms are equivalent:

(return-list *list***)**

(return (values-list *list***))**

The latter form is the preferred way to return list elements as multiple values from a block named **nil**. To direct the returned values to a named block, use:

(return-from *name* **(values-list** *list***)).**

19. Transfer of Control

tagbody and **go** are the primitive special forms for unstructured transfer of control. **tagbody** defines places that can receive a transfer of control, and **go** transfers control to such a place.

tagbody *tag-or-statement...* *Special Form*

The body of a **tagbody** form is a series of *tags* or *statements*. A *tag* is a symbol; a *statement* is a list. **tagbody** processes each element of the body in sequence. It ignores *tags* and evaluates *statements*, discarding the results. If it reaches the end of the body, it returns **nil**.

If a **(go** *tag***)** form is evaluated during evaluation of a *statement*, **tagbody** searches its body and the bodies of any **tagbody** forms that lexically contain it. Control is transferred to the innermost *tag* that is **eq** to the *tag* in the **go** form. Processing continues with the next *tag* or *statement* that follows the *tag* to which control is transferred.

The scope of the *tags* is lexical. That is, the **go** form must be inside the **tagbody** construct itself (or inside a **tagbody** form that that **tagbody** lexically contains), not inside a function called from the **tagbody**.

do, **prog**, and their variants use implicit **tagbody** constructs. You can provide *tags* within their bodies and use **go** forms to transfer control to the *tags*.

For example, the following two forms are equivalent:

```
(dotimes (i n) (print i))

(let ((i 0))
  (when (plusp n)
    (tagbody
      loop
      (print i)
      (setq i (1+ i))
      (when (< i n) (go loop)))))
```

go *tag* *Special Form*

Transfers control within a **tagbody** form or a construct like **do** or **prog** that uses an implicit **tagbody**.

The *tag* must be a symbol. It is not evaluated. **go** transfers control to the tag in the body of the **tagbody** that is **eq** to the *tag* in the **go** form. If the body has no such tag, the bodies of any lexically containing **tagbody** forms are examined as well. If no tag is found, an error is signalled.

The scope of *tag* is lexical. That is, the **go** form must be inside the

tagbody construct itself (or inside a **tagbody** form that that **tagbody** lexically contains), not inside a function called from the **tagbody**.

Example:

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

20. Iteration

do *Special Form*

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

do comes in two varieties.

The more general, so-called "new-style" **do** looks like:

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

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

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

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

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

Note that the *end-test* gets evaluated before the first time the body is evaluated. **do** first initializes the variables from the *init* forms, then it checks the *end-test*, then it processes the body, then it deals with the *repeat* forms, then it tests the *end-test* again, and so on. If the **end-test** returns a non-**nil** value the first time, then the body is never processed.

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

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

If a **return** special form is evaluated inside the body of a **do**, then the **do** immediately stops, unbinds its variables, and returns the values given to **return**. See the special form **return**, page 185. **return** and its variants are explained in more detail in that section. **go** special forms and **prog**-tags can also be used inside the body of a **do** and they mean the same thing that they do inside **prog** forms, but we discourage their use since they complicate the control structure in a hard-to-understand way.

The other, so-called "old-style" **do** looks like:

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

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

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

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

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

return forms are often useful to do simple searches:

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

Examples of the new form of **do**:

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

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

The following construction exploits parallel assignment to index variables:

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

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

The following example is like `(maplist 'f x y)`. (See the section "Mapping", page 201.)

```
(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.
```

For information about a general iteration facility based on a keyword syntax rather than a list-structure syntax: See the section "The **loop** Iteration Macro", page 205.

do* *Special Form*

Just like **do**, except that the variable clauses are evaluated sequentially rather than in parallel. When a **do** starts, all the initialization forms are

evaluated before any of the variables are set to the results; when a **do*** starts, the first initialization form is evaluated, then the first variable is set to the result, then the second initialization form is evaluated, and so on. The stepping forms work analogously.

do-named *Special Form*

Sometimes one **do** is contained inside the body of an outer **do**. The **return** function always returns from the innermost surrounding **do**, but sometimes you want to return from an outer **do** while within an inner **do**. You can do this by giving the outer **do** a name. You use **do-named** instead of **do** for the outer **do**, and use **return-from**, specifying that name, to return from the **do-named**.

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

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

If the symbol **t** is used as the name, it is made "invisible" to **returns**; that is, **returns** inside that **do-named** return to the next outermost level whose name is not **t**. **(return-from t ...)** returns from a **do-named** named **t**. You can also make a **do-named** invisible to **returns** by including immediately inside it the form **(declare (invisible-block t))**. This feature is not intended to be used by user-written code; it is for macros to expand into.

If the symbol **nil** is used as the name, it is as if this were a regular **do**. Not having a name is the same as being named **nil**.

progs and **loop**s can have names just as **do**s can. Since the same functions are used to return from all of these forms, all of these names are in the same namespace; a **return** returns from the innermost enclosing iteration form, no matter which of these it is, and so you need to use names if you nest any of them within any other and want to return to an outer one from inside an inner one.

do*-named *Special Form*

Just like **do-named**, except that the variable clauses are evaluated sequentially, rather than in parallel. See the special form **do***, page 191.

dotimes *(index count) body...* *Special Form*

A convenient abbreviation for the most common integer iteration. **dotimes** performs *body* the number of times given by the value of *count*, with *index* bound to **0**, **1**, and so forth on successive iterations. Example:

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

is equivalent to:

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

except that the name **count** is not used. Note that **i** takes on values starting at 0 rather than 1, and that it stops before taking the value **(// m n)** rather than after. You can use **return** and **go** and **prog**-tags inside the body, as with **do**. **dotimes** forms return **nil** unless returned from explicitly with **return**. For example:

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

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

dolist *(item list) body...* *Special Form*

A convenient abbreviation for the most common list iteration. **dolist** performs *body* once for each element in the list that 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*

Aids in writing functions that take keyword arguments in the standard fashion. You can also use the **&key** lambda-list keyword to create functions that take keyword arguments. **&key** is preferred and is substantially more efficient; **keyword-extract** is generally considered to be obsolete. See the section "Evaluating a Function Form", page 151.

The form:

```
(keyword-extract key-list iteration-var
        keywords flags other-clauses...)
```

parses the keywords out into local variables of the function. *key-list* is a form that evaluates to the list of keyword arguments; it is generally the function's **&rest** argument. *iteration-var* is a variable used to iterate over the list; sometimes *other-clauses* uses the form:

```
(car (setq iteration-var (cdr iteration-var)))
```

to extract the next element of the list. (Note that this is not the same as **pop**, because it does the **car** after the **cdr**, not before.)

keywords defines the symbols that are keywords to be followed by an argument. Each element of *keywords* is either the name of a local variable that receives the argument and is also the keyword, or a list of the keyword and the variable, for use when they are different or the keyword is not to go in the keyword package. Thus, if *keywords* is **(a (b c) d)** then the keywords recognized are **:a**, **b**, and **:d**. If **:a** is specified its argument is stored into **a**. If **:d** is specified its argument is stored into **d**. If **b** is specified, its argument is stored into **c**.

Note that **keyword-extract** does not bind these local variables; it assumes you have done that somewhere else in the code that contains the **keyword-extract** form.

flags defines the symbols that are keywords not followed by an argument. If a flag is seen its corresponding variable is set to **t**. (You are assumed to have initialized it to **nil** when you bound it with **let** or **&aux**.) As in *keywords*, an element of *flags* can be either a variable from which the keyword is deduced, or a list of the keyword and the variable.

If there are any *other-clauses*, they are **selectq** clauses selecting on the keyword being processed. These clauses are for handling any keywords that are not handled by the *keywords* and *flags* elements. These can be used to do special processing of certain keywords for which simply storing the argument into a variable is not good enough. Unless the *other-clauses* include an **otherwise** (or **t** clause, after them there is an **otherwise** clause to complain about any unhandled keywords found in *key-list*. If you write your own **otherwise** clause, it is up to you to take care of any unhandled keywords.

prog *Special Form*

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 can 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 integer 5, and that of **d** is the symbol **zz**. The binding and initialization of the variables is done in *parallel*; that is, all the initial values are computed before any of the variables are changed. **prog*** is the same as **prog** except that this initialization is sequential rather than parallel.

The part of a **prog** after the variable list is called the *body*. Each element of the body is either a symbol, in which case it is called a *tag*, or anything else (almost always a list), in which case it is called a *statement*.

After **prog** binds the variables, it processes each form in its body sequentially. *tags* are skipped over. *statements* are evaluated, and their returned values discarded. If the end of the body is reached, the **prog** returns **nil**. However, two special forms can 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.

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**.

See the function **do**, page 189. That uses a body similar to **prog**. The **do**, **catch**, and **throw** special forms are included as an attempt to encourage goto-less programming style, which often leads to more readable, more easily maintained code. You should use these forms instead of **prog** wherever reasonable.

If the first subform of a **prog** is a non-**nil** symbol (rather than a variable

list), it is the name of the **prog**, and **return-from** can be used to return from it. See the special form **do-named**, page 192. Example:

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

prog, **do**, and their variants are effectively constructed out of **let**, **block**, and **tagbody** forms. **prog** could have been defined as the following macro (except for processing of local **declare**, which has been omitted for clarity):

```
(defmacro prog (&rest x)
  (let ((block-name (and (symbolp (car x))
                         (neq (car x) nil)
                         (pop x)))
        (variables (car x))
        (tagbody (cdr x)))
    (if block-name
        `(block ,block-name
           (block nil
             (let ,variables
               (tagbody ,@tagbody))))
        `(block nil
           (let ,variables
             (tagbody ,@tagbody))))))
```

A variant of **defun** that incorporates a **prog** into the function body is described in another section: See the macro **defunp**, page 302.

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**.

21. Nonlocal Exits

catch and **throw** are special forms used for nonlocal exits. **catch** evaluates forms; if a **throw** occurs during the evaluation, **catch** immediately returns (possibly multiple) values specified by **throw**.

catch and **throw** differ from **block** and **tagbody** in their scoping rules. **catch** and **throw** have dynamic scope; **block** and **tagbody** have lexical scope. See the section "Blocks and Exits", page 183.

***catch** and ***throw** are supported for compatibility with Maclisp. **catch** can be used with ***throw**, and ***catch** can be used with **throw**. If control exits normally, the returned values depend on whether **catch** or ***catch** is used. If control exits abnormally, the returned values depend on whether **throw** or ***throw** is used.

The old Maclisp **catch** and **throw** macros are not supported.

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

Used with **throw** for nonlocal exits. **catch** first evaluates *tag* to obtain an object that is the "tag" of the catch. Then the *body* forms are evaluated in sequence, and **catch** returns the (possibly multiple) values of the last form in the body.

However, a **throw** or ***throw** form might be evaluated during the evaluation of one of the forms in *body*. In that case, if the throw "tag" is **eq** to the catch "tag" and if this **catch** is the innermost **catch** with that tag, the evaluation of the body is immediately aborted, and **catch** returns values specified by the **throw** or ***throw** form.

If the **catch** exits abnormally because of a **throw** form, it returns the (possibly multiple) values that result from evaluating **throw**'s second subform. If the **catch** exits abnormally because of a ***throw** form, it returns two values: the first is the result of evaluating ***throw**'s second subform, and the second is the result of evaluating ***throw**'s first subform (the tag thrown to).

(catch 'foo *form*) catches a **(throw 'foo** *form*) but not a **(throw 'bar** *form*). It is an error if **throw** is done when no suitable **catch** exists.

The scope of the *tags* is dynamic. That is, the **throw** does not have to be lexically within the **catch** form; it is possible to throw out of a function that is called from inside a **catch** form.

Example:

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

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

throw *tag form* *Special Form*

Used with **catch** to make nonlocal exits. It first evaluates *tag* to obtain an object that is the "tag" of the throw. It next evaluates *form* and saves the (possibly multiple) values. It then finds the innermost **catch** or ***catch** whose "tag" is **eq** to the "tag" that results from evaluating *tag*. It causes the **catch** or ***catch** to abort the evaluation of its body forms and to return all values that result from evaluating *form*. In the process, dynamic variable bindings are undone back to the point of the **catch**, and any **unwind-protect** cleanup forms are executed. An error is signalled if no suitable **catch** is found.

The scope of the *tags* is dynamic. That is, the **throw** does not have to be lexically within the **catch** form; it is possible to throw out of a function that is called from inside a **catch** form.

The value of *tag* cannot be the symbol **sys:unwind-protect-tag**; that is reserved for internal use.

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

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

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

The nonlocal exit facility of Lisp creates a situation in which the above code does not work. However, if **hairy-function** should do a **throw** to a **catch** that is outside of the **progn** form, **(turn-off-water-faucet)** is never evaluated (and the faucet is presumably left running). This is particularly likely if **hairy-function** gets an error and the user tells the Debugger to give up and abort the computation.

In order to allow the above program to work, it can be rewritten using **unwind-protect** as follows:

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

If **hairy-function** does a **throw** that attempts to quit out of the evaluation of the **unwind-protect**, the **(turn-off-water-faucet)** form is evaluated in between the time of the **throw** and the time at which the **catch** returns. If the **progn** returns normally, then the **(turn-off-water-faucet)** is evaluated, and the **unwind-protect** returns the result of the **progn**.

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

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

protected-form is evaluated, and when it returns or when it attempts to quit out of the **unwind-protect**, the *cleanup-forms* are evaluated. **unwind-protect** catches exits caused by **return-from** or **go** as well as those caused by **throw**. The value of the **unwind-protect** is the value of *protected-form*. Multiple values returned by the *protected-form* are propagated back through the **unwind-protect**.

The cleanup forms are run in the variable-binding environment that you would expect: that is, variables bound outside the scope of the **unwind-protect** special form can be accessed, but variables bound inside the *protected-form* cannot be. In other words, the stack is unwound to the point just outside the *protected-form*, then the cleanup handler is run, and then the stack is unwound some more.

unwind-protect-case (&optional *aborted-p-var*) *body-form* &rest *cleanup-clauses* *Macro*

body-form is executed inside an unwind-protect form. The cleanup forms of the unwind-protect are generated from *cleanup-clauses*. Each cleanup-clause is considered in order of appearance and has the form *(keyword forms ...)*. *keyword* can be **:normal**, **:abort** or **:always**. The forms in a **:normal** clause are executed only if *body-form* finished normally. The forms in an **:abort** clause are executed only if *body-form* exited before completion. The forms in an **:always** clause are always executed. The values returned are the values of *body-form*, if it completed normally.

aborted-p-var, if supplied, is **t** if the *body-form* was aborted, and **nil** if it finished normally. *aborted-p-var* can be used in forms within *cleanup-clauses* as a condition for executing abort instead of normal cleanup code. It can be set within *body-form*, but should be done so with great care. It should only be set to **nil** if the remaining subforms of *body-form* do not need protecting.

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

An obsolete version of **catch** that is supported for compatibility with Maclisp. It is equivalent to **catch** except that if ***catch** exits normally, it returns only two values: the first is the result of evaluating the last form in the body, and the second is **nil**. If ***catch** exits abnormally, it returns the same values as **catch** when **catch** exits abnormally: that is, the returned values depend on whether the exit results from a **throw** or a ***throw**. See the special form **catch**, page 197.

***throw** *tag form* *Function*

An obsolete version of **throw** that is supported for compatibility with Maclisp. It is equivalent to **throw** except that it causes the **catch** or ***catch** to return only two values: the first is the result of evaluating *form*, and the second is the result of evaluating *tag* (the tag thrown to). See the special form **throw**, page 198.

22. Mapping

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

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

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

In this case *f* must be a function of *n* arguments. **mapcar** proceeds down the lists *x1, x2, ..., xn* in parallel. The first argument to *f* comes from *x1*, the second from *x2*, and so on. The iteration stops as soon as any of the lists is exhausted. (If there are no lists at all, then there are no lists to be exhausted, so the function is called repeatedly over and over. This is an obscure way to write an infinite loop. It is supported for consistency.) If you want to call a function of many arguments where one of the arguments successively takes on the values of the elements of a list and the other arguments are constant, you can use a circular list for the other arguments to **mapcar**. The function **circular-list** is useful for creating such lists. See the function **circular-list**, page 50.

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

Often *f* is a lambda-expression, rather than a symbol. For example:

```
(mapcar (function (lambda (x) (cons x something)))
        some-list)
```

The functional argument to a mapping function must be a function, acceptable to **apply** — it cannot be a macro or the name of a special form.

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

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

There are also functions (**mapatoms** and **mapatoms-all**) for mapping over all symbols in certain packages. See the section "Package Iteration", page 608.

You can also do what the mapping functions do in a different way by using **loop**. See the section "The **loop** Iteration Macro", page 205.

map *fcn* &rest *lists* *Function*

Like **maplist**, except that it does not return any useful value. This function is used when the function is being called merely for its side effects, rather than its returned values. See the function **maplist**, page 202.

mapc *fcn* &rest *lists* *Function*

Like **mapcar**, except that it does not return any useful value. This function is used when the function is being called merely for its side effects, rather than its returned values. See the function **mapcar**, page 202.

maplist *fcn* &rest *lists* *Function*

Like **mapcar**, except that the function is applied to the list and successive cdr's of that list rather than to successive elements of the list. See the function **mapcar**, page 202.

mapcar *fcn* &rest *lists* *Function*

fcn is a function that takes as many arguments as there are lists in the call to **mapcar**. For example, since **expt** takes two arguments the following use of **mapcar** is incorrect:

```
(mapcar #'expt '(1 2 3 4 5) '(43 2 1 4 2) '(2 3 2 3 2))
```

This use of **mapcar** is correct:

```
(mapcar #'expt '(1 2 3 4 5) '(43 2 1 4 2))
```

In the correct example, **mapcar** calls **expt** repeatedly, each time using successive elements of the first list as its first argument and successive elements of the second list as its second argument. Thus, **mapcar** calls **expt** with the arguments 1 and 43, 2 and 2, 3 and 1, 4 and 4, and 5 and 2 and returns a list of the five results.

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

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

In this case *f* must be a function of *n* arguments. **mapcar** proceeds down the lists *x1, x2, ..., xn* in parallel. The first argument to *f* comes from *x1*, the second argument from *x2*, and so on. The iteration stops as soon as any of the lists is exhausted. If there are no lists at all, then there are no lists to be exhausted, so the function is called repeatedly over and over.

mapcon *fcn* &rest *lists* *Function*

Like **maplist**, except that it combines the results of the function using **nconc** instead of **list**. See the function **maplist**, page 202. That is, **mapcon** could have been defined by:

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

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

mapcan *fcn* &rest *lists* *Function*

Like **mapcar**, except that it combines the results of the function using **nconc** instead of **list**. See the function **mapcar**, page 202.

23. The loop Iteration Macro

23.1 Introduction to loop

loop *x* &optional *ignore* *Macro*

A Lisp macro that provides a programmable iteration facility. The same **loop** module operates compatibly in Symbolics-Lisp, Maclisp (PDP-10 and Multics), and NIL. **loop** was inspired by the "FOR" facility of CLISP in Interlisp; however, it is not compatible and differs in several details.

The general approach is that a form introduced by the word **loop** generates a single program loop, into which a large variety of features can be incorporated. The loop consists of some initialization (*prologue*) code, a body that can be executed several times, and some exit (*epilogue*) code. Variables can be declared local to the loop. The features are concerned with loop variables, deciding when to end the iteration, putting user-written code into the loop, returning a value from the construct, and iterating a variable through various real or virtual sets of values.

The **loop** form consists of a series of clauses, each introduced by a keyword symbol. Forms appearing in or implied by the clauses of a **loop** form are classed as those to be executed as initialization code, body code, and/or exit code; within each part of the template that **loop** fills in, they are executed strictly in the order implied by the original composition. Thus, just as in ordinary Lisp code, side effects can be used, and one piece of code might depend on following another for its proper operation. This is the principal philosophical 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.

Here are some examples to illustrate the use of **loop**.

print-elements-of-list prints each element in its argument, which should be a list. It returns **nil**.

```
(defun print-elements-of-list (list-of-elements)
  (loop for element in list-of-elements
        do (print element)))
```

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 gather-alist-entries (list-of-pairs)
  (loop for pair in list-of-pairs
        collect (car pair)))
```

extract-interesting-numbers takes two arguments, which should be integers, and returns a list of all the numbers in that range (inclusive) that satisfy the predicate **interesting-p**.

```
(defun extract-interesting-numbers (start-value end-value)
  (loop for number from start-value to end-value
        when (interesting-p number) collect number))
```

find-maximum-element returns the maximum of the elements of its argument, a one-dimensional array. For Maclisp, **aref** could be a macro that turns into either **funcall** or **arraycall** depending on what is known about the type of the array.

```
(defun find-maximum-element (an-array)
  (loop for i from 0 below (array-dimension-n 1 an-array)
        maximize (aref an-array i)))
```

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 **remove** function.

```
(defun my-remove (object list)
  (loop for element in list
        unless (equal object element) collect element))
```

find-frob returns the first element of its list argument that satisfies the predicate **frobp**. If none is found, an error is generated.

```
(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)))
```

23.2 Clauses

Internally, **loop** constructs a **prog** that includes variable bindings, preiteration (initialization) code, postiteration (exit) code, the body of the iteration, and stepping of variables of iteration to their next values (which happens on every iteration after executing the body).

A *clause* consists of the keyword symbol and any Lisp forms and keywords with which it deals. For example:

```
(loop for x in l
      do (print x))
```

contains two clauses, "`for x in l`" and "`do (print x)`". Certain parts of the clause

are described as being *expressions*, such as **(print x)** in the example above. An expression can be a single Lisp form, or a series of forms implicitly collected with **progn**. An expression is terminated by the next following atom, which is taken to be a keyword. This syntax allows only the first form in an expression to be atomic, but makes misspelled keywords more easily detectable.

loop uses print-name equality to compare keywords so that **loop** forms can be written without package prefixes; in Lisp implementations that do not have packages, **eq** is used for comparison.

Bindings and iteration variable steppings can be performed either sequentially or in parallel, which affects how the stepping of one iteration variable can depend on the value of another. The syntax for distinguishing the two is described with the corresponding clauses. When a set of things is "in parallel", all of the bindings produced are performed in parallel by a single lambda binding. Subsequent bindings are performed inside that binding environment.

23.2.1 Iteration-driving Clauses

These clauses all create a *variable of iteration*, which is bound locally to the loop and takes on a new value on each successive iteration. Note that if more than one iteration-driving clause is used in the same loop, several variables are created that all step together through their values; when any of the iterations terminates, the entire loop terminates. Nested iterations are not generated; for those, you need a second **loop** form in the body of the loop. In order to not produce strange interactions, iteration-driving clauses are required to precede any clauses that produce "body" code: that is, all except those that produce prologue or epilogue code (**initially** and **finally**), bindings (**with**), the **named** clause, and the iteration termination clauses (**while** and **until**).

Clauses that drive the iteration can be arranged to perform their testing and stepping either in series or in parallel. They are by default grouped in series, which allows the stepping computation of one clause to use the just-computed values of the iteration variables of previous clauses. They can be made to step "in parallel", as is the case with the **do** special form, by "joining" the iteration clauses with the keyword **and**. The form this typically takes is something like:

```
(loop ... for x = (f) and for y = init then (g x) ...)
```

which sets **x** to **(f)** on every iteration, and binds **y** to the value of *init* for the first iteration, and on every iteration thereafter sets it to **(g x)**, where **x** still has the value from the *previous* iteration. Thus, if the calls to **f** and **g** are not order-dependent, this would be best written as:

```
(loop ... for y = init then (g x) for x = (f) ...)
```

because, as a general rule, parallel stepping has more overhead than sequential stepping. Similarly, the example:

```
(loop for sublist on some-list
      and for previous = 'undefined then sublist
          ...)
```

which is equivalent to the **do** construct:

```
(do ((sublist some-list (cdr sublist))
     (previous 'undefined sublist))
    ((null sublist) ...)
  ...)
```

in terms of stepping, would be better written as:

```
(loop for previous = 'undefined then sublist
      for sublist on some-list
          ...)
```

When iteration-driving clauses are joined with **and**, if the token following the **and** is not a keyword that introduces an iteration-driving clause, it is assumed to be the same as the keyword that introduced the most recent clause; thus, the above example showing parallel stepping could have been written as:

```
(loop for sublist on some-list
      and previous = 'undefined then sublist
          ...)
```

The order of evaluation in iteration-driving clauses is that those expressions that are only evaluated once are evaluated in order at the beginning of the form, during the variable-binding phase, while those expressions that are evaluated each time around the loop are evaluated in order in the body.

One common and simple iteration-driving clause is **repeat**:

repeat *expression*

Evaluates *expression* (during the variable-binding phase), and causes the **loop** to iterate that many times. *expression* is expected to evaluate to an integer. If *expression* evaluates to a 0 or negative result, the body code is not executed.

All remaining iteration-driving clauses are subdispatches of the keyword **for**, which is synonymous with **as**. In all of them a *variable of iteration* is specified. Note that, in general, if an iteration-driving clause implicitly supplies an endtest, the value of this iteration variable as the loop is exited (that is, when the epilogue code is run) is undefined. See the section "The Iteration Framework", page 221.

Here are all of the varieties of **for** clauses. Optional parts are enclosed in curly brackets. See the section "Data Types Recognized by **loop**", page 219. The *data-types* as used here are discussed fully in that section.

for *var* {*data-type*} **in** *expr1* {**by** *expr2*}

Iterates over each of the elements in the list *expr1*. If the **by** subclause is present, *expr2* is evaluated once on entry to the loop to supply the function to be used to fetch successive sublists, instead of **cdr**.

for *var* {*data-type*} **on** *expr1* {**by** *expr2*}
Like the previous **for** format, except that *var* is set to successive sublists of the list instead of successive elements. Note that since *var* is always a list, it is not meaningful to specify a *data-type* unless *var* is a *destructuring pattern*, as described in the section on *destructuring*. Note also that **loop** uses a **null** rather than an **atom** test to implement both this and the preceding clause.

for *var* {*data-type*} = *expr*
On each iteration, *expr* is evaluated and *var* is set to the result.

for *var* {*data-type*} = *expr1* **then** *expr2*
var is bound to *expr1* when the loop is entered, and set to *expr2* (reevaluated) at all but the first iteration. Since *expr1* is evaluated during the binding phase, it cannot reference other iteration variables set before it; for that, use the following:

for *var* {*data-type*} **first** *expr1* **then** *expr2*
Sets *var* to *expr1* on the first iteration, and to *expr2* (reevaluated) on each succeeding iteration. The evaluation of both expressions is performed *inside* of the **loop** binding environment, before the **loop** body. This allows the first value of *var* to come from the first value of some other iteration variable, allowing such constructs as:

```
(loop for term in poly
      for ans first (car term) then (gcd ans (car term))
      finally (return ans))
```

for *var* {*data-type*} **from** *expr1* {**to** *expr2*} {**by** *expr3*}
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 can be written in either order. **downto** can 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 is terminated before *expr2* is reached, rather than after. Note that the **to** variant appropriate for the direction of stepping must be used for the endtest to be formed correctly; that is, the code does not work if *expr3* is negative or 0. If no limit-specifying clause is given, then the direction of the stepping can be specified as being decreasing by using **downfrom** instead of **from**. **upfrom** can 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 can appear. See the section "Iteration Paths", page 222.

23.2.2 Bindings

The **with** keyword can be used to establish initial bindings, that is, variables that are local to the loop but are only set once, rather than on each iteration. The **with** clause looks like:

with *var1* {*data-type*} {= *expr1*}
{**and** *var2* {*data-type*} {= *expr2*}}...

If no *expr* is given, the variable is initialized to the appropriate value for its data type, usually **nil**. **with** bindings linked by **and** are performed in parallel; those not linked are performed sequentially. That is:

```
(loop with a = (foo) and b = (bar) and c
        ...)
```

binds the variables like:

```
((lambda (a b c) ...)
 (foo) (bar) nil)
```

whereas:

```
(loop with a = (foo) with b = (bar a) with c ...)
```

binds the variables like:

```
((lambda (a)
   ((lambda (b)
      ((lambda (c) ...)
       nil))
    (bar a)))
 (foo))
```

All *expr*'s in **with** clauses are evaluated in the order they are written, in lambda-expressions surrounding the generated **prog**. The **loop** expression:

```
(loop with a = xa and b = xb
      with c = xc
      for d = xd then (f d)
      and e = xe then (g e d)
      for p in xp
      with q = xq
        ...)
```

produces the following binding contour, where **t1** is a **loop**-generated temporary:

```
((lambda (a b)
   ((lambda (c)
      ((lambda (d e)
         ((lambda (p t1)
            ((lambda (q) ...)
             xq))
          nil xp))
       xd xe))
    xc))
 xa xb)
```

Because all expressions in **with** clauses are evaluated during the variable-binding phase, they are best placed near the front of the **loop** form for stylistic reasons.

For binding more than one variable with no particular initialization, one can use the construct:

with *variable-list* {*data-type-list*} {**and** ...}

as in:

```
with (i j k t1 t2) (fixnum fixnum fixnum) ...
```

A slightly shorter way of writing this is:

```
with (i j k) fixnum and (t1 t2) ...
```

These are cases of *destructuring* which **loop** handles specially. See the section "Data Types Recognized by **loop**", page 219. See the section "Destructuring", page 220.

Occasionally there are various implementational reasons for a variable *not* to be given a local type declaration. If this is necessary, the **nodeclare** clause can 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 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 causes this clause to not take effect, and cannot be diagnosed.

23.2.3 Entrance and Exit

initially *expression*
Puts *expression* into the *prologue* of the iteration. It is 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*
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 can generate code that terminates the iteration without running the epilogue code; this behavior is noted with those clauses. See the section "Aggregated Boolean Tests", page 215. This clause can be used to cause the loop to return values in a nonstandard way:

```
(loop for n in l
      sum n into the-sum
      count t into the-count
      finally (return (quotient the-sum the-count)))
```

23.2.4 Side Effects

do *expression*

doing *expression*
expression is evaluated each time through the loop, as shown in the **print-elements-of-list** example. See the function **loop**, page 205.

23.2.5 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 is 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) has been set to the accumulated result and can be used by the epilogue code. In this way, a user can accumulate and somehow pass back multiple values from a single **loop**, or use them during the loop. It is safe to reference these variables during the loop, but they should not be modified until the epilogue code of the loop is reached. For example:

```
(loop for x in list
      collect (foo x) into foo-list
      collect (bar x) into bar-list
      collect (baz x) into baz-list
      finally (return (list foo-list bar-list baz-list)))
```

has the same effect as:

```
(do ((g0001 list (cdr g0001))
     (x) (foo-list) (bar-list) (baz-list))
    ((null g0001)
     (list (nreverse foo-list)
           (nreverse bar-list)
           (nreverse baz-list)))
  (setq x (car g0001))
  (setq foo-list (cons (foo x) foo-list))
  (setq bar-list (cons (bar x) bar-list))
  (setq baz-list (cons (baz x) baz-list)))
```

except that **loop** arranges to form the lists in the correct order, obviating the **nreverse**s at the end, and allowing the lists to be examined during the computation.

collect *expr* {**into** *var*}

collecting ...

Causes the values of *expr* on each iteration to be collected into a list.

nconc *expr* {**into** *var*}

nconcing ...

append ...

appending ...

These are like **collect**, but the results are **nconc**ed or **append**ed together as appropriate.

```
(loop for i from 1 to 3
      nconc (list i (* i i)))
  => (1 1 2 4 3 9)
```

count *expr* {**into** *var*} {*data-type*}

counting ...

If *expr* evaluates non-**nil**, a counter is incremented. The *data-type* defaults to **fixnum**.

sum *expr* {*data-type*} {**into** *var*}

summing ...

Evaluates *expr* on each iteration, and accumulates the sum of all the values. *data-type* defaults to **number**, which for all practical purposes is **notype**. Note that specifying *data-type* implies that *both* the sum and the number being summed (the value of *expr*) is 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 is meaningless. If **loop** can determine that the arithmetic being performed is not contagious (by virtue of *data-type* being **fixnum** or **flonum**), then it can 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 is of that type.

Not only can there be multiple *accumulations* in a **loop**, but a single *accumulation* can come from multiple places *within the same* **loop** *form*. Obviously, the types of the collection must be compatible. **collect**, **nconc**, and **append** can all be mixed, as can **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)))
```

23.2.6 Endtests

The following clauses can be used to provide additional control over when the iteration gets terminated, possibly causing exit code (due to **finally**) to be performed and possibly returning a value (for example, from **collect**).

while *expr*

If *expr* evaluates to **nil**, the loop is exited, performing exit code (if any), and returning any accumulated value. The test is placed in the body of the loop where it is written. It can appear between sequential **for** clauses.

until *expr*

Identical to **while (not** *expr*).

This might be needed, for example, to step through a strange data structure, as in:

```
(loop until (top-of-concept-tree? concept)
      for concept = expr then (superior-concept concept)
          ...)
```

Note that the placement of the **until** clause before the **for** clause is valid in this

case because of the definition of this particular variant of **for**, which *binds* **concept** to its first value rather than setting it from inside the **loop**.

The following can 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) is run, and any implicitly collected result is 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.

23.2.7 Aggregated Boolean Tests

All of these clauses perform some test, and can 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** is returned when the loop finishes, after the epilogue code has been run. If the loop terminates before *expr* is ever evaluated, the epilogue code is run and the loop returns **t**.

always *expr* is like **(and** *expr1 expr2* ...**)**, except that if no *expr* evaluates to **nil**, **always** returns **t** and **and** returns the value of the last *expr*. If the loop terminates before *expr* is ever evaluated, **always** is like **(and)**.

If you want a similar test, except that you want the epilogue code to run if *expr* evaluates to **nil**, use **while**.

never *expr*

Causes the loop to return **t** if *expr* **never** evaluates non-**null**. This is equivalent to **always (not** *expr***)**. If the loop terminates before *expr* is ever evaluated, the epilogue code is run and the loop returns **t**.

never *expr* is like **(and (not** *expr1***) (not** *expr2***)** ...**)**. If the loop terminates before *expr* is ever evaluated, **never** is like **(and)**.

If you want a similar test, except that you want the epilogue code to run if *expr* evaluates non-**null**, use **until**.

thereis *expr*

If *expr* evaluates non-**null**, the iteration is terminated and that value is returned, without running the epilogue code. If the loop terminates before *expr* is ever evaluated, the epilogue code is run and the loop returns **nil**.

thereis *expr* is like **(or** *expr1 expr2* ...**)**. If the loop terminates before *expr* is ever evaluated, **thereis** is like **(or)**.

If you want a similar test, except that you want the epilogue code to run if *expr* evaluates non-**null**, use **until**.

23.2.8 Conditionalization

These clauses can be used to "conditionalize" the following clause. They can 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 is skipped, otherwise not.

unless *expr*
This is equivalent to **when (not** *expr***)**.

Multiple conditionalization clauses can appear in sequence. If one test fails, then any following tests in the immediate sequence, and the clause being conditionalized, are skipped.

Multiple clauses can 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 can 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 can appear in an **else**-phrase, using **and** to join them in the same way as above.

Conditionals can be nested. For example:

```
(loop for i from a to b
      when (zerop (remainder i 3))
        do (print i)
        and when (zerop (remainder i 2))
              collect i)
```

returns a list of all multiples of **6** from **a** to **b**, and prints all multiples of **3** from **a** to **b**.

When **else** is used with nested conditionals, the "dangling else" ambiguity is resolved by matching the **else** with the innermost **when** not already matched with an **else**. Here is a complicated example.

```
(loop for x in l
      when (atom x)
        when (memq x *distinguished-symbols*)
          do (process1 x)
      else do (process2 x)
      else when (memq (car x) *special-prefixes*)
             collect (process3 (car x) (cdr x))
             and do (memorize x)
      else do (process4 x))
```

Useful with the conditionalization clauses is the **return** clause, which causes an explicit return of its "argument" as the value of the iteration, bypassing any epilogue code. That is:

when *expr1* **return** *expr2*

is equivalent to:

when *expr1* **do (return** *expr2***)**

Conditionalization of one of the "aggregated boolean value" clauses simply causes the test that would cause the iteration to terminate early not to be performed unless the condition succeeds. For example:

```
(loop for x in l
      when (significant-p x)
        do (print x) (princ "is significant.")
        and thereis (extra-special-significant-p x))
```

does not make the **extra-special-significant-p** check unless the **significant-p** check succeeds.

The format of a conditionalized clause is typically something like:

when *expr1 keyword expr2*

If *expr2* is the keyword **it**, then a variable is generated to hold the value of *expr1*, and that variable gets substituted for *expr2*. Thus, the composition:

when *expr* **return it**

is equivalent to the clause:

thereis *expr*

and one can collect all non-**null** values in an iteration by saying:

when *expression* **collect it**

If multiple clauses are joined with **and**, the **it** keyword can only be used in the first.

If multiple **whens**, **unlesses**, and/or **ifs** occur in sequence, the value substituted for **it** is that of the last test performed. The **it** keyword is not recognized in an **else**-phrase.

23.2.9 Miscellaneous Other Clauses

named *name*

Gives the **prog** that **loop** generates a name of *name*, so that you can 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 can be given to any particular **loop** construct. This feature does not exist in the Maclisp version of **loop**, since Maclisp does not support "named progs".

return *expression*

Immediately returns the value of *expression* as the value of the loop, without running the epilogue code. This is most useful with some sort of conditionalization, as discussed in the previous section. Unlike most of the other clauses, **return** is not considered to "generate body code", so it is allowed to occur between iteration clauses, as in:

```
(loop for entry in list
      when (not (numberp entry))
        return (error ...)
      as frob = (times entry 2)
         ...)
```

If you instead want the loop to have some return value when it finishes normally, you can place a call to the **return** function in the epilogue (with the **finally** clause). See the section "Entrance and Exit", page 212.

23.3 loop Synonyms

define-loop-macro *keyword* *Macro*

Can be used to make *keyword*, a **loop** keyword (such as **for**), into a Lisp macro that can introduce a **loop** form. For example, after evaluating:

```
(define-loop-macro for),
```

you can now write an iteration as:

```
(for i from 1 below n do ...)
```

This facility exists primarily for diehard users of a predecessor of **loop**. Its

unconstrained use is not recommended, as it tends to decrease the transportability of the code and needlessly uses up a function name.

23.4 Data Types Recognized by loop

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**), are both flonums; thus **loop** can 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 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 can be defined; for that, consult the source code):

fixnum	An implementation-dependent limited-range integer.
flonum	An implementation-dependent limited-precision floating-number.
integer	Any integer (no range restriction).
number	Any number.
notype	Unspecified type (that is, anything else).

Note that explicit specification of a nonnumeric type for an operation that is numeric (such as the **summing** clause) can cause a variable to be initialized to **nil** when it should be **0**.

If local data-type declarations must be inhibited, you can use the **nodeclare** clause.

23.5 Destructuring

Destructuring provides you with the ability to "simultaneously" assign or bind multiple variables to components of some data structure. Typically this is used with list structure. For example:

```
(loop with (foo . bar) = '(a b c) ...)
```

has the effect of binding **foo** to **a** and **bar** to **(b c)**.

loop's destructuring support is intended to parallel if not augment that provided by the host Lisp implementation, with a goal of minimally providing destructuring over list structure patterns. Thus, in Lisp implementations with no system destructuring support at all, you can still use list-structure patterns as **loop** iteration variables, and in **with** bindings. In NIL, **loop** also supports destructuring over vectors.

You can specify the data-types of the components of a pattern by using a corresponding pattern of the data type keywords in place of a single-data type keyword. This syntax remains unambiguous because wherever a data-type keyword is possible, a **loop** keyword is the only other possibility. Thus, if you want to do:

```
(loop for x in l
      as i fixnum = (car x)
      and j fixnum = (cadr x)
      and k fixnum = (cddr x)
        ...)
```

and no reference to **x** is needed, you can 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 Symbolics-Lisp, you can cause **loop** to use already provided destructuring support instead:

si:loop-use-system-destructuring? *Variable*

Exists *only* in **loop** implementations in Lisps that do not provide destructuring support in the default environment. It is by default **nil**. If changed, then **loop** behaves as it does in Lisps that *do* provide destructuring support: destructuring binding is performed using **let**, and destructuring assignment is performed using **desetq**. Presumably, if your personalized environment supplies these macros, then you should set this variable to **t**; there is, however, little (if any) efficiency loss if this is not done.

23.6 The Iteration Framework

This section describes the way **loop** constructs iterations. It is necessary if you are writing your own iteration paths, and can be useful in clarifying what **loop** does with its input.

loop considers the act of *stepping* to have four possible parts. Each iteration-driving clause has some or all of these four parts, which are executed in this order:

pre-step-endtest

This is an endtest that determines if it is safe to step to the next value of the iteration variable.

steps Variables that get "stepped". This is internally manipulated as a list of the form (*var1 val1 var2 val2* ...); all of those variables are stepped in parallel, meaning that all of the *val*s are evaluated before any of the *var*s are set.

post-step-endtest

Sometimes you cannot see if you are done until you step to the next value; that is, the endtest is a function of the stepped-to value.

pseudo-steps

Other things that need to be stepped. This is typically used for internal variables that are more conveniently stepped here, or to set up iteration variables that are functions of some internal variable(s) that are actually driving the iteration. This is a list like *steps*, but the variables in it do not get stepped in parallel.

The above alone is actually insufficient in just about all iteration-driving clauses that **loop** handles. What is missing is that in most cases, the stepping and testing for the first time through the loop is different from that of all other times. So, what **loop** deals with is two four-tuples as above; one for the first iteration, and one for the rest. The first can be thought of as describing code that immediately precedes the loop in the **prog**, and the second as following the body code — in fact, **loop**

does just this, but severely perturbs it in order to reduce code duplication. Two lists of forms are constructed in parallel: one is the first-iteration endtests and steps, the other the remaining-iterations endtests and steps. These lists have dummy entries in them so that identical expressions 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 *might* get some duplicated code if one has multiple iterations. Alternatively, **loop** might decide to use and test a flag variable that indicates whether one iteration has been performed. In general, sequential iterations have less overhead than parallel iterations, both from the inherent overhead of stepping multiple variables in parallel, and from the standpoint of potential code duplication.

Note also that although the user iteration variables are guaranteed to be stepped in parallel, the placement of the endtest for any particular iteration can 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**, might or might not do this depending on the particular construction.

This problem also means that it might not be safe to examine an iteration variable in the epilogue of the loop form. As a general rule, if an iteration-driving clause implicitly supplies an endtest, then you cannot know the state of the iteration variable when the loop terminates. Although you can guess on the basis of whether the iteration variable itself holds the data upon which the endtest is based, that guess *might* be wrong. Thus:

```
(loop for subl on expr
          ...
      finally (f subl))
```

is incorrect, but:

```
(loop as frob = expr while (g frob)
                           ...
      finally (f frob))
```

is safe because the endtest is explicitly dissociated from the stepping.

23.7 Iteration Paths

Iteration paths provide a mechanism for user extension of iteration-driving clauses. The interface is constrained so that the definition of a path need not depend on much of the internals of **loop**. The typical form of an iteration path is

for *var* {*data-type*} being {each|the} *pathname* {*preposition1 expr1*}...

pathname is an atomic symbol that is defined as a **loop** path function. The usage and defaulting of *data-type* is up to the path function. Any number of preposition/expression pairs can be present; the prepositions allowable for any particular path are defined by that path. For example:

```
(loop for x being the array-elements of my-array from 1 to 10
        ...)
```

To enhance readability, pathnames are usually defined in both the singular and plural forms; this particular example could have been written as:

```
(loop for x being each array-element of my-array from 1 to 10
        ...)
```

Another format, which is not so generally applicable, is:

for *var* {*data-type*} being *expr0* and its *pathname* {*preposition1 expr1*}...

In this format, *var* takes on the value of *expr0* the first time through the loop. Support for this format is usually limited to paths that step through some data structure, such as the "superiors" of something. Thus, we can hypothesize the **cdrs** path, such that:

```
(loop for x being the cdrs of '(a b c . d) collect x)
 => ((b c . d) (c . d) d)
```

but:

```
(loop for x being '(a b c . d) and its cdrs collect x)
 => ((a b c . d) (b c . d) (c . d) d)
```

To satisfy the anthropomorphic among you, **his**, **her**, or **their** can be substituted for the **its** keyword, as can **each**. Egocentricity is not condoned. See the section "Predefined Iteration Paths", page 225. Some example uses of iteration paths are shown in that section.

Very often, iteration paths step internal variables that the you do not specify, such as an index into some data structure. Although in most cases the user does not wish to be concerned with such low-level matters, it is occasionally useful to have a handle on such things. **loop** provides an additional syntax with which you can provide a variable name to be used as an "internal" variable by an iteration path, with the **using** "prepositional phrase".

The **using** phrase is placed with the other phrases associated with the path, and contains any number of keyword/variable-name pairs:

```
(loop for x being the array-elements of a using (index i) (sequence s)
      ...)
```

which says that the variable **i** should be used to hold the index of the array being stepped through, and the variable **s** should be bound to the array. The particular keywords that can be used are defined by the iteration path; the **index** and

sequence keywords are recognized by all **loop** sequence paths. See the section "Sequence Iteration", page 225. 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 is 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 can be in use, and the potential for syntactic ambiguity exists if *frob* is the name of a defined iteration path. This feature is not for casual use; it is intended for use by large systems that wish to use a special syntax for some feature they provide.

23.7.1 loop Iteration Over Hash Tables or Heaps

loop has iteration paths that support iterating over each entry in a hash table or a heap.

```
(loop for x being the hash-elements of new-coms ...)
(loop for x being the hash-elements of new-coms with-key k ...)

(loop for x being the heap-elements of priority-queue ...)
(loop for x being the heap-elements of priority-queue with-key k ...)
```

This allows *x* to take on the values of successive entries of hash tables or heaps. The body of the loop runs once for each entry of the hash table or heap. For heaps, *x* could have the same value more than once, since the key is not necessarily unique. When looping over hash tables or heaps, the ordering of the elements is undefined.

The **with-key** phrase is optional. It provides for the variable *k* to have the hash or heap key for the particular entry value *x* that you are examining.

The **heap-elements loop** iteration path returns the items in random order and does not provide for locking the heap.

23.7.2 Predefined Iteration Paths

loop comes with two predefined iteration path functions; one implements a **mapatoms**-like iteration path facility, and the other is used for defining iteration paths for stepping through sequences.

23.7.2.1 The interned-symbols Path

The **interned-symbols** iteration path is like a **mapatoms** for **loop**.

```
(loop for sym being interned-symbols ...)
```

iterates over all of the symbols in the current package and its superiors (or, in Maclisp, the current obarray). This is the same set of symbols that **mapatoms** iterates over, although not necessarily in the same order. The particular package to look in can be specified as in:

```
(loop for sym being the interned-symbols in package ...)
```

which is like giving a second argument to **mapatoms**.

In Lisp implementations such as Symbolics-Lisp with some sort of hierarchical package structure, you can restrict the iteration to be over just the package specified and not its superiors, by using the **local-interned-symbols** path:

```
(loop for sym being the local-interned-symbols {in package}
      ...)
```

Example:

```
(defun my-apropos (sub-string &optional (pkg package))
  (loop for x being the interned-symbols in pkg
        when (string-search sub-string x)
          when (or (boundp x) (fboundp x) (plist x))
            do (print-interesting-info x)))
```

In the Symbolics-Lisp and NIL implementations of **loop**, a package specified with the **in** preposition can be anything acceptable to the **pkg-find-package** function. The code generated by this path contains calls to internal **loop** functions, with the effect that it is 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.

23.7.2.2 Sequence Iteration

One very common form of iteration is that over the elements of some object that is accessible by means of an integer index. **loop** defines an iteration path function for doing this in a general way, and provides a simple interface to allow users to define iteration paths for various kinds of "indexable" data.

define-loop-sequence-path *path-name-or-names fetchfun sizefun* &optional *sequence-type element-type* *Macro*

path-name-or-names is either an atomic path name or list of path names.

fetchfun is a function of two arguments: the sequence, and the index of the item to be fetched. (Indexing is assumed to be zero-origined.) *sizefun* is a function of one argument, the sequence; it should return the number of elements in the sequence. *sequence-type* is the name of the data-type of the sequence, and *element-type* the name of the data-type of the elements of the sequence. These last two items are optional.

The Symbolics-Lisp implementation of **loop** utilizes the Symbolics-Lisp 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
```

steps *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 can be specified with the use of all the same keywords that are accepted by the **loop** arithmetic stepper (**for** *var* **from** ...); they are **by**, **to**, **downto**, **from**, **downfrom**, **below**, and **above**, and are interpreted in the same manner. Thus:

```
(loop for var being the array-elements of array
          from 1 by 2
          ...)
```

steps *var* over all of the odd elements of *array*, and:

```
(loop for var being the array-elements of array
          downto 0
          ...)
```

steps in "reverse" order.

```
(define-loop-sequence-path (vector-elements vector-element)
                           vref vector-length notype notype)
```

is how the **vector-elements** iteration path can be defined in NIL (which it is). You can then do such things as:

```
(defun cons-a-lot (item &restv other-items)
  (and other-items
       (loop for x being the vector-elements of other-items
             collect (cons item x))))
```

All such sequence iteration paths allow you to specify the variable to be used as the index variable, by use of the **index** keyword with the **using** prepositional phrase. You can also use the **sequence** keyword with the **using** prepositional phrase to specify the variable to be bound to the sequence. See the section "Iteration Paths", page 222.

23.7.3 Defining Iteration Paths

A **loop** iteration clause (for example, a **for** or **as** clause) produces, in addition to the code that defines the iteration, variables that must be bound, and preiteration (*prologue*) code. See the section "The Iteration Framework", page 221. This breakdown allows a user interface to **loop** that does not have to depend on or know about the internals of **loop**. To complete this separation, the iteration path mechanism parses the clause before giving it to the user function that returns those items. A function to generate code for a path can be declared to **loop** with the **define-loop-path** function:

define-loop-path *Macro*

(define-loop-path *pathname-or-names path-function*
list-of-allowable-prepositions
datum-1 datum-2 ...)

This defines *path-function* to be the handler for the path(s) *pathname-or-names*, which can 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 is called with the following arguments:

path-name The name of the path that caused the path function to be invoked.

variable The "iteration variable".

data-type The data type supplied with the iteration variable, or **nil** if none was supplied.

prepositional-phrases
A list with entries of the form *(preposition expression)*, in the order in which they were collected. This can also include some supplied implicitly (for example, an **of** phrase when the iteration is inclusive, and an **in** phrase for the **default-loop-path** path); the ordering shows the order of evaluation that should be followed for the expressions.

inclusive? **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* appears in *prepositional-phrases* with the **of** preposition; for example, **for x being foo and its cdrs** gets *prepositional-phrases* of **((of foo))**.

allowed-prepositions
The list of allowable prepositions declared for the pathname that caused the path function to be invoked. It and *data* can be used

by the path function such that a single function can handle similar paths.

data The list of "data" declared for the pathname that caused the path function to be invoked. It might, 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 might be able to handle different paths.

The handler should return a list of either six or ten elements:

variable-bindings
A list of variables that need to be bound. The entries in it can 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 are 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
A list of forms that should be included in the **loop** prologue.

the four items of the iteration specification
The four items: *pre-step-endtest*, *steps*, *post-step-endtest*, and *pseudo-steps*. See the section "The Iteration Framework", page 221.

another four items of iteration specification
If these four items are given, they apply to the first iteration, and the previous four apply to all succeeding iterations; otherwise, the previous four apply to *all* iterations.

Here are the routines that are used by **loop** to compare keywords for equality. In all cases, a *token* can be any Lisp object, but a *keyword* is expected to be an atomic symbol. In certain implementations these functions might be implemented as macros.

si:loop-tequal *token keyword* *Function*
The **loop** token comparison function. *token* is any Lisp object; *keyword* is the keyword it is to be compared against. It returns **t** if they represent the same token, comparing in a manner appropriate for the implementation.

si:loop-tmember *token keyword-list* *Function*
The **member** variant of **si:loop-tequal**.

si:loop-tassoc *token keyword-alist* *Function*
The **assoc** variant of **si:loop-tequal**.

If an iteration path function desires to make an internal variable accessible to the user, it should call the following function instead of **gensym**:

si:loop-named-variable *keyword* *Function*

Should only be called from within an iteration path function. If *keyword* has been specified in a **using** phrase for this path, the corresponding variable is returned; otherwise, **gensym** is called and that new symbol returned. Within a given path function, this routine should only be called once for any given keyword.

If you specify a **using** preposition containing any keywords for which the path function does not call **si:loop-named-variable**, **loop** informs you of the error.

23.7.3.1 An Example Path Definition

Here is an example function that defines the **string-characters** iteration path. This path steps a variable through all of the characters of a string. It accepts the format:

```
(loop for var being the string-characters of str ...)
```

The function is defined to handle the path by:

```
(define-loop-path string-characters string-chars-path
  (of))
```

Here is the function:

```
(defun string-chars-path (path-name variable data-type
                          prep-phrases inclusive?
                          allowed-prepositions data
                          &aux (bindings nil)
                               (prologue nil)
                               (string-var (gensym))
                               (index-var (gensym))
                               (size-var (gensym)))
  allowed-prepositions data ; unused variables
  ; To iterate over the characters of a string, we need
  ; to save the string, save the size of the string,
  ; step an index variable through that range, setting
  ; the user's variable to the character at that index.
  ; Default the data-type of the user's variable:
  (cond ((null data-type) (setq data-type 'fixnum)))
  ; We support exactly one "preposition", which is
  ; required, so this check suffices:
  (cond ((null prep-phrases)
         (ferror nil "OF missing in ~S iteration path of ~S"
                 path-name variable)))
  ; We do not support "inclusive" iteration:
  (cond ((not (null inclusive?))
         (ferror nil
           "Inclusive stepping not supported in ~S path ~
            of ~S (prep phrases = ~:S)"
           path-name variable prep-phrases)))
  ; Set up the bindings
  (setq bindings (list (list variable nil data-type)
                       (list string-var (cadar prep-phrases))
                       (list index-var 0 'fixnum)
                       (list size-var 0 'fixnum)))
  ; Now set the size variable
  (setq prologue (list '(setq ,size-var (string-length
                                          ,string-var))))
  ; and return the appropriate stuff, explained below.
  (list bindings
        prologue
        '(= ,index-var ,size-var)
        nil
        nil
       ; char-n is the NIL string referencing primitive.
       ; In Symbolics-Lisp, aref could be used instead.
        (list variable '(char-n ,string-var ,index-var)
              index-var '(1+ ,index-var))))
```

The first element of the returned list is the bindings. The second is a list of forms to be placed in the *prologue*. The remaining elements specify how the iteration is to be performed. This example is a particularly simple case, for two reasons: the actual "variable of iteration", **index-var**, is purely internal (being **gensym**med), and

the stepping of it (**1+**) is such that it can be performed safely without an endtest. Thus **index-var** can be stepped immediately after the setting of the user's variable, causing the iteration specification for the first iteration to be identical to the iteration specification for all remaining iterations. This is advantageous from the standpoint of the optimizations **loop** is able to perform, although it is frequently not possible due to the semantics of the iteration (for example, **for** *var* **first** *expr1* **then** *expr2*) or to subtleties of the stepping. It is safe for this path to step the user's variable in the *pseudo-steps* (the fourth item of an iteration specification) rather than the "real" steps (the second), because the step value can have no dependencies on any other (user) iteration variables. Using the pseudo-steps generally results in some efficiency gains.

If you wanted the index variable in the above definition to be user-accessible through the **using** phrase feature with the **index** keyword, the function would need to be changed in two ways. First, **index-var** should be bound to **(si:loop-named-variable 'index)** instead of **(gensym)**. Secondly, the efficiency hack of stepping the index variable ahead of the iteration variable must not be done. This is effected by changing the last form to be:

```
(list bindings prologue
      nil
      (list index-var '(1+ ,index-var))
      '(= ,index-var ,size-var)
      (list variable '(char-n ,string-var ,index-var))
      nil
      nil
      '(= ,index-var ,size-var)
      (list variable '(char-n ,string-var ,index-var)))
```

Note that although the second **'(= ,index-var ,size-var)** could have been placed earlier (where the second **nil** is), it is best for it to match up with the equivalent test in the first iteration specification grouping.

PART VI.

Arrays, Characters, and Strings

24. Arrays

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

The *dimensionality* of an array (or, the number of dimensions that the array has) is the number of subscripts used to refer to one of the elements of the array. The dimensionality can be any integer from zero to seven, inclusive.

The lowest value for any subscript is 0; the highest value is a property of the array. Each dimension has a size, which is the lowest number that is too great to be used as a subscript. For example, in a one-dimensional array of five elements, the size of the one and only dimension is five, and the acceptable values of the subscript are 0, 1, 2, 3, and 4.

The most basic primitive functions for handling arrays are:

- **make-array** — used for the creation of arrays
- **aref** — used for examining the contents of arrays
- **aset** — used for storing into arrays

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

Another way of handling arrays, inherited from Maclisp, is to treat them as functions. In this case each array has a name, which is a symbol whose function definition is the array. Symbolics-Lisp 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.

24.1 Array Types

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 integers or flonums. The array types are known by a set of symbols whose names begin with "**art-**" (for ARray Type).

24.1.1 art-q Array Type

The most commonly used type is called **art-q**. An **art-q** array simply holds Lisp objects of any type. This array type can store floating point numbers without any storage overhead.

24.1.2 art-q-list Array Type

Similar to the **art-q** type is the **art-q-list**. Like the **art-q**, its elements can be any Lisp object. The difference is that the **art-q-list** array "doubles" as a list; the function **g-l-p** takes an **art-q-list** array and returns a list whose elements are those of the array, and whose actual substance is that of the array. If you **rplaca** elements of the list, the corresponding element of the array changes, and if you store into the array, the corresponding element of the list changes the same way. An attempt to **rplacd** the list causes an error, since arrays cannot implement that operation.

24.1.3 art-*N*b Array Type

There is a set of types called **art-1b, art-2b, art-4b, art-8b,** and **art-16b**; these names are short for "1 bit", "2 bits", and so on. Each element of an **art-*n*b** array is a nonnegative integer, 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 find a 1 rather than a 5.

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

24.1.4 art-string Array Type

Character strings are implemented by the **art-string** array type. This type acts similarly to the **art-8b**; its elements must be integers, of which only the least significant eight bits are stored. However, many important system functions, including **read, print,** and **eval,** treat **art-string** arrays very differently from the other kinds of arrays. These arrays are usually called *strings*. See the section "Strings", page 277. That section deals with functions that manipulate these type of arrays.

24.1.5 art-fat-string Array Type

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.

24.1.6 art-boolean Array Type

An **art-boolean** array type is an array whose elements can take on the values **t** and **nil**. It uses only one bit of storage per element.

For example:

```
(setq boolean-array (make-array 7 :type 'art-boolean))
 => #<ART-BOOLEAN-7 22741370>

(listarray boolean-array)
 => (NIL NIL NIL NIL NIL NIL NIL)

(setf (aref boolean-array 0) T) => T
(setf (aref boolean-array 1) :some-keyword)
 => :SOME-KEYWORD
(setf (aref boolean-array 2) 42) => 42
(setf (aref boolean-array 3) boolean-array)
 => #<ART-BOOLEAN-7 25123011>

(listarray boolean-array) => (T T T T NIL NIL NIL)

(setf (aref boolean-array 1) (typep boolean-array :number))
 => NIL

(listarray boolean-array) => (T NIL T T NIL NIL NIL)
```

24.1.7 Multidimensional Arrays

Currently, multidimensional arrays are stored in column-major order rather than row-major order as in Maclisp. Row-major order means that successive memory locations differ in the last subscript, while column-major order means that successive memory locations differ in the first subscript. This has an effect on paging performance when using large arrays; if you want to reference every element in a multidimensional array and move linearly through memory to improve locality of reference, you must vary the first subscript fastest rather than the last.

24.2 Array Representation Tools

array-types *Variable*

The value of **array-types** is a list of all of the array type symbols such as **art-q**, **art-4b**, **art-string** and so on. The values of these symbols are internal array type code numbers for the corresponding type.

array-types *array-type-code* *Function*

Given an internal numeric array-type code, returns the symbolic name of that type.

array-elements-per-q *Variable*

array-elements-per-q is an association list that associates each array type symbol with the number of array elements stored in one word, for an array of that type. See the section "Association Lists", page 64. If the value is negative, it is instead the number of words per array element, for arrays whose elements are more than one word long.

array-elements-per-q *array-type-code* *Function*

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

array-bits-per-element *Variable*

The value of **array-bits-per-element** is an association list that associates each array type symbol with the number of bits of unsigned number it can hold, or **nil** if it can hold Lisp objects. This can be used to tell whether an array can hold Lisp objects or not. See the section "Association Lists", page 64.

array-bits-per-element *array-type-code* *Function*

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

array-element-size *array* *Function*

Given an array, returns the number of bits that fit in an element of that array. For arrays that can hold general Lisp objects, the result is 31; this assumes that you are storing fixnums in the array and manipulating their bits with **dpb** (rather than **%logdpb**). You can store any number of bits per element in an array that holds general Lisp objects, by letting the elements expand into bignums.

24.3 Extra Features of Arrays

24.3.1 Array Leaders

Any array can have an *array leader*. An array leader is similar to a one-dimensional **art-q** array that is attached to the main array. An array that has a leader acts like two arrays joined together. The leader can be stored into and examined by a special set of functions, different from those used for the main array: **array-leader** and **store-array-leader**. The leader is always one-dimensional, and can always 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 is a homogeneous set of objects, while the leader is used to remember a few associated nonhomogeneous pieces of data. In this case the leader is not used like an array; each slot is used differently from the others. Explicit numeric subscripts should not be used for the leader elements of such an array; instead the leader should be described by a **defstruct**. See the macro **defstruct**, page 383.

By convention, element zero of the array leader of an array is used to hold the number of elements in the array that are "active" in some sense. When the zeroth element is used this way, it is called a *fill pointer*. Many array-processing functions recognize the fill pointer. For instance, if a string (an array of type **art-string**) has seven elements, but its fill pointer contains the value 5, then only elements zero through four of the string are considered to be "active". This means that the string's printed representation is five characters long, string-searching functions stop after the fifth element, and so on.

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

Leader element one is used in conjunction with the "named structure" feature to associate a "data type" with the array. See the section "Named Structures", page 403. Element one is treated specially only if the array is flagged as a named structure.

24.3.2 Displaced Arrays

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 Symbolics Lisp Machine's address space, such as the area used for the control of input/output devices, or the bitmap memory that generates the TV image. These are known as *displaced arrays*. They 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** an integer or a locative as the value of the **:displaced-to** option, it creates a displaced array referring to that location of virtual memory and its successors.

References to elements of the displaced array 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.

24.3.3 Indirect Arrays

It is possible to have an array whose contents, instead of being located at a fixed place in virtual memory, are defined to be those of another array. Such an array is called an *indirect array*, and is created by giving **make-array** an array as the value of the **:displaced-to** option.

The effects of this are simple if both arrays have the same type; the two arrays share all elements. An object stored in a certain element of one can be retrieved from the corresponding element of the other. This, by itself, is not very useful. However, if the arrays have different dimensionality, the manner of accessing the elements differs. Thus, by creating a one-dimensional array of nine elements that was indirected to a second, two-dimensional array of three elements by three, then the elements could be accessed in either a one-dimensional or a two-dimensional manner.

Unexpected effects can be produced if the new array is of a different type than the old array; this is not generally recommended. Indirecting an **art-*m*b** array to an **art-*n*b** array does the "obvious" thing. For instance, if *m* is 4 and *n* is 1, each element of the first array contains four bits from the second array, in right-to-left order.

It is 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 an integer to **make-array** as the value of the **:displaced-index-offset** option. Similarly, the length of the indirect array need not be the full length of the array it indirects to; it can be smaller. The **nsubstring** function creates such arrays. When using index offsets with multidimensional arrays, there is only one index offset; it is added in to the "linearized" subscript which is the result of multiplying each subscript by an appropriate coefficient and adding them together.

24.3.3.1 Conformal Indirection

Multidimensional arrays remember their actual dimensions, separately from the magic numbers by which to multiply the subscripts before adding them together to get the index into the array.

As a result of this, multidimensional indirect arrays can have *conformal indirection*. If A is indirected to B, and they do not have the same width, then normally the part of B that is shared with A does not have the same shape as A. If conformal indirection is used, then it does have the same shape and there are gaps between the rows of A. For example:

```
(setq b (make-array '(10. 20.)))
(setq a (make-array '(3 5) :displaced-to b
                           :displaced-index-offset 12.))
```

Now:

```
(aref a 1 0) = (aref b 3 1) and (aref a 1 1) = (aref b 6 1).
```

In contrast:

```
(setq a (make-array '(3 5) :displaced-to b
                           :displaced-index-offset 12.
                           :displaced-conformally t))
```

`(aref a 1 0) = (aref b 3 1)` still, but `(aref a 1 1) = (aref b 3 2)`. Each row of A corresponds to part of a row of B, always starting at the same column `(2)`.

A graphic illustration:

```
(setq a (make-array '(6 20.))
      b (make-array '(3 5) :displaced-to a
                           :displaced-index-offset 22.)
      c (make-array '(3 5) :displaced-to a
                           :displaced-index-offset 22.
                           :displaced-conformally t))
```

```
  Normal case                    Conformal case
  0                   19         0                   19
  +--------------------+         +--------------------+
 0|aaaaaaaaaaaaaaaaaaaa|        0|aaaaaaaaaaaaaaaaaaaa|
  |aaBBBBBBBBBBBBBBBaaa|         |aaCCCCCaaaaaaaaaaaaa|
  |aaaaaaaaaaaaaaaaaaaa|         |aaCCCCCaaaaaaaaaaaaa|
  |aaaaaaaaaaaaaaaaaaaa|         |aaCCCCCaaaaaaaaaaaaa|
  |aaaaaaaaaaaaaaaaaaaa|         |aaaaaaaaaaaaaaaaaaaa|
 5|aaaaaaaaaaaaaaaaaaaa|        5|aaaaaaaaaaaaaaaaaaaa|
  +--------------------+         +--------------------+
```

Arrays are stored in column-major order, so the units in which the index-offset is measured should be read first from left to right and then from top to bottom.

The meaning of **adjust-array-size** for conformal indirect arrays is undefined.

copy-array-contents, **copy-array-contents-and-leader**, **copy-array-portion**, **fillarray**, **listarray**, **math:invert-matrix**, and all other operations that treat a multidimensional array as if it were one-dimensional do not work on conformally displaced arrays.

24.4 Basic Array Functions

make-array *dimensions* &rest *options* *Function*

This is the primitive function for making arrays. *dimensions* should be a list of integers that are the dimensions of the array; the length of the list is the dimensionality of the array. For convenience when making a one-dimensional array, the single dimension can be provided as an integer rather than a list of one integer.

options are alternating keywords and values. The keywords can be any of the following:

:area The value specifies in which area the array should be created. It should be either an area number (an integer), or **nil** to mean the default area. See the section "Areas" in *Internals, Processes, and Storage Management.*

: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 can only be integers or flonums, then every element of the array is initially 0 or 0.0; otherwise, every element is initially **nil**. See the section "Arrays", page 235. Array types are described in that section. The value of the option can also be the value of a symbol that is an array type name (that is, an internal numeric array type code).

:displaced-to
If this is not **nil**, then the array is a *displaced* array. If the value is an integer or a locative, **make-array** creates a regular displaced array that refers to the specified section of virtual address space. If the value is an array, **make-array** creates an indirect array. See the section "Displaced Arrays", page 239. See the section "Indirect Arrays", page 240.

:initial-value
This makes its value the initial value of every element of the array. Example:

```
(make-array 5 :type 'art-string :initial-value #/a)
                    => "aaaaa"
```

:leader-length
The value should be an integer. The array has a leader with that many elements. The elements of the leader are initialized to **nil** unless the **:leader-list** option is given.

:leader-list
The value should be a list. Call the number of elements in the list *n*. The first *n* elements of the leader are initialized from successive elements of this list. If the **:leader-length** option is not specified, then the length of the leader is *n*. If the **:leader-length** option is given, and its value is greater than *n*, then the *n*th and following leader elements are initialized to **nil**. If its value is less than *n*, an error is signalled. The leader elements are filled in forward order; that is, the **car** of the list is stored in leader element **0**, the **cadr** in element **1**, and so on.

:fill-pointer
It causes **make-array** to give the array a fill pointer and initializes it to the value following the keyword. Use this instead of **:leader-length** or **:leader-list** when you are using the leader only for a fill pointer. This keyword is compatible with the current Common Lisp design, which has no array leaders.

:displaced-index-offset
If this is present, the value of the **:displaced-to** option should be an array, and the value should be a nonnegative integer; it is made to be the index-offset of the created indirect array. See the section "Indirect Arrays", page 240.

:displaced-conformally
Can be used with the **:displaced-to** option. If the value is **t** and **make-array** is creating an indirect array, the array uses conformal indirection.

:named-structure-symbol
If this is not **nil**, it is a symbol to be stored in the named-structure cell of the array. The array is tagged as a named structure. See the section "Named Structures", page 403. If the array has a leader, then this symbol is stored in leader element **1** regardless of the value of the **:leader-list** option. If the array does not have a leader, then this symbol is stored in array element zero.

Examples:

```
;; Create a one-dimensional array of five elements.
(make-array 5)
;; Create a two-dimensional array,
;; three by four, with four-bit elements.
(make-array '(3 4) :type 'art-4b)
;; Create an array with a three-element leader.
(make-array 5 :leader-length 3)
;; Create an array with a leader, providing
;; initial values for the leader elements.
(setq a (make-array 100 :type 'art-1b
                        :leader-list '(t nil)))
(array-leader a 0) => t
(array-leader a 1) => nil
```

```
;; Create a named-structure with five leader
;; elements, initializing some of them.
(setq b (make-array 20 :leader-length 5
                       :leader-list '(0 nil foo)
                       :named-structure-symbol 'bar))
(array-leader b 0) => 0
(array-leader b 1) => bar
(array-leader b 2) => foo
(array-leader b 3) => nil
(array-leader b 4) => nil
```

make-array returns the newly created array, and also returns, as a second value, the number of words allocated in the process of creating the array, that is, the **%structure-total-size** of the array.

aref *array* &rest *subscripts* *Function*

Returns the element of *array* selected by the *subscripts*. The *subscripts* must be integers and their number must match the dimensionality of *array*.

ar-1 *array i* *Function*

This is an obsolete version of **aref** that only works for one-dimensional arrays. There is no reason ever to use it.

ar-2 *array i j* *Function*

This is an obsolete version of **aref** that only works for two-dimensional arrays. There is no reason ever to use it.

aset *x array* &rest *subscripts* *Function*

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

as-1 *x array i* *Function*

This is an obsolete version of **aset** that only works for one-dimensional arrays. There is no reason ever to use it.

as-2 *x array i j* *Function*

This is an obsolete version of **aset** that only works for two-dimensional arrays. There is no reason ever to use it.

aloc *array* &rest *subscripts* *Function*

Returns a locative pointer to the element-cell of *array* selected by the *subscripts*. The *subscripts* must be integers and their number must match the dimensionality of *array*. See the section "Locatives", page 83.

ap-1 *array i* *Function*

This is an obsolete version of **aloc** that only works for one-dimensional arrays. There is no reason ever to use it.

ap-2 *array i j* *Function*

This is an obsolete version of **aloc** that only works for two-dimensional arrays. There is no reason ever to use it.

The compiler turns **aref** into **ar-1** and **ar-2** according to the number of subscripts specified. It also turns **aset** into **as-1** and **as-2** and **aloc** into **ap-1** and **ap-2**. For arrays with more than two dimensions the compiler uses the slightly less efficient form since the special routines only exist for one and two dimensions. There is no reason for any program to call **ar-1**, **as-1**, **ar-2**, and so forth explicitly; they are documented because many old programs use them. New programs should use **aref**, **aset**, and **aloc**.

A related function, provided only for Maclisp compatibility, is **arraycall**.

array-leader *array i* *Function*

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

store-array-leader *x array i* *Function*

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

ap-leader *array i* *Function*

array should be an array with a leader, and *i* should be an integer. This returns a locative pointer to the *i*'th element of *array*'s leader. See the section "Locatives", page 83. This is analogous to **aloc**.

fill-pointer *array* *Function*

Returns the value of the fill pointer. *array* must have a fill pointer. **fill-pointer** is actually a subst, so it compiles inline instead of as a function call. **setf** can be used on a **fill-pointer** form to set the value of the fill pointer.

Programs access the fill pointer by explicitly asking for the zeroth element of the array leader.

24.5 Accessing Multidimensional Arrays as One-dimensional

sys:array-register-1d

The **sys:array-register-1d** declaration is used together with the **sys:%1d-aref** and **sys:%1d-aset** functions to access multidimensional arrays as if they were one-dimensional. This allows loop optimization of multidimensional array subscript calculations. The user must do the reduction from multiple subscripts to a single

subscript. See the section "Accessing Arrays Specially" in *Internals, Processes, and Storage Management.*

sys:%1d-aref *array index* *Function*

sys:%1d-aref is the same as **aref**, except that it ignores the number of dimensions of the array and acts as if it were a one-dimensional array by linearizing the multidimensional elements. In Zetalisp, arrays are stored in column-major order. **listarray** and **copy-array-portion** use this function.

For example:

```
(setq array (make-array '(20 30 50))) => #<Art-Q-20-30-50 26165512>

(setf (aref array 5 6 7) 'foo) => FOO

(aref array 5 6 7) => FOO

(sys:%1d-aref array (+ 5 (* 20 (+ 6 (* 30 7))))) => FOO
```

sys:%1d-aset *value array index* *Function*

sys:%1d-aset is the same as **aset**, except that it ignores the number of dimensions of the array and acts as if it were a one-dimensional array by linearizing the multidimensional elements. In Zetalisp, arrays are stored in column-major order. **copy-array-portion** uses this function.

Current style suggests that you should use **(setf (sys:%1d-aref ...))** instead of **sys:%1d-aset**.

sys:%1d-aloc *array index* *Function*

sys:%1d-aloc is the same as **aloc**, except that it ignores the number of dimensions of the array and acts as if it were a one-dimensional array by linearizing the multidimensional elements. In Zetalisp, arrays are stored in column-major order.

Current style suggests that you should use **(locf (sys:%1d-aref ...))** instead of **sys:%1d-aloc**.

24.6 Getting Information About an Array

array-type *array* *Function*

Returns the symbolic type of *array*. Example:

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

array-length *array* *Function*

array can 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 might want to use **array-active-length**.) **array-length** does not return the product of the dimensions for conformal arrays.

Example:

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

array-active-length *array* *Function*

If *array* does not have a fill pointer, then this returns whatever **(array-length** *array***)** would have. If *array* does have a fill pointer, **array-active-length** returns it. See the section "Array Leaders", page 238. A general explanation of the use of fill pointers is in that section.

array-#-dims *array* *Function*

Returns the dimensionality of *array*. Note that the name of the function includes a "#", which must be slashified if you want to be able to read your program in Maclisp. (It does not need to be slashified for the Symbolics-Lisp reader, which is smarter.) Example:

```
(array-#-dims (make-array '(3 5))) => 2
```

array-dimension-n *n array* *Function*

array can be any kind of array, and *n* should be an integer. If *n* is between 1 and the dimensionality of *array*, this returns the *n*th dimension of *array*. If *n* is **0**, this returns the length of the leader of *array*; if *array* has no leader it returns **nil**. If *n* is any other value, this returns **nil**. Examples:

```
(setq a (make-array '(3 5) :leader-length 7))
(array-dimension-n 1 a) => 3
(array-dimension-n 2 a) => 5
(array-dimension-n 3 a) => nil
(array-dimension-n 0 a) => 7
```

array-dimensions *array* *Function*

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

```
(setq a (make-array '(3 5)))
(array-dimensions a) => (3 5)
```

Note: the list returned by **(array-dimensions** *x***)** is equal to the cdr of the list returned by **(arraydims** *x***)**.

arraydims *array* *Function*

array can be any array; it also can be a symbol whose function cell contains an array, for Maclisp compatibility. See the section "Maclisp Array Compatibility", page 262. **arraydims** returns a list whose first element is

the symbolic name of the type of *array*, and whose remaining elements are its dimensions. Example:

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

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

This function checks whether *subscripts* is a valid set of subscripts for *array*, and returns **t** if they are; otherwise it returns **nil**.

array-column-major-index *array* &rest *subscripts* *Function*

This function takes an array and valid subscripts for the array and returns a single nonnegative integer less than the total size of the array that identifies the accessed element in the column-major ordering of the elements. The number of *subscripts* supplied must equal the rank of the array. Each subscript must be a nonnegative integer less than the corresponding array dimension. Like **aref**, **array-column-major-index** ignores fill pointers.

For example:

`window` is a conformal array whose 0,0 coordinate is at 256,256 of `big-array`. This example creates a 1/4 size portal into the center of `big-array`.

```
(setq big-array (make-array '(1024. 1024.) :type 'art-q
                                           :initial-value 0))
(setq window (make-array '(512. 512.) :type 'art-q
                         :displaced-to big-array
                         :displaced-index-offset (array-column-major-index
                                                   big-array 256. 256.)
                         :displaced-conformally t))
```

For a one-dimensional array, the result of **array-column-major-index** equals the supplied subscript. An error is signalled if some subscript is not valid.

array-displaced-p *array* *Function*

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

array-indirect-p *array* *Function*

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

array-indexed-p *array* *Function*

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

array-has-leader-p *array* *Function*

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

array-leader-length *array* *Function*

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

24.7 Changing the Size of an Array

adjust-array-size *array new-size* *Function*

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

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

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

If the size of the array is being increased, **adjust-array-size** might have to allocate a new array somewhere. In that case, it alters *array* so that references to it are made to the new array instead, by means of "invisible pointers". See the function **structure-forward** in *Internals, Processes, and Storage Management*. **adjust-array-size** returns this new array if it creates one, and otherwise it returns *array*. Be careful to be consistent about using the returned result of **adjust-array-size**, because you might end up holding two arrays that are not the same (that is, not **eq**), but that share the same contents.

The meaning of **adjust-array-size** for conformal indirect arrays is undefined.

array-grow *array* &rest *dimensions* *Function*

array-grow creates a new array of the same type as *array*, with the specified dimensions. Those elements of *array* that are still in bounds are copied into the new array. The elements of the new array that are not in the bounds of *array* are initialized to **nil** or **0** as appropriate. If *array* has a leader, the new array has a copy of it. **array-grow** returns the new array and also forwards *array* to it, like **adjust-array-size**.

Unlike **adjust-array-size**, **array-grow** always creates a new array rather than growing or shrinking the array in place. But **array-grow** of a multidimensional array can change all the subscripts and move the elements around in memory to keep each element at the same logical place in the array.

return-array *array* *Function*

This peculiar function attempts to return *array* to free storage. If it is displaced, this returns the displaced array itself, not the data that the array points to. Because of the way storage allocation works, **return-array** does nothing if the array is not at the end of its region, that is, if it was not the most recently allocated non-list object in its area. **return-array** returns **t** if storage was really reclaimed, or **nil** if it was not.

return-array is a subtle and dangerous feature that should be avoided by most users.

It is the responsibility of any program that calls **return-array** to ensure that there are no references to *array* anywhere in the Lisp world. This includes locative pointers to array elements, such as you might create with **aloc**. The results of attempting to use such a reference to the returned array are unpredictable. Simply holding such a reference in a local variable, without attempting to access it or to print it out, is allowed.

Other tools are available for manually allocating and freeing arrays. See the special form **sys:with-stack-array** in *Internals, Processes, and Storage Management*.

24.8 Arrays Overlaid with Lists

This function manipulates **art-q-list** arrays. See the section "**art-q-list** Array Type", page 236.

g-l-p *array* *Function*

array should be an **art-q-list** array. If *array* has a fill pointer, **g-l-p** returns a list that stops at the fill pointer. 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)
```

24.9 Adding to the End of an Array

array-push *array x* *Function*

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

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

array-push-extend is just like **array-push** except that if the fill pointer gets too large, the array is grown to fit the new element; that is, it never "fails" the way **array-push** does, and so never returns **nil**. *extension* is the number of elements to be added to the array if it needs to be grown. It defaults to something reasonable, based on the size of the array. **array-push-extend** returns the *former* value of the fill pointer, that is, the array index in which it stored *x*.

array-push-portion-extend *to-array from-array* &optional *(from-start* **0**) *from-end* *Function*

Copies a portion of one array to the end of another, updating the fill pointer of the other to reflect the new contents. The destination array must have a fill pointer. The source array need not. This is equivalent to numerous **array-push-extend** calls, but more efficient.

array-push-portion-extend returns the destination array and the index of the next location to be filled.

Example:

```
(setq to-string
      (array-push-portion-extend to-string from-string (or from 0) to))
```

It is similar to **array-push-extend** except that it copies more than one element and has different return values. The arguments default in the usual way, so that the default is to copy all of *from-array* to the end of *to-array*.

array-push-portion-extend adjusts the array size using **adjust-array-size**. It picks the new array size in the same way that **adjust-array-size** does, making it bigger than needed for the information being added. In this way, successive additions do not each end up consing a new array. **array-push-portion-extend** uses **copy-array-portion** internally.

array-pop *array* &optional (*default* **nil**) *Function*

array must be a one-dimensional array that has a fill pointer. The fill pointer is decreased by one, and the array element designated by the new value of the fill pointer is returned.

The second argument, if supplied, is the value to be returned if the array is empty. If **array-pop** is called with one argument and the array is empty, it signals an error.

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.

24.10 Copying an Array

fillarray *array source* *Function*

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

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

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

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

fillarray does not work on conformally displaced arrays.

listarray *array* &optional *limit* *Function*

array can 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 an integer, and only the first *limit* (if there are more than that many) elements of *array* are used, and so the maximum length of the returned list is *limit*.

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

listarray does not work on conformally displaced arrays.

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

array can 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 an integer, and only the first *limit* (if there are more than that many) elements of *array*'s leader are used, and so the maximum length of the returned list is *limit*. If *array* has no leader, **nil** is returned.

copy-array-contents *from-array to-array* *Function*

from and *to* must be arrays. The contents of *from* is copied into the contents of *to*, element by element. If *to* is shorter than *from*, the rest of *from* is ignored. If *from* is shorter than *to*, the rest of *to* is filled with **nil** if it is a q-type array, or 0 if it is a numeric array or a string, or 0.0 if it is a flonum array. This function always returns **t**.

Note that even if *from* or *to* has a leader, the whole array is used; the convention that leader element 0 is the "active" length of the array is not used by this function. The leader itself is not copied.

copy-array-contents works on multidimensional arrays. *from* and *to* are "linearized" subscripts, and column-major order is used, that is, the *first* subscript varies fastest (opposite from **fillarray**).

copy-array-contents does not work on conformally displaced arrays.

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

This is just like **copy-array-contents**, but the leader of *from* (if any) is also copied into *to*. **copy-array-contents** copies only the main part of the array.

copy-array-contents-and-leader does not work on conformally displaced arrays.

copy-array-portion *from-array from-start from-end to-array to-start to-end* *Function*

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

copy-array-portion does not work on conformally displaced arrays.

Currently, **copy-array-portion** (as well as **copy-array-contents** and **copy-array-contents-and-leader**) copies one element at a time in increasing

order of subscripts (this behavior might change in the future). This means that when copying from and to the same array, the results might be unexpected if *from-start* is less than *to-start*. You can safely copy from and to the same array as long as *from-start* ≥ *to-start*.

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

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

The top-left corner of the source rectangle is **(aref** *from-array from-x from-y***)**. The top-left corner of the destination rectangle is **(aref** *to-array to-x to-y***)**. *width* and *height* are the dimensions of both rectangles. If *width* or *height* is zero, **bitblt** does nothing.

from-array and *to-array* are allowed to be the same array. **bitblt** normally traverses the arrays in increasing order of *x* and *y* subscripts. If *width* is negative, then **(abs** *width***)** is used as the width, but the processing of the *x* direction is done backwards, starting with the highest value of *x* and working down. If *height* is negative it is treated analogously. When **bitblt**ing an array to itself, when the two rectangles overlap, it might 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* correspond to one element of the *to-array*. *width* is in units of elements of the *to-array*.

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

If *src* is an element of the source rectangle, and *dst* is the corresponding element of the destination rectangle, then **bitblt** changes the value of *dst* to **(boole** *alu src dst***)**. See the **boole** function. The following are the symbolic names for some of the most useful *alu* functions:

tv:alu-seta	plain copy
tv:alu-ior	inclusive or
tv:alu-xor	exclusive or

tv:alu-andca and with complement of source

bitblt is written in highly optimized microcode and goes very much faster than the same thing written with ordinary **aref** and **aset** operations would. Unfortunately this causes **bitblt** to have a couple of strange restrictions. Wraparound does not work correctly if *from-array* is an indirect array with an index offset. **bitblt** signals an error if the first dimensions of *from-array* and *to-array* are not both integral multiples of the machine word length. For **art-1b** arrays, the first dimension must be a multiple of 32., for **art-2b** arrays it must be a multiple of 16., and so on.

24.11 Array Registers

The **aref** and **aset** operations on arrays consist of two parts:

1. They "decode" the array, determining its type, its rank, its length, and the address of its first data element.

2. They read or write the requested element.

The first part of this operation is not dependent on the particular values of the subscripts; it is a function only of the array itself.

When you write a loop that processes one or more arrays, the first part of each array operation is invariant if the arrays are invariant inside the loop. You can improve performance by moving this array-decoding overhead outside the loop, doing it only once at the beginning of the loop, rather than repeating it on every trip around the loop. This is done using the **sys:array-register** and **sys:array-register-1d** declarations.

sys:array-register

The declared variable is compiled into an array-register variable. This causes references to the array that is the value of the variable to compile into special faster instructions. The **sys:array-register** declaration signals an error if given other than a one-dimensional array. Use **sys:array-register-1d** to access multidimensional arrays. See the section "Accessing Multidimensional Arrays as One-dimensional", page 245.

For more information: See the function **declare**, page 311.

24.11.1 Array Registers and Performance

The array-register feature makes optimization possible and convenient. Here is an example:

```
(defun foo (array-1 array-2 n-elements)
  (let ((a array-1)
        (b array-2))
    (declare (sys:array-register a b))
    (dotimes (i n-elements)
      (setf (aref b i) (aref a i)))))
```

This function copies the first **n-elements** elements of array **a** into array **b**. If the declaration is absent, it does the same thing more slowly. The variables **a** and **b** are compiled into "array register" variables rather than normal, local, variables. At the time **a** and **b** are bound, the arrays to which they are bound are decoded and the variables are bound to the results of the decoding. The compiler recognizes **aref** with a first argument that has been declared to be an array register and **aset** with a second argument that has been declared to be an array register and compiles them as special instructions that do only the second part of the operation. These instructions are **fast-aref** and **fast-aset**.

If you want to verify that your array register declarations are working, follow these steps:

1. Compile the function.

2. Disassemble it: **(disassemble 'foo)**.

3. Look for **fast-aref/fast-aset** instructions. For example, note instructions 11 and 13:

```
 0  ENTRY: 3 REQUIRED, 0 OPTIONAL
 1  PUSH-LOCAL FP|0            ;ARRAY-1
 2  BUILTIN SETUP-1D-ARRAY TO 4        ;creating A(FP|3)
 3  PUSH-LOCAL FP|1            ;ARRAY-2
 4  BUILTIN SETUP-1D-ARRAY TO 4        ;creating B(FP|7)
 5  PUSH-IMMED 0               ;creating I(FP|11)
 6  PUSH-LOCAL FP|2            ;N-ELEMENTS        creating NIL(FP|12)
 7  BRANCH 15
10  PUSH-LOCAL FP|11           ;I
11  FAST-AREF FP|4             ;A
12  PUSH-LOCAL FP|11           ;I
13  FAST-ASET FP|8             ;B
14  BUILTIN 1+LOCAL IGNORE FP|11       ;I
15  PUSH-LOCAL FP|11           ;I
16  PUSH-LOCAL FP|12
17  BUILTIN INTERNAL-< STACK
20  BRANCH-TRUE 10
21  PUSH-NIL
22  RETURN-STACK
```

The performance advantage of array registers over the simplest types of array (for example, no leader or no displacement) is fairly small, since the normal **aref** and

aset operations on those arrays are quite fast. The real advantage of array registers is that they are equally fast for the more complicated arrays, such as indirect arrays and those with leaders, as they are for simple arrays.

The performance advantage to be gained through the use of array registers depends on the type of the array. Using an array register is never slower, except for one peculiar case: an indirect byte array with an index offset that is not a multiple of the number of array elements per word; in other words, an array whose first element is not aligned on a word boundary. An example of this case is:

```
(setq a (make-array 100 :type 'art-1b))
(setq b (make-array 99 :type 'art-1b :displaced-to a :displaced-index-offset 1))
```

If the **:displaced-index-offset** had been a multiple of 32, array registers would enhance performance.

24.11.2 Hints for Using Array Registers

The expansion of the **loop** macro's **array-elements** path copies the array into a temporary variable. In order to get the benefits of array registers, you must write code in the following way:

Right:

```
(defun tst1 (array incr)
  (declare (sys:array-register a))
  (loop for elt being the array-elements of array using (sequence a)
        sum (* elt incr)))
```

Wrong:

```
(defun tst (array incr)
  (let ((a array)) (declare (sys:array-register a))
       (loop for elt being the array-elements of a
             sum (* elt incr))))
```

loop generates a temporary variable; the "using" clause forces the temporary variable to be named **a**. Since the user gets to control the name of the variable, it is possible to assign a declaration to the variable.

The other way to do it is to avoid the **array-elements** path, and instead use:

```
(defun tst (array incr)
  (let ((a array)) (declare (sys:array-register a))
    (loop for i from 0 below (array-active-length a)
          sum (* (aref a i) incr))))
```

This is a bit more efficient because it does not have the overhead of setting up the variable **elt**.

24.11.3 Array Register Restrictions

You cannot **setq** an array-register variable. It is, however, valid to read the value of an array-register variable; this yields the array, just as if the variable had not been declared.

It is also not valid to declare a variable simultaneously **special** and **sys:array-register**. You cannot declare a parameter (a variable that appears in the argument-list of a **defun** or a **lambda**) to be an array register; you must bind another variable (perhaps with the same name) to it with **let** and declare that variable.

For example:

```
(defun tst (x y)
  (let ((x x) (y y))
    (declare (sys:array-register x y))
    ...))
```

Note that the **array-register** declaration is in the **system** package (also known as **sys**), and therefore the declaration is **system:array-register** or **sys:array-register**. If you type **array-register** instead of **sys:array-register**, the compiler ignores this declaration and does not generate any warnings. In general, the compiler ignores any misspelled declarations. In this case, the code generated by the compiler runs slower. Additionally, if you type **sys:array-registar** instead of the correct spelling, the package system catches the misspelling because the **system** package is locked.

If the array decoded into an array register is altered (for example, with **adjust-array-size**) after the array register is created, the next reference through the array register re-decodes the array.

24.12 Matrices and Systems of Linear Equations

The functions in this section perform some useful matrix operations. The matrices are represented as two-dimensional Lisp arrays. These functions are part of the mathematics package rather than the kernel array system, hence the "**math:**" in the names.

math:multiply-matrices *matrix-1 matrix-2* &optional *matrix-3* *Function*

Multiplies *matrix-1* by *matrix-2*. If *matrix-3* is supplied, **multiply-matrices** stores the results into *matrix-3* and returns *matrix-3*; otherwise it creates an array to contain the answer and returns that. All matrices must be two-dimensional arrays, and the first dimension of *matrix-2* must equal the second dimension of *matrix-1*.

math:invert-matrix *matrix* &optional *into-matrix* *Function*

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

math:invert-matrix does not work on conformally displaced arrays.

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

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

math:determinant *matrix* *Function*

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

math:decompose and **math:solve** 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 their argument names, these two functions exist to solve the vector equation $A\ x = b$ for x. A is a matrix. b and x are vectors.

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

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

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

This function takes the LU decomposition and associated permutation array produced by **math:decompose**, and solves the set of simultaneous equations defined by the original matrix *a* and the right-hand sides in the vector *b*. If

x is supplied, the solutions are stored into it and it is returned; otherwise, an array is created to hold the solutions and that is returned. *b* must be a one-dimensional array.

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

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

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

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

24.13 Planes

A *plane* is an array whose bounds, in each dimension, are plus-infinity and minus-infinity; all integers are valid as indices. Planes are distinguished not by size and shape, but by number of dimensions alone. When a plane is created, a default value must be specified. At that moment, every component of the plane has that value. As you cannot ever change more than a finite number of components, only a finite region of the plane need actually be stored.

The regular array accessing functions do not work on planes. You can use **make-plane** to create a plane, **plane-aref** or **plane-ref** to get the value of a component, and **plane-aset** or **plane-store** to store into a component. **array-#-dims** works on a plane.

A plane is actually stored as an array with a leader. The array corresponds to a rectangular, aligned region of the plane, containing all the components in which a **plane-store** has been done (and others, in general, which have never been altered). The lowest-coordinate corner of that rectangular region is given by the **plane-origin** in the array leader. The highest coordinate corner can be found by adding the **plane-origin** to the **array-dimensions** of the array. The **plane-default** is the contents of all the elements of the plane that are not actually stored in the array. The **plane-extension** is the amount to extend a plane by in any direction when the plane needs to be extended. The default is 32.

If you never use any negative indices, then the **plane-origin** is all zeroes and you can use regular array functions, such as **aref** and **aset**, to access the portion of the plane which is actually stored. This can be useful to speed up certain algorithms. In this case you can even use the **bitblt** function on a two-dimensional plane of bits or bytes, provided you don't change the **plane-extension** to a number that is not a multiple of 32.

make-plane *rank* &rest *options* *Function*

Creates and returns a plane. *rank* is the number of dimensions. *options* is a list of alternating keyword symbols and values. The allowed keywords are:

:type The array type symbol (for example, **art-1b**) specifying the type of the array out of which the plane is made.

:default-value
The default component value.

:extension
The amount by which to extend the plane. See the section "Planes", page 260.

:initial-dimensions
A list of dimensions for the initial creation of the plane. You might want to use this option to create a plane whose first dimension is a multiple of 32, so you can use **bitblt** on it. Default: the result returned by **(make-list** *rank* **:initial-value 1)**.

:initial-origins
A list of origins for the initial creation of the plane. Default: the result returned by **(make-list** *rank* **:initial-value 0)**.

Example:

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

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

plane-origin *plane* *Function*

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

plane-default *plane* *Function*

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

plane-extension *plane* *Function*

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

plane-aref *plane* &rest *subscripts* *Function*

plane-aref and **plane-ref** return the contents of a specified element of a plane. They differ only in the way they take their arguments; **plane-aref** takes the subscripts as arguments, while **plane-ref** takes a list of subscripts.

plane-ref *plane subscripts* *Function*

plane-aref and **plane-ref** return the contents of a specified element of a plane. They differ only in the way they take their arguments; **plane-aref** takes the subscripts as arguments, while **plane-ref** takes a list of subscripts.

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

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

plane-store *datum plane subscripts* *Function*

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

24.14 Maclisp Array Compatibility

The functions in this section are provided only for Maclisp compatibility, and should not be used in new programs.

Integer arrays do not exist (however, see Symbolics-Lisp's small-positive-integer arrays). Flonum arrays exist but you do not use them in the same way; no declarations are required or allowed. "Un-garbage-collected" arrays do not exist.

Readtables and obarrays are represented as arrays, but unlike Maclisp special array types are not used. Information about readtables and obarrays (packages) can be found elsewhere: See the function **read** in *Reference Guide to Streams, Files, and I/O*. See the function **intern**, page 605. There are no "dead" arrays, nor are Multics "external" arrays provided.

The **arraycall** function exists for compatibility but should not be used. See the function **aref**, page 244.

Subscripts are always checked for validity, regardless of the value of ***rset** and whether the code is compiled or not.

Currently, multidimensional arrays are stored in column-major order rather than row-major order as in Maclisp. See the section "Multidimensional Arrays", page 237. This issue is discussed further in that section.

loadarrays and **dumparrays** are not provided. However, arrays can be put into compiled code files. See the section "Putting Data in Compiled Code Files" in *Reference Guide to Streams, Files, and I/O*.

The ***rearray** function is not provided, since not all of its functionality is available in Symbolics-Lisp. 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** and ***array** 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 *symbol type* &rest *dims* *Macro*

This creates an **art-q** type array in **default-array-area** with the given dimensions. (That is, *dims* is given to **make-array** as its first argument.) *type* is ignored. If *symbol* is **nil**, the array is returned; otherwise, the array is put in the function cell of *symbol*, and *symbol* is returned.

***array** *symbol type* &rest *dims* *Function*

This is just like **array**, except that all of the arguments are evaluated.

arraycall *ignore array* &rest *subscripts* *Function*

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

25. Characters

25.1 Character Objects

Zetalisp has always used positive integers to represent characters.

<, +, and **ldb** to perform various operations on characters. Common Lisp, on the other hand, has a separate data type for characters and specialized functions for operations on characters. In Common Lisp, it is possible to distinguish unambiguously between an integer and a character; characters print out in #\ notation.

This incompatibility between Zetalisp and Common Lisp affects strings as well as characters. A string is defined to be a one-dimensional array of characters, so in Zetalisp **aref** of a string returns an integer, but in Common Lisp **aref** of a string returns a character.

Eventually Zetalisp will be replaced with Symbolics Common Lisp, an extension of the standard Common Lisp language that also contains all of the advanced features of Zetalisp. System programs and most user programs will be written in Symbolics Common Lisp; old programs will continue to be supported by a Zetalisp compatibility package. To make it practical for Zetalisp and Symbolics Common Lisp programs to coexist in the same world and call each other freely, it is necessary for them to use compatible data types. If the two languages used two different representations for characters, they could not coexist conveniently, as characters and strings are ubiquitous throughout the Symbolics Lisp Machine system.

For this reason, in a future major release Zetalisp will be changed incompatibly to use a separate data type for characters, as Common Lisp does.

25.1.1 Character Object Details

A character object is a structured object containing several fields. Accessor functions, described later, are provided to extract and modify the fields. In an abstract sense the fields of a character object are:

code	the actual character, such as "upper-case A".
style	a modification of the character such as "italic" or "large".
bits	control, meta, super, and hyper.

In addition there are some derived fields, whose values depend on the values of the three fields listed above. For information about derived fields: See the section "Character Sets and Character Styles", page 267. See the section "The Device-font and Subindex Derived Fields", page 268.

Common Lisp calls the *style* field the *font* field. Within Symbolics-Lisp, the word "font" is not used because it has misleading prior associations in Zetalisp.

The precise meaning of the *code* and *style* fields might not be clear. Characters that are recognizably distinct always have different character codes. For example, the Roman a and the Greek α have two different character codes. The character code, which specifies the fundamental identity of a character, is modified by a style specification and by modifier bits from the keyboard. A modification of a character that leaves it recognizably the same is expressed in the style field and does not change the character code. For example, the Roman a, the bold **a**, and the italic *a* all have the same character code. The style field also expresses such attributes of a character as its displayed size and the typeface used, for example, whether it has serifs.

An operational definition of the difference between the *code* and *style* fields is provided by the **char-equal** function, which compares character codes but ignores the style and the bits. **char-equal** also ignores distinctions of alphabetic case. Because user-visible character comparisons, such as the Search and Replace commands in the editor, compare characters with **char-equal** these commands ignore differences in character style. In the Zetalisp releases that use integers to represent characters, set/style distinction is not fully implemented; therefore, a and α might be treated as the same character.

eq is not well-defined on character objects. Changing a field of a character object gives you a "new copy" of the object; it never modifies somebody else's "copy" of "the same" character object. In this way character objects are just like integers with fields accessed by **ldb** and changed by **dpb**. Because **eq** is not well-defined on character objects, they should be compared for identity with the **eql** function, not the **eq** function. This statement is true of integers as well. Integers can also be compared with =, but = is only for numbers and does not work for character objects. Currently on the 3600 family of machines, **eq** and **eql** are equivalent for characters, just as they are equivalent for integers, but programs should not be written to depend on this, for two reasons:

- "Extended" character objects could be introduced in the future, standing in the same relationship to "basic" character objects as bignums do to fixnums.
- **eq** might not work for characters in other implementations of Common-Lisp-compatible Lisp dialects.

In Zetalisp releases that use integers to represent characters, functions are provided to manipulate integers as if they were characters. In Zetalisp releases following the switch to character objects, the same functions will manipulate characters, providing compatibility.

25.1.2 Character Sets and Character Styles

The *code* field of a character can be broken down into a character set and an index into that character set. These are *derived fields* of a character.

A *character set* is a set of related characters that are recognizably different from other characters. Examples of character sets are the standard Symbolics Lisp Machine character set including the Roman alphabet and other characters, Cyrillic (the Cyrillic alphabet), and Japanese (comprising a large set of Kanji characters plus two syllabaries or alphabets). Not all character sets need contain the same number of characters. The indices in the standard character set range from 0 to 255, whereas the indices in the Kanji character set range from 0 to about 8000.

The standard Symbolics Lisp Machine character set is an upward-compatible extension of the 96 Common Lisp standard characters and the 6 Common Lisp semi-standard characters. It is almost an upward-compatible extension of ASCII; it uses a single Newline character and omits the ASCII control characters.

Character styles also exist. Among our existing fonts, Times-Roman, Centuryschoolbook, Jess, and CPTFONT are not different character sets; they are different styles of the Roman character set. Some styles can be applied to more than one character set; for example most character sets can be made boldface. It is possible to mix styles together; for example, a character can simultaneously be bold, italic, and 24 points high.

Format-effector characters such as Return, Tab, and Space exist only in the standard character set, but can be modified by styles that make them geometrically compatible with other character sets.

When comparing characters, there is no intrinsic ordering between characters in different character sets. Two characters of different character sets are never equal. Less-than is not well-defined between them. Within a single character set, less-than is defined so that characters (and strings) can be sorted alphabetically.

In Zetalisp releases that use integers to represent characters, the concepts of character set and character style are confused and conflated into a single concept: "font". Consequently there are no formal character-set and character-style objects in these pre-character object releases, just informal "font numbers". Furthermore, the exact meaning of these numbers depends on whether the Japanese system is loaded. Effectively, there is only one character set in Zetalisp releases that use integers to represent characters, and the "font" number is a style. However, when the optional Japanese system is loaded, there are two character sets — standard and Kanji — and the "font" number specifies both set and style. See the section "Support for Nonstandard Character Sets", page 275. Prior to the switch to character objects, no functions exist for dealing with character sets and styles. Programs that depend on this cannot be compatible between the two releases without source changes.

Several functions are based on Common Lisp functions that take an optional argument named "font". In Zetalisp releases that use integers to represent

characters, these functions do not take such an argument, since its meaning would be unclear and in any case it would change incompatibly in releases following the switch to character objects. After the switch to character objects has been made, these functions will probably permit either a character set or a style, or both, to be specified by optional arguments.

25.1.3 The Device-font and Subindex Derived Fields

In Zetalisp releases that use integers to represent characters, there are two additional *derived fields* of a character: the *device-font* number and the *subindex*. These two fields are derived from the *code* and *style* fields. Together they describe how to portray the character on an output device. Note: Programs that do output are not normally concerned with these fields; only programs that implement output devices need to know about them. Device-fonts will not exist after the switch to character objects has occurred. At that time, any program that uses them will have to be changed.

The *device-font* number is an integer that selects a device-dependent font; the *subindex* then selects a particular character image from that font. There is potentially a different device-font for each combination of character set, style, and output device. Each output device (such as a window or an LGP) has a table that maps *device-font* numbers into actual device-fonts.

The *subindex* can be an integer between 0 and 255. A character set can contain any number of characters; most character sets contain 256 or fewer characters, but the Kanji character set contains about 8000. A *device-font* always contains 256 characters, thus a large character set requires several device-fonts to portray all of the characters in that character set.

char-device-font accesses the *device-font* field and **char-subindex** accesses the *subindex* field of the specified character.

25.1.4 Two Kinds of Characters

A problem to be aware of when writing code to deal with characters is that releases that use integers to represent characters have two incompatible kinds of characters.

One kind, associated with the **%%kbd-** byte specifiers, is used for characters from the keyboard. The other kind, associated with the **%%ch-** byte specifiers, is used for characters in files, editor buffers, and strings. The keyboard characters can contain modifier bits, such as Control and Meta; the file characters can contain a device-font. The same bits in the number are used for both purposes, so in releases that use integers to represent characters, you cannot have a character with both bits and a font. Furthermore each character-processing function assumes that it was given a particular type of character as its argument; it has no way to tell which kind of character the caller intended, since all characters are just represented as numbers. Most functions assume file characters. Some functions work on either kind of

character, as long as all arguments are of the same kind, because they treat the bits and device-font attributes identically. If you are guaranteed to be dealing with characters in the common intersection of the two kinds, that is, characters whose bits and device-font attributes are both zero, you don't need to be concerned with these issues.

An example of the way you can get into trouble now is that **(alpha-char-p #\c-A)** returns **t**. Common Lisp specifies that it is supposed to return **nil**. In releases that use integers to represent characters, **alpha-char-p** expects a file character, so it regards the "control" bit as being a font number and ignores it. All problems of this type will be fixed by the switch to character objects, but in the meantime you need to be aware of them.

25.2 Character Fields

char-code *char* *Function*

Returns the code field of *char*, ignoring font. By default, the character code is the **%%ch-char** field. **char-code** works for both keyboard and file characters.

char-bits *char* *Function*

Returns the bits field of *char*. You can use **setf** on **(char-bits** *access-form*). **char-bits** works for keyboard characters only.

char-device-font *char* *Function*

Returns the device-font number field of *char*. This function works for file characters only.

char-subindex *char* *Function*

Returns the subindex field of *char*. This function works for both keyboard and file characters.

char-bit *char name* *Function*

Returns **t** if the specified bit is set in *char*, otherwise it returns **nil**. *name* can be **:control**, **:meta**, **:super**, **:hyper**, or **:shift**. **:shift** is a pseudo-bit that is not directly represented in the **char-bits** field. You can use **setf** on **(char-bit** *access-form name*). **char-bit** works for keyboard characters only.

set-char-bit *char name value* *Function*

Changes a bit in *char* and returns the new character. If *name* is **:shift**, *char* must be a letter with a nonzero **char-bits** field. *value* is **nil** to clear the bit or non-**nil** to set it. This function works for keyboard characters only.

code-char *code* &optional (*bits* **0**) *Function*
Constructs a character given its fields.

make-char *char* &optional (*bits* **0**) *Function*
Sets the bits field of *char*.

25.3 Character Predicates

graphic-char-p *char* *Function*
Returns **t** if *char* does not have any control bits set and is not a format effector. This function works for keyboard characters only.

upper-case-p *char* *Function*
Returns **t** if *char* is an upper-case letter. This function works for file characters only.

lower-case-p *char* *Function*
Returns **t** if *char* is a lower-case letter. This function works for file characters only.

both-case-p *char* *Function*
Returns **t** if *char* is a letter that exists in another case. This function works for file characters only.

alpha-char-p *char* *Function*
Returns **t** if *char* is a letter. This function works for file characters only.

digit-char-p *char* &optional (*radix* **10**) *Function*
Returns **t** if *char* is a valid digit in the specified radix. The value, if non-**nil**, is the weight of that digit (a number from zero to one less than the radix). This function works for file characters only. See the function **digit-char**, page 273.

alphanumericp *char* *Function*
Returns **t** if *char* is a letter or a base-10 digit. This function works for file characters only.

25.4 Character Comparisons

25.4.1 Character Comparisons Affected by Case, Style, and Bits

char= *char1* &rest *more-chars* *Function*
This is a comparison predicate that compares characters exactly, depending on

all fields including bits, style, and alphabetic case. This function works for both keyboard and file characters.

char≠ *char1* &rest *more-chars* *Function*
This is a comparison predicate that compares characters exactly, depending on all fields including bits, style, and alphabetic case. This function works for both keyboard and file characters.

char< *char1* &rest *more-chars* *Function*
This is a comparison predicate that compares characters exactly, depending on all fields including bits, style, and alphabetic case. This function works for both keyboard and file characters.

char> *char1* &rest *more-chars* *Function*
This is a comparison predicate that compares characters exactly, depending on all fields including bits, style, and alphabetic case. This function works for both keyboard and file characters.

char≤ *char1* &rest *more-chars* *Function*
This is a comparison predicate that compares characters exactly, depending on all fields including bits, style, and alphabetic case. This function works for both keyboard and file characters.

char≥ *char1* &rest *more-chars* *Function*
This is a comparison predicate that compares characters exactly, depending on all fields including bits, style, and alphabetic case. This function works for both keyboard and file characters.

25.4.2 Character Comparisons Ignoring Case, Style, and Bits

char-equal *char1 char2* *Function*
This is the primitive for comparing characters for equality; many of the string functions call it. *char1* and *char2* must be integers. **char-equal** ignores case and style, returning **t** if the characters are equal. Otherwise it returns **nil**.

char-not-equal *char1 char2* *Function*
This is a comparison predicate that compares only the code field and ignores distinctions of alphabetic case. This function works for both keyboard and file characters.

char-lessp *char1 char2* *Function*
This is the primitive for comparing characters for order; many of the string functions call it. *char1* and *char2* must be integers. The result is **t** if *char1* comes before *char2* ignoring case and font, otherwise **nil**. See the section "The Character Set" in *Reference Guide to Streams, Files, and I/O*. Details of the ordering of characters are in that section.

char-greaterp *char1 char2* *Function*

This is a comparison predicate that compares only the code field and ignores distinctions of alphabetic case. This function works for both keyboard and file characters.

char-not-greaterp *char1 char2* *Function*

This is a comparison predicate that compares only the code field and ignores distinctions of alphabetic case. This function works for both keyboard and file characters.

char-not-lessp *char1 char2* *Function*

This is a comparison predicate that compares only the code field and ignores distinctions of alphabetic case. This function works for both keyboard and file characters.

25.5 Character Conversions

character *x* *Function*

character coerces *x* to a single character, represented as an integer. If *x* is a number, it is returned. If *x* is a string or an array, its first element is returned. If *x* is a symbol, the first character of its pname is returned. Otherwise, an error occurs. See the section "The Character Set" in *Reference Guide to Streams, Files, and I/O*. The way characters are represented as integers is explained in that section.

char-int *char* *Function*

Converts *char* to an integer. This function works for both keyboard and file characters.

int-char *integer* *Function*

Converts *integer* to a character. This function works for both keyboard and file characters.

char-upcase *char* *Function*

If *char*, which must be an integer, is a lowercase alphabetic character in the standard character set, **char-upcase** returns its uppercase form; otherwise, it returns *char*. If font information is present it is preserved. The result of **char-upcase** is undefined for characters with modifier bits.

char-downcase *char* *Function*

If *char*, which must be an integer, is an uppercase alphabetic character in the standard character set, **char-downcase** returns its lowercase form; otherwise, it returns *char*. If font information is present it is preserved. The result of **char-downcase** is undefined for characters with modifier bits.

char-flipcase *char* *Function*

If *char*, which must be an integer, is a lowercase alphabetic character in the standard character set, **char-flipcase** returns its uppercase form. If *char* is an uppercase alphabetic character in the standard character set, **char-flipcase** returns its lowercase form. Otherwise, it returns *char*. If font information is present it is preserved. The result of **char-flipcase** is undefined for characters with modifier bits.

digit-char *weight* &optional (*radix* **10**) *Function*

Returns the character that represents a digit with a specified weight *weight*. Returns **nil** if *weight* is not between 0 and **(1-** *radix***)** or *radix* is not between 2 and 36.

See the function **digit-char-p**, page 270.

alphabetic-case-affects-string-comparison *Variable*

This variable is obsolete, and should always be left set to **nil**. Binding this variable non-**nil** makes some functions behave differently from the documentation.

Use **char=** to compare characters if you want alphabetic case to affect the result, otherwise use **char-equal**. Similarly, use **string=** or **string-equal** to compare strings.

25.6 Character Names

char-name *char* *Function*

Returns **nil** or the name of *char* (a string). **char-name** ignores bits and style.

name-char *string* *Function*

Returns a character given its name. **name-char** does not recognize names with modifier bit prefixes such as "hyper-space".

25.7 Mouse Characters

mouse-char-p *char* *Function*

Returns **t** if *char* is a mouse-character, representing the clicking of a mouse button. See the section "The Character Set" in *Reference Guide to Streams, Files, and I/O*. This function works for keyboard characters only.

char-mouse-button *char* *Function*

Returns the zero-origin button number. See the section "The Character Set" in *Reference Guide to Streams, Files, and I/O*. This function works for keyboard characters only.

char-mouse-n-clicks *char* *Function*

Returns one less than the number of times the mouse button was clicked. See the section "The Character Set" in *Reference Guide to Streams, Files, and I/O*. This function works for keyboard characters only.

make-mouse-char *button n-clicks* &optional (*bits* **0**) *Function*

Constructs a mouse character given its fields. See the section "The Character Set" in *Reference Guide to Streams, Files, and I/O*. This function produces keyboard characters only.

25.8 ASCII Characters

ascii-code *arg* *Function*

Returns an integer that is the ASCII code named by *arg*. If *arg* is a character, **char-to-ascii** is called. Otherwise, *arg* can be a string that is the name of one of the ASCII special characters.

Valid ASCII special character names are listed below. All numbers are in octal.

NUL 000	HT 011	DC1 021	SUB 032
SOH 001	LF 012	DC2 022	ESC 033
STX 002	NL 012	DC3 023	ALT 033
ETX 003	VT 013	DC4 024	FS 034
EOT 004	FF 014	NAK 025	GS 035
ENQ 005	CR 015	SYN 026	RS 036
ACK 006	SO 016	ETB 027	US 037
BEL 007	SI 017	CAN 030	SP 040
BS 010	DLE 020	EM 031	DEL 177
TAB 011			

char-to-ascii *char* *Function*

Converts *char* to the corresponding ASCII code. #\Return is converted to the ASCII CR character; the caller must supply an LF if desired. This function works only for characters with neither bits nor fonts. See the section "ASCII Strings", page 290.

ascii-to-char *code* *Function*

Converts *code* (an ASCII code) to the corresponding character. The caller must ignore LF after CR if desired. See the section "ASCII Strings", page 290.

25.9 Support for Nonstandard Character Sets

Symbolics-Lisp has a limited facility for defining multiple character sets, which will be replaced with a real multiple character set feature after the switch to character objects has been made. The standard Symbolics Lisp Machine character set is discussed in another section. See the section "The Character Set" in *Reference Guide to Streams, Files, and I/O*. You can have both standard and nonstandard character sets in multiple fonts.

Two functions, **char-standard** and **char-code**, provide support for nonstandard character sets that do not have the usual interpretations of case and font.

char-standard *char* *Function*

Returns **t** if *char* is a standard character, with the usual interpretations of case and font. By default, **char-standard** always returns **t**. You can redefine this function to introduce multiple character sets. This function will be removed after the switch to character objects has been made.

See the function **char-code**, page 269. You can redefine **char-code** to introduce multiple character sets. Always use **char-code** instead of **(ldb %%ch-char** *char***)** to determine the character code so that your programs can run without modification when the switch to character objects is made.

char-standard and **char-code** are hooks. You can redefine these functions to examine the value of the **%%ch-font** field of their argument and to use this in computing their result.

Make sure that replacement definitions for **char-standard** and **char-code** are thoroughly debugged using different names before redefining them. Defective versions of these functions can cause the system to crash.

By redefining these functions, you can control the behavior of **char-equal**, **char-lessp**, **char-upcase**, **string-equal**, **string-search**, and other system functions that ignore font information or that only make sense for the standard character set.

If **char-standard** is redefined and if
(eq (char-standard c1) (char-standard c2)) returns **nil**, then
(char-equal c1 c2) returns **nil**. **char-code** usually returns a number between 0 and **(dpb -1 %%ch-char 0)**, inclusive. If it is redefined, it can return numbers greater than **(dpb -1 %%ch-char 0)**.

For example, suppose you have three different Greek fonts and you want to define a Greek character set. Characters in fonts 1, 2, and 3 are assumed to be Greek. Characters in font 0 and in fonts 4 through 255 are assumed to be in the standard character set. Characters in font 0 must always be in the standard character set.

Suppose that the values of the **%%ch-char** field of the characters α, β, and γ are the same as that of the characters A, B, and C. We want

(char-equal #/A #/α) and **(string-equal "ABC" "αβγ")** to return **nil**. But we want **(string-equal "αβγ" "αβγ")** to return **t**, even if the first string and the second string are in two different fonts.

The system provides these definitions of **char-standard** and **char-code**:

```
(defun char-standard (ignore) t)
(defun char-code (char) (ldb %%ch-char char))
```

You can define a Greek character set, allowing three fonts using font numbers 1, 2, and 3, by using these definitions instead:

```
(defun char-standard (char)
  (let ((font (ldb %%ch-font char)))
    (or (zerop font) (≥ font 4))))

(defun char-code (char)
  (let ((font (ldb %%ch-font char))
        (code (ldb %%ch-char font)))
    (if (or (zerop font) (≥ font 4))
        code
        (dpb 1 %%ch-font code))))
```

You can define multiple character sets in a similar manner.

26. Strings

Strings are a type of array that represent a sequence of characters. The printed representation of a string is its characters enclosed in quotation marks, for example, "**foo bar**". Strings are constants, that is, evaluating a string returns that string. Strings are the right data type to use for text processing.

Strings are arrays of type **art-string**, where each element holds an eight-bit unsigned integer. This is because characters are represented as integers, 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 integer; the extra bits allow for multiple fonts or an expanded character set.

See the section "The Character Set" in *Reference Guide to Streams, Files, and I/O*. The way characters work, including multiple fonts and the extra bits from the keyboard, is explained in that section. Note that you can type in the integers that represent characters using "#/" and "#\"; for example, **#/f** reads in as the integer that represents the character "f", and **#\return** reads in as the integer that represents the special "return" character. See the section "Sharp-sign Reader Macros", page 27. Details of this syntax are explained there.

The functions described in this section provide a variety of useful operations on strings. In place of a string, most of these functions accept a symbol or an integer as an argument, and coerce it into a string. Given a symbol, its print name, which is a string, is used. Given an integer, a one-character string containing the character designated by that integer is used.

Since strings are arrays, the usual array-referencing function **aref** is used to extract the characters of the string as integers. 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 Symbolics-Lisp, everything is zero-based.

It is also valid 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 propagate to. When you are making strings that you intend to change later, you probably want to create an array with a fill-pointer so that you can change the length of the string as well as the contents. See the section "Array Leaders", page 238. The length of a string is always computed using **array-active-length**, so that if a string has a fill-pointer, its value is used as the length.

See also **intern**, which given a string returns "the" symbol with that print name.

26.1 Basic String Operations

string *x* *Function*

string coerces *x* into a string. Most of the string functions apply this to their string arguments. If *x* is a string, it is returned. If *x* is a symbol, its pname is returned. If *x* is a nonnegative integer less than **200000** octal, a one-character-long string containing it is created and returned. If *x* is a pathname, the "string for printing" is returned. See the section "Naming of Files" in *Reference Guide to Streams, Files, and I/O*. Otherwise, an error is signalled.

If you want to get the printed representation of an object into the form of a string, this function is *not* what you should use. You can use **format**, passing a first argument of **nil**. You might also want to use **with-output-to-string**.

string-length *string* *Function*

string-length returns the number of characters in *string*, which must be a string or an object that can be coerced into a string. See the function **string**, page 278. **string-length** returns the **array-active-length** if *string* is a string, or the **array-active-length** of the pname if *string* is a symbol.

substring *string from* &optional *to* (*area* **nil**) *Function*

This extracts a substring of *string*, starting at the character specified by *from* and going up to but not including the character specified by *to*. *string* is a string or an object that can be coerced to a string. See the function **string**, page 278. *from* and *to* are 0-origin indices. The length of the returned string is *to* minus *from*. If *to* is not specified it defaults to the length of *string*. The area in which the result is to be consed can be optionally specified. Example:

```
(substring "Nebuchadnezzar" 4 8) => "chad"
```

nsubstring *string from* &optional *to* (*area* **nil**) *Function*

nsubstring is the same as **substring** except that the substring is not copied; instead an indirect array is created that shares part of the argument *string*. See the section "Indirect Arrays", page 240. Modifying one string modifies the other.

string is a string or an object that can be coerced to a string. Since **nsubstring** is destructive, coercion should be used with care since a string internal to the object might be modified. See the function **string**, page 278.

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** 4 characters long. So you should not use this just "for efficiency"; it is intended for uses in which it is important to have a substring that, if modified, causes the original string to be modified too.

string-append &rest *strings* *Function*

Any number of strings are copied and concatenated into a single string. *strings* are strings or objects that can be coerced to strings. See the function **string**, page 278. With a single argument, **string-append** simply copies it. The result is an array of the same type as the argument with the greatest number of bits per element. For example, if the arguments are arrays of type **art-string** and **art-fat-string**, an array of type **art-fat-string** is returned. **string-append** can be used to copy and concatenate any type of one-dimensional array. Example:

```
(string-append #/! "foo" #/!) => "!foo!"
```

string-nconc *to-string* &rest *strings* *Function*

string-nconc is like **string-append** except that instead of making a new string containing the concatenation of its arguments, **string-nconc** modifies its first argument. *to-string* must be a string with a fill-pointer so that additional characters can be tacked onto it. Compare this with **array-push-extend**. The value of **string-nconc** is *to-string* or a new, longer copy of it; in the latter case the original copy is forwarded to the new copy (see **adjust-array-size**). Unlike **nconc**, **string-nconc** with more than two arguments modifies only its first argument, not every argument but the last.

string-nconc-portion *to-string {from-string from to} ...* *Function*

Adds information onto a string without consing intermediate substrings. *to-string* must be a string with a fill-pointer so that additional characters can be added onto it. The remaining arguments can be any number of "string portion specs", which are string/from/to triples. *from* and *to* are required but can be **nil** and **nil**. Even though the arguments are called strings, they can be anything that can be coerced to a string with **string** (for example, symbols or characters).

The value of **string-nconc-portion** is *to-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**).

string-nconc-portion is like **string-nconc** except that it takes parts of strings without consing substrings.

Example:

```
(let ((a (make-array 10 :type 'art-string :fill-pointer 0)))
  (format t "~A"
          (string-nconc-portion a 'xxxfoobar 3 nil
                                #\sp nil nil
                                "tempstuff" 0 4)))
     => FOOBAR temp
```

string-nconc-portion uses **array-push-portion-extend** internally, which uses **adjust-array-size** to take care of growing the *to-string* if necessary.

string-trim *char-set string* *Function*

This returns a **substring** of *string*, with all characters in *char-set* stripped off the beginning and end. *string* is a string or an object that can be coerced to a string. See the function **string**, page 278. *char-set* is a set of characters, which can be represented as a list of characters or a string of characters. Example:

```
(string-trim '(#\sp) "  Dr. No  ") => "Dr. No"
(string-trim "ab" "abbafooabb") => "foo"
```

string-left-trim *char-set string* *Function*

This returns a **substring** of *string*, with all characters in *char-set* stripped off the beginning. *string* is a string or an object that can be coerced to a string. See the function **string**, page 278. *char-set* is a set of characters, which can be represented as a list of characters or a string of characters.

string-right-trim *char-set string* *Function*

This returns a **substring** of *string*, with all characters in *char-set* stripped off the end. *string* is a string or an object that can be coerced to a string. See the function **string**, page 278. *char-set* is a set of characters, which can be represented as a list of characters or a string of characters.

string-reverse *string* *Function*

Returns a copy of *string* with the order of characters reversed. This reverses a one-dimensional array of any type. If *string* is not a string or another one-dimensional array, it is coerced into a string. See the function **string**, page 278.

string-nreverse *string* *Function*

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

If *string* is not a string or another one-dimensional array, it is coerced into a string. Since **string-nreverse** is destructive, coercion should be used with care since a string internal to the object might be modified. See the function **string**, page 278.

string-pluralize *string* *Function*

string-pluralize returns a string containing the plural of the word in the argument *string*. Any added characters go in the same case as the last character of *string*. *string* is a string or an object that can be coerced to a string. See the function **string**, page 278. Example:

```
(string-pluralize "event") => "events"
(string-pluralize "Man") => "Men"
(string-pluralize "Can") => "Cans"
(string-pluralize "key") => "keys"
(string-pluralize "TRY") => "TRIES"
```

For words with multiple plural forms depending on the meaning, **string-pluralize** cannot always do the right thing.

parse-number *string* &optional (*from* **0**) (*to nil*) (*radix nil*) (*fail-if-not-whole-string nil*) *Function*

parse-number takes a string and "reads" a number from it. *string* must be a string. It returns two values: the number found (or **nil**) and the character position of the next unparsed character in the string. It returns **nil** when the first character that it looks at cannot be part of a number. The function currently does not handle anything but integers. (**read-from-string** is a more general function that uses the Lisp reader; **prompt-and-read** reads a number from the keyboard.)

```
(parse-number "123    ")  =>  123 3
(parse-number "   123")  =>  NIL 0
(parse-number "-123")  => -123 4
(parse-number "25.3")  =>  25 2
(parse-number "$$$123" 3 4)  =>  1 4
(parse-number "123$$$" 0 nil nil nil) => 123 3
(parse-number "123$$$" 0 nil nil t) => NIL 0
```

Four optional arguments:

from The character position in the string to start parsing. The default is the first one, position 0.

to The character position past the last one to consider. The default, **nil**, means the end of the string.

radix The radix to read the string in. The default, **nil**, means base 10.

fail-if-not-whole-string
The default is **nil**. **nil** means to read up to the first character that is not a digit and stop there, returning the result of the parse so far. **t** means to stop at the first nondigit and to return **nil** and 0 length if that is not the end of the string.

number-into-array *array number* &optional (*radix* **base**) (*at-index* **0**) (*min-columns* **0**) *Function*

Deposits the printed representation of *number* into *array*. **number-into-array** is the inverse of **parse-number**. It has three optional arguments:

radix The radix to use when converting the number into its printed representation. It defaults to **base**.

at-index The character position in the array to start putting the number. If the number contains less characters than *min-columns*, the number is right-justified within the array. If the number contains more characters than *min-columns*, *min-columns* is ignored. An error is signalled if the number contains more characters than the length of the array minus *at-index*. The default is the first position, position 0.

min-columns The minimum number of characters required for the printed representation of the number.

The following example puts 23453243 into **string** starting at character position 5. Since *min-columns* is 10, the number is preceded by two spaces.

```
(let ((string (make-array 20. :type 'art-string :initial-value #\X)))
  (number-into-array string 23453243. 10. 5. 10.)
    string)

=>  "XXXXX  23453243XXXXX"
```

26.2 String Comparisons

26.2.1 String Comparisons Affected by Case, Style, and Bits

string= *string1 string2* &optional (*idx1* **0**) (*idx2* **0**) *lim1 lim2* *Function*
This is a comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including bits, style, and alphabetic case.

string≠ *string1 string2* &optional (*idx1* **0**) (*idx2* **0**) *lim1 lim2* *Function*
This is a comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including bits, style, and alphabetic case.

string< *string1 string2* &optional (*idx1* **0**) (*idx2* **0**) *lim1 lim2* *Function*
This is a comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including bits, style, and alphabetic case.

string> *string1 string2* &optional (*idx1* **0**) (*idx2* **0**) *lim1 lim2* *Function*
This is a comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including bits, style, and alphabetic case.

string≤ *string1 string2* &optional (*idx1* **0**) (*idx2* **0**) *lim1 lim2* *Function*
This is a comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including bits, style, and alphabetic case.

string≥ *string1 string2* &optional (*idx1* **0**) (*idx2* **0**) *lim1 lim2* *Function*
This is a comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including bits, style, and alphabetic case.

%string= *string1 index1 string2 index2 count* *Function*
This is a low-level string comparison, possibly more efficient than the other comparisons. Its only current efficiency advantage is its simplified arguments and minimized type-checking.

string-exact-compare *string1 string2* &optional (*idx1* **0**) (*idx2* **0**) *lim1 lim2* *Function*
This is a comparison predicate that compares two strings or substrings of them, exactly, depending on all fields including bits, style, and alphabetic case. **string-exact-compare** returns:

- a positive number if *string1* > *string2*
- zero if *string1* = *string2*
- a negative number if *string1* < *string2*

sys:%string-exact-compare *string1 index1 string2 index2 count* *Function*
This is a low-level string comparison, possibly more efficient than the other comparisons. Its only current efficiency advantage is its simplified arguments and minimized type-checking. **sys:%string-exact-compare** returns:

- a positive number if *string1* > *string2*
- zero if *string1* = *string2*
- a negative number if *string1* < *string2*

26.2.2 String Comparisons Ignoring Case, Style, and Bits

string-equal *string1 string2* &optional (*idx1* **0**) (*idx2* **0**) *lim1 lim2* *Function*
string-equal compares two strings, returning **t** if they are equal and **nil** if they are not. The comparison ignores the extra "font" bits in 16-bit strings and ignores alphabetic case. **equal** calls **string-equal** if applied to two strings. *string1* and *string2* are strings or objects that can be coerced to strings. See the function **string**, page 278.

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-not-equal *string1 string2* &optional (*idx1* **0**) (*idx2* **0**) *lim1* *lim2* *Function*

This compares two strings or substrings of them, ignoring bits, style, and alphabetic case.

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

This compares two strings using alphabetical order (as defined by **char-lessp**). The result is **t** if *string1* is the lesser, or **nil** if they are equal or *string2* is the lesser.

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

This compares two strings or substrings of them, ignoring bits, style, and alphabetic case.

string-not-greaterp *string1 string2* &optional (*idx1* **0**) (*idx2* **0**) *lim1* *lim2* *Function*

This compares two strings or substrings of them, ignoring bits, style, and alphabetic case.

string-not-lessp *string1 string2* &optional (*idx1* **0**) (*idx2* **0**) *lim1* *lim2* *Function*

This compares two strings or substrings of them, ignoring bits, style, and alphabetic case.

%string-equal *string1 index1 string2 index2 count* *Function*

This is a low-level string comparison, possibly more efficient than the other comparisons. Its only current efficiency advantage is its simplified arguments and minimized type-checking. **%string-equal** 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 an integer, *count* can 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 integers to strings is not performed. This function is documented because certain programs that require high efficiency and are willing to pay the price of less generality might 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-compare *string1 string2* &optional (*idx1* **0**) (*idx2* **0**) *lim1* *lim2* *Function*

Compares the characters of *string1* starting at *idx1* and ending just below *lim1* with the characters of *string2* starting at *idx2* and ending just below *lim2*. The comparison is in alphabetical order. *string1* and *string2* are strings or objects that can be coerced to strings. See the function **string**, page 278. *lim1* and *lim2* default to the lengths of the strings. **string-compare** returns:

- a positive number if *string1* > *string2*
- zero if *string1* = *string2*
- a negative number if *string1* < *string2*

If the strings are not equal, the absolute value of the number returned is one more than the index (in *string1*) at which the difference occurred.

sys:%string-compare *string1 index1 string2 index2 count* *Function*

This is a low-level string comparison, possibly more efficient than the other comparisons. Its only current efficiency advantage is its simplified arguments and minimized type-checking.

26.3 String Conversions

string-upcase *string* &optional (*from* **0**) *to* (*copy-p* **t**) *Function*

string is a string or an object that can be coerced to a string. See the function **string**, page 278. If *copy-p* is not **nil**, returns a copy of *string*, with lowercase alphabetic characters replaced by the corresponding uppercase characters. If *copy-p* is **nil**, uppercases characters in *string* itself and then returns the modified *string*. *from* is the index in *string* at which to begin uppercasing characters. If *to* is supplied, it is used in place of (**array-active-length** *string*) as the index one greater than the last character to be uppercased. Characters not in the standard character set are unchanged.

string-downcase *string* &optional (*from* **0**) *to* (*copy-p* **t**) *Function*

string is a string or an object that can be coerced to a string. See the function **string**, page 278. If *copy-p* is not **nil**, returns a copy of *string*, with uppercase alphabetic characters replaced by the corresponding lowercase characters. If *copy-p* is **nil**, lowercases characters in *string* itself and then returns the modified *string*. *from* is the index in *string* at which to begin lowercasing characters. If *to* is supplied, it is used in place of (**array-active-length** *string*) as the index one greater than the last

character to be lowercased. Characters not in the standard character set are unchanged.

string-flipcase *string* &optional (*from* **0**) *to* (*copy-p* **t**) *Function*
string is a string or an object that can be coerced to a string. See the function **string**, page 278. If *copy-p* is not **nil**, **string-flipcase** returns a copy of *string*, with uppercase alphabetic characters replaced by the corresponding lowercase characters, and with lowercase alphabetic characters replaced by the corresponding uppercase characters. If *copy-p* is **nil**, **string-flipcase** exchanges the case of characters in *string* itself and then returns the modified *string*. *from* is the index in *string* at which to begin exchanging the case of characters. If *to* is supplied, it is used in place of **(array-active-length** *string***)** as the index one greater than the last character whose case is to be exchanged. Characters not in the standard character set are unchanged.

string-capitalize-words *string* &optional (*copy-p* **t**) *Function*
Transforms *string* by changing hyphens to spaces and capitalizing each word. *string* is a string or an object that can be coerced to a string. See the function **string**, page 278.

```
(string-capitalize-words "Lisp-listener") => "Lisp Listener"
(string-capitalize-words "LISP-LISTENER")  => "Lisp Listener"
(string-capitalize-words "lisp--listener") => "Lisp  Listener"
(string-capitalize-words "symbol-processor-3") => "Symbol Processor 3"
```

copy-p indicates whether to return a copy of the string argument or to modify the argument itself. The default, **t**, returns a copy.

26.4 String Searching

26.4.1 String Searching Affected by Case, Style, and Bits

string-search-exact-char *char string* &optional (*from* **0**) *to* *Function*
This searches a string or a substring for a character, comparing characters exactly and depending on all fields including bits, style, and alphabetic case.

string-search-not-exact-char *char string* &optional (*from* **0**) *to* *Function*
This searches a string or a substring for a character, comparing characters exactly and depending on all fields including bits, style, and alphabetic case.

string-reverse-search-exact-char *char string* &optional *from* (*to* **0**) *Function*
This searches a string or a substring for a character, comparing characters exactly and depending on all fields including bits, style, and alphabetic case.

string-reverse-search-not-exact-char *char string* &optional *from* (*to* **0**) *Function*

This searches a string or a substring for a character, comparing characters exactly and depending on all fields including bits, style, and alphabetic case.

string-search-exact *key string* &optional (*from* **0**) *to* (*key-start* **0**) *key-end* *Function*

This searches one string for another, comparing characters exactly and depending on all fields including bits, style, and alphabetic case. Substrings of either argument can be specified.

string-reverse-search-exact *key string* &optional *from* (*to* **0**) (*key-start* **0**) *key-end* *Function*

This searches one string for another, comparing characters exactly and depending on all fields including bits, style, and alphabetic case. Substrings of either argument can be specified.

%string-search-exact-char *char string start end* *Function*

This is a low-level string search, possibly more efficient than the other searching functions. Its only current efficiency advantage is its simplified arguments and minimized type-checking.

26.4.2 String Searching Ignoring Case, Style, and Bits

string-search-char *char string* &optional (*from* **0**) *to* *Function*

string-search-char searches through *string* starting at the index *from*, which defaults to the beginning, and returns the index of the first character that is **char-equal** to *char*, or **nil** if none is found. If the *to* argument is supplied, it is used in place of **(string-length** *string***)** to limit the extent of the search. *string* is a string or an object that can be coerced to a string. See the function **string**, page 278. Example:

```
(string-search-char #/a "banana") => 1
```

string-search-not-char *char string* &optional (*from* **0**) *to* *Function*

string-search-not-char searches through *string* starting at the index *from*, which defaults to the beginning, and returns the index of the first character which is *not* **char-equal** to *char*, or **nil** if none is found. If the *to* argument is supplied, it is used in place of **(string-length** *string***)** to limit the extent of the search. *string* is a string or an object that can be coerced to a string. See the function **string**, page 278. Example:

```
(string-search-not-char #/b "banana") => 1
```

string-reverse-search-char *char string* &optional *from* (*to* **0**) *Function*

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

string, and returns the index of the first character that is **char-equal** to *char*, or **nil** if none is found. Note that the index returned is from the beginning of the string, although the search starts from the end. If the *to* argument is supplied, it limits the extent of the search. *string* is a string or an object that can be coerced to a string. See the function **string**, page 278. Example:

```
(string-reverse-search-char #/n "banana") => 4
```

string-reverse-search-not-char *char string* &optional *from* (*to* **0**) *Function*

string-reverse-search-not-char searches through *string* in reverse order, starting from the index one less than *from*, which defaults to the length of *string*, and returns the index of the first character that is *not* **char-equal** to *char*, or **nil** if none is found. Note that the index returned is from the beginning of the string, although the search starts from the end. If the *to* argument is supplied, it limits the extent of the search. *string* is a string or an object that can be coerced to a string. See the function **string**, page 278. Example:

```
(string-reverse-search-not-char #/a "banana") => 4
```

string-search *key string* &optional (*from* **0**) *to* (*key-start* **0**) *key-end* *Function*

string-search searches for the string *key* in the string *string*, using **string-equal** to do the comparison. 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. *string* is a string or an object that can be coerced to a string. See the function **string**, page 278. Example:

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

string-reverse-search *key string* &optional *from* (*to* **0**) (*key-start* **0**) *key-end* *Function*

string-reverse-search searches for the string *key* in the string *string*, using **string-equal** to do the comparison. 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. *string* is a string or an object that can be coerced to a string. See the function **string**, page 278. Example:

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

%string-search-char *char string from to* *Function*

This is a low-level string search, possibly more efficient than the other searching functions. Its only current efficiency advantage is its simplified arguments and minimized type-checking. *string* must be an array and *char*, *from*, and *to* must be integers. 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-set *char-set string* &optional (*from* **0**) *to* *Function*

string-search-set searches through *string* looking for a character that is in *char-set*. The search begins at the index *from*, which defaults to the beginning. It returns the index of the first character that is **char-equal** to some element of *char-set*, or **nil** if none is found. If the *to* argument is supplied, it is used in place of **(string-length** *string***)** to limit the extent of the search. *char-set* is a set of characters, which can be represented as a list of characters or a string of characters. *string* is a string or an object that can be coerced to a string. See the function **string**, page 278. Example:

```
(string-search-set '(#/n #/o) "banana") => 2
(string-search-set "no" "banana") => 2
```

string-search-not-set *char-set string* &optional (*from* **0**) *to* *Function*

string-search-not-set searches through *string* looking for a character that is not in *char-set*. The search begins at the index *from*, which defaults to the beginning. It returns the index of the first character that is not **char-equal** to any element of *char-set*, or **nil** if none is found. If the *to* argument is supplied, it is used in place of **(string-length** *string***)** to limit the extent of the search. *char-set* is a set of characters, which can be represented as a list of characters or a string of characters. *string* is a string or an object that can be coerced to a string. See the function **string**, page 278. Example:

```
(string-search-not-set '(#/a #/b) "banana") => 2
```

string-reverse-search-set *char-set string* &optional *from* (*to* **0**) *Function*

string-reverse-search-set searches through *string* in reverse order, starting from the index one less than *from*, which defaults to the length of *string*, and returns the index of the first character that is **char-equal** to some element of *char-set*, or **nil** if none is found. Note that the index returned is from the beginning of the string, although the search starts from the end. If the *to* argument is supplied, it limits the extent of the search. *char-set* is a set of characters, which can be represented as a list of characters or a string of characters. *string* is a string or an object that can be coerced to a string. See the function **string**, page 278.

```
(string-reverse-search-set "ab" "banana") => 5
```

string-reverse-search-not-set *char-set string* &optional *from* (*to* **0**) *Function*

string-reverse-search-not-set searches through *string* in reverse order, starting from the index one less than *from*, which defaults to the length of *string*, and returns the index of the first character that is not **char-equal** to any element of *char-set*, or **nil** if none is found. Note that the index returned is from the beginning of the string, although the search starts from the end. If the *to* argument is supplied, it limits the extent of the search. *char-set* is a set of characters, which can be represented as a list of characters or a string of characters. *string* is a string or an object that can be coerced to a string. See the function **string**, page 278.

```
(string-reverse-search-not-set '(#/a #/n) "banana") => 0
```

26.5 ASCII Strings

string-to-ascii *lispm-string* *Function*

Converts *lispm-string* to an **art-8b** array containing ASCII character codes. See the section "ASCII Characters", page 274.

Example:

```
(string-to-ascii "hello") => #<ART-8B-5 24443106>
```

ascii-to-string *ascii-array* *Function*

Converts *ascii-array*, an **art-8b** array representing ASCII characters, into a Lisp string. See the section "ASCII Characters", page 274.

Example:

```
(ascii-to-string (make-array 5 :type art-8b
                               :initial-value (char-code #\x)))
        => "xxxxx"
```

26.6 I/O to Strings

The special forms in this section allow you to create I/O streams that input from or output to a string rather than a real I/O device. See the section "Introduction to Streams" in *Reference Guide to Streams, Files, and I/O*. I/O streams are documented there.

with-input-from-string (*var string* &optional *index limit*) *body...* *Special Form*

The form:

```
(with-input-from-string (var string)
    body)
```

evaluates the forms in *body* with the variable *var* bound to a stream that

reads characters from the string which is the value of the form *string*. The value of the special form is the value of the last form in its body.

The stream is a function that only works inside the **with-input-from-string** special form, so be careful what you do with it. You cannot use it after control leaves the body, and you cannot nest two **with-input-from-string** special forms and use both streams since the special-variable bindings associated with the streams conflict. It is done this way to avoid any allocation of memory.

After *string* you can 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 is set to the length of the string. Since *index* is updated it cannot be a general expression; it must be a variable or a **setf**able reference. The *index* is not updated in the event of an abnormal exit from the body, such as a **throw**. The value of *index* is not updated until **with-input-from-string** returns, so you cannot use its value within the body to see how far the reading has proceeded.

Use of the *index* feature prevents multiple values from being returned out of the body, currently.

```
(with-input-from-string (var string index limit)
    body)
```

uses the value of the form *limit*, if the value is not **nil**, in place of the length of the string. If you want to specify a *limit* but not an *index*, write **nil** for *index*.

with-output-to-string (*var* &optional (*string* **nil** *string-p*) *index*) *Special Form*
body...

This special form provides a variety of ways to send output to a string through an I/O stream.

```
(with-output-to-string (var)
  body)
```

evaluates the forms in *body* with *var* bound to a stream that saves the characters output to it in a string. The value of the special form is the string.

```
(with-output-to-string (var string)
  body)
```

appends its output to the string that is the value of the form *string*. (This is like the **string-nconc** function). The value returned is the value of the

last form in the body, rather than the string. Multiple values are not returned. *string* must have an array-leader; element 0 of the array-leader is used as the fill-pointer. If *string* is too small to contain all the output, **adjust-array-size** is used to make it bigger.

If characters with font information are output, *string* must be of type **art-fat-string**. See the section "**art-fat-string** Array Type", page 236.

```
(with-output-to-string (var string index)
  body)
```

is similar to the above except that *index* is a variable or **setf**able reference that contains the index of the next character to be stored into. It must be initialized outside the **with-output-to-string** and is updated upon normal exit. The value of *index* is not updated until **with-output-to-string** returns, so you cannot use its value within the body to see how far the writing has gotten. The presence of *index* means that *string* is not required to have a fill-pointer; if it does have one it is 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 conflict. It is done this way to avoid any allocation of memory.

You can 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.

26.7 Maclisp-compatible Functions

The following functions are provided primarily for Maclisp compatibility.

alphalessp *string1 string2* *Function*

(alphalessp *string1 string2***)** is equivalent to **(string-lessp** *string1 string2***)**. If the arguments are not strings, **alphalessp** compares numbers numerically, lists by element, and random characters by printed representation. **alphalessp** is a Maclisp all-purpose alphabetic sorting function.

getchar *string index* *Function*

Returns the *index*th character of *string* as a symbol. Note that 1-origin indexing is used. This function is mainly for Maclisp compatibility; **aref** should be used to index into strings (however, **aref** does not coerce symbols or numbers into strings).

getcharn *string index* *Function*

Returns the *index*th character of *string* as an integer. Note that 1-origin indexing is used. This function is mainly for Maclisp compatibility; **aref** should be used to index into strings (however, **aref** does not coerce symbols or numbers into strings).

ascii *x* *Function*

ascii is like **character**, but returns a symbol whose printname is the character instead of returning an integer. Examples:

```
(ascii #0101) => A
(ascii #056) => /.
```

The symbol returned is interned in the current package.

maknam *char-list* *Function*

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

```
(maknam '(a b #/0 d)) => ab0d
```

implode *char-list* *Function*

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

samepnamep *sym1 sym2* *Function*

Returns **t** if the two symbols *sym1* and *sym2* have **string=** print-names, that is, if their printed representation is 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 for being in different packages. Examples:

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

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

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

This is the same function as **string=**. **samepnamep** is provided mainly so that you can write programs that work in Maclisp as well as Symbolics-Lisp; in new programs, you should just use **string=**.

PART VII.

Functions and Dynamic Closures

27. Functions

27.1 What is a Function?

Functions are the basic building blocks of Lisp programs. There are many different kinds of functions in Symbolics-Lisp. Here are the printed representations of examples of some of them:

```
foo
(lambda (x) (car (last x)))
(si:digested-lambda (lambda (x) (car (last x)))
                    (foo) 2049 262401 nil (x) nil (car (last x)))
#<dtp-compiled-function append 1424771>
#<lexical-closure (lambda ** **) 7371705>
#<lexical-closure (:internal foo 0) 7372462>
#<dtp-closure 1477464>
```

These all have one thing in common: a function is a Lisp object that can be applied to arguments. All of the above objects can be applied to some arguments and will return a value. Functions are Lisp objects and so can be manipulated in all the usual ways: you can pass them as arguments, return them as values, and make other Lisp objects refer to them. See the function **functionp**, page 8.

27.2 Function Specs

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

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

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

Some kinds of functions remember their own names, and some do not. The "name" remembered by a function can be any kind of function spec, although it is usually a symbol. (See the section "What is a Function?", page 297.) In that section, the example starting with the symbol **si:digested-lambda** and the one whose printed representation includes **dtp-compiled-function**, remember names (the function specs **foo** and **append** respectively). The others do not remember their names, except that the ones starting with **lexical-closure** and **dtp-closure** might contain functions that do remember their names. The second **lexical-closure** example contains the function whose name is **(:internal foo 0)**.

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

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

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

A function spec's definition in general consists of a *basic definition* surrounded by *encapsulations*. Both the basic definition and the encapsulations are functions, but of recognizably different kinds. What **defun** creates is a basic definition, and usually that is all there is. Encapsulations are made by function-altering functions such as **trace** and **advise**. When the function is called, the entire definition, which includes the tracing and advice, is used. If the function is "redefined" with **defun**, only the basic definition is changed; the encapsulations are left in place. See the section "Encapsulations", page 325.

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

a symbol
: The function is remembered in the function cell of the symbol. See the section "The Function Cell of a Symbol", page 563. Function cells and the primitive functions to manipulate them are explained in that section.

(:property *symbol property***)**
: The function is remembered on the property list of the symbol; doing

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

(:method *flavor-name message***)**
(:method *flavor-name method-type message***)**
The function is remembered inside internal data structures of the flavor system.

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

(:location *pointer***)**
The function is stored in the cdr of *pointer*, which can be a locative or a list. This is for pointing at an arbitrary place which there is no other way to describe. This form of function spec is not useful in **defun** (and related special forms) because the reader has no printed representation for locative pointers and always creates new lists; these function specs are intended for programs that manipulate functions. See the section "How Programs Manipulate Definitions", page 316.

(:within *within-function function-to-affect***)**
This refers to the meaning of the symbol *function-to-affect*, but only where it occurs in the text of the definition of *within-function*. If you define this function spec as anything but the symbol *function-to-affect* itself, then that symbol is replaced throughout the definition of *within-function* by a new symbol which is then defined as you specify. See the section "Encapsulations", page 325.

(: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 is numbered 0, the next 1, and so on.

(:internal *function-spec number name***)**
Some Lisp functions contain internal functions, created by **flet** or **labels** forms. *function-spec* is the containing function. *number* is a sequence number; the first internal function the compiler comes across in a given function is numbered 0, the next 1, and so on. *name* is the name of the internal function.

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

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

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

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

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

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

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

(*symbol property*)

as a function name. This is translated into:

(:property *symbol property*)

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

27.3 Simple Function Definitions

defun *Special Form*

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

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

name is the function spec you wish to define as a function. The *lambda-list* is a list of the names to give to the arguments of the function. Actually, it is a little more general than that; it can contain *lambda-list keywords* such as **&optional** and **&rest**. (Keywords are explained in other sections. See the section "Evaluating a Function Form", page 151. See the section "Lambda-list Keywords", page 309.) Additional syntactic features of **defun** are explained in another section. See the section "Function-defining Special Forms", page 305.

defun creates a list which looks like:

```
(si:digested-lambda...)
```

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 add-a-number (x &optional (inc 1))
  (+ x inc))

(defun average (&rest numbers &aux (total 0))
  (loop for n in numbers
        do (setq total (+ total n)))
  (// total (length numbers)))
```

addone is a function that expects a number as an argument, and returns a number one larger. **add-a-number** takes one required argument and one optional argument. **average** takes any number of additional arguments that are given to the function as a list named **numbers**.

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

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

is equivalent to and preferable to:

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

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

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

complete sentence that makes sense read by itself, since there are two editor commands to get at the documentation, one of which is "brief" and prints only the first line. Example:

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

defunp *Macro*

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

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

expands into:

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

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

See the section "Function-defining Special Forms", page 305. Information on defining functions, and other ways of doing so, are discussed in that section.

27.4 Operations the User Can Perform on Functions

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

To print out the definition of the function spec with indentation to make it legible, use **grindef**. This works only for interpreted functions. If the definition is a

compiled function, it cannot be printed out as Lisp code, but its compiled code can be printed by the **disassemble** function.

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

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

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

27.5 Kinds of Functions

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

Before we start classifying the functions, we will first discuss something about how the evaluator works. When the evaluator is given a list whose first element is a symbol, the form can be a function form, a special form, or a macro form. If the definition of the symbol is a function, then the function is just applied to the result of evaluating the rest of the subforms. If the definition is a cons whose car is **macro**, then it is a macro form. See the section "Macros", page 337. What about special forms?

Conceptually, the evaluator knows specially about all special forms (hence their name). However, the Symbolics-Lisp 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 call *special functions*. Each of these functions is the part of the Lisp interpreter that knows how to deal with that special form. Normally you do not have to know about this; it is just part of how the evaluator works.

Many of the special forms in Zetalisp are implemented as macros. They are implemented this way because it is easier to write a macro than to write both a new part of the interpreter (a special function) and a new *ad hoc* module in the compiler. However, they are sometimes documented as special forms, rather than macros, because you should not in any way depend on the way they are implemented.

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

1. *Interpreted* functions, which are defined with **defun**, represented as list structure, and interpreted by the Lisp evaluator.

2. *Compiled* functions, which are defined by **compile** or by loading a bin file, are represented by a special Lisp data type, and are executed directly by the machine.

3. Various types of Lisp objects that can be applied to arguments, but when they are applied they dig up another function somewhere and apply it instead. These include symbols, dynamic and lexical closures, and instances.

4. Various types of Lisp objects that, when used as functions, do something special related to the specific data type. These include arrays and stack groups.

27.5.1 Interpreted Functions

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

There are two kinds of interpreted functions: **lambda**s and **si:digested-lambda**s. A **lambda** function is the simplest kind. It is a list that looks like this:

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

The symbol **lambda** identifies this list as a **lambda** function. *lambda-list* is a description of what arguments the function takes. See the section "Evaluating a Function Form", page 151. 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.

An **si:digested-lambda** is like a **lambda**, but contains extra elements in which the system remembers the function's name, its documentation, a preprocessed form of its lambda-list, and other information. Having the function's name there allows the Debugger and other tools to give the user more information. This is the kind of function that **defun** creates. The interpreter turns any lambdas it is asked to apply into digested-lambdas, using **rplaca** and **rplacd** to modify the list structure of the original lambda-expression.

27.5.2 Compiled Functions

The Lisp function compiler converts **lambda** functions into compiled functions. A compiled function's printed representation looks like:

```
#<dtp-compiled-function append 1424771>
```

The object contains 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", compiled functions are full-fledged objects and can be passed as arguments, stored in data structure, and applied to arguments.

27.5.3 Other Kinds of Functions

A dynamic closure is a kind of function that contains another function and a set of special variable bindings. When the closure is applied, it puts the bindings into effect and then applies the other function. When that returns, the closure bindings are removed. Dynamic closures are created by the **closure** function and the **let-closed** special form. See the section "Dynamic Closures", page 331.

A lexical closure is a kind of function that contains another function and a set of local variable bindings. A lexical closure is created by reference to an internal functions. Invocation of a lexical closure simply provides the necessary data linkage for a function to run in the environment in which the closure was made. See the section "Lexical Scoping", page 137.

An instance is a message-receiving object that has some state and a table of message-handling functions (called *methods*). See the section "Flavors", page 415.

An array can be used as a function. The arguments to the array are the indices and the value is the contents of the element of the array. This works this way for Maclisp compatibility and is not recommended usage. Use **aref** instead.

A stack group can be called as a function. This is one way to pass control to another stack group. See the section "Stack Groups" in *Internals, Processes, and Storage Management.*

27.6 Function-defining Special Forms

defun is a special form that is put in a program to define a function. **defsubst** and **macro** are others. This section explains how these special forms work, how they relate to the different kinds of functions, and how they connect to the rest of the function-manipulation system.

Function-defining special forms typically take as arguments a function spec and a description of the function to be made, usually in the form of a list of argument names and some forms that constitute the body of the function. They construct a

function, give it the function spec as its name, and define the function spec to be the new function. Different special forms make different kinds of functions. **defun** and **defsubst** both make an **si:digested-lambda** function. **macro** makes a macro; though the macro definition is not really a function, it is like a function as far as definition handling is concerned.

These special forms are used in writing programs because the function names and bodies are constants. Programs that define functions usually want to compute the functions and their names, so they use **fdefine**.

All of these function-defining special forms alter only the basic definition of the function spec. Encapsulations are preserved. See the section "Encapsulations", page 325.

The special forms only create interpreted functions. There is no special way of defining a compiled function. Compiled functions are made by compiling interpreted ones. The same special form that defines the interpreted function, when processed by the compiler, yields the compiled function.

Note that the editor understands these and other "defining" special forms (for example, **defmethod**, **defvar**, **defmacro**, and **defstruct**) to some extent, so that when you ask for the definition of something, the editor can find it in its source file and show it to you. The general convention is that anything that is used at top level (not inside a function) and starts with **def** should be a special form for defining things and should be understood by the editor. **defprop** is an exception.

defun

The **defun** special form (and the **defunp** macro that expands into a **defun**) are used for creating ordinary interpreted functions. See the section "Simple Function Definitions", page 300.

For Maclisp compatibility, a *type* symbol can be inserted between *name* and *lambda-list* in the **defun** form. The following types are understood:

expr The same as no type.

fexpr Defines a special form that operates like a Maclisp fexpr. The special form can only be used in interpreted functions and in forms evaluated at top-level, since the compiler has not been told how to compile it.

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.

defsubst The **defsubst** special form is used to create substitutable functions. It is used just like **defun** but produces a function that acts normally when applied, but can also be open-coded (incorporated into its callers) by the compiler. See the section "Substitutable Functions", page 351.

macro The **macro** special form is the primitive means of creating a macro. It gives a function spec a definition that is a macro definition rather than a actual function. A macro is not a function because it cannot be applied, but it *can* appear as the car of a form to be evaluated. Most macros are created with the more powerful **defmacro** special form.

defselect The **defselect** special form defines a select-method function.

deff Unlike the above special forms, **deff** does not create new functions. It simply serves as a hint 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*

Unlike the above special forms, **def** does not create new functions. It simply serves as a hint 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.

If a function is created in some strange way, wrapping a **def** special form around the code that creates it informs the editor of the connection. The form:

```
(def function-spec
  form1 form2...)
```

simply evaluates the forms *form1*, *form2*, and so on. It is assumed that these forms 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 that define the function. The editor only uses it to tell which line to put the cursor on.

deff *function-spec definition-creator* *Special Form*

deff is a simplified version of **def**. It evaluates the form *definition-creator*, which should produce a function, and makes that function the definition of *function-spec*, which is not evaluated. **deff** is used for giving a function spec a definition that is not obtainable with the specific defining forms such as **defun** and **macro**. For example:

```
(deff foo 'bar)
```

makes **foo** equivalent to **bar**, with an indirection so that if **bar** changes, **foo** likewise changes;

```
(deff foo (function bar))
```

copies the definition of **bar** into **foo** with no indirection, so that further changes to **bar** 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 that define objects (such as functions) that @ should cross-reference.

defselect *Special Form*

defselect defines a function that is a select-method. This function contains a table of subfunctions; when it is called, the first argument, a symbol on the keyword package called the *message name*, is looked up in the table to determine which subfunction to call. Each subfunction can take a different number of arguments, and have a different pattern of **&optional** and **&rest** arguments. **defselect** is useful for a variety of "dispatching" jobs. By analogy with the more general message passing facilities in flavors, the subfunctions are sometimes called *methods* and the first argument is sometimes called a *message*.

The special form looks like:

```
(defselect (function-spec default-handler no-which-operations)
  (message-name (args...)
   body...)
  (message-name (args...)
   body...)
  ...)
```

function-spec is the name of the function to be defined. *default-handler* is optional; it must be a symbol and is a function that gets called if the select-method is called with an unknown message. If *default-handler* is unsupplied or **nil**, then an error occurs if an unknown message is sent. If *no-which-operations* is non-**nil**, the **:which-operations** method that would normally be supplied automatically is suppressed. The **:which-operations** method takes no arguments and returns a list of all the message names in the **defselect**.

The **:operation-handled-p** and **:send-if-handles** methods are automatically supplied. See the message **:operation-handled-p**, page 453. See the message **:send-if-handles**, page 454.

If *function-spec* is a symbol, and *default-handler* and *no-which-operations* are not supplied, then the first subform of the **defselect** can be just *function-spec* by itself, not enclosed in a list.

The remaining subforms in a **defselect** define methods. *message-name* is the message name, or a list of several message names if several messages are to be handled by the same subfunction. *args* is a lambda-list; it should not include the first argument, which is the message name. *body* is the body of the function.

A method subform can instead look like:

(*message-name* . *symbol*)

In this case, *symbol* is the name of a function that is called when the *message-name* message is received. It is called with the same arguments as the select-method, including the message symbol itself.

27.7 Lambda-list Keywords

This section documents all the keywords that can appear in the lambda-list (argument list) of a function, a macro, or a special form. See the section "Evaluating a Function Form", page 151. Some of them are allowed everywhere, while others are only allowed in one of these contexts; those are so indicated.

lambda-list-keywords *Variable*

The value of this variable is a list of all of the allowed "&" keywords. Some of these are obsolete and do not do anything; the remaining ones (some of which are also obsolete) are listed below. See the section "Evaluating a Function Form", page 151. Example functions which use each of these keywords are provided in that section.

&optional

Declares the following arguments to be optional. See the section "Evaluating a Function Form", page 151.

&rest Declares the following argument to be a rest argument. There can be only one **&rest** argument.

It is important to realize that the list of arguments to which a rest-parameter is bound is set up in whatever way is most efficiently implemented, rather than in the way that is most convenient for the function receiving the arguments. It is not guaranteed to be a "real" list. Sometimes the rest-args list is stored in the function-calling stack, and loses its validity when the function returns. If a rest-argument is to be returned or made part of permanent list-structure, it must first be copied, as you must always assume that it is one of these special lists. See the function **copylist**, page 50.

The system does not detect the error of omitting to copy a rest-argument; you simply find that you have a value that seems to change behind your back. At other times the rest-args list is an

argument that was given to **apply**; therefore it is not safe to **rplaca** this list as you might modify permanent data structure. An attempt to **rplacd** a rest-args list is unsafe in this case, while in the first case it causes an error, since lists in the stack are impossible to **rplacd**.

&key Separates the positional parameters and rest parameter from the keyword parameters. See the section "Evaluating a Function Form", page 151.

&allow-other-keys
In a lambda-list that accepts keyword arguments, **&allow-other-keys** says that keywords that are not specifically listed after **&key** are allowed. They and the corresponding values are ignored, as far as keyword arguments are concerned, but they do become part of the rest argument, if there is one.

&aux It separates the arguments of a function from the auxiliary variables. Following **&aux** you can put entries of the form:

(*variable initial-value-form*)

or just *variable* if you want it initialized to **nil** or do not care what the initial value is.

&special
Declares the following arguments and/or auxiliary variables to be special within the scope of this function. **&special** can appear anywhere any number of times.

&local Turns off a preceding **&special** for the variables that follow. **&local** can appear anywhere any number of times.

&quote
Using **&quote** is an obsolete way to define special functions. **&quote** declares that the following arguments are not to be evaluated. You should implement language extensions as macros rather than through special functions, because macros directly define a Lisp-to-Lisp translation and therefore can be understood by both the interpreter and the compiler.

Special functions, on the other hand, only extend the interpreter. The compiler has to be modified to understand each new special function so that code using it can be compiled. Since all real programs are eventually compiled, writing your own special functions is strongly discouraged.

&eval This is obsolete. Use macros instead to define special functions. **&eval** turns off a preceding **&quote** for the arguments which follow.

&list-of
This is not supported. Use **loop** or **mapcar** instead of **&list-of**.

&body This is for macros defined by **defmacro** or **macrolet** only. It is

similar to **&rest**, but declares to **grindef** and the code-formatting module of the editor that the body forms of a special form follow and should be indented accordingly.

See the section "&-Keywords Accepted by **defmacro**", page 373.

&whole
This is for macros defined by **defmacro** or **macrolet** only. **&whole** is followed by *variable*, which is bound to the entire macro-call form or subform. *variable* is the value that the macro-expander function receives as its first argument. **&whole** is allowed only in the top-level pattern, not in inside patterns.

See the section "&-Keywords Accepted by **defmacro**", page 373.

&environment
This is for macros defined by **defmacro** or **macrolet** only. **&environment** is followed by *variable*, which is bound to an object representing the lexical environment where the macro call is to be interpreted. This environment might not be the complete lexical environment. It should be used only with the **macroexpand** function for any local macro definitions that the **macrolet** construct might have established within that lexical environment. **&environment** is allowed only in the top-level pattern, not in inside patterns. See the section "Lexical Environment Objects and Arguments", page 138.

See the section "&-Keywords Accepted by **defmacro**", page 373.

27.8 Declarations

Declarations are optional Lisp expressions that provide the Lisp system, typically the interpreter and the compiler, with information about your program, for example, documentation.

The special operator **declare** is the most common mechanism for making declarations. The other special operator, **local-declare** should *not* be used for new code.

declare &rest *ignore* *Special Form*
The **declare** special form can be used in two ways: at top level or within function bodies. For information on top-level **declare** forms: See the section "How the Stream Compiler Handles Top-level Forms" in *Program Development Utilities*.

declare forms that appear within function bodies provide information to the Lisp system (for example, the interpreter and the compiler) about this particular function. Expressions appearing within the function-body **declare** are declarations; they are not evaluated. **declare** forms must appear at the

front of the body of certain special forms, such as **let** and **defun**. Some declarations apply to function definitions and must appear as the first forms in the body of that function; otherwise they are ignored.

Function-body **declare** forms understand the following declarations. The first group of declarations can be used only at the beginning of a function body, for example, **defun**, **defmacro**, **defmethod**, **lambda**, or **flet**.

(arglist . *arglist*)
This declaration saves *arglist* as the argument list of the function, to be used instead of its lambda-list if c-sh-A or the **arglist** function need to determine the function's arguments. The **arglist** declaration is used purely for documentation purposes.

Example:

```
(defun example (&rest options)
  (declare (arglist &key x y z))
  (lexpr-funcall #'example-2 "Print" options))
```

(values . *values*)
This declaration saves *values* as the return values list of the function, to be used if c-sh-A or the **arglist** function asks what values it returns. The **values** declaration is used purely for documentation purposes.

(sys:function-parent *name type*)
Helps the editor and source-finding tools (like m-.) locate symbol definitions produced as a result of macro expansion. (The accessor, constructor, and alterant macros produced by a **defstruct** are an example.)

The **sys:function-parent** declaration should be inserted in the source definition to record the name of the outer definition of which it is a part. *name* is the name of the outer definition. *type* is its type, which defaults to **defun**. See the section "Using the **sys:function-parent** Declaration", page 319.

(sys:downward-function)
The declaration **sys:downward-function**, in the body of an internal lambda, guarantees to the system that lexical closures of the lambda in which it appears are only used as downward funargs, and never survive the calls to the procedure that produced them. This allows the system to allocate these closures on the stack.

```
(defun special-search-table (item)
  (block search
    (send *hash-table* :map-hash
          #'(lambda (key object)
              (declare (sys:downward-function))
              (when (magic-function key object item)
                (return-from search object))))))
```

Here, the **:map-hash** message to the hash table calls the closure of the internal lambda many times, but does not store it into permanent variables or data structure, or return it "around" **special-search-table**. Therefore, it is guaranteed that the closure does not survive the call to **special-search-table**. It is thus safe to allow the system to allocate that closure on the stack.

Stack-allocated closures have the same lifetime (*extent*) as **&rest** arguments and lists created by **with-stack-list** and **with-stack-list***, and require the same precautions.

(sys:downward-funarg *var1 var2 ...***)** or **(sys:downward-funarg *)**

The **sys:downward-funarg** declaration (not to be confused with **sys:downward-function**) permits a procedure to declare its intent to use one or more of its arguments in a downward manner. For instance, **sort**'s second argument is a funarg, which is only used in a downward manner, and is declared this way. The second argument to **process-run-function** is a good example of a funarg that is not downward. Here is an example of a function that uses and declares its argument as a downward funarg.

```
(defun search-alist-by-predicate (alist predicate)
  (declare (sys:downward-funarg predicate))
  ;; Traditional "recursive" style, for variety.
  (if (null alist)
      nil
      (let ((element (car list))
            (rest (cdr list))
        (if (funcall predicate (car element))
            (cdr element)
            (search-alist-by-predicate rest predicate))))))
```

This function only calls the funarg passed as the value of **predicate**. It does not store it into permanent structure, return it, or throw it around **search-alist-by-predicate**'s activation.

The reason you so declare the use of an argument is to allow the system to deduce guaranteed downward use of a funarg without need for the **sys:downward-function** declaration. For instance, if **search-alist-by-predicate** were coded as above, we could write

```
(defun look-for-element-in-tolerance (alist required-value tolerance)
  (search-alist-by-predicate alist
    #'(lambda (key)
        (< (abs (- key required-value)) tolerance))))
```

to search the keys of the list for a number within a certain tolerance of a required value. The lexical closure of the internal lambda is

automatically allocated by the system on the stack because the system has been told that any funarg used as the first argument to **search-alist-by-predicate** is used only in a downward manner. No declaration in the body of the lambda is required.

All appropriate parameters to system functions have been declared in this way.

There are two possible forms of the **downward-funarg** declaration:

(declare (sys:downward-funarg *var1 var2 ...* **)**
Declares the named variables, which must be parameters (formal arguments) of the function in which this declaration appears, to have their values used only in a downward fashion. This affects the generation of closures as functional arguments to the function in which this declaration appears: it does not directly affect the function itself. Due to an implementation restriction, *var-i* cannot be a keyword argument.

(declare (sys:downward-funarg *))
Declares guaranteed downward use of all functional arguments to this function. This is to cover closures of functions passed as elements of **&rest** arguments and keyword arguments.

The following group of declarations can be used at the beginning of any body, for example, a **let** body.

(special *sym1 sym2 ...***)**
The symbols *sym1*, *sym2*, and so on, are treated as special variables within the form containing the **declare**; the Lisp system (both the compiler and the interpreter) implements the variables using the value cells of the symbols.

(unspecial *sym1 sym2 ...***)**
The symbols *sym1*, *sym2*, and so on, are treated as local variables within the form containing the **declare**.

Example:

```
(defun print-integer (number base)
  (declare (unspecial base))
  (when (≥ number base)
    (print-integer (floor number base) base))
  (tyo (digit-char (mod number base) base)))
```

(sys:array-register *variable1 variable2 ...***)**
Indicates to the compiler that *variable1*, *variable2*, and so on, are holding single-dimensional arrays as their values. Henceforth, each of

these variables must *always* hold a single-dimensional array. The compiler can then use special faster array element referencing and setting instructions for the **aref** and **aset** functions. Whether or not this declaration is worthwhile depends on the type of array and the number of times that referencing and setting instructions are executed. For example, if the number of referencing instructions is more than ten, this declaration makes your program run faster; for one or two references, it actually slows execution.

(sys:array-register-1d *variable1 variable2 ...***)**
Indicates to the compiler that *variable1*, *variable2*, and so on, are holding single- or multidimensional arrays as their values, and that the array is going to be referenced as a one-dimensional array. Henceforth, each of these variables must *always* hold an array. The compiler can then use special faster array element referencing and setting instructions for the **sys:%1d-aref** and **sys:%1d-aset** functions. Whether or not this declaration is worthwhile depends on the type of array and the number of times that referencing and setting instructions are executed. For example, if the number of referencing instructions is more than ten, this declaration makes your program run faster; for one or two references, it actually slows execution.

The compiler also recognizes any number of **declare** forms as the first forms in the bodies of the following special forms. This means that you can have **special** declarations that are local to any of these blocks. In addition, declarations can appear at the front of the body of a function definition, like **defun**, **defmacro**, **defsubst**, and so on.

destructuring-bind	**multiple-value-bind**
let	**let***
do	**do***
do-named	**do*-named**
prog	**prog***
lambda	

local-declare *declarations body...* *Special Form*
local-declare, while available in Release 6, should *not* be used for new code.

A **local-declare** form looks like this:

```
(local-declare (declaration declaration ...)
  form1
  form2
  ...)
```

Example:

```
(local-declare ((special foo1 foo2))
(defun larry ()
     )
(defun george ()
     )
  ); end of local-declare
```

local-declare understands the same declarations as **declare**.

Each local declaration is consed onto the list **local-declarations** while the *forms* are being evaluated (in the interpreter) or compiled (in the compiler). This list has two uses. First, it can be used to pass information from outer macros to inner macros. Secondly, the compiler specially interprets certain declarations as local declarations, which apply only to the compilation of the *forms*.

local-declarations *Variable*

local-declarations is a list of local declarations. Each declaration is itself a list whose car is an atom which indicates the type of declaration. The meaning of the rest of the list depends on the type of declaration. For example, in the case of **special** and **unspecial** the cdr of the list contains the symbols being declared.

The compiler is interested only in **special**, **unspecial**, **macro**, and **arglist** declarations.

Local declarations are added to **local-declarations** in two ways:

- Inside a **local-declare**, the specified declarations are bound onto the front.
- If **undo-declarations-flag** is **t**, some kinds of declarations in a file that is being compiled are consed onto the front of the list; they are not popped until **local-declarations** is unbound at the end of the file.

Many forms, such as **defun**, **defvar**, and **defconst**, have declarative aspects. For example, **defun** tells the system that a function of a certain name and number of arguments is defined and where it is defined. **defvar** and **defconst** tell the system that certain symbols are special.

27.9 How Programs Manipulate Definitions

fdefine *function-spec definition* &optional *(carefully* **nil***) (no-query* **nil***)* *Function*

This is the primitive that **defun** and everything else in the system use to change the definition of a function spec. If *carefully* is non-**nil**, which it usually should be, then only the basic definition is changed, the previous basic

definition is saved if possible (see **undefun**), and any encapsulations of the function such as tracing and advice are carried over from the old definition to the new definition. *carefully* also causes the user to be queried if the function spec is being redefined by a file different from the one that defined it originally. However, this warnings is suppressed if either the argument *no-query* is non-**nil**, or if the global variable **inhibit-fdefine-warnings** is **t**.

If **fdefine** is called while a file is being loaded, it records what file the function definition came from so that the editor can find the source code.

If *function-spec* was already defined as a function, and *carefully* is non-**nil**, the function-spec's **:previous-definition** property is used to save the previous definition. If the previous definition is an interpreted function, it is also saved on the **:previous-expr-definition** property. These properties are used by the **undefun** function, which restores the previous definition, and the **uncompile** function, which restores the previous interpreted definition. The properties for different kinds of function specs are stored in different places; when a function spec is a symbol its properties are stored on the symbol's property list.

defun and the other function-defining special forms all supply **t** for *carefully* and **nil** or nothing for *no-query*. Operations that construct encapsulations, such as **trace**, are the only ones that use **nil** for *carefully*.

inhibit-fdefine-warnings *Variable*

This variable is normally **nil**. Setting it to **t** prevents **fdefine** from warning you and asking about questionable function definitions such as a function being redefined by a different file than defined it originally, or a symbol that belongs to one package being defined by a file that belongs to a different package. Setting it to **:just-warn** allows the warnings to be printed out, but prevents the queries from happening; it assumes that your answer is "yes", that is, that it is all right to redefine the function.

record-source-file-name *function-spec* &optional *(type* **'defun***)* *Function*
no-query

record-source-file-name associates the definition of a function with its source files, so that tools such as Edit Definition (m-.) can find the source file of a function. It also detects when two different files both try to define the same function, and warns the user.

record-source-file-name is called automatically by **defun**, **defmacro**, **defstruct**, **defflavor**, and other such defining special forms. Normally you do not invoke it explicitly. If you have your own defining macro, however, that does not expand into one of the above, then you can make its expansion include a **record-source-file-name** form.

Normally, **record-source-file-name** returns **t**. If a definition of the same name and type was already made by another file, the user is asked whether

the definition should be performed. If the user answers "no", **record-source-file-name** returns **nil**. When **nil** is returned the caller should not perform the definition.

function-spec The function spec for the entity being defined.

type The type of entity being defined, with **defun** as the default. *type* can be any symbol, typically the name of the corresponding special form for defining the entity. Some standard examples:

defun
defvar
defflavor
defstruct

Both macros and substs are subsumed under the type defun, because you cannot have a function named **x** in one file and a macro named **x** in another file.

no-query Controls queries about redefinitions. **t** means to suppress queries about redefining. The default value of *no-query* depends on the value of **inhibit-fdefine-warnings**. When **inhibit-fdefine-warnings** is **t**, *no-query* is **t**; otherwise it is **nil**. Regardless of the value for *no-query*, queries are suppressed when the definition is happening in a patch file.

You cannot specify the source file name with this function. The function is always associated with the pathname for the file being loaded (**fdefine-file-pathname**).

When redefining functions, some users try to avoid redefinition warnings and queries by using the form **(remprop** *symbol* **:source-file-name)**. The preferred way to do this is to use the form **(record-source-file-name** *function-spec* **'defun t)**. The former method causes the system to forget both the original definition and other definitions for the same symbol (as a variable, flavor, structure, and so forth). **record-source-file-name** lets the system know that the function is defined in two places, and it avoids redefinition warnings and queries.

Of course, if you are redefining something other than a function, use the appropriate definition type symbol instead of **defun** as the second argument to **record-source-file-name**. For example, if you are redefining a flavor, use **defflavor** as the second argument. See the section "Using the **sys:function-parent** Declaration", page 319.

sys:fdefine-file-pathname *Variable*

While loading a file, this is the generic-pathname for the file. The rest of the time it is **nil**. **fdefine** uses this to remember what file defines each function.

sys:function-parent *function-spec* *Function*

When a symbol's definition is produced as the result of macro expansion of a source definition, so that the symbol's definition does not appear textually in the source, the editor cannot find it. The accessor, constructor, and alterant macros produced by a **defstruct** are an example of this. The **sys:function-parent** declaration can be inserted in the source definition to record the name of the outer definition of which it is a part.

The declaration consists of the following:

(**sys:function-parent** *name type*)

name is the name of the outer definition. *type* is its type, which defaults to **defun**. See the section "Using the **sys:function-parent** Declaration", page 319. Declarations are explained in another section. See the section "Declarations", page 311.

sys:function-parent is a function related to the declaration. It takes a function spec and returns **nil** or another function spec. The first function spec's definition is contained inside the second function spec's definition. The second value is the type of definition.

Two examples:

```
(defsubst foo (x y)
  (declare (sys:function-parent bar))
  ...)

(defmacro defxxx (name ...)
  `(local-declare ((sys:function-parent ,name defxxx))
        (defmacro ...)
        (defmacro ...)
        ))
```

Using the sys:function-parent Declaration

A *definition* is a Lisp expression that appears in a source program file and has a name by which a user would like to refer to it. Definitions come in a variety of types. The main point of definition types is that two definitions with the same name and different types can exist simultaneously, but two definitions with the same name and the same type redefine each other when evaluated. Some examples of definition type symbols and special forms that define such definitions are:

Type symbol	*Type name in English*	*Special form names*
defun	function	**defun, defmacro, defmethod**
defvar	variable	**defvar, defconst, defconstant**
defflavor	flavor	**defflavor**
defstruct	structure	**defstruct**

Things to note: More than one special form can define a given kind of definition. The name of the most representative special form is typically chosen as the type symbol. This symbol typically has a **si:definition-type-name** property of a string that acts as a prettier form of the name for people to read.

```
(defprop feature "Feature" si:definition-type-name)
(defprop defun "Function" si:definition-type-name)
```

record-source-file-name and related functions take a name and a type symbol as arguments. The editor understands certain definition-making special forms, and knows how to parse them to get out the name and the type. This mechanism has not yet been made user-extensible. Currently the editor assumes that any top-level form it does not know about that starts with "(def" must be defining a function (a definition of type **defun**) and assumes that the cadr of that form is the name of the function. The starting left parenthesis must be at the left margin (not indented) for the editor to recognize the "(def" form. Heuristics appropriate for **defun** are applied to this name if it is a list.

In general, a definition whose name is not a symbol and whose type is not **defun** does not work properly. This will be fixed in a future release.

The declaration **sys:function-parent** is of interest to users. The function with the same name is probably not of interest to users; it is part of the mechanism by which the Zmacs command Edit Definition (m-.) figures out what file to look in.

Example:

We have functions called "frobulators" that are stored on the property list of symbols and require some special bindings wrapped around their bodies. Frobulator definitions are not considered function definitions, because the name of the frobulator does not become defined as a Lisp function. Indeed, we could have a frobulator named **list** and Lisp's **list** function would continue to work. Instead we make a new definition type.

```
(defmacro define-frobulator (name arg-list &body body)
  `(progn
     (add-to-list-of-known-frobulators ',name)
     (record-source-file-name ',name 'define-frobulator)
     (defun (:property ,name frobulator) (self ,@arg-list)
       (declare (sys:function-parent ,name define-frobulator))
       (let (,(make-frobulator-bindings name arg-list))
         ,@body))))

(defprop define-frobulator "Frobulator" si:definition-type-name)
```

Here we would tell the editor how to parse **define-frobulator** if its parser were user-extensible. Because it is not, we rely on its heuristics to make M-. work adequately for frobulators.

Next we define a frobulator. This is not an interesting definition, for we do not actually know what the word "frobulate" means. We could always recast this example as a symbolic differentiator: We would define the + frobulator to return a list of + and the frobulations of the arguments, the * frobulator to return sums of products of factors and derivatives of factors, and so forth.

```
(define-frobulator list ()
  (frobulate-any-number-of-args self))
```

In **define-frobulator**, we call **record-source-file-name** so that when a file containing frobulator definitions is loaded, we know what file those definitions came from. Inside the function that is generated, we include a function-parent declaration because no definition of that function is apparent in any source file. The system takes care of doing
(record-source-file-name '(:property list frobulator) 'defun), as it always does when a function definition is loaded. Suppose an error occurs in a frobulator function — in the **list** example above, we might try to call **frobulate-any-number-of-args**, which is not defined — and we use the Debugger C-E command to edit the source. This is trying to edit **(:property list frobulator)**, the function in which we were executing. The definition that defines this function does not have that name; rather, it is named **list** and has type **define-frobulator**. The **sys:function-parent** declaration enables the editor to know that fact.

If your definition-making special form and your definition type symbol do not have the same name, you should define the special form's **zwei:definition-function-spec** property to be the definition type symbol. This helps the editor parse such special forms. This is useful when several special forms exist to make definitions of a single type.

For another example, more complicated but real, use **mexp** or the Zmacs command Macro Expand Expression (C-sh-M) to look at the macro expansion of:

```
(defstruct (foo :conc-name) one two)
```

The macro **sys:defsubst-with-parent** that it calls is just **defsubst** with a **sys:function-parent** declaration inside. It exists only because of a bug in an old implementation of **defsubst** that made doing it the straightforward way not work.

fset-carefully *symbol definition* &optional *force-flag* *Function*

This function is obsolete. It is equivalent to:

```
(fdefine symbol definition t force-flag)
```

fdefinedp *function-spec* *Function*

This returns **t** if *function-spec* has a definition, or **nil** if it does not.

fdefinition *function-spec* *Function*
This returns *function-spec*'s definition. If it has none, an error occurs.

sys:fdefinition-location *function-spec* *Function*
This returns a locative pointing at the cell that contains *function-spec*'s definition. For some kinds of function specs, though not for symbols, this can cause data structure to be created to hold a definition. For example, if *function-spec* is of the **:property** kind, then an entry might have to be added to the property list if it isn't already there. In practice, you should write **(locf (fdefinition** *function-spec*)) instead of calling this function explicitly.

fundefine *function-spec* *Function*
Removes the definition of *function-spec*. For symbols this is equivalent to **fmakunbound**. If the function is encapsulated, **fundefine** removes both the basic definition and the encapsulations. Some types of function specs (**:location** for example) do not implement **fundefine**. **fundefine** on a **:within** function spec removes the replacement of *function-to-affect*, putting the definition of *within-function* back to its normal state. **fundefine** on a **:method** function spec removes the method completely, so that future messages will be handled by some other method.

si:function-spec-get *function-spec indicator* *Function*
Returns the value of the *indicator* property of *function-spec*, or **nil** if it doesn't have such a property.

si:function-spec-putprop *function-spec value indicator* *Function*
Gives *function-spec* an *indicator* property whose value is *value*.

undefun *function-spec* *Function*
If *function-spec* has a saved previous basic definition, this interchanges the current and previous basic definitions, leaving the encapsulations alone. This undoes the effect of a **defun, compile**, and so on. (See the function **uncompile** in *Program Development Utilities*.)

27.10 How Programs Examine Functions

These functions take a function as argument and return information about that function. Some also accept a function spec and operate on its definition. The others do not accept function specs in general but do accept a symbol as standing for its definition. (Note that a symbol is a function as well as a function spec).

documentation *function* *Function*
Given a function or a function spec, this finds its documentation string, which is stored in various different places depending on the kind of function. If there is no documentation, **nil** is returned.

See the section "The Document Examiner" in *User's Guide to Symbolics Computers*.

debugging-info *function* *Function*

This returns the debugging info alist of *function*. Most of the elements of this alist are an internal interface between the compiler and the Debugger.

arglist *function* &optional *real-flag* *Function*

arglist is given a function or a function spec, and returns its best guess at the nature of the function's lambda-list. It can also return a second value which is a list of descriptive names for the values returned by the function. The third value is a symbol specifying the type of function:

Returned Value	*Function Type*
nil	ordinary function
subst	substitutable function
special	special form
macro	macro
si:special-macro	both a special form and a macro
array	array

If *function* is a symbol, **arglist** of its function definition is used.

Some functions' real argument lists are not what would be most descriptive to a user. A function can take an **&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))
  .....)
```

Note that since the declared argument list is supplied by the user, it does not necessarily correspond to the function's actual argument list.

real-flag allows the caller of **arglist** to say that the real argument list should be used even if a declared argument list exists.

If *real-flag* is **t** or a declared argument list does not exist, **arglist** computes its return value using information associated with the function. Normally the computed argument list is the same as that supplied in the source definition, but occasionally some differences occur. However, **arglist** always returns a functionally correct answer in that the number and type of the arguments is correct.

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

values declaration in the function's definition, entirely analogous to the **arglist** declaration above, you can specify a list of mnemonic names for the returned values. This list is returned by **arglist** as the second value.

```
(arglist 'arglist)
  => (function &optional real-flag) and (arglist values type)
```

args-info *function* *Function*

args-info returns an integer 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 integer, which are referenced by the symbolic names shown below. By the usual Symbolics Lisp Machine convention, those starting with a single "%" are bit-masks (meant to be **logand**ed or **bit-test**ed with the number), and those starting with "%%" are byte descriptors (meant to be used with **ldb** or **ldb-test**).

Here are the fields:

%%arg-desc-min-args
This is the minimum number of arguments that can be passed to this function, that is, the number of "required" parameters.

%%arg-desc-max-args
This is the maximum number of arguments that can be passed to this function, that is, the sum of the number of "required" parameters and the number of "optional" parameters. If there is an **&rest** argument, this is not really the maximum number of arguments that can 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-rest-arg
If this is nonzero, the function takes an **&rest** argument or **&key** arguments. A greater number of arguments than **%%arg-desc-max-args** can be passed.

%arg-desc-interpreted
This function is not a compiled-code object.

%%arg-desc-interpreted
This is the byte field corresponding to the **%arg-desc-interpreted** bit.

%%arg-desc-quoted
This is obsolete. In Release 5 this was used by the **&quote** feature.

%args-info *function* *Function*

This is an internal function; it is like **args-info** but does not work for interpreted functions. Also, *function* must be a function, not a function spec.

27.11 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**, **advise**, or **breakon** 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**, **advise**, and **breakon**.

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, but it can be a compiled function also, if the application that created it wants to compile it.

uninterned-symbol's function definition is the thing that the encapsulation contains, usually the basic definition of the function spec. Or it can be another encapsulation, which has in it another debugging info item containing another uninterned symbol. Eventually you get to a function that is not an encapsulation; it does not have the sort of debugging info item that encapsulations all have. That function is the basic definition of the function spec.

Literally speaking, the definition of the function spec is the outermost encapsulation, period. The basic definition is not the definition. If you are asking for the definition of the function spec because you want to apply it, the outermost encapsulation is exactly what you want. But the basic definition can be found mechanically from the definition, by following the debugging info alists. So it makes sense to think of it as a part of the definition. In regard to the function-defining special forms such as **defun**, it is convenient to think of the encapsulations as connecting between the function spec and its basic definition.

An encapsulation is created with the macro **si:encapsulate**.

si:encapsulate *Macro*

A call to **si:encapsulate** looks like:

```
(si:encapsulate function-spec outer-function type
                body-form
                extra-debugging-info)
```

All the subforms of this macro are evaluated. In fact, the macro could almost be replaced with an ordinary function, except for the way *body-form* is handled.

function-spec evaluates to the function spec whose definition the new encapsulation should become. *outer-function* is another function spec, which should often be the same one. Its only purpose is to be used in any error messages from **si:encapsulate**.

type evaluates to a symbol that identifies the purpose of the encapsulation; it says what the application is. For example, it could be **advise** or **trace**. The list of possible types is defined by the system because encapsulations are supposed to be kept in an order according to their type. See the variable **si:encapsulation-standard-order**, page 327. *type* should have an **si:encapsulation-grind-function** property that tells **grindef** what to do with an encapsulation of this type.

body-form is a form that evaluates to the body of the encapsulation-definition, the code to be executed when it is called. Backquote is typically used for this expression. See the section "Backquote", page 345. **si:encapsulate** is a macro because, while *body* is being evaluated, the variable **si:encapsulated-function** is bound to a list of the form **(function** *uninterned-symbol***)**, referring to the uninterned symbol used to hold the prior definition of *function-spec*. If **si:encapsulate** were a function, *body-form* would just get evaluated normally by the evaluator before **si:encapsulate** ever got invoked, and so there would be no opportunity to bind **si:encapsulated-function**. The form *body-form* should contain **(apply si:encapsulated-function arglist)** somewhere if the encapsulation is to live up to its name and truly serve to encapsulate the original definition. (The variable **arglist** is bound by some of the code that the **si:encapsulate** macro produces automatically. When the body of the encapsulation is run, **arglist**'s value is the list of the arguments that the encapsulation received.)

extra-debugging-info evaluates to a list of extra items to put into the debugging info alist of the encapsulation function (besides the one starting with **si:encapsulated-definition** that every encapsulation must have). Some applications find this useful for recording information about the encapsulation for their own later use.

When a special function is encapsulated, the encapsulation is itself a special

function with the same argument quoting pattern. (Not all quoting patterns can be handled; if a particular special form's quoting pattern cannot be handled, **si:encapsulate** signals an error.) Therefore, when the outermost encapsulation is started, each argument has been evaluated or not as appropriate. Because each encapsulation calls the prior definition with **apply**, no further evaluation takes place, and the basic definition of the special form also finds the arguments evaluated or not as appropriate. The basic definition can 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 breakon trace si:rename-within)
```

advise encapsulations are used to hold advice. **breakon** and **trace** encapsulations are used for implementing tracing. **si:rename-within** encapsulations are used to record the fact that function specs of the form **(:within** *within-function altered-function***)** have been defined. The encapsulation goes on *within-function*. See the section "Rename-within Encapsulations", page 329.

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** calls to process encapsulations of that type. This function need not take care of printing the encapsulated function, because **grindef** does that itself. But it should print any information about the encapsulation itself that the user ought to see. Refer to the code for the grind function for **advise** to see how to write one. See the special form **advise** in *Program Development Utilities*.

To find the right place in the ordering to insert a new encapsulation, it is necessary to parse existing ones. This is done with the function **si:unencapsulate-function-spec**.

si:unencapsulate-function-spec *function-spec* &optional *encapsulation-types* *Function*

This takes one function spec and returns another. If the original function spec is undefined, or has only a basic definition (that is, its definition is not an encapsulation), then the original function spec is returned unchanged.

If the definition of *function-spec* is an encapsulation, then its debugging info is examined to find the uninterned symbol that holds the encapsulated definition, and also the encapsulation type. If the encapsulation is of a type that is to be skipped over, the uninterned symbol replaces the original function spec and the process repeats.

The value returned is the uninterned symbol from inside the last encapsulation skipped. This uninterned symbol is the first one that does not have a definition that is an encapsulation that should be skipped. Or the value can be *function-spec* if *function-spec*'s definition is not an encapsulation that should be skipped.

The types of encapsulations to be skipped over are specified by *encapsulation-types*. This can be a list of the types to be skipped, or **nil**, meaning skip all encapsulations (this is the default). Skipping all encapsulations means returning the uninterned symbol that holds the basic definition of *function-spec*. That is, the *definition* of the function spec returned is the *basic definition* of the function spec supplied. Thus:

```
(fdefinition (si:unencapsulate-function-spec 'foo))
```

returns the basic definition of **foo**, and:

```
(fdefine (si:unencapsulate-function-spec 'foo) 'bar)
```

sets the basic definition (just like using **fdefine** with *carefully* supplied as **t**).

encapsulation-types can also be a symbol, which should be an encapsulation type; then we skip all types that are supposed to come outside of the specified type. For example, if *encapsulation-types* is **trace**, then we skip all types of encapsulations that come outside of **trace** encapsulations, but we do not skip **trace** encapsulations themselves. The result is a function spec that is where the **trace** encapsulation ought to be, if there is one. Either the

definition of this function spec is a **trace** encapsulation, or there is no **trace** encapsulation anywhere in the definition of *function-spec*, and this function spec is where it would belong if there were one. For example:

```
(let ((tem (si:unencapsulate-function-spec spec 'trace)))
  (and (eq tem (si:unencapsulate-function-spec tem '(trace)))
       (si:encapsulate tem spec 'trace '(...body...))))
```

finds the place where a **trace** encapsulation ought to go, and makes one unless there is already one there.

```
(let ((tem (si:unencapsulate-function-spec spec 'trace)))
  (fdefine tem (fdefinition (si:unencapsulate-function-spec
                              tem '(trace)))))
```

eliminates any **trace** encapsulation by replacing it by whatever it encapsulates. (If there is no **trace** encapsulation, this code changes nothing.)

These examples show how a subsystem can insert its own type of encapsulation in the proper sequence without knowing the names of any other types of encapsulations. Only the **si:encapsulation-standard-order** variable, which is used by **si:unencapsulate-function-spec**, knows the order.

27.11.1 Rename-within Encapsulations

One special kind of encapsulation is the type **si:rename-within**. This encapsulation goes around a definition in which renamings of functions have been done.

How is this used?

If you define, advise, or trace **(:within foo bar)**, then **bar** gets renamed to **altered-bar-within-foo** wherever it is called from **foo**, and **foo** gets a **si:rename-within** encapsulation to record the fact. The purpose of the encapsulation is to enable various parts of the system to do what seems natural to the user. For example, **grindef** notices the encapsulation, and so knows to print **bar** instead of **altered-bar-within-foo**, when grinding the definition of **foo**.

Also, if you redefine **foo**, or trace or advise it, the new definition gets the same renaming done (**bar** replaced by **altered-bar-within-foo**). To make this work, everyone who alters part of a function definition should pass the new part of the definition through the function **si:rename-within-new-definition-maybe**.

si:rename-within-new-definition-maybe *function-spec new-structure* *Function*

Given *new-structure* that is going to become a part of the definition of *function-spec*, perform on it the replacements described by the **si:rename-within** encapsulation in the definition of *function-spec*, if there is one. The altered (copied) list structure is returned.

It is not necessary to call this function yourself when you replace the basic definition because **fdefine** with *carefully* supplied as **t** does it for you.

si:encapsulate does this to the body of the new encapsulation. So you only need to call **si:rename-within-new-definition-maybe** yourself if you are rplac'ing part of the definition.

For proper results, *function-spec* must be the outer-level function spec. That is, the value returned by **si:unencapsulate-function-spec** is *not* the right thing to use. It has had one or more encapsulations stripped off, including the **si:rename-within** encapsulation if any, and so no renamings are done.

28. Dynamic Closures

A *closure* is a type of Lisp functional object useful for implementing certain advanced access and control structures. Closures give you more explicit control over the environment, by allowing you to save the environment created by the entering of a dynamic contour (that is, a **lambda**, **do**, **prog**, **progv**, **let**, or any of several other special forms), and then use that environment elsewhere, even after the contour has been exited.

28.1 What is a Dynamic Closure?

We use a particular view of lambda-binding in this section because it makes it easier to explain what closures do. In this view, when a variable is bound, a new value cell is created for it. The old value cell is saved away somewhere and is inaccessible. Any references to the variable get the contents of the new value cell, and any **setq**'s 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 **let** 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 **let** 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 special variables and *function* is any function, creates and returns a closure. When this closure is applied to some arguments, all the value cells of the variables on *var-list* are saved away, and the value cells that those variables had *at the time* **closure** *was called* (that is, at the time the closure was created) are made to be the value cells of the symbols. Then *function* is applied to the arguments.

Here is another, lower level explanation. The closure object stores several things inside of it. First, it saves the *function*. Secondly, for each variable in *var-list*, it

remembers what that variable's value cell was when the closure was created. Then when the closure is called as a function, it first temporarily restores the value cells it has remembered inside the closure, and then applies *function* to the same arguments to which the closure itself was applied. When the function returns, the value cells are restored to be as they were before the closure was called.

Now, if we evaluate the form (assuming that **x** has been declared special):

```
(setq a
      (let ((x 3))
        (closure '(x) 'frob)))
```

what happens is that a new value cell is created for **x**, and its contents is an integer **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 is restored and the value of **x** is **3** again. If **frob** uses **x** as a free variable, it sees **3** as the value.

A closure can be made around any function, using any form that evaluates to a function. The form could evaluate to a lambda expression, as in **'(lambda () x)**, or to a compiled function, as would **(function (lambda () x))**. In the example above, the form is **'frob** and it evaluates to the symbol **frob**. A symbol is also a good function. It is usually better to close around a symbol that is the name of the desired function, so that the closure points to the symbol. Then, if the symbol is redefined, the closure uses the new definition. If you actually prefer that the closure continue to use the old definition that was current when the closure was made, then close around the definition of the symbol rather than the symbol itself. In the above example, that would be done by:

```
(closure '(x) (function frob))
```

Because of the way dynamic closures are implemented, the variables to be closed over must be declared special. This can be done with an explicit **declare**, with a special form such as **defvar**, or with **let-closed**. In simple cases, a **declare** just inside the binding does 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 warns you if this happens.

In Symbolics-Lisp's implementation of dynamic closures, lambda-binding of special variables never really allocates any storage to create new value cells. Value cells are created only by the **closure** function itself, when they are needed. Thus, implementors of large systems need not worry about storage allocation overhead from this mechanism if they are not using dynamic closures.

Symbolics-Lisp dynamic closures are not closures in the true sense, as they do not save the whole variable-binding environment; however, most of that environment is

irrelevant, and the explicit declaration of which variables are to be closed allows the implementation to have high efficiency. They also allow you to explicitly choose for each variable whether it is to be bound at the point of call or bound at the point of definition (for example, creation of the closure), a choice which is not conveniently available in other languages. In addition, the program is clearer because the intended effect of the closure is made manifest by listing the variables to be affected.

Symbolics-Lisp also offers lexical closures, which save the variable bindings of all accessible local and instance variables. Lexical closures do not affect the bindings of special variables. There is no function to create a lexical closure; one is created automatically wherever you use a function with captured free references. See the section "Kinds of Variables", page 126. See the section "Funargs and Lexical Closure Allocation", page 139.

The implementation of dynamic closures (which is not usually necessary for you to understand) involves two kinds of value cells. Every symbol has an *internal value cell*, which is where its value is normally stored. When a variable is closed over by a closure, the variable gets an *external value cell* to hold its value. The external value cells behave according to the lambda-binding model used earlier in this section. The value in the external value cell is found through the usual access mechanisms (such as evaluating the symbol, calling **symeval**, and so on), because the internal value cell is made to contain an invisible pointer to the external value cell currently in effect. A symbol uses such an invisible pointer whenever its current value cell is a value cell that some closure is remembering; at other times, there is not an invisible pointer, and the value just resides in the internal value cell.

Most special variables that live in A-memory cannot be closed over.

28.2 Examples of the Use of Dynamic Closures

One thing we can do with dynamic closures is to implement a *generator*, which is a kind of function that is called successively to obtain successive elements of a sequence. We will implement a function **make-list-generator**, which takes a list and returns a generator that returns successive elements of the list. When it gets to the end it should return **nil**.

The problem is that in between calls to the generator, the generator must somehow remember where it is up to in the list. Since all of its bindings are undone when it is exited, it cannot save this information in a bound variable. It could save it in a global variable, but the problem is that if we want to have more than one list generator at a time, they all try to use the same global variable and get in each other's way.

Here is how we can use dynamic 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 do not get in each other's way because each has its own (external) value cell for **l**. Each of these value cells was created when the **make-list-generator** function was entered, and the value cells are remembered by the closures. We could also use lexical closures to solve the same problem.

```
(defun make-list-generator (l)
  (function (lambda ()
              (prog1 (car l)
                     (setq l (cdr l))))))
```

The following example 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** is run, and it changes the contents of the remembered value cell of **a** to **5**. If we then do **(funcall y)**, the function **bar** returns **(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.

To do this example with lexical closures, **foo** and **bar** would have to be defined with **flet** or **labels** so that they would share a lexical environment and contain captured free references to the same local variables **a** and **b**.

28.3 Dynamic Closure-manipulating Functions

closure *var-list function* *Function*

This creates and returns a dynamic closure of *function* over the variables in *var-list*. Note that all variables on *var-list* must be declared special.

To test whether an object is a dynamic closure, use the **closurep** predicate. See the section "Predicates", page 7. The **typep** function returns the symbol **closure** if given a dynamic closure. **(typep** *x* **:closure)** is equivalent to **(closurep** *x***)**.

symeval-in-closure *closure symbol* *Function*

This returns the binding of *symbol* in the environment of *closure*; that is, it returns what you would get if you restored the value cells known about by *closure* and then evaluated *symbol*. This allows you to "look around inside" a dynamic closure. If *symbol* is not closed over by *closure*, this is just like **symeval**.

set-in-closure *closure symbol x* *Function*

This sets the binding of *symbol* in the environment of *closure* to *x*; that is, it does what would happen if you restored the value cells known about by *closure* and then set *symbol* to *x*. This allows you to change the contents of the value cells known about by a dynamic closure. If *symbol* is not closed over by *closure*, this is just like **set**.

locate-in-closure *closure symbol* *Function*

This returns the location of the place in *closure* where the saved value of *symbol* is stored. An equivalent form is
(locf (symeval-in-closure *closure symbol***))**.

closure-alist *closure* *Function*

Returns an alist of **(***symbol* **.** *value***)** pairs describing the bindings which the dynamic closure performs when it is called. This list is not the same one that is actually stored in the closure; that one contains pointers to value cells rather than symbols, and **closure-alist** translates them back to symbols so you can understand them. As a result, clobbering part of this list does not change the closure.

If any variable in the closure is unbound, this function signals an error.

closure-function *closure* *Function*

Returns the closed function from *closure*. This is the function that was the second argument to **closure** when the dynamic closure was created.

let-closed *((variable value)...) function* *Special Form*

When using dynamic 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". **let-closed** is a special form that does all of this. It is best described by example:

```
(let-closed ((a 5) b (c 'x))
   (function (lambda () ...)))
```

macro-expands into

```
(let ((a 5) b (c 'x))
  (declare (special a b c ))
  (closure '(a b c)
           (function (lambda () ...)))))
```

copy-closure *closure* *Function*

Creates and returns a new closure by copying *closure*, which should be a dynamic closure. **copy-closure** generates new external value cells for each variable in the closure and initializes their contents from the external value cells of *closure*.

closure-variables *closure* *Function*

Creates and returns a list of all of the variables in *closure*, which should be a dynamic closure. It returns a copy of the list that was passed as the first argument to **closure** when *closure* was created.

boundp-in-closure *closure symbol* *Function*

Returns **t** if *symbol* is bound in the environment of *closure*; that is, it does what **boundp** would do if you restored the value cells known about by *closure*. If *symbol* is not closed over by *closure*, this is just like **boundp**.

makunbound-in-closure *closure symbol* *Function*

Makes *symbol* be unbound in the environment of *closure*; that is, it does what **makunbound** would do if you restored the value cells known about by *closure*. If *symbol* is not closed over by *closure*, this is just like **makunbound**.

A note about all of the *xxx*-**in-closure** functions (**set-**, **symeval-**, **boundp-**, and **makunbound-**): if the variable is not directly closed over, the variable's value cell from the global environment is used. That is, if closure A closes over closure B, *xxx*-**in-closure** of A does not notice any variables closed over by B.

PART VIII.

Macros

29. 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 macro, it contains an expander function. **eval** applies the expander function to two arguments, the form that **eval** is trying to evaluate and an object representing the lexical environment. The expander function returns a new form and **eval** evaluates that in lieu of the original form.

A macro definition is represented as a list whose first element is the symbol **special**, whose second element is the expander function, and whose third element is missing or **nil**. (The third element is used by a special form definition.) **eval** applies the function to the form it was originally given, takes the expansion that is returned, and evaluates that in lieu of the original form.

Here is a simple example. Suppose the definition of the symbol **first** is:

```
(special (lambda (x env)
           (list 'car (cadr x))))
```

This thing is a macro. What happens if we try to evaluate a form **(first '(a b c))**? **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 the definition is a macro.

eval gets the macro's *expander function*, and calls it providing as arguments the original form that **eval** was handed and the lexical environment. So it calls **(lambda (x env) (list 'car (cadr x)))** with arguments **(first '(a b c))** and the lexical environment. Whatever this returns is the *expansion* of the macro call. It is evaluated in place of the original form.

In this case, **x** is bound to **(first '(a b c))**, **(cadr x)** evaluates to **'(a b c)**, and **(list 'car (cadr x))** evaluates to **(car '(a b c))**, which is the expansion. **eval** now evaluates the expansion. **(car '(a b c))** returns **a**, and so the result is that **(first '(a b c))** returns **a**.

What have we done? We have defined a macro called **first**. What the macro does is to *translate* the form to some other form. Our translation is very simple — it just translates forms that look like **(first** *x***)** into **(car** *x***)**, for any form *x*. We can do much more interesting things with macros, but first we show how to define a macro.

macro *Special Form*

The primitive special form for defining macros is **macro**. A macro definition looks like this:

```
(macro name (form env)
  body)
```

name can be any function spec. *form* and *env* must be variables. *body* is a sequence of Lisp forms that expand the macro; the last form should return the expansion. **defmacro** is usually preferred in practice.

To define our **first** macro, we would say:

```
(macro first (x ignore)
   (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 ignore)
  (list 'plus '1 (cadr x))))
```

Now say we wanted a macro that would translate **(increment** *x***)** into **(setq** *x* **(1+** *x***)**. This would be:

```
(macro increment (x ignore)
   (list 'setq (cadr x) (list '1+ (cadr x)))
```

Of course, this macro is of limited usefulness, because the form in the *cadr* of the **increment** form should 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 are interested in how to fix this problem: See the macro **setf**, page 147. However, this is irrelevant to how macros work.)

As you can see, 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 can look at the whole form and see what it is doing. Macros are not functions; if **first** is defined as a macro, it is not meaningful to apply **first** to arguments. A macro does not take arguments at all; its expander function takes a Lisp form and turns it into another Lisp form.

The purpose of functions is to *compute*; the purpose of macros is to *translate*. Macros are used for a variety of purposes, the most common being extensions to the Lisp language. For example, Lisp is powerful enough to express many different control structures, but it does not provide every control structure anyone might ever possibly want. Instead, if you want some kind of control structure with a syntax that is not provided, you can translate it into some form that Lisp *does* know about.

For example, you might want a limited iteration construct that increments a variable by one until it exceeds a limit (like the FOR statement of the BASIC language). You might want it to look like:

```
(for a 1 100 (print a) (print (* a a)))
```

To get this, you could write a macro to translate it into:

```
(do a 1 (1+ a) (> a 100) (print a) (print (* a a)))
```

A macro to do this could be defined with:

```
(macro for (x ignore)
  (cons 'do
        (cons (cadr x)
              (cons (caddr x)
                    (cons (list '1+ (cadr x))
                          (cons (list '> (cadr x) (cadddr x))
                                (cddddr x)))))))
```

Now you have defined your own new control structure primitive, and it will act just as if it were a special form provided by Lisp itself.

30. Aids for Defining Macros

The main problem with the following definition is that it is verbose and clumsy:

```
(macro for (x ignore)
  (cons 'do
        (cons (cadr x)
              (cons (caddr x)
                    (cons (list '1+ (cadr x))
                          (cons (list '> (cadr x) (cadddr x))
                                (cddddr x)))))))
```

If it is that hard to write a macro to do a simple specialized iteration construct, you might wonder how anyone could write macros of any real sophistication.

There are two things that make the definition so inelegant. One is that you must write things like **(cadr x)** and **(cddddr x)** to refer to the parts of the form you want to do things with. The other problem is that the long chains of calls to the **list** and **cons** functions are very hard to read.

Two features are provided to solve these two problems. The **defmacro** macro solves the former, and the "backquote" (`) reader macro solves the latter.

30.1 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))))))
```

(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 first argument to the macro's expander function). If **defmacro** tries to match the following two lists:

```
(var lower upper . body)
(a 1 100 (print a) (print (* a a)))
```

var gets bound to the symbol **a**, **lower** to the integer **1**, **upper** to the integer **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* can 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 occurs in *pattern*, the symbol is bound to the corresponding part of the form. If the corresponding part of the form is **nil**, it goes off the end of the form. **&optional**, **&rest**, **&key**, and **&body** can be used to indicate where optional pattern elements are allowed.

All of the symbols in *pattern* can be used as variables within *body*. *name* is the name of the macro to be defined; it can be any function spec. See the section "Function Specs", page 297. *body* is evaluated with these bindings in effect, and its result is returned to the evaluator as the expansion of the macro.

defmacro could have been defined in terms of **destructuring-bind** as follows, except that the following is a simplified example of **defmacro** showing no error-checking and omitting the **&environment** and **&whole** features.

```
(defmacro defmacro (name pattern &body body)
  `(macro ,name (form env)
     (destructuring-bind ,pattern (cdr form)
       . ,body)))
```

See the special form **destructuring-bind**, page 132.

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 are very simple macros and have very simple patterns, but they show that we can replace **(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** that defines displacing macros. See the section "Displacing Macros", page 371. **defmacro** has other, more complex features. See the section "&-Keywords Accepted by **defmacro**", page 373. A way to define local macros is discussed elsewhere. See the function **macrolet**, page 144.

30.2 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***)**, some more new special characters make it easier to type forms that create new list structure. The functionality provided by these characters is called the *backquote* facility.

The backquote facility is used by giving a backquote character (`), followed by a form. If the form does not contain any use of the comma character, the backquote acts just like a single quote: it creates a form that, when evaluated, produces the form following the backquote. For example:

```
'(a b c) => (a b c)
`(a b c) => (a b c)
```

So in the simple cases, backquote is just like the regular single-quote macro. The way to get it to do interesting things is to include a comma somewhere inside the form following the backquote. The comma is followed by a form, and that form gets evaluated even though it is inside the backquote. For example:

```
(setq b 1)
`(a b c)  => (a b c)
`(a ,b c) => (a 1 c)
`(abc ,(+ b 4) ,(- b 1) (def ,b)) => (abc 5 0 (def 1))
```

In other words, backquote quotes everything *except* things preceded by a comma; those things get evaluated.

A list following a backquote can be thought of as a template for some new list structure. The parts of the list that are preceded by commas are forms that fill in slots in the template; everything else is just constant structure that appears 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 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 might use **list**, **cons**, **append**, or whatever might be a good idea; you should never have to concern yourself with what it actually turns into. All you need to care about is what it evaluates to. Actually, it does not use regular functions such as **cons** and **list**, but uses special ones instead so that the grinder can recognize a form that was created with the backquote syntax, and print it using backquote so that it looks like what you typed in. You should never write any program that depends on this, anyway, because backquote makes no guarantees about how it does what it does. In particular, in some circumstances it might decide to create constant forms that cause sharing of list structure at run time, or it might decide to create forms that create new list structure at run time. For example, if the reader sees **'(r . ,nil)**, it might produce the same thing as **(cons 'r nil)**, or **'(r . nil)**. Be careful that your program does not depend on one of these.

The following examples might make this behavior clearer. Here is how we would write our three simple macros using both the **defmacro** and backquote facilities.

```
(defmacro first (the-list)
    `(car ,the-list))

(defmacro addone (form)
   `(plus 1 ,form))

(defmacro increment (symbol)
   `(setq ,symbol (1+ ,symbol)))
```

Finally, to demonstrate how easy it is to define macros with these two facilities, here is the final form of the **for** macro.

```
(defmacro for (var lower upper . body)
  `(do ,var ,lower (1+ ,var) (> ,var ,upper) . ,body))
```

Look at how much simpler that is than the original definition. Also, look how closely it resembles the code it is producing. The functionality of the **for** stands out when written this way.

If a comma inside a backquote form is followed by an at-sign character (@), it has a special meaning. The **,@** should be followed by a form whose value is a *list*; then each of the elements of the list is put into the list being created by the backquote.

In other words, instead of generating a call to the **cons** function, backquote generates a call to **append**. For example, if **a** is bound to **(x y z)**, then **'(1 ,a 2)** would evaluate to **(1 (x y z) 2)**, but **'(1 ,@a 2)** would evaluate to **(1 x y z 2)**.

Here is an example of a macro definition that uses the ",@" construction. Suppose you wanted to extend Lisp by adding a kind of special form called **repeat-forever**, which evaluates all of its subforms repeatedly. One way to implement this would be to expand:

```
(repeat-forever form1 form2 form3)
```

into:

```
(prog ()
    a form1
      form2
      form3
      (go a))
```

You could define the macro by:

```
(defmacro repeat-forever body
       '(prog ()
            a ,@body
              (go a)))
```

A similar construct is ",." (comma, dot). This means the same thing as ",@" except that the list that is the value of the following form can be freely smashed; backquote uses **nconc** rather than **append**. This should of course be used with caution.

Backquote does not make any guarantees about what parts of the structure it shares and what parts it copies. You should not do destructive operations such as **nconc** on the results of backquote forms such as:

```
'(,a b c d)
```

since backquote might choose to implement this as:

```
(cons a '(b c d))
```

and **nconc** would smash the constant. On the other hand, it would be safe to **nconc** the result of:

```
'(a b ,c ,d)
```

since there is nothing this could expand into that does not involve making a new list, such as:

```
(list 'a 'b c d)
```

Backquote of course guarantees not to do any destructive operations (**rplaca, rplacd, nconc**) on the components of the structure it builds, unless the ,. syntax is used.

Advanced macro writers sometimes write "macro-defining macros": forms that expand into forms that, when evaluated, define macros. In such macros it is often

useful to use nested backquote constructs. The following example illustrates the use of nested backquotes in the writing of macro-defining macros.

This example is a very simple version of **defstruct**. You should first understand the basic description of **defstruct** before proceeding with this example. The **defstruct** below does not accept any options, and allows only the simplest kind of items; that is, it only allows forms like:

```
(defstruct (name)
     item1
     item2
     item3
     item4
     ...)
```

We would like this form to expand into:

```
(progn
 (defmacro item1 (x)
      '(aref ,x 0))
 (defmacro item2 (x)
      '(aref ,x 1))
 (defmacro item3 (x)
      '(aref ,x 2))
 (defmacro item4 (x)
      '(aref ,x 3))
 ...)
```

The meaning of the **(progn ...)** is discussed in another section. See the section "Macros Expanding Into Many Forms", page 360. Here is the macro to perform the expansion:

```
(defmacro defstruct ((name) . items)
     (do ((item-list items (cdr item-list))
          (ans nil)
          (i 0 (1+ i)))
         ((null item-list)
          '(progn . ,(nreverse ans)))
       (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, ",'," acts like a comma that matches the outer backquote, while "," preceding the "**x**" matches with the inner backquote. Thus, the symbol **i** is evaluated when the **defstruct** form is expanded, whereas the symbol **x** is evaluated when the accessor macros are expanded.

Backquote can be useful in situations other than the writing of macros. Whenever there is a piece of list structure to be consed up, most of which is constant, the use of backquote can make the program considerably clearer.

31. Substitutable Functions

A substitutable function is a function that is open-coded by the compiler. It is like any other function when applied, but it can be expanded instead, and in that regard resembles a macro.

defsubst *Special Form*

Used for defining substitutable functions. It is used just like **defun** and does almost the same thing.

`(defsubst` *name lambda-list . body*`)`

defsubst defines a function that executes identically to the one that a similar call to **defun** would define. The difference comes when a function that *calls* this one is compiled. Then, the call is open-coded by substituting the substitutable function's definition into the code being compiled. Such a function is called a **subst**. For example, if we define:

```
(defsubst square (x) (* x x))

(defun foo (a b) (square (+ a b)))
```

then if **foo** is used interpreted, **square** works just as if it had been defined by **defun**. If **foo** is compiled, however, the squaring is substituted into it and it compiles just like:

```
(defun foo (a b) (* (+ a b) (+ a b)))
```

A similar **square** could be defined as a macro, with:

```
(defmacro square (x) '(* ,x ,x))
```

When the compiler open-codes a **subst**, it binds the argument variables to the argument values with **let**, so they get evaluated only once and in the right order. Then, when possible, the compiler optimizes out the variables. In general, anything that is implemented as a **subst** can be reimplemented as a macro, just by changing the **defsubst** to a **defmacro** and putting in the appropriate backquote and commas, except that this does not get the simultaneous guarantee of argument evaluation order and generation of optimal code with no unnecessary temporary variables. 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 **subst**s 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**.

As with **defun**, *name* can be any function spec, but you get the "**subst**" effect only when *name* is a symbol.

The difference between a **subst** and a **lambda** is the way they are handled by the compiler. A call to a normal function is compiled as a *closed subroutine*; the compiler generates code to compute the values of the arguments and then apply the function to those values. A call to a **subst** is compiled as an *open subroutine*; the compiler incorporates the body forms of the **subst** into the function being compiled, substituting the argument forms for references to the variables in the **subst**'s *lambda-list*. This is a simple but useful facility for *open* or *in-line coded* functions.

32. Symbol Macros

A symbol macro translates a symbol into a substitute form. When the Lisp evaluator is given a symbol, it checks whether the symbol has been defined as a symbol macro. If so, it evaluates the symbol's replacement form instead of the symbol itself. Use **define-symbol-macro** to define a symbol macro.

define-symbol-macro *name form* *Special Form*

Defines a symbol macro. *name* is a symbol to be defined as a symbol macro. *form* is a Lisp form to be substituted for the symbol when the symbol is evaluated. A symbol macro is more like a subst than a macro: *form* is the form to be substituted for the symbol, not a form whose evaluation results in the substitute form.

Example:

```
(define-symbol-macro foo (+ 3 bar))
(setq bar 2)
foo => 5
```

A symbol defined as a symbol macro cannot be used in the context of a variable. You cannot use **setq** on it, and you cannot bind it. You can use **setf** on it: **setf** substitutes the replacement form, which should access something, and expands into the appropriate update function.

For example, suppose you want to define some new instance variables and methods for a flavor. You want to test the methods using existing instances of the flavor. For testing purposes, you might use hash tables to simulate the instance variables, using one hash table per instance variable with the instance as the key. You could then implement an instance variable **x** as a symbol macro:

```
(defvar x-hash-table (make-hash-table))
(define-symbol-macro x (send x-hash-table :get-hash self))
```

To simulate setting a new value for **x**, you could use **(setf x** *value***)**, which would expand into **(send x-hash-table :put-hash self** *value***)**.

33. Lambda Macros

Lambda macros are similar to regular Lisp macros, except that regular Lisp macros replace, and expand into, Lisp forms, whereas lambda macros replace, and expand into, Lisp functions. They are an advanced feature, used only for certain special language extensions or embedded programming systems.

To understand what lambda macros do, consider how regular Lisp macros work. When the evaluator is given a Lisp form to evaluate, it inspects the car of the form to figure out what to do. If the car is the name of a function, the function is called. But if the car is the name of a macro, the macro is expanded, and the result of the expansion is considered to be a Lisp form and is evaluated. Lambda macros work analogously, but in a different situation. When the evaluator finds that the car of a form is a list, it looks at the car of this list to figure out what to do. If this car is the symbol **lambda**, the list is an ordinary function, and it is applied to its arguments. But if this car is the name of a lambda macro, the lambda macro is expanded, and the result of the expansion is considered to be a Lisp function and is applied to the arguments.

Like regular macros, lambda macros are named by symbols and have a body, which is a function of one argument. To expand the lambda macro, the evaluator applies this body to the entire lambda macro function (the list whose car is the name of the lambda macro), and expects the body to return another function as its value.

Several special forms are provided for dealing with lambda macros. The primitive for defining a new lambda macro is **lambda-macro**; it is analogous to the **macro** special form. For convenience, **deflambda-macro** and **deflambda-macro-displace** are defined; these work like **defmacro** to provide easy parsing of the function into its component parts. The special form **deffunction** creates a new Lisp function that uses a named lambda macro instead of **lambda** in its definition.

lambda-macro *name lambda-list body...* *Special Form*

Like **macro**, defines a lambda macro to be called *name*. *lambda-list* should be a list of one variable, which is bound to the function being expanded. The lambda macro must return a function. Example:

```
(lambda-macro ilisp (x)
  '(lambda (&optional ,@(second x) &rest ignore) . ,(cddr x)))
```

This defines a lambda macro called **ilisp**. After it has been defined, the following list is a valid Lisp function:

```
(ilisp (x y z) (list x y z))
```

The above function takes three arguments and returns a list of them, but all of the arguments are optional and any extra arguments are ignored. (This shows how to make functions that imitate Interlisp functions, in which all

arguments are always optional and extra arguments are always ignored.) So, for example:

```
(funcall #'(ilisp (x y z) (list x y z)) 1 2)  =>  (1 2 nil)
```

deflambda-macro *Special Form*

Like **defmacro**, but defines a lambda macro instead of a normal macro.

deflambda-macro-displace *Special Form*

Like **defmacro-displace**, but defines a displacing lambda macro instead of a displacing normal macro.

deffunction *function-spec lambda-macro-name lambda-list body...* *Special Form*

Defines a function using an arbitrary lambda macro in place of **lambda**. A **deffunction** form is like a **defun** form, except that the function spec is immediately followed by the name of the lambda macro to be used. **deffunction** expands the lambda macro immediately, so the lambda macro must already be defined before **deffunction** is used. For example, suppose the **ilisp** lambda macro were defined as follows:

```
(lambda-macro ilisp (x)
  '(lambda (&optional ,@(second x) &rest ignore) . ,(cddr x)))
```

Then the following example would define a function called **new-list** that would use the lambda macro called **ilisp**:

```
(deffunction new-list ilisp (x y z)
  (list x y z))
```

new-list's arguments are optional, and any extra arguments are ignored. Examples:

```
(new-list 1 2) => (1 2 nil)
(new-list 1 2 3 4) -> (1 2 3)
```

Lambda macro-expander functions can be accessed with the (**:lambda-macro** *name*) function spec.

34. Hints to Macro Writers

Over the years, Lisp programmers have discovered useful techniques for writing macros, and have identified pitfalls that must be avoided. This section discusses some of these techniques, and illustrates them with examples.

The most important thing to keep in mind as you learn to write macros is that you should first figure out what the macro form is supposed to expand into, and only then should you start to actually write the code of the macro. If you have a firm grasp of what the generated Lisp program is supposed to look like, from the start, you will find the macro much easier to write.

In general any macro that can be written as a substitutable function should be written as one, not as a macro, for several reasons:

- Substitutable functions are easier to write and to read.
- They can be passed as functional arguments (for example, you can pass them to **mapcar**).
- Some subtleties can occur in macro definitions that need not be worried about in substitutable functions.

See the section "Substitutable Functions", page 351. A macro can be a substitutable function only if it has the exact semantics of a function, rather than a special form.

34.1 Name Conflicts

One of the most common errors in writing macros is best illustrated by example. Suppose we wanted to write **dolist** as a macro that expanded into a **do**. The first step, as always, is to figure out what the expansion should look like. Let's pick a representative example form, and figure out what its expansion should be. Here is a typical **dolist** form.

```
(dolist (element (append a b))
  (push element *big-list*)
  (foo element 3))
```

We want to create a **do** form that does the thing that the above **dolist** form says to do. That is the basic goal of the macro: it must expand into code that does the same thing that the original code says to do, but it should be in terms of existing Lisp constructs. The **do** form might look like this:

```
(do ((list (append a b) (cdr list))
     (element))
    ((null list))
  (setq element (car list))
  (push element *big-list*)
  (foo element 3))
```

Now we could start writing the macro that would generate this code, and in general convert any **dolist** into a **do**, in an analogous way. However, there is a problem with the above scheme for expanding the **dolist**. The above expansion works fine. But what if the input form had been the following:

```
(dolist (list (append a b))
  (push list *big-list*)
  (foo list 3))
```

This is just like the form we saw above, except that the user happened to decide to name the looping variable **list** rather than **element**. The corresponding expansion would be:

```
(do ((list (append a b) (cdr list))
     (list))
    ((null list))
  (setq list (car list))
  (push list *big-list*)
  (foo list 3))
```

This does not work at all! In fact, this is not even a valid program, since it contains a **do** that uses the same variable in two different iteration clauses.

Here is another example that causes trouble:

```
(let ((list nil))
  (dolist (element (append a b))
    (push element list)
    (foo list 3)))
```

If you work out the expansion of this form, you 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, it might conflict with some other name that is in the user's program.

One way to avoid this problem is to choose a name that is very unlikely to be picked by the user, simply by choosing an unusual name. This will probably work, but it is inelegant since there is no guarantee that the user will not happen to choose the same name. The only sure way to avoid the name conflict is to use an uninterned symbol as the variable in the generated code. The function **gensym** is useful for creating such symbols.

Here is the expansion of the original form, using an uninterned symbol created by **gensym**.

```
(do ((#:g0005 (append a b) (cdr #:g0005))
     (element))
    ((null #:g0005))
  (setq element (car #:g0005))
  (push element *big-list*)
  (foo element 3))
```

This is the right kind of thing to expand into. Now that we understand how the expansion works, we are ready to actually write the macro. Here it is:

```
(defmacro dolist ((var form) . body)
  (let ((dummy (gensym)))
    `(do ((,dummy ,form (cdr ,dummy))
          (,var))
         ((null ,dummy))
       (setq ,var (car ,dummy))
       . ,body)))
```

Many system macros do not use **gensym** for the internal variables in their expansions. Instead they use symbols whose print names begin and end with a dot. This provides meaningful names for these variables when looking at the generated code and when looking at the state of a computation in the Debugger. However, this convention means that users should avoid naming variables this way.

34.2 prog-Context Conflicts

A related problem occurs when you write a macro that expands into a **prog** (or a **do**, or something that expands into **prog** or **do**) behind the user's back (unlike **dolist**, which is documented to be like **do**).

Consider the **error-restart** special form; suppose we wanted to implement it as a macro that expands into a **prog**. If it expanded into a standard **prog**, then the following (contrived) Lisp program would not behave correctly:

```
(prog ()
   (setq a 3)
   (error-restart (error "Try again")
     (cond ((> a 10)
            (return 5))
           ((> a 4)
            (fsignal 'lose "You lose."))))
   (setq b 7))
```

The problem is that the **return** returns 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** passes right through the **error-restart** form and returns 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 return to the right place, unless the macro is documented as generating a **prog/do**-like form that can be exited with **return**.

34.3 Macros Expanding Into Many Forms

Sometimes a macro wants to do several different things when its expansion is evaluated. Another way to say this is that sometimes a macro wants to expand into several things, all of which should happen sequentially at run time (not macro-expand time). For example, suppose you wanted to implement **defconst** as a macro. **defconst** must do two things: declare the variable to be special, and set the variable to its initial value. To simplify the example, we implement a simplified **defconst** that does only these two things, and does not have any options. What should a **defconst** form expand into? What we would like is for an appearance of:

```
(defconst a (+ 4 b))
```

in a file to be the same thing as the appearance of the following two forms:

```
(declare (special a))
(setq a (+ 4 b))
```

However, because of the way that macros work, they expand into only one form, not two. So we need to have a **defconst** form expand into one form that is just like having two forms in the file.

There is such a form. It looks like this:

```
(progn
  (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. 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. When it sees such a form, it processes each of the 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
     (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, the **define-command** form provides two things: a function that executes the command, and a text string that contains the documentation for the command (in order to provide an online interactive documentation feature). This macro is a simplified version of a macro that is actually used in the Zwei editor. Suppose that in this subsystem, commands are always functions of no arguments, documentation strings are placed on the **help** property of the name of the command, and the names of all commands are put onto a list. A typical call to **define-command** would look like:

```
(define-command move-to-top
   "This command moves you to the top."
   (do ()
       ((at-the-top-p))
     (move-up-one)))
```

This could expand into:

```
(progn
       (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. For more discussion of the differences among compile time, load time, and eval time: See the function **eval-when** in *Program Development Utilities*.

This technique makes Lisp a powerful language in which to implement your own language. When you write a large system in Lisp, frequently you can make things much more convenient and clear by using macros to extend Lisp into a customized

language for your application. In the above example, we have created a little language extension: a new special form that defines commands for our system. It lets the writer of the system put documentation strings right next to the code that they document, so that the two can be updated and maintained together. The way that the Lisp environment works, with load-time evaluation able to build data structures, lets the documentation database and the list of commands be constructed automatically.

34.4 Macros That Surround Code

There is a particular kind of macro that is very useful for many applications. This is a macro that you place "around" some Lisp code, in order to make the evaluation of that code happen in some context. For a very simple example, we could define a macro called **with-output-in-base**, that executes the forms within its body with any output of numbers that is done defaulting to a specified base.

```
(defmacro with-output-in-base ((base-form) &body body)
   `(let ((base ,base-form))
      . ,body))
```

A typical use of this macro might look like:

```
(with-output-in-base (*default-base*)
   (print x)
   (print y))
```

that would expand into:

```
(let ((base *default-base*))
  (print x)
  (print y))
```

This example is too trivial to be very useful; it is intended to demonstrate some stylistic issues. Some special forms are similar to this macro. See the special form **with-open-file** in *Reference Guide to Streams, Files, and I/O*. See the special form **with-input-from-string**, page 290. 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 is 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 *with* the following things true". Another convention is that any "parameters" to the special form should appear in a list that is the first subform of the special form, and that the rest of the subforms should make up a body of forms that are evaluated sequentially with the last one returned. All of the examples cited above work this way. In our **with-output-in-base** example, there was one parameter (the base), which appears as the first (and only) element of a list that is the first subform of the special form. The extra level of parentheses in the printed representation serves to separate the "parameter" forms from the "body" forms so that it is textually apparent which is which; it also provides a convenient way to provide default parameters (a good example is the **with-input-from-string** special form, which takes two required and two optional "parameters"). Another convention/technique is to use the **&body** keyword in the **defmacro** to tell the editor how to correctly indent the special form. See the section "&-Keywords Accepted by **defmacro**", page 373.

The other thing to remember is that control can leave the special form either by the last form's returning, or by a nonlocal exit (that is, something doing a **throw**). You should write the special form in such a way that everything is cleaned up appropriately no matter which way control exits. In our **with-output-in-base** example, there is no problem, because nonlocal exits undo lambda-bindings. However, in even slightly more complicated cases, an **unwind-protect** form is needed: the macro must expand into an **unwind-protect** that surrounds the body, with "cleanup" forms that undo the context-setting-up that the macro did. For example, **using-resource** is implemented as a macro that does an **allocate-resource** and then performs the body inside of an **unwind-protect** that has a **deallocate-resource** in its "cleanup" forms. This way the allocated resource item is deallocated whenever control leaves the **using-resource** special form.

34.5 Multiple and Out-of-order Evaluation

In any macro, you should always pay attention to the problem of multiple or out-of-order evaluation of user subforms. Here is an example of a macro with such a problem. This macro defines a special form with two subforms. The first is a reference, and the second is a form. The special form is defined to create a cons whose car and cdr are both the value of the second subform, and then to set the reference to be that cons. Here is a possible definition:

```
(defmacro test (reference form)
   `(setf ,reference (cons ,form ,form)))
```

Simple cases work all right:

```
(test foo 3) ==>
  (setf foo (cons 3 3))
```

But a more complex example, in which the subform has side effects, can produce surprising results:

```
(test foo (setq x (1+ x))) ==>
  (setf foo (cons (setq x (1+ x))
                  (setq x (1+ x))))
```

The resulting code evaluates the **setq** form twice, and so **x** is increased by two instead of by one. A better definition of **test** that avoids this problem is:

```
(defmacro test (reference form)
   (let ((value (gensym)))
     `(let ((,value ,form))
        (setf ,reference (cons ,value ,value)))))
```

With this definition, the expansion works as follows:

```
(test foo (setq x (1+ x))) ==>
  (let ((#:g0005 (setq x (1+ x))))
    (setf foo (cons #:g0005 #:g0005)))
```

In general, when you define a new special form that has some forms as its subforms, you have to be careful about when those forms get evaluated. If you are not careful, they can get evaluated more than once, or in an unexpected order, and this can be semantically significant if the forms have side effects. There is nothing fundamentally wrong with multiple or out-of-order evaluation if that is really what you want and if it is what you document your special form to do. However, it is very common for special forms to simply behave like functions, and when they are doing things like what functions do, it is natural to expect them to be function-like in the evaluation of their subforms. Function forms have their subforms evaluated, each only once, in left-to-right order, and special forms that are similar to function forms should try to work that way too for clarity and consistency.

The macro **once-only** makes it easier for you to follow the principle explained above. 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 evaluated only once, and references to **form** inside the macro body refer to that value. **once-only** automatically introduces a lambda-binding of a generated symbol to hold the value of the form. Actually, it is more clever than that; it avoids introducing the lambda-binding for forms whose evaluation is trivial and may be repeated without harm or cost, such as numbers, symbols, and quoted structure. This is just an optimization that helps produce more efficient code.

The **once-only** macro makes it easier to follow the principle, but it does not completely nor automatically solve the problems of multiple and out-of-order evaluation. It is just a tool that can solve some of the problems some of the time; it is not a panacea.

The following description attempts to explain what **once-only** does, but it is much

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, it first inspects the values of each of the variables in *var-list*; these values are assumed to be Lisp forms. It binds each variable either to its current value, if the current value is a trivial form, or to a generated symbol. Next, **once-only** evaluates the *forms*, in this new binding environment, and when they have been evaluated it undoes the bindings. The result of the evaluation of the last *form* is presumed to be a Lisp form, typically the expansion of a macro. If all of the variables had been bound to trivial forms, then *once-only* just returns that result. Otherwise, **once-only** returns the result wrapped in a lambda-combination that binds the generated symbols to the result of evaluating the respective nontrivial forms.

The effect is that the program produced by evaluating the **once-only** form is coded in such a way that it only evaluates each form once, unless evaluation of the form has no side effects, for each of the forms that were the values of variables in *var-list*. At the same time, no unnecessary lambda-binding appears in this program, but the body of the **once-only** is not cluttered up with extraneous code to decide whether or not to introduce lambda-binding in the program it constructs.

Note well: while **once-only** attempts to prevent multiple evaluation, it does *not* necessarily preserve the *order* of evaluation of the forms! Since it generates the new bindings, the evaluation of complex forms (for which a new variable needs to be created) may be moved ahead of the evaluation of simple forms (such as variable references). **once-only** does not solve all of the problems mentioned in this section.

Caution! A number of system macros, **setf** for example, fail to follow this convention. Unexpected multiple evaluation and out-of-order evaluation can occur with them. This was done for the sake of efficiency and is prominently mentioned in the documentation of these macros. It would be best not to compromise the semantic simplicity of your own macros in this way. (**cl:setf** and related macros follow the convention correctly.)

34.6 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? **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** 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 first sees the **with-collection** form, and calls in the expander function to expand it. The expander function creates the expansion, and returns to the evaluator, which then evaluates the expansion. The expansion includes in it a **let** form to bind ***collect-variable*** to the generated symbol. When the evaluator sees this **let** form during the evaluation of the expansion of the **with-collection** form, it sets up the binding and recursively evaluates 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 is able to see the value of **collection-variable** and incorporate the generated symbol into its own expansion.

Writing the macros this way is not quite right. It works fine interpreted, but the problem is that it does not work when we try to compile Lisp code that uses these special forms. When code is being compiled, there is no interpreter to do the binding in our new **let** form; macro expansion is done at compile time, but generated code is not run until the results of the compilation are loaded and run. The way to fix our definitions is to use **compiler-let** instead of **let**. **compiler-let** is a special form that exists specifically to do the sort of thing we are trying to do here. **compiler-let** is identical to **let** as far as the interpreter is concerned, so changing our **let** to a **compiler-let** does not affect the behavior in the interpreter; it continues to work. When the compiler encounters a **compiler-let**, however, it actually performs the bindings that the **compiler-let** specifies, and proceeds to compile the body of the **compiler-let** with all of those bindings in effect. In other words, it acts as the interpreter would.

Here is the right way to write these macros:

```
(defvar *collect-variable*)

(defmacro with-collection (&body body)
  (let ((var (gensym)))
     `(let ((,var nil))
        (compiler-let ((*collect-variable* ',var))
           . ,body)
        (nreverse ,var))))

(defmacro collect (argument)
  `(push ,argument ,*collect-variable*))
```

Another way to write this type of macro is to use **macrolet** to create a definition of **collect** local to the body of **with-collection**. Example:

```
(defmacro with-collection (&body body)
  (let ((var (gensym)))
    `(let ((,var nil))
       (macrolet ((collect (argument)
                    `(push ,argument ,',var)))
         . ,body)
       (nreverse ,var))))

;To make COLLECT known to editing tools, and to get a better error
;message if it is used in the wrong place, we define a global definition
;that will be shadowed by the MACROLET.  The error message for misuse of
;COLLECT comes out at both compile-time and run-time.

(defmacro collect (argument)
  (compiler:warn () "~S used outside of ~S"
                 'collect 'with-collection)
  `(ferror "~S used outside of ~S"
           '(collect ,,argument) 'with-collection))
```

34.7 Functions Used During Expansion

The technique of defining functions to be used during macro expansion deserves explicit mention. A macro-expander function is a Lisp program like any other Lisp program, and it can benefit in all the usual ways by being broken down into a collection of functions that do various parts of its work. Usually macro-expander functions are pretty simple Lisp programs that take things apart and put them together slightly differently, but some macros are quite complex and do a lot of work. Several features of Symbolics-Lisp, 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 **(eval-when (compile load eval) ...)**. Doing this means that at compile time the definition of the function is interpreted, not compiled, and thus runs more slowly. Another approach is to separate macro definitions and the functions they call during expansion into a separate file, often called a "defs" (definitions) file. This file defines all the macros but does not use any of them. It can be separately compiled and loaded up before compiling the main part of the program, which uses the macros. The *system* facility helps keep these various files straight, compiling and loading things in the right order. See the section "Maintaining Large Programs" in *Program Development Utilities*.

34.8 Aid for Debugging Macros

mexp *Function*

mexp goes into a loop in which it reads forms and sequentially expands them, printing out the result of each expansion (using the grinder to improve readability). See the section "Formatting Lisp Code" in *Reference Guide to Streams, Files, and I/O*. It terminates when you press the END key. If you type in a form that is not a macro form, there are no expansions and so it does not type anything out, but just prompts you for another form. This allows you to see what your macros are expanding into, without actually evaluating the result of the expansion.

See the section "Expanding Lisp Expressions in Zmacs" in *Text Editing and Processing*. That section describes two editor commands that allow you to expand macros — c-sh-M and m-sh-M.

35. Displacing Macros

Every time 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 is put where the macro form used to be, and the next time that form is seen, it is already expanded. A macro that does this is called a *displacing macro*, since it displaces the macro form with its expansion.

The major problem with this is that the Lisp form gets changed by its evaluation. If you were to write a program that used such a macro, call **grindef** to look at it, then run the program and call **grindef** again, you would see the expanded macro the second time. Presumably the reason the macro is there at all is that it makes the program look nicer; we would like to prevent the unnecessary expansions, but still let **grindef** display the program in its more attractive form. This is done with the function **displace**.

Another thing to worry about with displacing macros is that if you change the definition of a displacing macro, then your new definition does not take effect in any form that has already been displaced. If you redefine a displacing macro, an existing form using the macro uses the new definition only if the form has never been evaluated.

displace *form expansion* *Function*

Replaces the car and cdr of *form* so that it looks like:

```
(si:displaced original-form expansion)
```

form must be a list. *original-form* is equal to *form* but has a different top-level cons so that the replacing mentioned above does not affect it. **si:displaced** is a macro, which returns the caddr of its own macro form. So when the **si:displaced** form is given to the evaluator, it "expands" to *expansion*. **displace** returns *expansion*.

The grinder knows specially about **si:displaced** forms, and grinds 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 (See the section "Introduction to Macros", page 339.) 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 convenient 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*

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.

36. &-Keywords Accepted by defmacro

The pattern in a **defmacro** is like the lambda-list of a normal function. **defmacro** is allowed to contain certain **&**-keywords.

defmacro destructures all levels of patterns in a consistent way. The inside patterns can also contain **&**-keywords and there is checking of the matching of lengths of the pattern and the subform. See the function **destructuring-bind**, page 132. This behavior exists for all of **defmacro**'s parameters, except for **&environment**, **&whole**, and **&aux**.

You must use **&optional** in the parameter list if you want to call the macro with less than its full complement of subforms. There must be an exact one-to-one correspondence between the pattern and the data unless you use **&optional** in the parameter destructuring pattern.

```
(defmacro with-output-to-string
          ((var &optional string index) &body body)
  `(let ((with-output-to-string-internal-string
           ,(or string '(make-array 100 :type 'art-string)))
         ...)
     ...
     ,@body))
```

defmacro accepts these keywords:

&optional **&optional** is followed by *variable*, (*variable*), (*variable default*), or (*variable default present-p*), exactly the same as in a function. Note that *default* is still a form to be evaluated, even though *variable* is not being bound to the value of a form. *variable* does not have to be a symbol; it can be a pattern. In this case the first form is disallowed because it is syntactically ambiguous. The pattern must be enclosed in a singleton list.

&rest Using **&rest** is the same as using a dotted list as the pattern, except that it might be easier to read and leaves a place to put **&aux**.

&key Separates the positional parameters and rest parameter from the keyword parameters. See the section "Evaluating a Function Form", page 151.

&allow-other-keys
In a lambda-list that accepts keyword arguments, **&allow-other-keys** says that keywords that are not specifically listed after **&key** are allowed. They and the corresponding values are ignored, as far as keyword arguments are concerned, but they do become part of the rest argument, if there is one.

&aux **&aux** is the same in a macro as in a function, and has nothing to do with pattern matching. It separates the destructuring pattern of a macro from the auxiliary variables. Following **&aux** you can put entries of the form:

(*variable initial-value-form*)

or just *variable* if you want it initialized to **nil** or do not care what the initial value is.

&body **&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.

&whole **&whole** is followed by *variable*, which is bound to the entire macro-call form or subform. *variable* is the value that the macro-expander function receives as its first argument. **&whole** is allowed only in the top-level pattern, not in inside patterns.

&environment **&environment** is followed by *variable*, which is bound to an object representing the lexical environment where the macro call is to be interpreted. This environment might not be the complete lexical environment. It should be used only with the **macroexpand** function for any local macro definitions that the **macrolet** construct might have established within that lexical environment. **&environment** is allowed only in the top-level pattern, not in inside patterns. See the section "Lexical Environment Objects and Arguments", page 138.

&list-of is not supported as a result of making **defmacro** Common-Lisp compatible. Use **loop** or **mapcar** instead of **&list-of**.

37. Functions to Expand Macros

The following 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 to examine and understand code that might contain macros.

macroexpand-1 *form* &optional *env dont-expand-special-forms* *Function*

If *form* is a macro form, **macroexpand-1** expands it (once) and returns the expanded form and **t**. Otherwise it returns *form* and **nil**. **macroexpand-1** expands **defsubst** function forms as well as macro forms. *env* is a lexical environment that can be supplied to specify the lexical environment of the expansions. See the section "Lexical Environment Objects and Arguments", page 138. *dont-expand-special-forms* prevents macro expansion of forms that are both special forms and macros. See the variable **si:*macroexpand-hook***, page 375.

macroexpand *form* &optional *env dont-expand-special-forms* *Function*

If *form* is a macro form, **macroexpand** expands it repeatedly until it is not a macro form and returns two values: the final expansion and **t**. Otherwise, it returns *form* and **nil**. **macroexpand** expands **defsubst** function forms as well as macro forms. *env* is a lexical environment that can be supplied to specify the lexical environment of the expansions. See the section "Lexical Environment Objects and Arguments", page 138. *dont-expand-special-forms* prevents macro expansion of forms that are both special forms and macros.

si:*macroexpand-hook* *Variable*

The value of this variable is used as the expansion interface hook by **macroexpand-1**. When **macroexpand-1** determines that a symbol names a macro, it obtains the expansion function for that macro. The value of **si:*macroexpand-hook*** is called as a function of three arguments: the expansion function, *form*, and *env*. The value returned from this call is the expansion of the macro call.

The initial value of **si:*macroexpand-hook*** is **funcall**, and the net effect is to invoke the expansion function, giving it *form* and *env* as its two arguments.

PART IX.

Structure Macros

38. Introduction to Structure Macros

You can extend Lisp's data structures with the **defstruct** macro. **defstruct** provides a facility in Lisp for creating and using aggregate data types with named elements. These are similar to PL/I structures, or records in Pascal.

For information about using macros to extend the control structures of Lisp:

See the section "Macros", page 337.

See the section "The **loop** Iteration Macro", page 205.

To understand the basic idea, assume you are writing a Lisp program that deals 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?

The representation could be either a list of the elements (x-position, y-position, and so on) or an array of five elements, the zero element being the x-position, the first being the y-position, and so on. The problem with both these representations is that the "elements" (such as x-position) occupy places in the object that are arbitrary and hard to remember; for example, was the mass the third or the fourth element of the array? 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**.

It would be more useful to have names that are easy to remember and understand. If the symbol **foo** were bound to a representation of a space ship, then the following could return its x-position:

```
(ship-x-position foo)
```

The following could return its y-position:

```
(ship-y-position foo)
```

And so forth. The **defstruct** facility does just this.

defstruct itself is a macro that defines a structure. For the space ship example, the structure could be defined by using:

```
(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. The evaluation of this form does several things. First, it defines **ship-x-position** to be a function

that, given a ship, returns the x component of its position. This is called an *accessor function*, because it accesses a component of a structure. **defstruct** defines the other four accessor functions in a similar way.

defstruct also defines **make-ship** to be a macro that expands into the necessary Lisp code to create a **ship** object. **setq** is used to set **x** to the new ship:

```
(setq x (make-ship))
```

The macro **make-ship** is called a *constructor macro*, because it constructs a new structure.

It is also possible to change the contents of a structure. To do this, the **setf** macro is used as follows:

```
(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 is 100.

How does all this map into the familiar primitives of Lisp? In this simple example, the choice of implementation technique is left up to **defstruct**; it chooses to represent a ship as an array. The array has five elements, which are the five components of the ship. The accessor functions are defined thus:

```
(defun ship-x-position (ship)
  (aref ship 0))
```

The constructor macro **(make-ship)** expands into **(make-array 5)**, which makes an array of the appropriate size to be a ship. Note that a program that uses ships need not contain any explicit knowledge that ships are represented as five-element arrays; this is kept hidden by **defstruct**.

The accessor functions are not ordinary functions; they are **substs**. 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, although this is not normally done in Symbolics-Lisp.

You can use the **describe-defstruct** function to examine the contents of the **ship**:

```
(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>
```

defstruct has many other features not demonstrated by the space ship example. First of all, you can specify the kind of Lisp object to use for the "implementation" of the structure. The space ship example 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 also allows you to specify to the constructor macro what the various elements of the structure should be initialized to. And it lets you give, in the **defstruct** form, default values for the initialization of each element.

The **defstruct** in Symbolics-Lisp also works in various dialects of Maclisp and thus has some features that are not useful in Symbolics-Lisp. When possible, the Maclisp-specific features attempt to do something reasonable or harmless in Symbolics-Lisp so that it is easier to write code that runs equally well in Symbolics-Lisp and Maclisp.

39. Using defstruct

defstruct *Macro*

Defines a record-structure data type. 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 other Lisp programs and that want to display the contents of structures in a helpful way. *name* is used for other things; for more information: See the section "Named Structures", page 403.

Each *option* can be either a symbol, which should be one of the recognized option names, or a list, whose car should be one of the option names and the rest of which should be "arguments" to the option. Some options have arguments that default; others require that arguments be given explicitly. For more information about options: See the section "Options to **defstruct**", page 385.

Each *slot-description* can 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 is defined with the name *slot-name*. The **:conc-name** option allows you to specify a prefix and have it concatenated onto the front of all the slot names to make the names of the accessor functions. Form (2) is similar, but allows a default initialization for the slot. Form (3) lets you pack several slots into a single element of the physical underlying structure, using the byte field feature of **defstruct**.

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 later definition is the one that takes effect, destroying the earlier definition. (This is the same as the requirement that each **defun** that is intended to define a distinct function must have a distinct name.)

You should always prefix the names of all accessor functions with some text unique to the structure. See the section "Introduction to Structure Macros", page 379. In the space ship example there, all the names start with **ship-**. The **:conc-name** option can be used to provide such prefixes automatically. Similarly, the conventional name for the constructor macro in the space ship example is **make-ship**, and the conventional name for the alterant macro is **alter-ship**.

The **describe-defstruct** function lets you examine an instance of a structure.

describe-defstruct *instance* &optional *name* *Function*

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 this name so that **describe-defstruct** can know of what structure *instance* is an instance, and thus figure out the names of *instance*'s slots.

If *instance* is a named structure, you do not have to provide *name*, since it is just the named structure symbol of *instance*. Normally the **describe** function calls **describe-defstruct** if it is asked to describe a named structure; however, some named structures have their own idea of how to describe themselves. See the section "Named Structures", page 403.

40. 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 the kind of Lisp object to be used to implement the structure. The option 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 by using **defstruct-define-type**.

:array Use an array, storing components in the body of the array.

:named-array Like **:array**, but make the array a named structure using the name of the structure as the named structure symbol. See the section "Named Structures", page 403. Element 0 of the array holds the named structure symbol and so is not used to hold a component of the structure.

:array-leader Use an array, storing components in the leader of the array.

:named-array-leader Like **:array-leader**, but make the array a named structure using the name of the structure as the named structure symbol. See the section "Named Structures", page 403. Element 1 of the leader holds the named structure symbol and so is not used to hold a component of the structure.

:list Use a list.

:named-list Like **:list**, but the first element of the list holds the symbol that is the name of the structure and so is not used as a component.

: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 can only have one slot. This is only useful with the byte field feature; it lets you store a bunch of small numbers within fields of a fixnum, giving the fields names. See the section "Using Byte Fields and **defstruct**", page 399.

:grouped-array See the section "Grouped Arrays", page 401. This option is described there.

:constructor This option takes one argument, which specifies the name of the constructor macro. If the argument is not provided or if the option itself is not provided, the name of the constructor is made by concatenating the string "**make-**" to the name of the structure. If the argument is provided and is **nil**, no constructor is defined. For more information about the use of the constructor macro: See the section "Constructor Macros", page 395.

:export The **:export** option exports the specified symbols from the package in which the structure is defined. This option accepts the following as arguments: the names of slots and the following options: **:alterant**, **:constructor**, **:copier**, **:predicate**, **:size-macro**, and **:size-symbol**.

The following example shows the use of **:export**.

```
(defstruct (2d-moving-object
             (:type :array)
             :conc-name
             ;; export all accessors and make-2d-moving-object
             (:export :accessors :constructor))
  mass
  x-pos
  y-pos
  x-velocity
  y-velocity)
```

See the section "Importing and Exporting Symbols", page 573.

:alterant This option takes one argument, which specifies the name of the alterant macro. If neither the argument nor the option itself is 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. For more information about the use of the alterant macro: See the section "Alterant Macros", page 397.

: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 does the usual thing, but if you invoke it without its argument, it behaves as if you had invoked it on the result of evaluating the form that is the argument to the **:default-pointer** argument. Here is an example:

```
(defstruct (room (:default-pointer *default-room*))
   room-name
   room-contents)

(room-name x) ==> (aref x 0)
(room-name)   ==> (aref *default-room* 0)
```

If the argument to the **:default-pointer** argument is not given, it defaults to the name of the structure.

:conc-name The **:conc-name** option allows you to specify a prefix and have it concatenated onto the front of all the slot names to make the names of the accessor functions. 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 argument should be a symbol; its print name is used as the prefix. If **:conc-name** is specified without an argument, the name of the structure followed by a hyphen is used as the prefix. If you do not specify the **:conc-name** option, the names of the accessors are the same as the slot names, and you should then 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**, because it causes the slot and accessor names to be different.

The following example turns the slot name **knob-color** into the an accessor with the name **door-knob-color**. It uses **door**, the name of the structure, as the default for **:conc-name**.

```
(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**, then the two defined in **astronaut** itself. The accessor functions defined by the **person** structure can be applied to instances of the **astronaut** structure. The following illustrates 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 applies only 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, apply only 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, the slot has no initial value in the new structure. Otherwise its initial value form is replaced by the *init-form*. The old (included) structure is unmodified.

For example, to define **astronaut** so that the default age for an astronaut is **45.**, then the following can be used:

```
(defstruct (astronaut (:include person (age 45.)))
  helmet-size
  (favorite-beverage 'tang))
```

:named This means that you want to use one of the "named" types. If you specify a type of **:array**, **:array-leader**, or **:list**, and give the **:named** option, then the **:named-array**, **:named-array-leader**, or **:named-list** type is used instead. Asking for type **:array** and giving the **:named** option as well is the same as asking for the type **:named-array**; the only difference is stylistic.

:make-array If the structure being defined is implemented as an array, this option can 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 might want to specify the type of the array. Of course, this option is only meaningful if the structure is, in fact, being implemented by an array.

The argument to the **:make-array** option should be a list of alternating keyword symbols to the **make-array** function, and forms whose values are the arguments to those keywords. 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.

When necessary, **defstruct** overrides any of the **:make-array** options. For example, if your structure is of type **:array**, then **defstruct** supplies the size of that array regardless of what you say in the **:make-array** option.

Constructor macros for structures implemented as arrays all allow the keyword **:make-array** to be supplied. Attributes supplied therein override any **:make-array** option attributes supplied in the original **defstruct** form. If some attribute appears in neither the invocation of the constructor nor in the **:make-array** option to **defstruct**, then the constructor chooses appropriate defaults.

The **:make-array** option lets you control the initialization of arrays created by **defstruct** as instances of structures. **make-array** initializes the array before the constructor code does. Therefore, any initial value supplied via the new **:initial-value** keyword for **make-array** is overwritten in any slots where you gave **defstruct** an explicit initialization.

If a structure is of type **:array-leader**, you probably want to specify the dimensions of the array. The dimensions of an array are given to **:make-array** as a position argument rather than a keyword argument, so there is no way to specify them in the above syntax. To solve this problem, you can use the keyword **:dimensions** or the keyword **:length** (they mean the same thing) with a value that is anything acceptable as **make-array**'s first argument.

:times This option is used for structures of type **:grouped-array** to control the number of repetitions of the structure that are allocated by the constructor macro. The constructor macro also allows **:times** to be used as a keyword that overrides the value given in the original **defstruct** form. If **:times** appears in neither the invocation of the constructor nor in the **:make-array** option to **defstruct**, then the constructor allocates only one instance of the structure.

:size-symbol The **:size-symbol** option allows you to specify a global variable whose value is the "size" of the structure; this variable is declared with **defconst**. The exact meaning of the size varies, but in general this number is the one you would need to know if you were going to allocate one of these structures yourself. The symbol has this value both at compile time and at run time. If this option is present without an argument, then the name of the structure is concatenated with **"-size"** to produce the symbol.

:size-macro This is similar to the **:size-symbol** option. A macro of no arguments is defined that expands into the size of the structure. The name of this macro defaults as with **:size-symbol**.

:initial-offset This allows you to tell **defstruct** to skip over a certain number of slots before it starts allocating the slots described in the body. This option requires an argument (which must be a fixnum) that is the number of slots you want **defstruct** to skip. To use this option, you must understand how **defstruct** is implementing your structure; otherwise, you will be unable to make use of the slots that **defstruct** has left unused.

:but-first This option is best explained by example:

```
(defstruct (head (:type :list)
                 (:default-pointer person)
                 (:but-first person-head))
  nose
  mouth
  eyes)
```

The accessors expand like this:

```
(nose x)        ==> (car (person-head x))
(nose)          ==> (car (person-head person))
```

The idea is that **:but-first**'s argument is an accessor from some other structure, and it is expected that this structure will never be found outside that slot of that other structure. Actually, you can use any one-argument function, or a macro that acts like a one-argument function. It is an error for **:but-first** to be used without an argument.

:callable-accessors

This option controls whether accessors are really functions, and therefore "callable", or whether they are really macros. With an argument of **t**, or with no argument, or if the option is not provided, then the accessors are really functions. Specifically, they are **subst**s, so that they have all the efficiency of macros in compiled programs, while still being function objects that can be manipulated (passed to **mapcar**, and so on). If the argument is **nil** then the accessors will really be macros.

:eval-when

Normally the functions and macros defined by **defstruct** are defined at eval time, compile time, and load time. This option allows you to control this behavior. The argument to the **:eval-when** option is just like the list that is the first subform of an **eval-when** special form. For example: **(:eval-when (:eval :compile))** causes the functions and macros to be defined only when the code is running interpreted or inside the compiler.

:property

For each structure defined by **defstruct**, a property list is maintained for the recording of arbitrary properties about that structure. (That is, there is one property list per structure definition, not one for each instantiation of the structure.)

The **:property** option can be used to give a **defstruct** an arbitrary property. **(:property** *property-name value***)** gives the **defstruct** a *property-name* property of *value*. Neither argument is evaluated. To access the property list, the user should look inside the **defstruct-description** structure. See the section "**defstruct** Internal Structures", page 407.

:print

The **:print** option gives you implementation-independent control over the printed representation of a structure.

```
(defstruct (foo :named
                (:print "#<Foo S S>" (foo-a foo) (foo-b foo)))
  foo-a
  foo-b)
```

The **:print** option takes a format string and its arguments. The arguments are evaluated in an environment in which the name symbol for the structure is bound to the structure instance being printed.

The **:print** option makes obsolete the use of a **named-structure-invoke** handler to define **:print** handlers.

:predicate The **:predicate** option causes **defstruct** to generate a predicate that recognizes instances of the structure. The first example defines a single-argument function, **foo-p**, that returns **t** only for instances of structure **foo**. The second example defines a function called **is-it-a-foo?**.

```
(defstruct (foo :named :predicate)
  foo-a
  foo-b)
(defstruct (foo :named (:predicate is-it-a-foo?))
  foo-a
  foo-b)
```

The **:predicate** option has one optional argument, the name for the function being generated. The default name for the generated function is formed by appending **-p** to the structure name.

The **:predicate** option works only for named types.

:copier The **:copier** option causes **defstruct** to generate a function for copying instances of the structure.

```
(defstruct (foo (:type list) :copier)
  foo-a
  foo-b)
```

This example would generate a function named **copy-foo**, with a definition approximately like this:

```
(defun copy-foo (x)
  (list (car x) (cadr x)))
```

type In addition to the documented options to **defstruct**, any currently defined type (any valid argument to the **:type** option) can be used as an option. This is mostly for compatibility with the old version of **defstruct**. It allows you to say just *type* instead of **(:type** *type***)**. It is an error to give an argument to one of these options.

other

Finally, if an option is not found among the other options to **defstruct**, **defstruct** checks the property list of the name of the option to see if it has a non-**nil** **:defstruct-option** property. If it does have such a property, then if the option was of the form (*option-name value*), it is treated just like (**:property** *option-name value*). That is, the **defstruct** is given an *option-name* property of *value*. It is an error to use such an option without a value.

This provides a primitive way for you to define your own options to **defstruct**, particularly in connection with user-defined types. See the section "Extensions to **defstruct**", page 409. Several options to **defstruct** are implemented using this mechanism.

41. Using the Constructor and Alterant Macros

This section describes how to create instances of structures and altering the values of its slots. After you have defined a new structure with **defstruct**, you can create instances of this structure using the constructor macro, and you can alter the values of its slots using the alterant macro. By default, **defstruct** defines both the constructor and the alterant, forming their names by concatenating "**make-**" and "**alter-**", respectively, onto the name of the structure. You can specify the names yourself by passing the name you want to use as the argument to the **:constructor** or **:alterant** options, or specify that you do not want the macro created at all by passing **nil** as the argument.

41.1 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* can 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), that element of the created structure is initialized to the value of *form*. If no *symbol* is present for a given slot, then the slot is initialized to the result of evaluating the default initialization form specified in the call to **defstruct**. (In other words, the initialization form specified to the constructor overrides the initialization form specified to **defstruct**.) If the **defstruct** itself also did not specify any initialization, the element's initial value is undefined. You should always specify the initialization, either in the **defstruct** or in the constructor macro, if you care about the initial value of the slot.

Notes: The order of evaluation of the initialization forms is *not* necessarily the same as the order in which they appear in the constructor call, nor the order in which they appear in the **defstruct**; you should make sure your code does not depend on the order of evaluation. The forms are reevaluated on every constructor-macro call, so that if, for example, the form **(gensym)** were used as an initialization form, either in a call to a constructor macro or as a default initialization in the **defstruct**, then every call to the constructor macro would create a new symbol.

Two symbols are specially recognized by the constructor: **:make-array**, which should only be used for **:array** and **:array-leader** type structures (or the named

versions of those types), and **:times**, which should only be used for **:grouped-array** type structures. If one of these symbols appears instead of a slot name, then it is interpreted just as the **:make-array** option or the **:times** option, and it overrides what was requested in that option. For example:

```
(make-ship ship-x-position 10.0
           ship-y-position 12.0
           :make-array (:leader-length 5 :area disaster-area))
```

41.2 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, you can use the keywords **&optional**, **&rest**, and **&aux** in the argument list. They work as you might expect, but note the following:

```
(:constructor make-foo
        (a &optional b (c 'sea) &rest d &aux e (f 'eff)))
```

This defines **make-foo** to be a constructor of one or more arguments. The first argument is used to initialize the **a** slot. The second argument is used to initialize the **b** slot. If there is no second argument, then the default value given in the body of the **defstruct** (if given) is used instead. The third argument is used to initialize the **c** slot. If there is no third argument, then the symbol **sea** is used instead. Any arguments following the third argument are collected into a list and used to initialize the **d** slot. If there are three or fewer arguments, then **nil** is placed in the **d** slot. The **e** slot *is not initialized*; its initial value is undefined. Finally, the **f** slot is initialized to contain the symbol **eff**.

The actions taken in the **b** and **e** cases were carefully chosen to allow you to specify all possible behaviors. Note that the **&aux** "variables" can be used to completely override the default initializations given in the body.

Note that you are allowed to give the **:constructor** option more than once, so that you can define several different constructors, each with a different syntax.

The following restrictions should also be noted:

- Even these "function-style" constructors do not guarantee that their arguments will be evaluated in the order that you wrote them.
- You cannot specify the **:make-array** or **:times** information in this form of constructor macro.

41.3 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 code than using consecutive **setf**s when you are altering two byte fields of the same object, or when you are using the **:but-first** option.

You can use alterant macros on structures whose accessors require additional arguments. Put the additional arguments before the list of slots and values, in the same order as required by the accessors.

42. Using Byte Fields and defstruct

The byte field feature of **defstruct** allows you to specify that several slots of your structure are bytes in an integer stored in one element of the structure. For example, consider the following structure:

```
(defstruct (phone-book-entry (:type :list))
  name
  address
  (area-code 617.)
  exchange
  line-number)
```

Although this works correctly, 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. Because Symbolics 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 octal numbers **#o1212** and **#o0012** are byte specifiers to be used with the functions **ldb** and **dpb**. The accessors, constructor, and alterant macros 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 read and write the second element of the list only 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.

If the byte specifier is **nil**, the accessor is defined to be the usual kind that accesses the entire Lisp object, thus returning all the byte field components as a fixnum. These slots can have default initialization forms.

The byte specifier need not be a constant; you can use a variable (or any Lisp form). It is evaluated each time the slot is accessed. Of course, you do not ordinarily want the byte specifier to change between accesses.

Constructor macros initialize words divided into byte fields as if they were deposited in the following order:

1. Initializations for the entire word given in the **defstruct** form.
2. Initializations for the byte fields given in the **defstruct** form.
3. Initializations for the entire word given in the constructor macro form.
4. Initializations for the byte fields given in the constructor macro form.

Alterant macros work similarly: the modification for the entire Lisp object is done first, followed by modifications to specific byte fields. If any byte fields being initialized or altered overlap each other, the actions of the constructor and alterant macros are unpredictable.

43. 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. 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 also 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.

44. Named Structures

44.1 Introduction to Named Structures

The *named structure* feature provides a very simple form of user-defined data type. Any array can be made a named structure, although usually the **:named** option of **defstruct** is used to create named structures. The principal advantages of a named structure are that it has a more informative printed representation than a normal array and that the **describe** function knows how to give a detailed description of it. (You do not have to use **describe-defstruct**, because **describe** can figure out the names of the structure's slots by looking at the named structure's name.) It is recommended, therefore, that "system" data structures be implemented with named structures.

Another kind of user-defined data type, more advanced but less efficient when used only as a record structure, is provided by the *flavor* feature. See the section "Flavors", page 415.

A named structure has an associated symbol called its "named structure symbol"; it represents the user-defined type of which it is an instance. The **typep** function applied to the named structure returns this symbol. If the array has a leader, 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 an instance of a named structure and the second its named structure symbol, **typep** returns **t**. **t** is also returned if the second argument is the named structure symbol of a **:named defstruct** included (using the **:include** option), directly or indirectly, by the **defstruct** for this structure. For example, if the structure **astronaut** includes the structure **person**, and **person** is a named structure, then giving **typep** an instance of an **astronaut** as the first argument, and the symbol **person** as the second argument, returns **t**. This reflects the fact that an astronaut is, in fact, a person, as well as an astronaut.

44.2 Handler Functions for Named Structures

You can associate a function that handles various operations that can be done on the named structure with a named structure. You can control both how the named structure is printed and what **describe** will do with it.

To provide such a handler function, make the function the **named-structure-invoke** property of the named structure symbol. The functions that know about named structures 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. What the function is expected to do depends on the keyword it is passed as its first argument. The following keywords are defined:

:which-operations
Returns a list of the names of the operations handled by the function.

:print-self
The arguments are **:print-self**, the named structure, the stream to which to output, the current depth in list-structure, and **t** if slashification is enabled (**prinl** 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 defaults to a reasonable printed representation. For example:

#<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 checks whether the named structure was created by using the **:named** option of **defstruct**; if so, the names and values of the structure's fields are enumerated.

Here is an example of a simple named-structure handler function.

For this example to have any effect, the person **defstruct** used as an example there must be modified to include the **:named** attribute.

```
(defselect ((:property person named-structure-invoke)
            (:print-self (person stream ignore slashify-p)
             (format stream
                     (if slashify-p "#<person ~a>" "~a")
                     (person-name person))))
```

This example 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 there **:includes** the person structure, this named-structure handler is not invoked when an astronaut is printed, and an astronaut does not include his name in his printed representation. This is because named structures are not as general as flavors.

In this example, the **:which-operations** handler is automatically generated, as well as the handlers for **:operation-handled-p** and **:send-if-handles**.

Another way to write this handler is as follows:

```
(defselect ((:property person named-structure-invoke))
  (:print-self (person stream ignore slashify-p)
   (if slashify-p
       (si:printing-random-object (person stream :typep)
         (princ (person-name person) stream))
       (princ (person-name person) stream))))
```

This example uses the **si:printing-random-object** special form, which is a more advanced way of printing #< ... >. It interacts with the **si:print-readably** variable and special form.

44.3 Functions That Operate on Named Structures

named-structure-p *x* *Function*

This semi-predicate returns **nil** if *x* is not a named structure; otherwise it returns *x*'s named structure symbol.

named-structure-symbol *x* *Function*

x should be a named structure. This returns *x*'s named structure symbol: if *x* has an array leader, element 1 of the leader is returned, otherwise element 0 of the array is returned.

make-array-into-named-structure *array* *Function*

array is made to be a named structure, and is returned.

named-structure-invoke *operation structure* &rest *args* *Function*

operation should be a keyword symbol, and *structure* should be a named structure. The handler function of the named structure symbol, found as the value of the **named-structure-invoke** property of the symbol, is called with appropriate arguments.

See also the **:named-structure-symbol** keyword to **make-array**.

45. defstruct Internal Structures

If you want to write a program that examines structures and displays them the way **describe** and the Inspector do, your program will work by examining the internal structures used by **defstruct**. In addition to discussing these internal structures, this section also provides the information necessary to define your own structure types.

Whenever you use **defstruct** to define a new structure, **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 following example shows a simplified version of how **si:defstruct-description** structure is actually defined. **si:defstruct-description** is defined in the **system-internals** (or **si:**) package and includes additional slots that are not shown in this example:

```
;;;simplied version of defstruct-description structure
(defstruct (defstruct-description
             (:default-pointer description)
             (:conc-name defstruct-description-))
           name
           size
           property-alist
           slot-alist)
```

The **name** slot contains the symbol supplied by the user to be the name of the structure, such as **spaceship** or **phone-book-entry**.

The **size** slot contains the total number of locations in an instance of this kind of structure. This is *not* the same number as that obtained from the **:size-symbol** or **:size-macro** options to **defstruct**. A named structure, for example, usually uses up an extra location to store the name of the structure, so the **:size-macro** option gets 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 section "Options to **defstruct**", page 385.

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:

```
;;;simplified version of the actual implementation
(defstruct (defstruct-slot-description
             (:default-pointer slot-description)
             (:conc-name defstruct-slot-description-))
           number
           ppss
           init-code
           ref-macro-name)
```

Note that this is a simplified version of the real definition and does not fully represent the complete implementation. The **number** slot contains the number of the location of this slot in an instance of the structure. Locations are numbered starting with **0**, and continuing up to one less than the size of the structure. The actual location of the slot is determined by the reference-consing function associated with the type of the structure. See the section "Options to **defstruct-define-type**", page 410.

The **ppss** slot contains the byte specifier code for this slot if this slot is a byte field of its location. If this slot is the entire location, then the **ppss** slot contains **nil**.

The **init-code** slot contains the initialization code supplied for this slot by the user in the **defstruct** form. If there is no initialization code for this slot, then the init-code slot contains the symbol **si:%%defstruct-empty%%**.

The **ref-macro-name** slot contains the symbol that is defined as a macro or a subst that expands into a reference to this slot (that is, the name of the accessor function).

46. Extensions to defstruct

This section describes the use of **defstruct-define-type**.

46.1 An Example of defstruct-define-type

This section provides an explanation of how **defstruct-define-type** works by examining a call to the macro. 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 are bound while the constructor-creating form is evaluated. The first, **initialization-list**, is bound to a list of the initialization forms for the slots of the structure. The second, **description**, is bound to the **defstruct-description** structure for the structure. See the section "**defstruct** Internal Structures", page 407. For a description of the third variable, **keyword-options**, and the **:list** keyword: See the section "Options to **defstruct-define-type**", page 410.

The keyword **:ref** is followed by a list of three variables that are bound while the accessor-creating form is evaluated. The first, **slot-number**, is bound to the number of the slot that the new accessor should reference. The second, **description**, is bound to the **defstruct-description** structure for the structure. The third, **argument**, is bound to the form that was provided as the argument to the accessor.

defstruct-define-type *type* &body *options* *Macro*

Teaches **defstruct** about new types that it can use to implement structures.

The body of this function is shown in the following example:

```
(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.

46.2 Options to defstruct-define-type

:cons

The **:cons** option to **defstruct-define-type** is how you supply **defstruct** with the code necessary to cons up a form that constructs 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 constructs, initializes, and returns an instance of a structure of this type.

The symbol *inits* is 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:

- **:list** — *inits* is bound to a list of initializations, in the correct order, with **nil**s in uninitialized slots.
- **:alist** — *inits* is bound to an alist with pairs of the form (*slot-number . init-code*).

The symbol *description* is bound to the instance of the **defstruct-description** structure that **defstruct** maintains for this particular structure. See the section "**defstruct** Internal Structures", page 407. 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* is bound to an alist with pairs of the form (*keyword . value*), where each *keyword* was a keyword supplied to the constructor macro that was not the name of a slot, and *value* was the Lisp object that followed the keyword. This is how you can make your own special keywords, such as the existing **:make-array** and **:times** keywords. See the section "Constructor Macros", page 395. You specify the list of acceptable keywords with the **:keywords** option.

It is an error not to supply the **:cons** option to **defstruct-define-type**.

:ref

The **:ref** option to **defstruct-define-type** is how you supply

defstruct with the necessary code that it needs to cons up a form that will reference an instance of a structure of this type.

The **:ref** option has the syntax:

```
(:ref (number description arg-1 ... arg-n)
      body)
```

body is some code that should construct and return a piece of code that will reference an instance of a structure of this type.

The symbol *number* is bound to the location of the slot that is to be referenced. This is the same number that is found in the number slot of the **defstruct-slot-description** structure. See the section "**defstruct** Internal Structures", page 407.

The symbol *description* is 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 is defaulted by the **:default-pointer** option. See the section "Options to **defstruct**", page 385. **defstruct** checks that the user has supplied exactly *n* arguments to the accessor function before calling the reference consing code.

It is an error not to supply the **:ref** option to **defstruct-define-type**.

:overhead The **:overhead** option to **defstruct-define-type** is how you declare to **defstruct** that the implementation of this particular type of structure "uses up" some number of locations in the object actually constructed. This option is used by various "named" types of structures that store the name of the structure in one location.

The syntax of **:overhead** is: **(:overhead** *n***)** where *n* is a fixnum that says how many locations of overhead this type needs.

This number is used only by the **:size-macro** and **:size-symbol** options to **defstruct**. See the section "Options to **defstruct**", page 385.

:named The **:named** option to **defstruct-define-type** controls the use of the **:named** option to **defstruct**. With no argument, the **:named** option means that this type is an acceptable "named structure". With an argument, as in **(:named** *type-name***)**, the symbol *type-name* should be the name of some other structure type that **defstruct** should use if someone asks for the named version of this type. (For example, in the definition of the **:list**

type the **:named** option is used like this:
(:named :named-list).)

:keywords The **:keywords** option to **defstruct-define-type** allows you to define additional constructor keywords for this type of structure. (The **:make**-array constructor keyword for structures of type **:array** is an example.) The syntax is:
(:keywords *keyword-1 ... keyword-n***)**, where each *keyword* is a symbol that the constructor conser expects to find in the *keywords* alist.

:defstruct The **:defstruct** option to **defstruct-define-type** allows you to run some code and return some forms as part of the expansion of the **defstruct** macro.

The **:defstruct** option has the syntax:

```
(:defstruct (description)
            body)
```

body is a piece of code that runs whenever **defstruct** is expanding a **defstruct** form that defines a structure of this type. The symbol *description* is bound to the instance of the **defstruct-description** structure that **defstruct** maintains for this particular structure.

The value returned by the *body* should be a *list* of forms to be included with those that the **defstruct** expands into. Thus, if you only want to run some code at **defstruct**-expand time, and you do not want to actually output any additional code, then you should be careful to return **nil** from the code in this option.

:predicate The **:predicate** option specifies how to construct a **:predicate** option for **defstruct.**

```
(:predicate (description name)
            `(defun ,name (x)
               (and (frobbozp x)
                    (eq (frobbozref x 0)
                        ',(defstruct-description-name)))))
```

The syntax for the option follows.

```
(:predicate (description name)
            body)
```

The variable *description* is bound to the **defstruct-description** structure maintained for the structure for which a predicate is generated. The variable *name* is bound to the symbol that is to be defined as a predicate. *body* is a piece of code that is evaluated to return the defining form for the predicate.

:copier The **:copier** option specifies how to copy a particular type of

structure for situations when it is necessary to provide a copying function other than the one that **defstruct** would generate.

```
(:copier (description name)
  `(fset-carefully ',name 'copy-frobboz))
```

The syntax for the option follows.

```
(:copier (description name)
         body)
```

description is bound to an instance of the **defstruct-description** structure, *name* is bound to the symbol to be defined, and *body* is some code to evaluate to get the defining form.

PART X.

Flavors

47. Introduction to the Flavor System

The Flavor System is the part of Symbolics-Lisp that supports object-oriented programming. The Flavor System is used 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. *Flavors* are the abstract types of objects; *methods* are the generic operators. The objects are *flavor instances* that you manipulate by sending *messages*, which are requests for specific operations.

47.1 Objects and the Flavor System

It is often convenient to model a program in terms of *objects*, which are conceptual entities that can be likened to real-world things. Choosing the 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 objects, the next task of the design is to determine 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, the program is built around a set of objects, each of which has a set of operations that can be performed on it. More specifically, the program defines several *types* of object (the editor example 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.

A simple example of this 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: **get**, **putprop**, and **remprop**.

Another example is the use of **defstruct** to create an object called a **ship**. (For more information about this example: See the section "Structure Macros", page 377.) **defstruct** automatically defined some operations on this object to access its elements. Other functions could be defined that do useful things with the **ship**

objects, such as computing their speed, angle of travel, momentum, or velocity, stopping them, or moving them elsewhere.

In these cases, the conceptual object is represented by one Lisp object. The Lisp object 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, the object keeps track of an *internal state*, which can be *examined* and *altered* by the operations available for that type of object. **get** examines the state of a property list, and **putprop** alters it; **ship-x-position** examines the state of a ship, and **(setf (ship-mass ship) 5.0)** alters it.

This creation and modification is 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.

100.2 Modularity and Object-oriented Programming

An important benefit of the object-oriented style is that it lends itself to a particularly simple and lucid kind of modularity. Programs that use modular programming constructs and techniques make it easier to write programs that are easy to read and understand, and are more reliable and maintainable. Object-oriented programming lets a programmer implement a useful facility that presents the caller (another program) 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. This set of defined external interfaces is known as the contract between the caller and the callee.

For example, a program that uses disembodied property lists does not need to know that the property lists are being maintained as lists of alternating indicators and values; the program simply performs the operations, passing inputs to the property lists and getting back outputs. The program depends only on the external definition of these operations; for example, if it uses **putprop** to give a property list a property of x, it can use **get** on the property list and the indicator to get x returned, as long as no other changes were made to the property list.

Thus, if you read a program that uses disembodied property lists, you need only understand what they do, not how they are implemented. This also means that if the representation of property lists could be changed, your 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. (See the section "Introduction to Structure Macros", page 379.) 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 this example, **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, the velocity is stored in polar coordinates instead of rectangular coordinates. This is purely an implementation decision; the caller has no idea which of the two ways the implementation uses, because the caller only performs the operations on the object by calling the appropriate functions.

In this example, new types of objects, whose implementations are hidden from the programs that use them, have been created. 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.

The quantities being held by the elements of the **ship** structure are referred to as *instance variables*. Each instance of a type has the same operations defined on it; what distinguishes one instance from another (besides identity (**eq**ness)) is the values that reside in its instance variables. The example above illustrates that a caller of operations does not know what the instance variables are; these two ways of writing the **ship** operations have different instance variables, but from the outside they perform exactly the same operations.

You might ask: "But what if the caller evaluates **(aref ship 2)** and notices that the x-velocity is returned rather than the speed? Then you could tell which of the two implementations were used." This is true; however, when a facility is implemented in the object-oriented style, only certain functions are documented and advertised — those that are considered to be operations on the type of object. The contract between **ship** and its callers covers only what happens if the caller calls these functions; it does not guarantee what would happen if the caller were to use **aref**. If **ship** were reimplemented, the code that does the **aref** would have a different effect entirely and would 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, Symbolics-Lisp does not automatically forbid constructs that circumvent the contract. This is intentional. One reason for this is that the Symbolics 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 Symbolics 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, you can create new 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.

The implementation of an abstract type can be hidden by making its operations into functions that the user might call. The important thing is not that they are functions because in Lisp everything is done with functions. The important thing is that a new conceptual operation has been defined and been given a name, rather than requiring the user who wants to do the operation to write it out step-by-step. Thus, **(ship-x-velocity s)** can be used, rather than **(aref s 2)**.

Like ordinary functions, such abstract-operation functions are sometimes simple

enough that it is useful to have the compiler compile special code for them rather than actually calling the function. Compiling special code in this way 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 using this optimization, the implementation of the abstract type is hidden only 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 could 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 need to be recompiled.

47.3 Generic Operations on Objects

Consider the rest of the program that uses the **ship** abstraction. See the section "Introduction to Structure Macros", page 379. It might 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 the amount of iron it contains.

However, all kinds of movable objects have positions, velocities, and masses, and the system will contain some programs that deal with these quantities in a uniform way, regardless of what kind of object the attributes apply to. For example, a piece of the system that calculates every object's orbit in space need not worry about the other, more peripheral attributes of various types of objects; it works the same way for all objects. Unfortunately, a program that tries to calculate the orbit of a ship needs to know the ship's attributes, and must call **ship-x-position**, **ship-y-velocity**, and so on.

But these functions do not work for meteors. We could have a second program to calculate orbits for meteors that would be identical to the first, except that where the first one calls **ship-x-position**, the second would call **meteor-x-position**, and so on. However, this would require multiple copies of almost identical code, all of which would have to be maintained in parallel.

We need an operation that can be performed on objects of several different types, doing the thing appropriate for each type. Such operations are called *generic* operations. The classic example of generic operations is the arithmetic functions in most programming languages, including Symbolics-Lisp. The + (or **plus**) function accepts either fixnums or flonums, and performs either fixnum addition or flonum addition, as appropriate, based on the data types of the objects being manipulated.

In this example, a generic **x-position** operation is needed that can be performed on either ships, meteors, or any other kind of mobile object represented in the system. This way, a single program can be written to calculate orbits. When it wants to know the *x* position of an object, it simply invokes the generic **x-position** operation on the object; the correct operation is performed, and the *x* position is returned.

Performing a generic operation is called *sending a message.* The objects in the program are sent messages and respond with answers. In the ship example, the objects are sent **x-position** messages, to which they respond with their *x* position. This is known as *message passing.*

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, but rather a message name and an object; that is, the caller 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 must therefore be found to handle the message. The function called is based on the *type* of the object and the *name* of the message. The same set of functions is used for all instances of a given type, so the type is the only attribute of the object used to determine which function to call. The rest of the message besides the name is information that is 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 an **x-position** message is sent to an object of type **ship**, then the function that is used 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**.

Using this 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 were of type **ship**, then the **ship** type's **x-position** method was invoked; if it were of type **meteor**, then the **meteor** type's **x-position** method was invoked. The orbit-calculating program merely sends the message, and the right function is invoked based on the type of the object. These are true generic functions in the form of message passing: the same operation can mean different things depending on the type of the object.

47.4 Message Passing in the Flavor System

By convention, objects that receive messages are always *functional* objects (that is, you can apply them to arguments), and a message is sent to an object by calling that object as a function, passing the name of the message as the first argument, and the arguments of the message as the rest of the arguments. Message names are represented by symbols; normally these symbols are in the keyword package, since messages are a protocol for communication between different programs, which might reside in different packages. For example, to determine the x position of variable **my-ship** whose value is an object of type **ship**, a message is sent 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**, the following message is sent:

```
(funcall my-ship :set-x-position 3.0)
```

It should be stressed that no new features are added to Lisp for message sending; instead, a convention has been defined 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 that 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 cannot be **defstructs** any more, because those are not functions. But the structure defined by **defstruct** was doing something useful: it was holding the instance variables (the internal state) of the object. A function with an internal state, a coroutine, is needed.

One possible way to provide this internal state is with a closure. A message-receiving object could be implemented as a closure over a set of instance variables. The function inside the closure would have a large **selectq** form to dispatch on its first argument. While using closures does work, it creates several serious problems. The main one is that in order to add a new operation to a system, you must modify a great deal of code to find all the types that understand that operation, and add a new clause to the **selectq**. But then you cannot textually separate the implementation of your new operation from the rest of the system; the methods must be interleaved with the other operations for the type. Adding a new operation should only require adding Lisp code; it should not require modifying Lisp code.

The conventional way of making generic operations is to have a procedure for each operation, which has a big **selectq** for all the types; this means you have to modify code to add a type. The way described above is to have a procedure for each type, which has a big **selectq** for all the operations; this means you have to modify code to add an operation. Neither of these has the desired property that extending the system should only require adding code, rather than modifying code.

Closures are also somewhat clumsy and crude. A far more streamlined, convenient, and powerful system for creating message-receiving objects exists; it is called the *Flavor* mechanism. Flavors provide a mechanism for doing many common and useful things on Lisp Machines. With Flavors, you can add a new method simply by adding code without modifying anything you can also manage and maintain large amounts of code easily.

48. Using the Flavor System

48.1 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 would be implemented. (See the section "Introduction to Structure Macros", page 379.)

```
(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))
```

This code 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: **x-position**, **y-position**, **x-velocity**, **y-velocity**, and **mass**. The next subform, **()**, can contain the names of flavors which get included in this flavor. (For more information about combining flavors: See the section "Mixing Flavors", page 431.) The last subform is the body of the **defflavor**; it specifies an option about this flavor. In this example, there is only one option, **: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 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, 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** is evaluated in an environment in which the instance variables of **ship** refer to the instance variables of this instance (the one to which the message was sent). So when the arguments of **atan** are evaluated, the values of instance variables of the

object to which the message was sent are used as the arguments. **atan** is invoked, and the result it returns is returned by the instance itself.

In general, this is how to create a new abstract type: a new flavor. Every instance of this flavor has the five instance variables named in the **defflavor** form, and seven methods (five that were automatically generated because of the **:gettable-instance-variables** option, and two that were written in the example). The way to create an instance of our new flavor is with the **make-instance** function. For example:

```
(setq my-ship (make-instance 'ship))
```

This returns an object whose printed representation is:

```
#<SHIP 13731210>
```

The argument to **make-instance** is the name of the flavor to be instantiated. Additional arguments, not used here, are *init options*; these are commands to the flavor that is being made an instance, selecting optional features.

This flavor is useless as it stands, because there is no way to set any of the parameters. This can be fixed 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.

You can add another option to the **defflavor** form, **:initable-instance-variables**, to initialize the values of the instance variables when an instance is first created. **:initable-instance-variables** does not create any methods; instead, it makes *initialization keywords* named **:x-position**, **:y-position**, and so on, that can be used as init option arguments to **make-instance** to initialize the corresponding instance variables. The set of init options is sometimes called the *init-plist* because it resembles 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)
```

By evaluating this new **defflavor**, the existing flavor definition is updated and includes the new methods and initialization options; in addition, the instance generated earlier can now be able to accept these new messages.

The mass of the ship can be set created by evaluating the following:

```
(send my-ship :set-mass 3.0)
```

This sets the **mass** instance variable of **my-ship** to **3.0**.

If you want to experiment with flavors, it is useful to know that **describe** of an instance tells you the flavor of the instance and the values of its instance variables. If **(describe my-ship)** were evaluated 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", the following example shows how to create another ship and initialize some of the instance variables using the init-plist:

```
(setq her-ship (make-instance 'ship :x-position 0.0
                                    :y-position 2.0
                                    :mass 3.5))
                ==> #<SHIP 13756521>
```

The following shows the results of using **describe** on the new ship:

```
(describe her-ship)
#<SHIP 13756521>, an object of flavor SHIP,
 has instance variable values:
        X-POSITION:         0.0
        Y-POSITION:         2.0
        X-VELOCITY:         unbound
        Y-VELOCITY:         unbound
        MASS:               3.5
```

A flavor can also establish default initial values for instance variables. These default values are used when a new instance is created if the values are not initialized any other way. The syntax for specifying a default initial value is to replace the name of the instance variable by a list whose first element is the name and whose second is a form to evaluate to produce the default initial value. For example:

```
(defvar *default-x-velocity* 2.0)
(defvar *default-y-velocity* 3.0)

(defflavor ship ((x-position 0.0)
                 (y-position 0.0)
                 (x-velocity *default-x-velocity*)
                 (y-velocity *default-y-velocity*)
                 mass)
                ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)

(setq another-ship (make-instance 'ship :x-position 3.4))
```

```
(describe another-ship)
#<SHIP 14563643>, an object of flavor SHIP,
 has instance variable values:
        X-POSITION:         3.4
        Y-POSITION:         0.0
        X-VELOCITY:         2.0
        Y-VELOCITY:         3.0
        MASS:               unbound
```

x-position was initialized explicitly, so the default was ignored. **y-position** was initialized from the default value, which was **0.0**. The two velocity instance variables were initialized from their default values, which came from two global variables. **mass** was not explicitly initialized and did not have a default initialization, so it was left unbound.

These are only some of the options that can be used for **defflavor**. Additionally, init options can used to do more than initialize instance variables. But even with the small set of features described so far, it is possible to write object-oriented programs using flavors. (For more information about **defflavor** options: See the section "**defflavor** Options", page 441.)

48.1.1 Functions for Creating Flavors

defflavor *Macro*

A flavor is defined by a form such as:

```
(defflavor flavor-name (var1 var2...) (flav1 flav2...)
      opt1 opt2...)
```

flavor-name is a symbol that serves to name this flavor. It gets 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*, returns the symbol *flavor-name*. **(typep** *obj flavor-name***)** is **t** if *obj* is an instance of a flavor, one of whose components (possibly itself) is *flavor-name*.

var1, *var2*, and so on, are the names of the instance variables containing the local state for this flavor. A list of the name of an instance variable and a default initialization form is also acceptable; the initialization form is evaluated when an instance of the flavor is created if no other initial value for the variable is obtained. If no initialization is specified, the variable remains unbound.

flav1, *flav2*, and so on, are the names of the component flavors out of which this flavor is built. The features of those flavors are inherited as described previously.

opt1, *opt2*, and so on, are options; each option could be either a keyword symbol or a list of a keyword symbol and arguments.

The options to **defflavor** are described elsewhere: See the section "**defflavor** Options", page 441.

all-flavor-names *Variable*

This is a list of the names of all the flavors that have ever been created by **defflavor**.

defmethod *Macro*

A method is a function to handle a particular message sent to an instance of a particular flavor. It is defined by a form such as:

```
(defmethod (flavor-name method-type message) lambda-list
  form1 form2...)
```

flavor-name is a symbol that is the name of the flavor that 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 that names the message to be handled.

The meaning of the *method-type* depends on the kind of method combination is declared for this message. For instance, for daemons **:before** and **:after** are allowed. For a complete description of method types and the way in which methods are combined: See the section "Method Combination", page 455.

lambda-list describes the arguments and "aux variables" of the function; the first argument to the method, which is the message keyword, is handled automatically, and so it is not included in the *lambda-list*. Note that methods cannot have **&quote** arguments; that is, they must be functions, not special forms. *form1*, *form2*, and so on, are the function body; the value of the last form is returned.

The following variant form, where *function* is a symbol, says that *flavor-name*'s method for *message* is *function*, a symbol that names a function:

```
(defmethod (flavor-name message) function)
```

The first three arguments are the object receiving the message, the mapping table (which can safely be ignored), and the message keyword.

If you redefine a method that is already defined, the old definition is replaced by the new one. Given a flavor, a message name, and a method type, there can only be one function, so if you define a **:before** daemon method for the **foo** flavor to handle the **:bar** message, 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 specification for a method looks like:

(:method *flavor-name message*) or
(:method *flavor-name method-type message*)

This is useful to know if you want to trace or advise a method, or if you want to manipulate (for example, disassemble) the method function itself.

make-instance *flavor-name init-option1 value1 init-option2 value2...* *Function*

Creates and returns an instance of the specified flavor. Arguments after the first are alternating init-option keywords and arguments to those keywords. These options are used to initialize instance variables and to select arbitrary options. 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**.

46.1.2 Functions for Passing Messages

In order to improve the clarity of heavily object-oriented programs, **funcall** is not used to send messages. Instead, the **send** function, which has a shorter, more specific name, is used.

send *object message-name* &rest *arguments* *Function*

Sends the message named *message-name* to the *object*. *arguments* are the arguments passed. **send** does exactly the same thing as **funcall**. For stylistic reasons, it is preferable to use **send** instead of **funcall** when sending messages because **send** clarifies the programmer's intent.

lexpr-send *object message-name* &rest *arguments* *Function*

Sends the message named *message-name* to the *object*. *arguments* are the arguments passed, except that the last element of *arguments* should be a list, and all the elements of that list are passed as arguments. Example:

```
(send some-window :set-edges 10 10 40 40)
```

does the same thing as

```
(setq new-edges '(10 10 40 40))
(lexpr-send some-window :set-edges new-edges)
```

lexpr-send is to **send** as **lexpr-funcall** is to **funcall**.

send-if-handles, **lexpr-send-if-handles**, and **operation-handled-p** work by sending **:operation-handled-p** and **:send-if-handles** messages. For example, **(send-if-handles** *object message arguments*) sends *message* to *object* with the argument *arguments*.

If you explicitly need to have the **:operation-handled-p** message sent, you should use **(send** *object* **:operation-handled-p** *message*) rather than **(operation-handled-p** *object message*).

send-if-handles *object message-name* &rest *arguments* *Function*
Sends the message named *message-name* to *object* if the flavor associated with *object* has a method defined for *message-name*. If it does not have a method defined, **nil** is returned. *message-name* is a message name and *arguments* is a list of arguments for that message.

lexpr-send-if-handles *object message-name* &rest *arguments* *Function*
Sends the message named *message-name* to *object* if the flavor associated with *object* has a method defined for *message-name*. *message-name* is a message name and *arguments* is a list of arguments for that message. If *object* does not have a method defined, **nil** is returned.

The difference between **lexpr-send-if-handles** and **send-if-handles** is that for **lexpr-send-if-handles**, the last element of arguments should be a list; all the elements of that list are passed as arguments. **lexpr-send-if-handles** is to **send-if-handles** as **lexpr-send** is to **send**.

operation-handled-p *object message-name* &rest *arguments* *Function*
Returns **t** if the flavor associated with *object* has a method defined for *message-name* and **nil** if a method is not defined for *message-name*.

48.2 Mixing Flavors

This section discusses how to mix flavors to create new flavors. For information about a system for defining message-receiving objects that provide generic operations: See the section "Simple Use of Flavors", page 425.

To create a new type called **meteor** that would accept the same generic operations as **ship**, we could write another **defflavor** and two more **defmethods** identical to those of **ship**; meteors and ships would then both accept the same operations. **ship** would have additional instance variables for holding attributes specific to ships and methods for operations that would not be generic, but that would be defined only for ships; the same would be true of **meteor**.

However, this would be wasteful. Code would be duplicated and maintained in several places, and several instance variables would have to be repeated. The power of flavors (and the name "flavors") comes from the ability to mix several flavors and get a new flavor. Since the characteristics of **ship** and **meteor** partially overlap, the common characteristics can be identified and moved into their own flavor, which might be called **moving-object**. **moving-object** would be defined the same way as **ship**. **ship** and **meteor** could then 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 is a list of flavors to be combined to form the new flavor; such flavors are called *components*. **ship** 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. So with the new definition, **ship** instances still accept the **:x-velocity** and **:speed** messages, and they do the same thing. However, the **:engine-power** message is also understood (and returns the value of the **engine-power** instance variable).

In this example, two more specialized and powerful abstract types have been built on top of another abstract type, **moving-object**. 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; a flavor called **ship-with-passenger** could be defined 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. All the components taken together form a big tree of flavors. A flavor is built from its components, its components' components, and so on. Sometimes the term "components" is used 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, as some flavors might be components through more than one path. It is really a directed graph. Note, though, that the structure cannot be cyclic; circular flavor definitions are not permitted.)

The order in which the components are combined to form a flavor is important. The tree of flavors is turned into an ordered list by performing a *top-down, depth-first* walk of the tree, including nonterminal nodes *before* the subtrees they head, and eliminating duplicates. For example, if **flavor-1**'s immediate components are **flavor-2** and **flavor-3**, and **flavor-2**'s components are **flavor-4** and **flavor-5**, and **flavor-3**'s component was **flavor-4**, then the complete list of components of **flavor-1** would be:

```
flavor-1, flavor-2, flavor-4, flavor-5, flavor-3
```

The flavors earlier in this list are the more specific, less basic ones; in our example, **ship-with-passengers** would be first in the list, followed by **ship**, followed by **moving-object**. A flavor is always the first in the list of its own components. Notice that **flavor-4** does not appear twice in this list. Only the first occurrence of a flavor appears; duplicates are removed. (The elimination of duplicates is done

during the walk; if there is a cycle in the directed graph, it will not cause a nonterminating computation.)

The set of instance variables for the new flavor is the union of all the sets of instance variables in all the component flavors. If both **flavor-2** and **flavor-3** have instance variables named **foo**, then **flavor-1** will have an instance variable named **foo**, and any methods that refer to **foo** will refer to this same instance variable. Thus different components of a flavor can communicate with one another using shared instance variables. Typically, only one component ever sets the variable, and the others only look at it. The default initial value for an instance variable comes from the first component flavor to specify one.

The way the methods of the components are combined is the heart of the flavor system. When a flavor is defined, a single function, called a *combined method*, is constructed for each message supported by the flavor. This function is constructed out of all the methods for that message from all the components of the flavor. Methods can be combined in many different ways; you can select when a flavor is defined and you can create new forms of combination.

There are several kinds of methods, but so far, the only kinds of methods discussed 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. This means that if you start with a flavor **foo** and build a flavor **bar** on top of it, you can then 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; you could use it to make a new flavor **bar** that is just like **foo** except that it reacts completely differently to a few messages. However, often you do not 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 you use combination of methods.

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, and that other flavors only want to know that the message was sent, or only 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 was shown 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 two kinds of *daemon* methods, *before* and *after*. There is a special syntax in **defmethod** for defining such methods. For example, to give the **ship** flavor an after-daemon method for the **:speed** message, you would use the following syntax:

```
(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 **flavor-1** built on **flavor-2** is created, it should not matter what **flavor-2** is built out of. The new before-daemons go before all methods of **flavor-2**, and the 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**.

To clarify this, consider an example using the **:print-self** method. The Lisp printer (that is, the **print** function) prints instances of flavors by sending them **:print-self** messages. The first argument to the **:print-self** message is a stream (the others can be ignored for now), and the receiver of the message is supposed to print its printed representation on the stream. In the **ship** example, instances of the **ship** flavor printed the way they did 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 **ship** is given its own primary method for the **:print-self** message, then that method takes over the job of printing completely; **vanilla-flavor**'s method is not called at all. However, if **ship** is given a before-daemon method for the **:print-self** message, then it is invoked before the **vanilla-flavor** message, and so whatever it prints appears before what **vanilla-flavor** prints. Thus, before-daemons can be used 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.

For information about more advanced ways of combining methods: See the section "Method Combination", page 455.

For more information about the flavor, **vanilla-flavor**: See the section "Vanilla Flavor", page 453.

49. Flavor Functions

instantiate-flavor *flavor-name init-plist* &optional *send-init-message-p return-unhandled-keywords area* *Function*

This is an extended version of **make-instance**, giving you more features. Note that it takes the init-plist as an argument, rather than taking an **&rest** argument of init options and values.

The *init-plist* argument must be a disembodied property list; **locf** of an **&rest** argument is acceptable. But note that this property list can be modified; the properties from the default-init-plist that do not simply initialize instance variables are **putprop**ed on if not already present, and some **:init** methods do explicit **putprop**s onto the init-plist.

If **:init** methods do **remprop** of properties already on the init-plist (rather than simply doing **get** and **putprop**), then the init-plist is **rplacd**ed. This means that the actual list of options is modified. It also means that **locf** of an **&rest** argument does not work; the caller of **instantiate-flavor** must copy its rest argument (for example, with **copylist**); this is because **rplacd** is not allowed on **&rest** arguments.

First, if the flavor's method-table and other internal information have not been computed or are not up to date, they are computed. This could take a substantial amount of time and invoke the compiler, but happens only once for a particular flavor no matter how many instances you make, unless you change something.

Next, the instance variables are initialized in one of several ways. If an instance variable is declared initable, and a keyword with the same spelling 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 cannot depend on any instance variables nor on **self**; it is not evaluated in the "inside" environment in which methods are called. If an instance variable does not get initialized either of these ways it is left unbound; presumably an **:init** method should initialize it.

Note that a simple empty disembodied property list is **(nil)**, which is what you should give if you want an empty init-plist. If you use **nil**, the property list of **nil** will be used, which is probably not what you want.

If any keyword appears in the *init-plist* but is not used to initialize an instance variable and is not declared in an **:init-keywords** option, it is presumed to be a misspelling. Thus, 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 cause an error to be signalled. 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**.

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.

change-instance-flavor *instance new-flavor* &optional (*error-p* **t**) *Function*

Changes the flavor of an instance to another flavor that has compatible instance variables. If you specify a third argument of **nil**, the function returns **t** if it works and **nil** if it does not. If it does not work, you could create an instance of the new flavor and **structure-forward** the old instance to the new.

The caller is responsible for sending any messages to the instance that the caller requires to let it know what has happened. In some cases the caller might want to sent an **:init** message, in other cases some other message.

With the default third argument of **t**, **change-instance-flavor** either returns **t** or calls **ferror** and explains why the instance's flavor cannot be changed.

defun-method *function-spec flavor argument-list body...* *Special Form*

Sometimes you write a function that is not itself a method, but that is to be called by methods and should be able to access the instance variables of the object **self**. **defun-method** is like **defun**, but the function is able to access the instance variables of *flavor*. It is valid to call the function only while executing inside a method or a **defun-method** for an object of the specified flavor, or of some flavor built upon it.

function-spec must be a symbol.

defun-method works by defining two functions, *function-spec* and (**:defun-method** *function-spec*). An optimizer is also added to *function-spec* (since optimizers currently can be added only to symbols, *function-spec* is constrained to be a symbol for now). The function named *function-spec* can be called from anywhere, as long as **self** is bound to an appropriate instance. The environment is correctly set up, and the internal **:defun-method** is called. This requires calling into the Flavor System and has some performance penalty over sending a message. However, if *function-spec* is

called from a context where the compiler can know the current flavor (in other words, some constraints on what **self** can be), the optimizer on *function-spec* turns into a call to the **:defun-method** internal function, generating inline code to pass the correct environment.

Also, because of the optimizer, **defun-method** acts like a subst in that better code is generated if the **defun-method** is defined in a file before it is used. However, **defun-methods** are not much faster than message-passing, even when the optimized version of the call is being used.

Note that it is faster to send a computed message than it is to call a computed function that is a **defun-method**. It is slower to use **funcall** to call the function being defined with a **defun-method** than it is to send a message in which you have to have a form that computes the name of the message at run time.

defselect-method *function-spec flavor body...* *Special Form*

Defines a function that is a select-method; it differs from **defselect** in that the **defselect-method** forms of *body* are able to access the instance variables of *flavor*.

undefflavor *flavor-name* *Function*

Removes the flavor named by *flavor-name*.

undefmethod *(flavor [type] message)* *Macro*

```
(undefmethod (flavor :before :message))
```

removes the method created by

```
(defmethod (flavor :before :message) (args) ...)
```

To remove a wrapper, use **undefmethod** with **:wrapper** as the method type.

undefmethod is simply an interface to **fundefine**. **undefmethod** accepts the same syntax as **defmethod**.

undefun-method *function-spec* *Special Form*

Undoes the effect of **defun-method** in the same way that **undefmethod** undoes the effect of **defmethod**. **undefun-method** is a special form, not a function, so *function-spec* is not evaluated.

When you redefine a **defun-method** to no longer be a **defun-method**, you must use **undefun-method** for the **:defun-method** function generated internally by it. Otherwise the compiler thinks that the function is still a **defun-method** and thus generates the wrong code.

self *Variable*

When a message is sent to an object, the variable **self** is automatically bound

to that object, for the benefit of methods that want to manipulate the object itself (as opposed to its instance variables).

recompile-flavor *flavor-name* &optional *single-message* *Function*
(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. If *use-old-combined-methods* is **t**, then the existing combined method functions are used if possible. New ones are generated only if the set of methods to be called has changed. This is the default. If *use-old-combined-methods* is **nil**, automatically generated functions to call multiple methods or to contain code generated by wrappers are regenerated unconditionally. If *do-dependents* is **nil**, only the specific flavor you specified are recompiled. Normally it and all flavors that depend on it are recompiled.

recompile-flavor affects only flavors that have already been compiled. Typically this means it affects flavors that have been instantiated, but does not affect mixins.

compile-flavor-methods *flavor...* *Macro*

The form (**compile-flavor-methods** *flavor-name-1 flavor-name-2...*), placed in a file to be compiled, causes the compiler to include the automatically generated combined methods for the named flavors in the resulting **bin** file, provided all of the necessary flavor definitions have been made. Furthermore, when the **bin** file is loaded, internal data structures (such as the list of all methods of a flavor) are generated.

This means that the combined methods get compiled at compile time, and the data structures get generated at load time, rather than both happening at run time. **compile-flavor-methods** is thus 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 is still called if incompatible changes have been made, such as addition or deletion of methods that must be called by a combined method.)

You should use **compile-flavor-methods** only for flavors that are going to be instantiated. For a flavor that will never be instantiated (that is, one that only serves to be a component of other flavors that actually do get instantiated), it is useless, except in the unusual case where the other flavors can all inherit the combined methods of this flavor instead of each having its own copy of a combined method that happens to be identical to the others.

The **compile-flavor-methods** forms should be compiled after all of the information needed to create the combined methods is available. You should put these forms after all of the definitions of all relevant flavors, wrappers, and methods of all components of the flavors mentioned.

When a **compile-flavor-methods** form is seen by the interpreter, the combined methods are compiled and the internal data structures are generated.

get-handler-for *function operation* &optional (*superiors-p* **t**) *Function*
Given an object and a message, **get-handler-for** returns 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 Zmacs command List Combined Methods (m-X) to find out what it does.

get-handler-for is related to the **:handler** function spec. It can also be used with other things than flavors.

get-flavor-handler-for *flavor-name operation* &optional (*superiors-p* **t**) *Function*
Given a flavor name and a message, **get-flavor-handler-for** returns that flavor's method for that message or **nil** if it has none.

flavor-allows-init-keyword-p *flavor-name keyword* *Function*
Returns non-**nil** if the flavor named *flavor-name* allows *keyword* in the init options when it is instantiated, or **nil** if it does not. The non-**nil** value is the name of the component flavor that contributes the support of that keyword.

si:flavor-allowed-init-keywords *flavor-name* *Function*
Returns a list of all symbols that are valid init options for the flavor, sorted alphabetically. *flavor-name* should be the name of a flavor (a symbol). This function is primarily useful for people, rather than programs, to call to get information. You can use this to help remember the name of an init option or to help write documentation about a particular flavor.

symeval-in-instance *instance symbol* &optional *no-error-p* *Function*
Finds the value of an instance variable inside a particular instance. *instance* is the instance to be examined, and *symbol* is the instance variable whose value should be returned. If there is no such instance variable, an error is signalled, unless *no-error-p* is non-**nil**, in which case **nil** is returned.

set-in-instance *instance symbol value* *Function*
Alters the value of an instance variable inside a particular instance. *instance* is the instance to be altered, *symbol* is the instance variable whose value should be set, and *value* is the new value. If there is no such instance variable, an error is signalled.

locate-in-instance *instance symbol* *Function*

Returns a locative pointer to the cell inside *instance* that holds the value of the instance variable named *symbol*.

describe-flavor *flavor-name* *Function*

Prints out descriptive information about a flavor. One important piece of information is the combined list of component flavors; this list is what is printed after the phrase "and directly or indirectly depends on".

si:*flavor-compilations* *Variable*

Contains a history of when the flavor mechanism invoked the compiler. It is a list; elements toward the front of the list represent more recent compilations. Elements are typically of the form

```
(:method flavor-name type message-name)
```

type is typically **:combined**.

You can **setq** this variable to **nil** at any time, for example, before loading some files that you suspect might have missing or obsolete **compile-flavor-methods** in them.

si:*flavor-compile-trace* *Variable*

A string containing a textual description of each invocation of the compiler by the flavor system. New elements are appended to the end of the string (it has a fill pointer).

si:flavor-default-init-putprop *flavor value property* *Function*

si:flavor-default-init-putprop is like **putprop** except that its first argument is either a flavor structure or the name of a flavor. It puts the property on the default init-plist of the specified flavor.

si:flavor-default-init-get *flavor property* *Function*

si:flavor-default-init-get is like **get** except that its first argument is either a flavor structure or the name of a flavor. It retrieves the property from the default init-plist of the specified flavor. You can use **setf**:

```
(setf (si:flavor-default-init-get f p) x)
```

si:flavor-default-init-remprop *flavor property* *Function*

si:flavor-default-init-remprop is like **remprop** except that its first argument is either a flavor structure or the name of a flavor. It removes the property from the default init-plist of the specified flavor.

50. defflavor Options

This section describes all the options to **defflavor**, including those that are for specialized purposes and are seldom used. 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 that 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. You can declare things about instance variables inherited from a component flavor, but to do so you must list these instance variables explicitly in the instance variable list at the top of the **defflavor**.

:gettable-instance-variables

Enables automatic generation of methods for getting the values of instance variables. The message name is the name of the variable, in the keyword package (that is, put a colon in front of it.)

Note that there is nothing special about these methods; you could easily define them yourself. This option generates them automatically to save you the trouble of writing out many very simple method definitions. (The same is true of methods defined by the **:settable-instance-variables** option.) If you define a method for the same message name as one of the automatically generated methods, the new definition overrides the old one, just as if you had manually defined two methods for the same message name.

:settable-instance-variables

Enables automatic generation of methods for setting the values of instance variables. The message name is "**:set-**" followed by the name of the variable. All settable instance variables are also automatically made gettable and initable. (See the note in the description of the **:gettable-instance-variables** option.)

:initable-instance-variables

The instance variables listed as arguments, or all instance variables listed in this **defflavor** if the keyword is given alone, are made *initable*. This means that they can be initialized through use of a keyword (a colon followed by the name of the variable) as an init-option argument to **make-instance**.

:init-keywords The arguments are declared to be keywords in the initialization property list that is 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, required-init-keywords, or elements of this list. If the caller misspells a keyword or otherwise uses a keyword that no component flavor handles, **make-instance** signals an error. When you write an **: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, the following would provide a default "frob array" for any instance for which the user did not provide one explicitly:

```
(:default-init-plist :frob-array
                     (make-array 100))
```

:default-init-plist entries that initialize instance variables are not added to the init-plist seen by the **:init** methods.

:required-instance-variables
Declares that any flavor incorporating this one that is instantiated into an object must contain the specified instance variables. An error occurs if there is an attempt to instantiate a flavor that incorporates this one if it does not have these in its set of instance variables. Note that this option is not one of those that checks the spelling of its arguments (if it did, it would be useless).

Required instance variables can be freely accessed by methods just like normal instance variables. The difference between listing instance variables here and listing them at the front of the **defflavor** is that the latter declares that this flavor "owns" those variables and will take care of initializing them, while the former declares that this flavor depends on those variables but that some other flavor must be provided to manage them and whatever features they imply.

:required-init-keywords
Specifies keywords that must be supplied. The arguments are keywords. It is an error to try to make an instance of this flavor or any incorporating it without specifying these keywords as arguments to **make-instance** (or to **instantiate-flavor**) or as a **:default-init-plist** option in a component flavor. This error can often be detected at compile time.

:required-methods
The arguments are names of messages that any flavor incorporating this one must handle. An error occurs if an attempt is made to instantiate such a flavor and it lacks a method for one of these messages. Typically this option appears in the **defflavor** for a base flavor. Usually this is used when a base flavor does a **(send self ...)** to send itself a message that is not handled by the base flavor itself; the idea is that the base flavor will not be instantiated alone, but only with other components (mixins) that do handle the message. This keyword allows the error of having no handler for the message be detected when the flavor is defined (which usually means at compile time) rather than at run time.

:required-flavors The arguments are names of flavors that any flavor incorporating this one must include as components, directly or indirectly. The difference between declaring flavors as required and listing them directly as components at the top of the **defflavor** is that declaring flavors to be required does not make any commitments about where those flavors will appear in the ordered list of components; that is left up to whoever does specify them as components. Declaring a flavor as required mainly allows instance variables declared by that flavor to be accessed. It also provides error checking: an attempt to instantiate a flavor that does not include the required flavors as components signals an error.

The **ship** example demonstrates the use of required flavors. To define a **relativity-mixin** that increases the mass dependent on the speed, the following could be used:

```
(defflavor relativity-mixin () (moving-object))
(defmethod (relativity-mixin :mass) ()
  (// mass (sqrt (- 1 (^ (// (send self :speed)
                             *speed-of-light*)
                         2)))))
```

However, this would not work because any flavor that had **relativity-mixin** as a component would get **moving-object** right after **relativity-mixin** 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** should not change 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))
```

This puts **moving-object** last (inheriting it from **ship**).

So, the following is used instead:

```
(defflavor relativity-mixin () ()
        (:required-flavors moving-object))
```

This 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. If the component is included this way, the component is inserted between the last flavor that included it and that flavor's first normal component. The difference between declaring flavors with **:included-flavors** 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, the included flavor 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. See the section ":Required-flavors Option for Defflavor ". 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. For this reason, it is suggested that **:required-flavors** be used rather than **:included-flavors**.

The following example shows the order in which included flavors are actually included.

```
(defflavor a () ())
(defflavor b () ())
(defflavor c () ())
(defflavor d () (a))
(defflavor e () (c) (:included-flavors a b))
(defflavor f () (d e) (:included-flavors a b))
```

F's components are:

```
(F D A E B C SI:VANILLA-FLAVOR)
```

- A goes after D, rather than after E, because the included flavor (A) is that flavor's (D) normal component.

- B goes after E, rather than after F, because the included flavor (B) goes after the last place to include it, which was E.
- B goes before C because the included component (B) goes before the normal components. In this case, C was a normal component of D, which was a normal component of F.
- **si:vanilla-flavor** is always last.

:no-vanilla-flavor Normally when a flavor is defined, the special flavor **si:vanilla-flavor** is included automatically at the end of its list of components. The vanilla flavor provides some default methods for the standard messages that all objects are supposed to understand. These include **:print-self**, **:describe**, **:which-operations**, and several other messages.

If any component of a flavor specifies the **:no-vanilla-flavor** option, **si:vanilla-flavor** will not be included in that flavor. This option should not be used casually.

:mixture Defines a family of related flavors. When **make-instance** (or **instantiate-flavor**) is called, it uses keywords in the init-plist to decide which flavor of the family to instantiate. Thus, init options can be used to select the flavor as well as instance-variable values.

The *ancestral* flavor is the one that includes the **:mixture** option in its **defflavor**. The flavors in the family are automatically constructed by mixing various mixins with the ancestral flavor. The names for the family members are chosen automatically. The name of such an automatically constructed flavor is a concatenation of the names of its components, separated by hyphens; however, obvious redundancies are removed heuristically.

:mixture does not inherit. If you have a mixin that uses **:mixture**, the mixin uses only the ancestor and ignores any other flavors.

defflavor of the ancestral flavor also defines the automatically constructed flavors. **compile-flavor-methods** of the ancestral flavor also compiles combined methods of the automatically constructed flavors.

The **:mixture** option has the following form:

(:mixture *spec spec* ...**)**

Each *spec* is processed independently, and all the resulting mixins are mixed together. A *spec* can be any of the following:

(*keyword mixin*)
Add mixin if the value of *keyword* is **t**; add nothing if **nil**.

(*keyword* (*value mixin*) (*value mixin*) ...)
Look up the value of *keyword* in this alist and add the specified mixin.

(*keyword mixin subspec subspec* ...)

(*keyword* (*value mixin subspec subspec* ...) ...)
Subspecs take on the same forms as specs. Subspecs are processed only when the specified keyword has the specified value. Use them when there are interdependencies among keywords.

A *mixin* is one of the following:

symbol	The name of a flavor to be mixed in.
nil	No flavor needs to be mixed in if the keyword takes on this value.
string	This value is invalid: Signal an error with the string as the message.

make-instance and **instantiate-flavor** check that the keywords are given with valid values.

Example:

```
(defflavor cereal-stream (...) (stream)
  ...
  (:init-keywords :characters :direction
                  :ascii :hang-up-when-close)
  (:mixture (:characters
             (t nil (:direction
                      (:in buffered-line-input-stream)
                      (:out buffered-output-character-stream))
                    (:ascii ascii-translating-character-stream))
             (nil nil (:direction (:in buffered-input-stream)
                                  (:out buffered-output-stream))
                      (:ascii "Ascii translation is not
                               meaningful for binary streams")))
            (:hang-up-when-close hang-up-when-close-mixin)))
```

Note the need for an **:init-keywords** declaration for any keywords that are used only in the **:mixture** declaration.

In this declaration, any kind of stream can have a **:hang-up-when-close** option. The **:characters** option does not itself add any mixins (hence the **nil**), but the processing of the **:direction** option depends on whether it is used with a character

stream or a binary stream. The **:ascii** option is allowed only for character streams, and an error message is specified if it is used with a binary stream. If **:ascii** had not been mentioned in the **:characters nil** case, the keyword would have been ignored by **make-instance** on the assumption that an **:init** method was going to do something with it.

:default-handler The argument is the name of a function that is to be called when a message is received for which there is no method. The function is called with the arguments the instance was called with, including the message name; the values it returns are returned. If this option is not specified on any component flavor, it defaults to a function that signals an error.

The function specified with the **:default-handler** option to **defflavor** receives two additional arguments. The first argument is **self** and the second is always **nil**.

This is equivalent to using the **:unclaimed-message** message. Because of this, in some cases, the **:unclaimed-message** message might be preferable to the **:default-handler** option.

The following example shows the use of **:default-handler**.

```
(defflavor lisp-stream (forward) ()
  (:default-handler lisp-stream-forward))

(defun lisp-stream-forward (self ignore message &rest arguments)
  (lexpr-funcall (send self :forward) message arguments))
```

:ordered-instance-variables
This option is mostly for internal system uses. The arguments are names of instance variables that must appear first (and in this order) in all instances of this flavor or any flavor depending on this flavor. This is used for instance variables that are specially known about by microcode, and 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 that are to be accessible from "outside" this object, that is, from functions other than methods. A macro (actually a **defsubst**) is defined that takes an object of this flavor as an argument and returns the value of the instance variable; **setf** can 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**.

This feature works in two different ways, depending on whether the instance variable has been declared to have a fixed slot in all instances, via the **:ordered-instance-variables** option.

If the variable is not ordered, the position of its value cell in the instance must be computed at run time. This takes longer than actually sending a message. An error is signalled if the argument to the accessor macro is not an instance or is an instance that does not have an instance variable with the appropriate name. However, there is no error check that the flavor of the instance is the flavor for which the accessor macro was defined, or a flavor built on that flavor.

If the variable is ordered, the compiler compiles a call to the accessor macro into a subprimitive that simply accesses that variable's assigned slot by number. This subprimitive is only three or four times slower than **car**. The only error-checking performed is to verify that the argument is an instance and is large enough to contain that slot. There is no check that the accessed slot belongs to an instance variable of the appropriate name. Any functions that use these accessor macros must be recompiled if the number or order of instance variables in the flavor is changed. The system does not know automatically to do this recompilation. It is very easy to forget to compile something and thus encounter a 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 that owns the flavor, and the "general public" should send messages.

:accessor-prefix Normally the accessor macro created by the **:outside-accessible-instance-variables** option to access the flavor *f*'s instance variable *v* is named *f-v*. Specifying **(:accessor-prefix get$)** would cause it to be named **get$***v* instead.

:special-instance-variables
Use the **:special-instance-variables** option if you need instance variables to be bound as special variables when an instance is entered. Its format is like that of **:gettable-instance-variables**; that is, the option can be **:special-instance-variables** to declare all of the instance variables to be special variables, or it can be of the format **(:special-instance-variables a b c)** to declare only the instance variables **a**, **b**, and **c** to be special variables. When any method is called, these special variables are bound to the values in the instance, and references to these variables from methods are compiled as special variable references. This detracts from performance and should be avoided.

:export-instance-variables

Exports the symbols from the package in which the flavor is defined. The following example shows the use of **:export-instance-variables**.

```
(defflavor box
        (x-dim y-dim z-dim)
        ()
  :gettable-instance-variables
  ;; export all the instance variables
  :export-instance-variables)
```

See the section "Importing and Exporting Symbols", page 573.

:method-order The old name, **:select-method-order**, is still accepted, but it might not be supported in a future release. The arguments are names of messages that are frequently used or for which speed is important. Their combined methods are inserted into the handler hash table first, so that they are found by the first hash probe.

:method-combination

Declares the way that methods from different flavors are to be combined. Each "argument" to this option is a list (*type order message1 message2...*). *Message1*, *message2*, and so on, are names of messages whose methods are to be combined in the declared fashion. *type* is a keyword that is a defined type of combination.

Order is a keyword whose interpretation is up to *type*; typically it is either **:base-flavor-first** or **:base-flavor-last**, depending on the type that was specified.

Any component of a flavor can 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, they must agree on the specification, or else an error is signalled.

For more information about combining methods: See the section "Method Combination", page 455.

: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. A string is documentation on what the flavor is for; this could 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 might 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 follows.

: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 purpose by a particular program, which is not intended for general use.

This documentation can be viewed with the **describe-flavor** function or the Describe Flavor (m-X) Zmacs command.

:abstract-flavor Declares that the flavor exists only to define a protocol; it is not intended to be instantiated by itself. Instead, it is intended to have more specialized flavors mixed in before being instantiated.

Trying to instantiate an abstract flavor signals an error.

:abstract-flavor is an advanced feature that affects paging. It decreases paging and usage of virtual memory by allowing abstract flavors to have combined methods. Normally, only instantiated flavors get combined methods, which are small Lisp functions that are automatically built and compiled by the flavor system to call all of the methods that are being combined to make the effective method. Sometimes many different instantiated flavors use the same combination of methods. If this is the case, and the abstract flavor's combined methods are the same ones that are needed by the instantiated flavors, then all instantiated flavors can simply share the combined methods of the abstract flavor instead of having to each make their own. This sharing improves performance because it reduces the working set.

compile-flavor-methods is permitted on an abstract flavor. It is useful for combined methods that most specializations of that flavor would be able to share.

51. 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 ways to get 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 an existing flavor, it can still be useful since you can use its **:default-init-plist** to select options of its component flavors and you can define one or two methods to customize it.

52. Vanilla Flavor

The messages described in this section are a standard protocol that all message-receiving objects are assumed to understand. The standard methods that implement this protocol are automatically supplied by the Flavor System unless the user specifically tells it not to do so. These methods are associated with the flavor **si:vanilla-flavor**.

si:vanilla-flavor *Flavor*

Unless you specify otherwise (with the **:no-vanilla-flavor** option to **defflavor**), every flavor includes the "vanilla" flavor, which has no instance variables but provides some basic useful methods.

:print-self *stream prindepth slashify-p* *Message*

The object should output its printed representation to a stream. The printer sends this message when it encounters an instance. The arguments are the stream, the current depth in list-structure (for comparison with **prinlevel**), and whether slashification is enabled (**prinl** vs **princ**). For more information about printed representations: See the section "What the Printer Produces", page 14.

The **:print-self** method of **si:vanilla-flavor** ignores the last two arguments, and prints something like #<*flavor-name octal-address*>. The *flavor-name* tells you the type of object it is, and the *octal-address* allows you to tell different objects apart (provided the garbage collector does not move them).

:describe *Message*

The object should describe itself, printing a description onto the **standard-output** stream. The **describe** function sends this message when it encounters an instance or an entity. The **:describe** method of **si:vanilla-flavor** outputs the object, the name of its flavor, and the names and values of its instance-variables, in a reasonable format.

:which-operations *Message*

The object should return a list of the messages it can handle. The **:which-operations** method of **si:vanilla-flavor** generates the list once per flavor and remembers it, minimizing consing and compute time. If a new method is added, the list is regenerated the next time someone asks for it.

:operation-handled-p *operation* *Message*

operation is a message name. The object should return **t** if it has a handler for the specified message, **nil** if it does not.

:get-handler-for *operation* *Message*

operation is a message name. The object should return the method it uses to handle *operation*. If it has no handler for that message, it should return **nil**. This is like the **get-handler-for** function, but you can only use it on objects known to accept messages.

:send-if-handles *operation* &rest *arguments* *Message*

operation is a message name and *arguments* is a list of arguments for that message. The object should send itself that message with those arguments if it handles the message. If it does not handle the message it should return **nil**.

:eval-inside-yourself *form* *Message*

The argument is a form that is evaluated in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables. You can use **setq** on one of these special variables to modify the instance variable; this is mainly for debugging. An especially useful value of *form* is **(break t)**; this gets you a Lisp top-level loop inside the environment of the flavor's methods, allowing you to examine and alter instance variables, and run functions that use the instance variables.

:funcall-inside-yourself *function* &rest *args* *Message*

function is applied to *args* in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables. You can use **setq** on one of these special variables to modify the instance variable; this is mainly for debugging.

:unclaimed-message *message* &rest *arguments* *Message*

For each message, the Flavor System checks to be sure that a method exists for the message. If no method is found, it checks for a handler for **:unclaimed-message**. If such a handler exists, it is invoked with arguments *message* (the unclaimed message) and all the arguments that were sent to the unclaimed message.

This is equivalent to using the **:default-handler** option to **defflavor**.

53. Method Combination

There are many ways to combine methods. The simplest way is the **:daemon** type of combination, which is the default. To use one of the other types, you use the **:method-combination** option to **defflavor** to specify that all the methods for a certain message to the flavor, or a flavor built on it, should be combined in a certain way.

Note that for most types of method combination other than **:daemon**, you must define the order in which the methods are combined in the **:method-combination** option; the order can be either **:base-flavor-first** or **:base-flavor-last**, depending on the method combination type that is specified. In this context, base-flavor means the last element of the flavor's fully expanded list of components. The method type keywords that are allowed depend on the type of method combination selected. There are also certain method types used for internal purposes.

The combined methods are compiled at compile time. At load time, a **compile-flavor-methods** form initializes all the cached information.

You can define your own types of method combination. The types of method combination that are supplied by the system are as follows:

:daemon This is the default type of method combination. All the **:before** methods are called, then the primary (untyped) method for the outermost flavor that has one is called, then all the **:after** methods are called. The value returned is the value of the primary method.

:progn All the methods are called, inside a **progn** special form. Methods can have a **:progn** type for documentation. This means that all the methods are called, and the result of the combined method is whatever the last of the methods returns.

:or All the methods are called, inside an **or** special form. Methods can have an **:or** type for documentation. This means that each of the methods is called in turn. If a method returns a non-**nil** value, that value is returned and none of the rest of the methods are called; otherwise, the next method is called. Thus each method is given a chance to handle the message; if it does not want to handle the message, it should return **nil**, to give the next method a chance to try.

:and All the methods are called, inside an **and** special form. Methods can have an **:and** type for documentation. The basic idea is much like **:or**.

:list Calls all the methods and returns a list of their returned values. Methods can have a **:list** type for documentation.

:inverse-list Calls each method with one argument; these arguments are successive elements of the list that is the sole argument to the message. Methods can have an **:inverse-list** type for documentation. Returns no particular value. If the result of a **:list**-combined message is sent back with an **:inverse-list**-combined message, with the same ordering and with corresponding method definitions, each component flavor receives the value that came from that flavor.

:pass-on Calls each method on the values returned by the preceding one. The values returned by the combined method are those of the outermost call. Methods can have a **:pass-on** type for documentation. The format of the declaration in the **defflavor** is as follows, where *ordering* is **:base-flavor-first** or **:base-flavor-last**:

```
(:method-combination (:pass-on (ordering . arglist)) . operation-names)
```

arglist can include the **&aux** and **&optional** keywords.

:append All the component methods are called as arguments to **append**. It expects each of the methods to return a list; the final result is the result of appending all these lists. Methods can have an **:append** type for documentation.

:nconc All the component methods are called as arguments to **nconc**. It expects each of the methods to return a list; the final result is the result of concatenating these lists. Methods can have an **:nconc** type for documentation.

:daemon-with-or This is like the **:daemon** method combination type, except that the primary method is wrapped in an **or** special form with all **:or** methods. Multiple values are returned from the primary method, but not the **:or** methods. This produces combined methods like the following (simplified to ignore multiple values):

```
(progn (foo-before-method)
       (or (foo-or-method)
           (foo-primary-method))
       (foo-after-method))
```

This is primarily useful for flavors in which a mixin introduces an alternative to the primary method. Each **:or** message gets a chance to run before the primary method and to decide whether or not the primary method should be run; if any **:or** method returns a non-**nil** value, the primary method is not run (nor are the rest of the **:or** methods). Note that the ordering of the combination of the **:or** methods is controlled by the *order* keyword in the **:method-combination** option to **defflavor**.

:daemon-with-and
This is similar to **:daemon-with-or**, except that it combines **:and** methods in an **and** special form. The primary method is run only if all the **:and** methods return non-**nil** values.

:daemon-with-override
This is similar to the **:daemon** method combination type, except an **or** special form is wrapped around the entire combined method with all **:override** typed methods before the combined method. This differs from **:daemon-with-or** in that the **:before** and **:after** daemons are not run unless *none* of the **:override** methods returns non-**nil**. The combined method looks something like this:

```
(or (foo-override-method)
    (progn (foo-before-method)
           (foo-primary-method)
           (foo-after-method)))
```

:case Takes a subsidiary message name. It dispatches on this message name just as the original message name caused a primary dispatch. This facility is used in the condition handling system.

```
(defmethod (sys:subscript-out-of-bounds :case :proceed :new-subscript)
           (&optional (sub (prompt-and-read :number
                                            "Subscript to use instead: ")))
  "Supply a different subscript"
  (values :new-subscript sub))

(send obj :proceed :new-subscript new-sub)
```

The following are all the method types used in the standard system (you 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 This is used for the before-daemon methods used by **:daemon** method combination.

:after This is used for the after-daemon methods used by **:daemon** method combination.

:override This allows some of the features of **:or** method combination to be used with daemons. An **:override** method can choose at run time to act like a primary method or as if it were not there. Typically, the **:override** method returns **nil** and does nothing, but in exceptional circumstances it takes over the handling of the message. **:override** is used only with the **:daemon-with-override** method combination.

:default If there are any untyped methods among any of the flavors being

combined, the **:default** methods are ignored. If there are no untyped methods among the flavors being combined, the **:default** methods are treated as if they were untyped.

Typically a base-flavor defines some default methods for certain messages understood by its family.

:or This is used for **:daemon-with-or** and **:daemon-with-and** method combination.

:and This is used for **:daemon-with-or** and **:daemon-with-and** method combination.

:wrapper This type is used internally by **defwrapper**.

:whopper This type is used internally by **defwhopper**.

:combined This type is used internally for automatically generated combined methods.

The most common form of combination is **:daemon**. 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, it is necessary to 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.

Whether you use a **:before** or **:after** daemon matters only if some of these methods interact. This interaction 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 a flavor; a method cannot make any assumptions about how the 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 the message have run yet. But the **:after** daemon knows that the **:before** daemon for each of the other flavors has run. So if one flavor wants to convey information to the other, the first one should "transmit" the information in a **:before** daemon, and the second one should "receive" it in an **:after** daemon. So while the **:before** daemons are run, information is "transmitted"; that is, instance

variables get set up. Then, when the **:after** daemons are run, they can look at the instance variables and act on their values.

In the case of the **:init** method, the **:before** daemons typically set up instance variables of the object based on the init-plist, while the **:after** daemons actually do things, relying on the fact that all of the instance variables have been initialized by the time they are called.

Of course, since flavors are not hierarchically organized, the notion of levels of abstraction is not strictly applicable. However, it remains a useful way of thinking about systems.

Combination Method Types

Methods used with **:progn, :append, :nconc, :and, :or, :list, :inverse-list,** and **:pass-on** combination types can use the combination type as the method type. This is useful in documenting how the method is used.

In the following example, **(:method foo :or :find-frabjous-frob)** could have been defined as **(:method foo :find-frabjous-frob)**. The only difference is one of style; using **:or** as the method type makes it clear that the methods are combined using **:or** combination.

```
(defflavor foo (frobl) (bar)
  (:method-combination (:or :base-flavor-last :find-frabjous-frob)))

(defmethod (foo :or :find-frabjous-frob) (type)
  (dolist (frob frobl)
    (when (send frob :frabjous-p type)
      (return frob))))
```

The macro **si:define-simple-method-combination** provides a simple means of defining a method combination.

si:define-simple-method-combination *combination-type operator* *Macro*
&optional *single-arg-is-value*

Defines a method combination with the name *combination-type*, which must be a symbol and is usually a keyword, such as **:progn** or **:list**. *operator* is the name of either a special form or function (such as **progn** or **list**) that is used as the function to apply to the results of the individual methods or a special form.

As an optimization, some functions return their first argument if they are given exactly one argument. In this case, the single method can be used and no combined method need be created. *single-arg-is-value* indicates that *operator* has this property. **progn** has this property. **list** does not have this property; it returns a list length of 1.

If *combination* is the name (symbol) of the method combination, and *component-n* is a form that when evaluated performs the computation

contributed by the *n*th component, the combination produced has exactly the same values and effect as the form
(combination component-1 component-2 ...), just as though this form were actually the "source code body" for the resulting combination.

The "combination" can be a symbol naming either any function or any macro or special form that treats all the operands supplied to it as indivisible forms. Examples of this are: **+**, **-**, **set**, **and**, **if**. Examples that do *not* work include: **let**, **multiple-value-bind**, **setq**, and **cond**. Note: It is the user's responsibility to ensure that the correct number of methods are supplied for functions such as **set**, which do not have an **&rest** argument.

The *component-n* form can be thought of as having the semantics of a form such as **(funcall #'component-n-function...)**. However, the combination should not depend on the structure of the components and should treat them as indivisible, atomic forms, as their implementation could change.

54. Whoppers and Wrappers

Wrappers and whoppers provide a means for combining methods and flavors. A wrapper is a kind of macro that can be used to handle a message to an object of some flavor. Whoppers are related to wrappers and can do most of the things that wrappers can do, but have several advantages. Because they involve the interaction of several complex mechanisms, you should use great care when using wrappers and whoppers.

Both wrappers and whoppers are used in certain cases in which **:before** and **:after** daemons are not powerful enough. **:before** and **:after** daemons let you put some code before or after the execution of a method; wrappers and whoppers let you put some code *around* the execution of the method. For example, you might want to bind a special variable to some value around the execution of a method. You might also want to establish a condition handler or set up a ***catch**. Wrappers and whoppers can also decide whether or not the method should be executed.

The main difference between wrappers and whoppers is that a wrapper is like a macro, whereas a whopper is like a function. If you modify a wrapper, all the combined methods that use that wrapper have to be recompiled; the system does this automatically, but it still takes time. If you modify a whopper, only the whopper has to be recompiled; the combined methods need not be changed. Another disadvantage of wrappers is that a wrapper's body is expanded in all the combined methods in which it is involved, and if that body is very large and complex, all that code is duplicated in many different compiled-code objects instead of being shared. Using whoppers is also somewhat easier than using wrappers. Whoppers are slightly slower than wrappers since they require two extra function calls each time a message is sent.

Wrappers are defined with the following macro:

defwrapper *Macro*

Sometimes the way the Flavor System combines the methods of different flavors (the daemon system) is not powerful enough. In that case you can use **defwrapper** to define a macro that expands into code that is wrapped around the invocation of the methods. This is best explained by an example.

Suppose you need a lock locked during the processing of the **:foo** message to the **bar** flavor, which takes two arguments. You have a **lock-frobboz** special form that knows how to lock the lock (presumably it generates an **unwind-protect**). **lock-frobboz** needs to see the first argument to the message; perhaps that tells it the sort of operation, read or write, that is going to be performed.

```
(defwrapper (bar :foo) ((arg1 arg2) . body)
  `(lock-frobboz (self arg1)
     . ,body))
```

The use of the **body** macro-argument prevents the **defwrappered** macro from knowing the exact implementation and allows several **defwrappers** from different flavors to be combined properly.

Note that the argument variables, **arg1** and **arg2**, are not referenced with commas before them. Although these look like **defmacro** "argument" variables, 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 macroexpansion and back-quoting is code that, when a message is sent, binds 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 that puts a ***catch** around the processing of a message so that any one of the methods could throw out in the event of an unexpected condition.

Redefining a wrapper automatically performs the necessary recompilation of the combined method of the flavor. If a wrapper is given a new definition, the combined method is recompiled so that it gets the new definition. If a wrapper is redefined with the same old definition, the existing combined methods continue to be used, since they are still correct.

Like daemon methods, wrappers work in outside-in order; when you add a **defwrapper** to a flavor built on other flavors, the new wrapper is placed

outside any wrappers of the component flavors. However, *all* wrappers happen before *any* daemons happen. When the combined method is built, the calls to the before-daemon methods, primary methods, and after-daemon methods are all placed together, and then the wrappers are wrapped around them. Thus, if a component flavor defines a wrapper, methods added by new flavors execute within that wrapper's context.

Whoppers are defined with the following special form:

defwhopper (*flavor-name operation*) *lambda-list* &body *body* *Special Form*
Defines a whopper for the specified message to the specified flavor. *arglist* is the list of arguments, which should be the same as the argument list for any method handling the specified message.

When a message is sent to an object of some flavor, and a whopper is defined for that message, the whopper runs before any of the methods (primary or daemon). The arguments are passed, and the body of the whopper is executed. Unlike daemon combination, whoppers always return their own value, not the value of the primary handler. If a message is sent for value rather than effect, the whopper has to take responsibility for getting the value back to the caller. If the whopper does not do anything special, the methods themselves are never run and the result of the whopper is returned as the result of sending the message. However, most whoppers usually run the methods for the message. To make this happen, the body of the whopper calls one of the following two functions:

continue-whopper &rest *arguments* *Function*
Calls the methods for the message that was intercepted by the whopper. *arguments* is the list of arguments passed to those methods. This function must be called from inside the body of a whopper. Normally the whopper passes down the same arguments that it was given. However, some whoppers might want to change the values of the arguments and pass new values; this is valid.

lexpr-continue-whopper &rest *arguments* *Function*
Calls the methods for the message that was intercepted by the whopper in the same way that **continue-whopper** does, but the last element of *arguments* is a list of arguments to be passed. It is useful when the arguments to the intercepted message include an **&rest** argument.

The following whopper binds the value of the special variable **base** to 3 around the execution of the **:print-integer** message to flavor **foo** (this message takes one argument):

```
(defwhopper (foo :print-integer) (n)
  (let ((base 3))
    (continue-whopper n)))
```

The following whopper sets up a ***catch** around the execution of the **:compute-height** message to flavor **giant**, no matter what arguments this message uses:

```
(defwhopper (giant :compute-height) (&rest args)
  (*catch 'too-high
    (lexpr-continue-whopper args)))
```

Like daemon methods, whoppers work in outward-in order; when you add a **defwhopper** to a flavor built on other flavors, the new whopper is placed outside any whoppers of the component flavors. However, *all* whoppers happen before *any* daemons happen. Thus, if a component defines a whopper, methods added by new flavors are considered part of the continuation of that whopper and are called only when the whopper calls its continuation.

Whoppers and wrappers are considered equal for purposes of combination. If two flavors are combined, one having a wrapper and the other having a whopper for some method, then the wrapper or whopper of the flavor that is further out is on the outside. If, for some reason, the very same flavor has both a wrapper and a whopper for the same message, the wrapper goes outside the whopper.

defwhopper-subst (*flavor-name operation*) *lambda-list* &body *body* *Macro*

Defines a wrapper for the specified message to the specified flavor by combining the use of **defwhopper** with the efficiency of **defwrapper**. The body is expanded in-line in the combined method, providing improved time efficiency but decreased space efficiency unless the body is small. The symbols **continue-whopper** and **lexpr-continue-whopper** should not be used for any purpose other than calls to the functions with those names; for example, they should not be used as variable names and should not appear in quoted constants.

The following example shows the use of **defwhopper-subst**.

```
(defwhopper-subst (xns :add-checksum-to-packet) (checksum &optional (bias 0))
  (when (= checksum #o177777)
    (setq checksum 0))
  (continue-whopper checksum bias))
```

55. Copying Instances

The Flavor System does not include any built-in way to copy instances. Copying instances raises a number of issues:

- Do you or do you not send an **:init** message to the new instance? If you do, what init-plist options do you supply?
- If the instance has a property list, you should copy the property list (for example, with **copylist**) so that sending a **:putprop** or **:remprop** message to one of the instances does not affect the properties of the other instance.
- The instance might be contained in data structure maintained by the program of which it is a part. For example, a graphics system might have a list of all the objects that are currently visible on the screen. Copying such an instance requires making the appropriate entries in the data structure.
- If the instance is a pathname, the concept of copying is not even meaningful. Pathnames are *interned*, which means that there can only be one pathname object with any given set of instance-variable values.
- If the instance is a stream connected to a network, some of the instance variables represent an agent in another host elsewhere in the network. Copying the instance requires that a copy of that agent somehow be constructed.
- If the instance is a stream connected to a file, should copying the stream make a copy of the file or should it make another stream open to the same file? Should the choice depend on whether the file is open for input or for output?

In order to copy an instance you must understand a lot about the instance. You must know what the instance variables mean so that their values variables can be copied if necessary. You must understand the instance's relations to the external environment so that new relations can be established for the new instance. You must even understand what the general concept "copy" means in the context of this particular instance, and whether it means anything at all.

Copying is a generic operation, whose implementation for a particular instance depends on detailed knowledge relating to that instance. Modularity dictates that this knowledge be contained in the instance's flavor, not in a "general copying function". Thus the way to copy an instance is to send it a message.

The Flavor System chooses not to provide any default method for copying an instance, and does not even suggest a standard name for the copying message, because copying involves so many semantic issues.

One way that people have organized copying of instances is to define a message, **:copy**, whose methods are combined with **:append** method combination. Each method supplies some init-plist options. Thus each component flavor controls the copying of its own aspect of the instance's behavior. The resulting appended list of init-plist options is used to create the new instance. Each component flavor has an **:init** method that extracts the init-plist options that are relevant to it and initializes the appropriate aspect of the new instance. A wrapper can be used to clean up the interface to the **:copy** message seen from the outside. A simple example follows:

```
(defflavor basic-copyable-object () ()
  (:method-combination (:append :base-flavor-last :copy)))

(defwrapper (basic-copyable-object :copy) (() . body)
  `(lexpr-funcall #'make-instance (typep self) (progn ,@body)))

(defflavor copyable-property-list-mixin () (si:property-list-mixin))

(defmethod (copyable-property-list-mixin :copy) ()
  `(:property-list ,(copylist (send self :property-list))))

(defflavor example () (copyable-property-list-mixin basic-copyable-object))

(setq a (make-instance 'example))

(send a :putprop 1 'value)

(setq b (send a :copy))

(send b :get 'value)  =>  1

(send b :putprop 1.5 'value)

(send b :get 'value)  =>  1.5

(send a :get 'value)  =>  1
```

A related feature is the **:fasd-form** message, which provides a way for an instance to tell the compiler how to copy it from one Lisp world into another, via a **bin** file. This is different from making a second copy of the instance in the same Lisp world. **:fasd-form** is a way to get an *equivalent* instance when the **bin** file is loaded.

56. Implementation of Flavors

An object that is an instance of a flavor is implemented using the data type **dtp-instance**. The representation is a structure whose first word, tagged with a header data type, points to a structure (known to the microcode as an "instance descriptor") containing the internal data for the flavor. The remaining words of the structure are value cells containing the values of the instance variables. The instance descriptor is a **defstruct** that appears on the **si:flavor** property of the flavor name. It contains, among other things, the name of the flavor, the size of an instance, the table of methods for handling messages, and information for accessing the instance variables.

defflavor creates such a data structure for each flavor, and links them together according to the dependency relationships between flavors.

A message is sent to an instance simply by calling it as a function, with the first argument being the message keyword. The instance descriptor contains a hash table that associates the message keyword with 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, that calls the appropriate methods in the right order.

Any wrappers are incorporated into this combined method. The function that handles the message is called with three special arguments preceding the arguments of the message:

- **self** (the object to which the message was sent)
- **self-mapping-table** (an internal data structure used in the accessing of instance variables)
- The message keyword

fdefine and related functions understand the function-specifier syntax **(:method** *flavor-name optional-method-type message-name*).

56.1 Ordering Flavors, Methods, and Wrappers

You have a certain amount of freedom in the order in which you do **defflavors**, **defmethods**, and **defwrappers**. This freedom makes 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 is because the system needs a place to remember the method, and because it must know the flavor's instance variables if the method is to be compiled.

- A flavor can be defined (with **defflavor**) even before all of its component flavors have been defined. This allows **defflavors** to be spread between files according to a program's modularity, 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 produces a warning that an instance variable was declared special (because the system did not realize it was an instance variable). Such a warning indicates that the compiled code will not work.

- The methods automatically generated by the **:gettable-instance-variables** and **:settable-instance-variables** options to **defflavor** are generated at the time the **defflavor** is done.

- The first time a flavor is instantiated, the system looks through all the component flavors and gathers various information. At this point an error is signalled if not all of the components have been **defflavored**. This is also the time at which certain other errors are detected, such as lack of a required instance variable. The combined methods are generated at this time also, unless they already exist. They 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 flavors are made.

56.2 Changing a Flavor

You can change anything about a flavor at any time, even after it has been instantiated. You can change the flavor's general attributes by doing another **defflavor** with the same name. You can add or modify methods by using **defmethod**. If you do a **defmethod** with the same flavor-name, message-name, and (optional) method-type as an existing method, that method is replaced with the new definition. You can remove a flavor with **undefflavor** and a method with **undefmethod**.

These changes always propagate to all flavors that depend on the changed flavor. Normally the system propagates 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 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, 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 does not work.

57. Zmacs Commands for Flavors

This section documents some Zmacs commands that are useful for flavors.

m-.
: The m-. (Edit Definition) command finds the definition of a flavor in the same way that it can find the definition of a function.

  Edit Definition finds the definition of a method if you give the following as the function name:

  (:method *flavor type message*)

  The keyword **:method** can be omitted. Completion occurs on the flavor name and message name.

Describe Flavor (m-X)
: Asks you for a flavor name; when typing the 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). Names of flavors and methods in the resulting display are mouse sensitive; as usual the right mouse button gives you a menu of operations and the left button does the most common operation, typically positioning the editor to the source code for the thing at which you are pointing.

List Methods (m-X)
: Asks you for a message name. Lists the methods of all flavors that handle the message, in a mouse-sensitive display that allows you to select methods to edit.

Edit Methods (m-X)
: Asks you for a message name and lists all the flavors that have a method for that message. You can type in the message name, point to it with the mouse, or let it default to the message that is being sent by the Lisp form the cursor is inside. Unlike List Methods, Edit Methods does not produce a display of selected methods, but prepares to edit the methods.

  As usual with this type of command, the Zmacs command c-. is redefined to advance the cursor to the next method in the list, reading in its source file if necessary. Pressing c-. while the display is on the screen edits the first method.

List Combined Methods (m-X)
: Asks for a message name, then for a flavor name. It lists the methods for a specified message to a specified flavor. Error messages appear when the flavor does not handle the message and when the flavor requested is not a composed, instantiated flavor.

List Combined Methods (m-X) 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; press c-. to get to successive ones.

Edit Combined Methods (m-X)

Asks you for a message name and a flavor name. It lists all the methods that would be called if that message were sent to an instance of that flavor. You can point to the message and flavor with the mouse; completion is available for the flavor name. As in Edit Methods (m-X), the command skips the display and proceeds directly to the editing phase.

58. 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.

si:property-list-mixin *Flavor*

This mixin flavor provides messages that perform the basic operations on property lists. The messages for **si:property-list-mixin** are as follows:

:get *indicator* *Message*

Gets the value of this object's *indicator* attribute. *indicator* is a keyword symbol. If there is no such attribute, returns **nil**.

```
(send net:*local-host* :get :system-type) => :lispm
```

:getl *indicator-list* *Message*

The **:getl** message is like the **:get** message, except that the argument is a list of indicators. The **:getl** message searches down the property list for any of the indicators in *indicator-list* until it finds a property whose indicator is one of those elements. It returns the portion of the property list beginning with the first such property that it found. If it does not find any, it returns **nil**.

:putprop *property indicator* *Message*

Gives the object an *indicator*-property of *property*.

:remprop *indicator* *Message*

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* *Message*

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 special form **push**, page 150.

:property-list *Message*

Returns the list of alternating indicators and values that implements the property list.

:set-property-list *list* *Message*

Sets the list of alternating indicators and values that implements the property list to *list*.

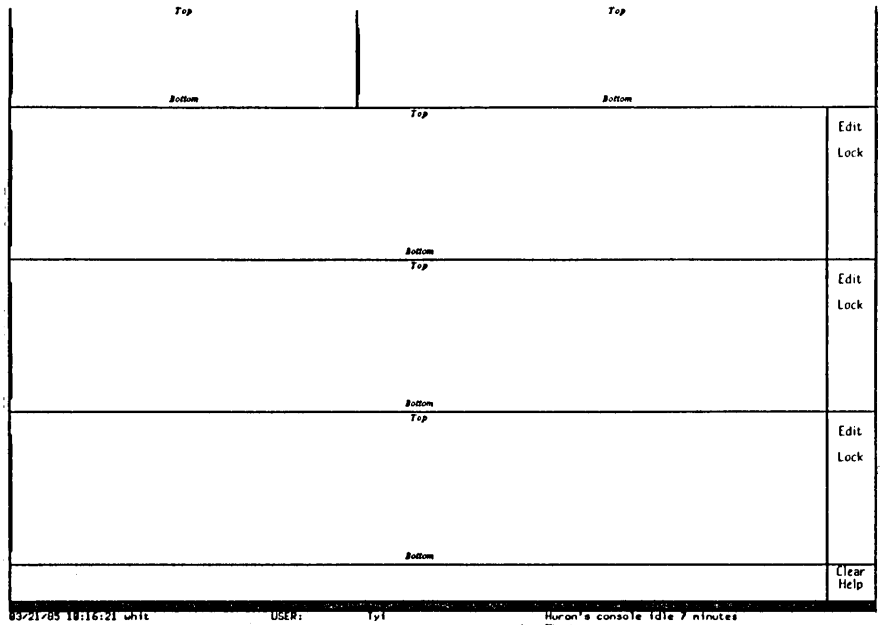
:property-list *list* (for **si:property-list-mixin**) *Init Option*

Initializes the list of alternating indicators and values that implements the property list to *list*.

59. Flavor Examiner

The Flavor Examiner utility examines the structure of flavors defined in the Lisp environment. You can select the Flavor Examiner with SELECT X, the System menu, or the Select Activity Flavor Examiner or Select Activity Flavex commands.

The Flavor Examiner window is divided into six panes.



The examiner panes (the three middle panes) list the answer to a query. The edit item of each examiner pane places the contents of the pane into a Zmacs possibilities buffer. The lock item for a examiner pane prevents the pane from being updated.

You enter a flavor name or method-spec into the interaction pane (the bottom pane).

To get started, type the name of a flavor in the interaction pane.

Methods are listed in the following format:

```
MESSAGE-NAME method-type method-combination-type FLAVOR
```

If the method-combination-type is **:case**, this format is used:

```
MESSAGE-NAME SUBMESSAGE-NAME method-type method-combination-type FLAVOR
```

Clicking on a flavor results in these actions:

- A left click on a flavor presents a menu of flavors and methods related to the flavor. (Note that automatically generated methods to get and set instance variables and methods associated with **si:vanilla-flavor** are not listed.)
- A middle click on a flavor presents a menu of related instance variables.

- A right click on a flavor presents a menu of operations on the flavor, including edit and inspect.
- Any click places on a flavor it in the flavor history pane if it is not already there.

Clicking on a method results in these actions:

- A left click on a method lists the instance variables to which the method refers.
- A middle click on a combined method lists the methods used to build the combined method.
- A middle click on a noncombined method lists all methods for that message from any flavor.
- A right click on a method presents a menu of operations on the method, including [arglist], [documentation], [edit], [inspect], [method spec], [trace], and [disassemble], unless the method is pseudocombined.
- Any click on a method places it in the method history if it is not already there.

Clicking on an instance variable results in these actions:

- A left click on an instance variable lists the methods that refer to the instance variable.
- A middle click on an instance variable shows the default value of the instance variable.

PART XI.

Conditions

60. Introduction

This documentation is tailored for applications programmers. It contains descriptions of all conditions that are signalled by Symbolics Lisp Machine software. With this information, you can write your own handlers for events detected by the system or define and handle classes of events appropriate for your own application.

The documentation describes the following major topics.

- Mechanisms for handling conditions that have been signalled by system or application code.
- Mechanisms for defining new conditions.
- Mechanisms that are appropriate for application programs to use to signal conditions.
- All of the conditions that are defined by and used in the system software.

60.1 Overview and Definitions

An *event* is "something that happens" during execution of a program. That is, it is some circumstance that the system can detect, like the effect of dividing by zero. Some events are errors — which means something happened that was not part of the contract of a given function — and some are not. In either case, a program can report that the event has occurred, and it can find and execute user-supplied code as a result.

The reporting process is called *signalling*, and subsequent processing is called *handling*. A *handler* is a piece of user-supplied code that assumes control when it is invoked as a result of signalling. Symbolics Lisp Machine software includes default mechanisms to handle a standard set of events automatically.

The mechanism for reporting the occurrence of an event relies on flavors. Each standard class of events has a corresponding flavor called a *condition*. For example, occurrences of the event "dividing by zero" correspond to the condition **sys:divide-by-zero**.

The mechanism for reporting the occurrence of an event is called *signalling a condition*. The signalling mechanism creates a *condition object* of the flavor appropriate for the event. The condition object is an instance of that flavor. The instance contains information about the event, such as a textual message to report, and various parameters of the condition. For example, when a program divides a number by zero, the signalling mechanism creates an instance of the flavor **sys:divide-by-zero**.

Handlers are pieces of user or system code that are bound for a particular condition or set of conditions. When an event occurs, the signalling mechanism searches all of the currently bound handlers to find the one that corresponds to the condition. The handler can then access the instance variables of the condition object to learn more about the condition and hence about the event.

Handlers have dynamic scope, so that the handler that is invoked for a condition is the one that was bound most recently.

The condition system provides flexible mechanisms for determining what to do after a handler runs. The handler can try to *proceed*, which means that the program might be able to continue execution past the point at which the condition was signalled, possibly after correcting the error. Any program can designate *restart* points. This facility allows a user to retry an operation from some earlier point in a program.

Some conditions are very specific to a particular set of error circumstances and others are more general. For example, **fs:delete-failure** is a specialization of **fs:file-operation-failure** which is in turn a specialization of **fs:file-error**. You choose the level of condition that is appropriate to handle according to the needs of the particular application. Thus, a handler can correspond to a single condition or to a predefined class of conditions. This capability is provided by the flavor inheritance mechanism.

61. How Applications Programs Treat Conditions

This section provides an overview of how applications programs treat conditions.

- A program signals a condition when it wants to report an occurrence of an event.
- A program binds a handler when it wants to gain control when an event occurs.

When the system or a user function detects an error, it signals an appropriate condition and some handler bound for that condition then deals with it.

Conditions are flavors. Each condition is named by a symbol that is the name of a flavor, for example, **sys:unbound-variable, sys:divide-by-zero, fs:file-not-found.** As part of signalling a condition, the program creates a condition object of the appropriate flavor. The condition object contains information about the event, such as a textual message to report and various parameters. For example, a condition object of flavor **fs:file-not-found** contains the pathname that the file system failed to find.

Handlers are bound with dynamic scope, so the most recently bound handler for the condition is invoked. When an event occurs, the signalling mechanism searches all of the current handlers, starting with the innermost handler, for one that can handle the condition that has been signalled. When an appropriate handler is found, it can access the condition object to learn more about the error.

61.1 Example of a Handler

condition-case is a simple form for binding a handler. For example:

```
(condition-case ()
    (// a b)
  (sys:divide-by-zero *infinity*))
```

This form does two things.

- Evaluates **(// a b)** and returns the result.
- Binds a handler for the **sys:divide-by-zero** condition which applies during the evaluation of **(// a b)**.

In this example, it is a simple handler that just returns a value. If division by zero happened in the course of evaluating **(// a b)**, the form would return the value of ***infinity*** instead. If any other error occurred, it would be handled by the system's default handler for that condition or by some other user handler of higher scope.

You can also bind a handler for a predefined class of conditions. For example, the symbol **fs:file-operation-failure** refers to the set of all error conditions in file system operations, such as "file not found" or "directory not found" or "link to nonexistent file", but not to such errors as "network connection closed" or "invalid arguments to **open**", which are members of different classes.

61.2 Signalling

You can signal a condition by calling either **signal** or **error**. **signal** is the most general signalling function; it can signal any condition. It allows either a handler or the user to proceed from the error. **error** is a more restrictive version that accepts only error conditions and does not allow proceeding. **error** is guaranteed never to return to its caller.

Both **signal** and **error** have the same calling sequence. The first argument is a symbol that names a condition; the rest are keyword arguments that let you provide extra information about the error. See the section "Signalling Conditions", page 501. Full details on using the signalling mechanism are in that section.

Applications programs rarely need to signal system conditions although they can. Usually when you have a signalling application, you need to define a new condition flavor to signal it. Two simpler signalling functions, called **ferror** and **fsignal**, are applicable when you want to signal without defining a new condition.

It is very important to understand that signalling a condition is not just the same thing as throwing to a tag. ***throw** is a simple control-structure mechanism allowing control to escape from an inner form to an outer form. Signalling is a convention for finding and executing a piece of user-supplied code when one of a class of events occurs. A condition handler might in fact do a ***throw**, but it is under no obligation to do so. User programs can continue to use ***throw**; it is simply a different capability with a different application.

61.3 Condition Flavors

Symbols for conditions are the names of flavors; sets of conditions are defined by the flavor inheritance mechanism. For example, the flavor **lmfs:lmfs-file-not-found** is built on the flavor **fs:file-not-found**, which is built on **fs:file-operation-failure**, which is in turn built on the flavor **error**.

The flavor inheritance mechanism controls which handler is invoked. For example, when a Symbolics Lisp Machine file system operation fails to find a file, it could signal **lmfs:lmfs-file-not-found**. The signalling mechanism invokes the first appropriate handler that it finds, in this case, a handler for **fs:file-not-found**, one for **fs:file-operation-failure**, or one for **error**. In general, if a handler is bound

for flavor **a**, and a condition object **c** of flavor **b** is signalled, then the handler is invoked if **(typep c 'a)** is true; that is, if **a** is one of the flavors that **b** is built on.

The symbol **condition** refers to all conditions, including simple, error, and debugger conditions. The symbol **error** refers to the set of all error conditions. Figure 1 shows an overview of the flavor hierarchy.

error is a base flavor for many conditions, but not all. *Simple conditions* are those built on **condition**; *debugger conditions* are those built on **dbg:debugger-condition**. *Error conditions* or *errors* are those built on **error**. For your own condition definitions, whether you decide to treat something as an error or as a simple condition is up to the semantics of the application.

From a more technical viewpoint, the distinction between simple conditions and debugger conditions hinges on what action occurs when the program does not provide its own handler. For a debugger condition, the system invokes the Debugger; for a simple condition, **signal** simply returns **nil** to the caller.

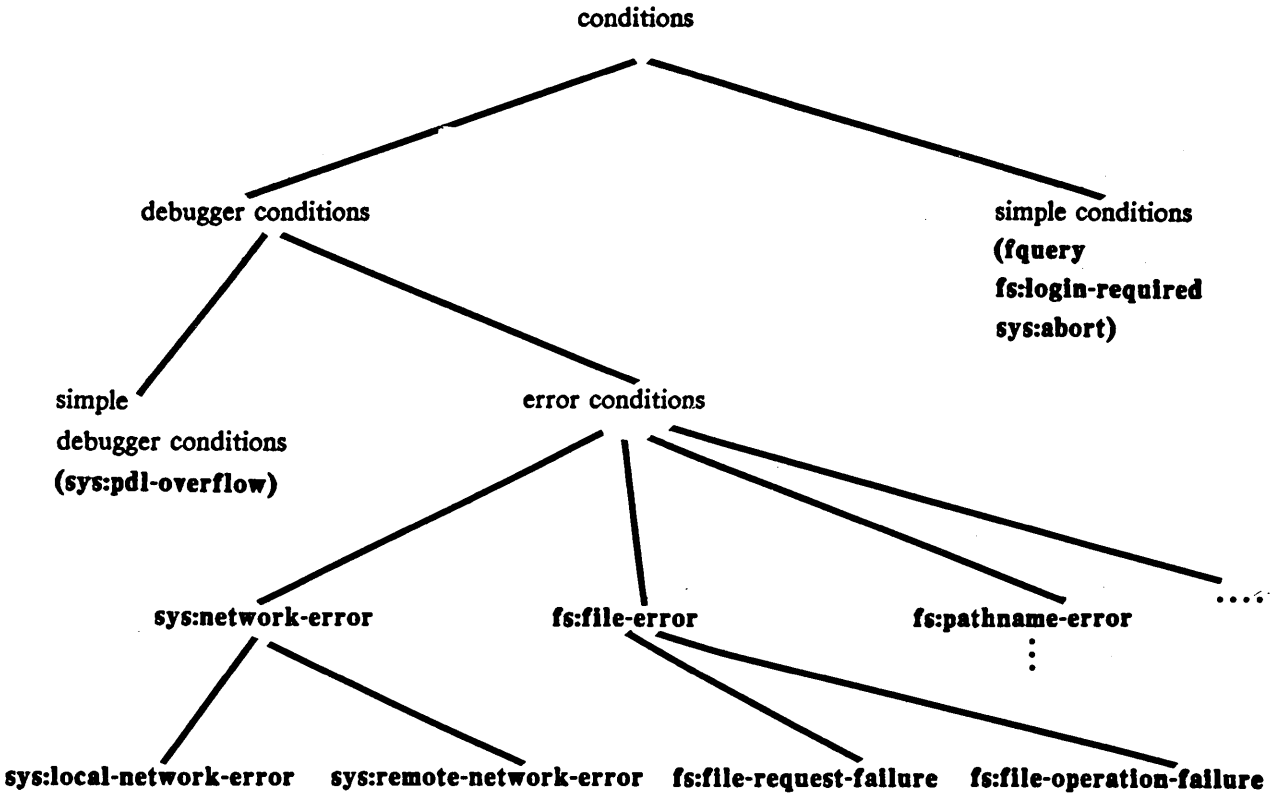


Figure 1. Condition flavor hierarchy

62. Creating New Conditions

An application might need to detect and signal events that are specific to the application. To support this, you need to define new conditions.

Defining a new condition is straightforward. For simple cases, you need only two forms: one defines the flavor, and the other defines a **:report** message. Build the flavor definition on either **error** or **condition**, depending on whether or not the condition you are defining represents an error. The following example defines an error condition.

```
(defflavor block-wrong-color () (error))

(defmethod (block-wrong-color :report) (stream)
   (format stream "The block was of the wrong color."))
```

Your program can now signal the error as follows:

```
(error 'block-wrong-color)
```

:report requires one argument, which is a stream for it to use in printing an error message. Its message should be a sentence, ending with a period and with no leading or trailing newlines.

The **:report** method must not depend on the dynamic environment in which it is invoked. That is, it should not do any free references to special variables. It should use only its own instance variables. This is because the condition object might receive a **:report** message in a dynamic environment that is different from the one in which it was created. This situation is common with **condition-case**.

The above example is adequate but does not take advantage of the power of the condition system. For example, the error message tells you only the class of event detected, not anything about this specific event. You can use instance variables to make condition objects unique to a particular event. For example, add instance variables **block** and **color** to the flavor so that **error** can use them to build the condition object:

```
(defflavor block-wrong-color (block color) (error)
  :initable-instance-variables
  :gettable-instance-variables)

(defmethod (block-wrong-color :report) (stream)
  (format stream "The block ~S was ~S, which is the wrong color."
          block color))
```

The **:initable-instance-variables** option defines **:block** and **:color** init options; the **:gettable-instance-variables** option defines methods for the **:block** and **:color** messages, which handlers can send to find out details of the condition.

Your program would now signal the error as follows:

```
(error 'block-wrong-color :block the-bad-block
                          :color the-bad-color)
```

The only other interesting thing to do when creating a condition is to define proceed types. See the section "Proceeding", page 517.

It is a good idea to use **compile-flavor-methods** for any condition whose instantiation is considered likely, to avoid the need for run-time combination and compilation of the flavor. See the macro **compile-flavor-methods**, page 438. Otherwise, the flavor must be combined and compiled the first time the event occurs, which causes perceptible delay.

62.1 Creating a Set of Condition Flavors

You can define your own sets of conditions and condition hierarchies. Just create a new flavor and build the flavors on each other accordingly. The base flavor for the set does not need a **:report** method if it is never going to be signalled itself. For example:

```
(defflavor block-world-error () (error))

(defflavor block-wrong-color (block color) (block-world-error)
  :initable-instance-variables)

(defflavor block-too-big (block container) (block-world-error)
  :initable-instance-variables)

(defmethod (block-too-big :report) (stream)
  (format stream "The block ~S is too big to fit in the ~S."
                 block container))

(defmethod (block-wrong-color :report) (stream)
  (format stream "The block ~S was ~S, which is the wrong color."
          block color))
```

63. Establishing Handlers

63.1 What is a Handler?

A handler consists of user-supplied code that is invoked when an appropriate condition signal occurs. Symbolics Lisp Machine software includes default handlers for all standard conditions. Application programs need not handle all conditions explicitly but can provide handlers for just the conditions most relevant to the needs of the application.

63.2 Classes of Handlers

The simplest form of handler handles every error condition, each in the same way. The form for binding this handler is **ignore-errors**. In addition, four basic forms are available to bind handlers for particular conditions. Each of these has a standard version and a conditional variant:

- **condition-bind** and **condition-bind-if**
  condition-bind is the most general form. It allows the handler to run in the dynamic environment in which the error was signalled and to try to proceed from the error.

- **condition-bind-default** and **condition-bind-default-if**
  condition-bind-default is a variant of **condition-bind**. It binds a handler on the default condition list instead of the bound condition list. The distinction is described in these two sections. See the section "Signalling Conditions", page 501. See the section "Default Handlers and Complex Modularity", page 509.

- **condition-case** and **condition-case-if**
  condition-case is the simplest form to use. It returns to the dynamic environment in which the handler was bound and so does not allow proceeding.

- **condition-call** and **condition-call-if**
  condition-call is a more general version of **condition-case**. It uses user-specified predicates to select the clause to be executed.

In the conditional variants, the handlers are bound only if some expression is true.

63.3 Reference Material

condition-bind *bindings body...* *Special Form*

condition-bind binds handlers for conditions and then evaluates its body with those handlers bound. One of the handlers might be invoked if a condition signal occurs while the body is being evaluated. The handlers bound have dynamic scope.

The following simple example sets up application-specific handlers for two standard error conditions, **fs:file-not-found** and **fs:delete-failure**.

```
(condition-bind ((fs:file-not-found 'my-fnf-handler)
                 (fs:delete-failure 'my-delete-handler))
  (deletef pathname))
```

The format for **condition-bind** is:

```
(condition-bind ((condition-flavor-1 handler-1)
                 (condition-flavor-2 handler-2)
                 ...
                 (condition-flavor-m handler-m))
  form-1
  form-2
  ...
  form-n)
```

condition-flavor-j The name of a condition flavor or a list of names of condition flavors. The *condition-flavor-j* need not be unique or mutually exclusive. (See the section "Finding a Handler", page 501. Search order is explained in that section.)

handler-j A form that is evaluated to produce a handler function. One handler is bound for each condition flavor clause in the list. The forms for binding handlers are evaluated in order from *handler-1* to *handler-m*. All the *handler-j* forms are evaluated and then all handlers are bound.

When *handler* is a lambda-expression, it is compiled. The handler function is a lexical closure, capable of referring to the lexical variables of the containing block.

form-i A body, constituting an implicit **progn**. The forms are evaluated sequentially. The **condition-bind** form returns whatever values *form-n* returns (**nil** when the body contains no forms). The handlers that are bound disappear when the **condition-bind** form is exited.

If a condition signal occurs for one of the *condition-flavor-j* during evaluation of the body, the signalling mechanism examines the bound handlers in the

order in which they appear in the **condition-bind** form, invoking the first appropriate handler. You can think of the mechanism as being analogous to **typecase** or **selectq**. It invokes the handler function with one argument, the condition object. The handler runs in the dynamic environment in which the error occurred; no ***throw** is performed.

Any handler function can take one of three actions:

- It can return **nil** to indicate that it does not want to handle the condition after all. The handler is free to decide not to handle the condition, even though the *condition-flavor-j* matched. (In this case the signalling mechanism continues to search for a condition handler.)
- It can throw to some outer catch-form, using ***throw**.
- If the condition has any proceed types, it can proceed from the condition by sending a **:proceed** message to the condition object and returning the resulting values. In this case, **signal** returns all of the values returned by the handler function. (Proceed types are not available for conditions signalled with **error**. See the section "Proceeding", page 517.)

condition-bind-if *cond-form bindings body...* *Special Form*

condition-bind-if binds its handlers conditionally. In all other respects, it is just like **condition-bind**. It has extra subform called *cond-form*, for the conditional. Its format is:

```
(condition-bind-if cond-form
                   ((condition-flavor-1 handler-1)
                    (condition-flavor-2 handler-2)
                    ...
                    (condition-flavor-m handler-m))
  form-1
  form-2
  ...
  form-n)
```

condition-bind-if first evaluates *cond-form*. If the result is **nil**, it evaluates the handler forms but does not bind any handlers. It then executes the body as if it were a **progn**. If the result is not **nil**, it continues just like **condition-bind** binding the handlers and executing the body.

condition-bind-default *bindings body...* *Special Form*

condition-bind-default-if *cond-form bindings body...* *Special Form*

These forms bind their handlers on the default handler list instead of the bound handler list. (See the section "Finding a Handler", page 501.) In other respects **condition-bind-default** is just like **condition-bind**, and

condition-bind-default-if is just like **condition-bind-if**. Such default handlers are examined by the signalling mechanism only after all of the bound handlers have been examined. Thus, a **condition-bind-default** can be overridden by a **condition-bind** outside of it. This advanced feature is described in more detail in another section. See the section "Default Handlers and Complex Modularity", page 509.

condition-case *(vars...) form clause...* *Special Form*

condition-case binds handlers for conditions, expressing the handlers as clauses of a case-like construct instead of as functions. The handlers bound have dynamic scope.

Examples:

```
(condition-case ()
    (time:parse string)
  (time:parse-error *default-time*))

(condition-case (e)
    (time:parse string)
  (time:parse-error
   (format error-output "~A, using default time instead." e)
   *default-time*))

(do () (nil)
  (condition-case (e)
      (return (time:parse string))
    (time:parse-error
      (setq string
            (prompt-and-read
              :string
              "~A~%Use what time instead? " e)))))
```

The format is:

```
(condition-case (var1 var2 ...)
    form
  (condition-flavor-1 form-1-1 form-1-2 ... form-1-n)
  (condition-flavor-2 form-2-1 form-2-2 ... form-2-n)
  ...
  (condition-flavor-m form-m-1 form-m-2 ... form-m-n))
```

Each *condition-flavor-j* is either a condition flavor, a list of condition flavors, or **:no-error**. If **:no-error** is used, it must be the last of the handler clauses. The remainder of each clause is a body, a list of forms constituting an implicit **progn**.

condition-case binds one handler for each clause. The handlers are bound simultaneously.

If a condition is signalled during the evaluation of *form*, the signalling

mechanism examines the bound handlers in the order in which they appear in the definition, invoking the first appropriate handler.

condition-case normally returns the values returned by *form*. If a condition is signalled during the evaluation of *form*, the signalling mechanism determines whether the condition is one of the *condition-flavor-j*. If so, the following actions occur:

1. It automatically performs a ***throw** to unwind the dynamic environment back to the point of the **condition-case**. This discards the handlers bound by the **condition-case**.

2. It executes the body of the corresponding clause.

3. It makes **condition-case** return the values produced by the last form in the handler clause.

While the clause is executing, *var1* is bound to the condition object that was signalled and the rest of the variables (*var2*, ...) are bound to **nil**. If none of the clauses needs to examine the condition object, you can omit *var1*.

```
(condition-case () ...)
```

As a special case, *condition-flavor-m* (the last one) can be the special symbol **:no-error**. If *form* is evaluated and no error is signalled during the evaluation, **condition-case** executes the **:no-error** clause instead of returning the values returned by *form*. The variables *var1*, *var2*, and so on are bound to the values produced by *form*, in the style of **multiple-value-bind**, so that they can be accessed by the body of the **:no-error** case. Any extra variables are bound to **nil**.

When an event occurs that none of the cases handles, the signalling mechanism continues to search the dynamic environment for a handler. You can provide a case that handles any **error** condition by using **error** as one *condition-flavor-j*.

condition-case-if *cond-form (vars...) form clause...* *Special Form*

condition-case-if binds its handlers conditionally. In all other respects, it is just like **condition-case**. Its syntax includes *cond-form*, a subform that controls binding handlers:

```
(condition-case-if cond-form (var)
    form
  (condition-flavor-1 form-1-1 form-1-2 ... form-1-n)
  (condition-flavor-2 form-2-1 form-2-2 ... form-2-n)
  ...
  (condition-flavor-m form-m-1 form-m-2 ... form-m-n))
```

condition-case-if first evaluates *cond-form*. If the result is **nil**, it does not

set up any handlers; it just evaluates the form. If the result is not **nil**, it continues just like **condition-case**, binding the handlers and evaluating the form.

The **:no-error** clause applies whether or not *cond-form* is **nil**.

condition-call *(vars...) form clause...* *Special Form*

condition-call binds handlers for conditions, expressing the handlers as clauses of a case-like construct instead of as functions. These handlers have dynamic scope.

condition-call and **condition-case** have similar applications. The major distinction is that **condition-call** provides the mechanism for using a complex conditional criterion to determine whether or not to use a handler. **condition-call** clauses do not have the ability to decline to handle a condition because the clause is selected on the basis of the type of a condition.

The format is:

```
(condition-call (var)
    form
  (predicate-1 form-1-1 form-1-2 ... form-1-n)
  (predicate-2 form-2-1 form-2-2 ... form-2-n)
  ...
  (predicate-m form-m-1 form-m-2 ... form-m-n))
```

Each *predicate-j* must be a function of one argument. The predicates are called, rather than evaluated. The *form-j-i* are a body, a list of forms constituting an implicit **progn**. The handler clauses are bound simultaneously.

When a condition is signalled, each predicate in turn (in the order in which they appear in the definition) is applied to the condition object. The corresponding handler clause is executed for the first predicate that returns a value other than **nil**. The predicates are called in the dynamic environment of the signaller.

condition-call takes the following actions when it finds the right predicate:

1. It automatically performs a ***throw** to unwind the dynamic environment back to the point of the **condition-call**. This discards the handlers bound by the **condition-call**.

2. It executes the body of the corresponding clause.

3. It makes **condition-call** return the values produced by the last form in the clause.

During the execution of the clause, the variable *var* is bound to the condition

object that was signalled. If none of the clauses needs to examine the condition object, you can omit *var:*

```
(condition-call () ...)
```

condition-call and :no-error

As a special case, *predicate-m* (the last one) can be the special symbol **:no-error**. If *form* is evaluated and no error is signalled during the evaluation, **condition-case** executes the **:no-error** clause instead of returning the values returned by *form*. The variables *vars* are bound to the values produced by *form*, in the style of **multiple-value-bind**, so that they can be accessed by the body of the **:no-error** case. Any extra variables are bound to **nil**.

Some limitations on predicates:

- Predicates must not have side effects. The number of times that the signalling mechanism chooses to invoke the predicates and the order in which it invokes them are not defined. For side effects in the dynamic environment of the signal, use **condition-bind**.
- The predicates are not lexical closures and therefore cannot access variables of the lexically containing form, unless those variables are declared **special**.
- Lambda-expression predicates are not compiled.

condition-call-if *cond-form (vars...) form clause...* *Special Form*

condition-call-if binds its handlers conditionally. In all other respects, it is just like **condition-call**. Its format includes *cond-form*, the subform that controls binding handlers:

```
(condition-call-if cond-form (var)
  form
  (predicate-1 form-1-1 form-1-2 ... form-1-n)
  (predicate-2 form-2-1 form-2-2 ... form-2-n)
  ...
  (predicate-m form-m-1 form-m-2 ... form-m-n))
```

condition-call-if first evaluates *cond-form*. If the result is **nil**, it does not set up any handlers; it just evaluates the form. If the result is not **nil**, it continues just like **condition-call**, binding the handlers and evaluating the form.

The **:no-error** clause applies whether or not *cond-form* is **nil**.

ignore-errors *body...* *Special Form*

ignore-errors sets up a very simple handler on the bound handlers list that

handles all error conditions. Normally, it executes *body* and returns the first value of the last form in *body* as its first value and **nil** as its second value. If an error signal occurs while *body* is executing, **ignore-errors** immediately returns with **nil** as its first value and something not **nil** as its second value.

ignore-errors replaces **errset** and **catch-error**.

63.4 Application: Handlers Examining the Stack

condition-bind handlers are invoked in the dynamic environment in which the error is signalled. Thus the Lisp stack still holds the frames that existed when the error was signalled. A handler can examine the stack using the functions described in this section.

One important application of this facility is for writing error logging code. For example, network servers might need to keep providing service even though no user is available to run the Debugger. By using these functions, the server can record some information about the state of the stack into permanent storage, so that a maintainer can look at it later and determine what went wrong.

These functions return information about stack frames. Each stack frame is identified by a *frame pointer*, represented as a Lisp locative pointer. In order to use any of these functions, you need to have appropriate environment bindings set up. The macro **dbg:with-erring-frame** both sets up the environment properly and provides a frame pointer to the stack frame that got the error. Within the body of that macro, use the appropriate functions to move up and down stack frames; these functions take a frame pointer and return a new frame pointer by following links in the stack.

These frame-manipulating functions are actually *subprimitives*, even though they do not have a % sign in their name. If given an argument that is not a frame pointer, they stand a good chance of crashing the machine. Use them with care.

The functions that return new frame pointers work by going to the *next frame* or the *previous frame* of some category. "Next" means the frame of a procedure that was invoked more recently (the frame called by this one; toward the top of the stack). "Previous" means the frame of a procedure that was invoked less recently (the caller of this frame; towards the base of the stack).

These functions assume three categories of frames: *interesting active* frames, *active* frames, and *open* frames.

- An active frame simply means a procedure that is currently running (or active) on the stack.

- Interesting active frames include all of the active frames except those that are parts of the internals of the Lisp interpreter, such as frames for **eval**, **apply**,

funcall, **let**, and other basic Lisp special forms. The list of such functions is the value of the system constant **dbg:*uninteresting-functions***.

- Open frames include all the active frames as well as frames that are still under construction, for functions whose arguments are still being computed. Open frames and active frames are synonymous.

63.4.1 Reference Material

The functions in this section all take a frame pointer as an argument. For functions that indicate a direction on the stack, using **nil** as the argument indicates the frame at relevant end of the stack. For example, when you are using a function that looks up the stack, **nil** as the argument indicates the top-most stack frame.

Remember to use the functions in this section only within the context of the **dbg:with-erring-frame** macro.

dbg:with-erring-frame *(var object)* *Macro*

dbg:with-erring-frame sets up an environment with appropriate bindings for using the rest of the functions that examine the stack. It binds *var* with the frame pointer to the stack frame that signalled the error. *var* is always a pointer to an interesting stack frame. *object* is the condition object for the error, which was the first argument given to the **condition-bind** handler.

```
(defun my-handler (condition-object)
  (dbg:with-erring-frame (frame-ptr condition-object)
    body...))
```

Inside *body*, the variable **frame-ptr** is bound to the frame pointer of the frame that got the error.

Sometimes, you might want to use the special variable **dbg:*current-frame*** as *var* because some functions expect this special variable to be bound to the stack frame that signalled the error.

You would use this special variable if you are sending the **:bug-report-description** message to the condition object, which calls stack-examination routines that depend on the idea of a current frame, in addition to the other things that **dbg:with-erring-frame** sets up. **:bug-report-description** is the message that generates the text that the Debugger c-M command puts in the mail composition window. See the message **:bug-report-description**, page 525.

dbg:get-frame-function-and-args *frame-pointer* *Function*

dbg:get-frame-function-and-args returns a list containing the name of the function for *frame-pointer* and the values of the arguments.

dbg:frame-next-active-frame *frame-pointer* *Function*

dbg:frame-next-active-frame returns a frame pointer to the next active frame following *frame-pointer*. If *frame-pointer* is the last active frame on the stack, it returns **nil**.

dbg:frame-next-interesting-active-frame *frame-pointer* *Function*

dbg:frame-next-interesting-active-frame returns a frame pointer to the next interesting active frame following *frame-pointer*. If *frame-pointer* is the last interesting active frame on the stack, it returns **nil**.

dbg:frame-next-open-frame *frame-pointer* *Function*

dbg:frame-next-open-frame returns a frame pointer to the next open frame following *frame-pointer*. If *frame-pointer* is the last open frame on the stack, it returns **nil**.

dbg:frame-previous-active-frame *frame-pointer* *Function*

dbg:frame-previous-active-frame returns a frame pointer to the previous active frame before *frame-pointer*. If *frame-pointer* is the first active frame on the stack, it returns **nil**.

dbg:frame-previous-interesting-active-frame *frame-pointer* *Function*

dbg:frame-previous-interesting-active-frame returns a frame pointer to the previous interesting active frame before *frame-pointer*. If *frame-pointer* is the first interesting active frame on the stack, it returns **nil**.

dbg:frame-previous-open-frame *frame-pointer* *Function*

dbg:frame-previous-open-frame returns a frame pointer to the previous open frame before *frame-pointer*. If *frame-pointer* is the first open frame on the stack, it returns **nil**.

dbg:frame-next-nth-active-frame *frame-pointer* &optional *(count 1)* *Function*

dbg:frame-next-nth-active-frame goes up the stack by *count* active frames from *frame-pointer* and returns a frame pointer to that frame. It returns a second value that is not **nil**. When count is positive, this is like calling **dbg:frame-next-active-frame** *count* times; *count* can also be negative or zero. If either end of the stack is reached, it returns a frame pointer to the first or last active frame and **nil**.

dbg:frame-next-nth-interesting-active-frame *frame-pointer* *Function*
&optional *(count 1)*

dbg:frame-next-nth-interesting-active-frame goes up the stack by *count* interesting active frames from *frame-pointer* and returns a frame pointer to that frame. It returns a second value that is not **nil**. When count is positive, this is like calling **dbg:frame-next-interesting-active-frame** *count* times; *count* can also be negative or zero. If either end of the stack is reached, it returns a frame pointer to the first or last active frame and **nil**.

dbg:frame-next-nth-open-frame *frame-pointer* &optional *(count 1)* *Function*

dbg:frame-next-nth-open-frame goes up the stack by *count* open frames from *frame-pointer* and returns a frame pointer to that frame. It returns a second value that is not **nil**. When count is positive, this is like calling **dbg:frame-next-open-frame** *count* times; *count* can also be negative or zero. If either end of the stack is reached, it returns a frame pointer to the first or last active frame and **nil**.

dbg:frame-out-to-interesting-active-frame *frame-pointer* *Function*

dbg:frame-out-to-interesting-active-frame returns either *frame-pointer* (if it points to an interesting active frame) or the previous interesting active frame before *frame-pointer*. (This is what the c-n-U command in the debugger does.)

dbg:frame-active-p *frame-pointer* *Function*

dbg:frame-active-p indicates whether *frame-pointer* is an active frame.

Value	*Meaning*
nil	Frame is not active
not **nil**	Frame is active

dbg:frame-real-function *frame-pointer* *Function*

dbg:frame-real-function returns either the function object associated with *frame-pointer* or **self** when the frame was the result of sending a message to an instance.

dbg:frame-total-number-of-args *frame-pointer* *Function*

dbg:frame-total-number-of-args returns the number of arguments that were passed in *frame-pointer*. For functions that take an **&rest** parameter, each argument is counted separately. Sending a message to an instance results in two implicit arguments being passed internally along with the other arguments. These implicit arguments are included in the count.

dbg:frame-number-of-spread-args *frame-pointer* &optional *(type* **:supplied***)* *Function*

dbg:frame-number-of-supplied-args returns the number of "spread" arguments that were passed in *frame-pointer*. (These are the arguments that are not part of a **&rest** parameter.) Sending a message to an instance results in two implicit arguments being passed internally along with the other arguments. These implicit arguments are included in the count.

type requests more specific definition of the number:

Value	*Meaning*
:supplied	Returns the number of arguments that were actually passed by the caller, except for arguments that were bound to a **&rest** parameter. This is the default.

:expected Returns the number of arguments that were expected by the function being called.

:allocated Returns the number of arguments for which stack locations have been allocated. In the absence of a **&rest** parameter, this is the same as **:expected** for compiled functions, and the same as **:supplied** for interpreted functions. If stack locations were allocated for arguments that were bound to a **&rest** parameter, they are included in the returned count.

These values would all be the same except in cases where a wrong-number-of-arguments error occurred, or where there are optional arguments (expected but not supplied).

dbg:frame-arg-value *frame-pointer n* &optional *callee-context no-error-p* *Function*

dbg:frame-arg-value returns the value of the *n*th argument to *frame-pointer*. It returns a second value, which is a locative pointer to the word in the stack that holds the argument. If *n* is out of range, then it takes action based on *no-error-p*: if *no-error-p* is **nil**, it signals an error, otherwise it returns **nil**. *n* can also be the name of the argument (a symbol, but it need not be in the right package). Each argument passed for an **&rest** parameter counts as a separate argument when *n* is a number. **dbg:frame-arg-value** controls whether you get the caller or callee copy of the argument (original or possibly modified.)

dbg:frame-number-of-locals *frame-pointer* *Function*

dbg:frame-number-of-locals returns the number of local variables allocated for *frame-pointer*.

dbg:frame-local-value *frame-pointer n* &optional *no-error-p* *Function*

dbg:frame-local-value returns the value of the *n*th local variable in *frame-pointer*. *n* can also be the name of the local variable (a symbol, but it need not be in the right package). It returns a second value, which is a locative pointer to the word in the stack that holds the local variable. If *n* is out of range, then the action is based on *no-error-p*: if *no-error-p* is **nil**, it signals an error, otherwise it returns **nil**.

dbg:frame-self-value *frame-pointer* &optional *instance-frame-only* *Function*

dbg:frame-self-value returns the value of **self** in *frame-pointer*, or **nil** if **self** does not have a value. If *instance-frame-only* is not **nil** then it returns **nil** unless this frame is actually a message-sending frame created by **send**.

dbg:frame-real-value-disposition *frame-pointer* *Function*

dbg:frame-real-value-disposition returns a symbol indicating how the calling function is going to handle the values to be returned by this frame.

If the calling function just returns the values to its caller, then the symbol indicates how the final recipient of the values is going to handle them.

Value	*Meaning*
:ignore	The values would be ignored; the function was called for effect.
:single	The first value would be received and the rest would not; the function was called for value.
:multiple	All the values would be received; the function was called for multiple values. It returns a second value indicating the number of values expected. **nil** indicates an indeterminate number and is always returned.

dbg:print-function-and-args *frame-pointer* &optional *show-pc-p* *Function*

dbg:print-function-and-args prints the name of the function executing in *frame-pointer* and the names and values of its arguments, in the same format as the Debugger uses. If *show-pc-p* is true, the program counter value of the frame, relative to the beginning of the function, is printed in octal. **dbg:print-function-and-args** returns the number of local slots occupied by arguments.

dbg:print-frame-locals *frame-pointer local-start* &optional *(indent 0)* *Function*

dbg:print-frame-locals prints the names and values of the local variables of *frame-pointer*. *local-start* is the first local slot number to print; the value returned by **dbg:print-function-and-args** is often suitable for this. *indent* is the number of spaces to indent each line; the default is no indentation.

64. Signalling Conditions

64.1 Signalling Mechanism

The following functions and macros invoke the signalling mechanism, which finds and invokes a handler for the condition.

error
signal
ferror
fsignal
signal-proceed-case

64.1.1 Finding a Handler

The signalling mechanism finds a handler by inspecting four lists of handlers, in this order:

1. It first looks down the list of *bound* handlers, which are handlers set up by **condition-bind**, **condition-case**, and **condition-call** forms.

2. Next, it looks down the list of *default* handlers, which are set up by **condition-bind-default**.

3. Next, it looks down the list of *interactive* handlers. This list normally contains only one handler, which enters the Debugger if the condition is based on **dbg:debugger-condition** and declines to handle it otherwise.

4. Finally, it looks down the list of *restart* handlers, which are set up by **error-restart**, **error-restart-loop**, and **catch-error-restart**. See the section "Default Handlers and Complex Modularity", page 509. See the section "Restart Handlers", page 513.

5. If it gets to the end of the last list without finding a willing handler, one of two things happens.

   - **signal** returns **nil** when both of the following are true:

     - The condition was signalled with **signal**, **fsignal**, or **signal-proceed-case**.

     - The condition object is not an instance of a condition based on **error**.

- The Debugger assumes control.

The signalling mechanism checks each handler to see if it is willing to handle the condition. Some handlers have the ability to decline to handle the condition, in which case the signalling mechanism keeps searching. It calls the first willing handler it finds.

As we have seen, the signalling mechanism searches for handlers in a Specific order. It looks at all the bound handlers before any of the default handlers and all of the default handlers before any of the restart handlers. Thus, it tries any **condition-bind** handler before any handler bound by **condition-bind-default**, even though the **condition-bind-default** is within the dynamic scope of the **condition-bind**. Similarly, it considers a **condition-bind** handler before an **error-restart** handler, even when the **error-restart** handler was bound more recently. See the section "Default Handlers and Complex Modularity", page 509.

While a bound or default handler is executing, that handler and all handlers inside it are removed from the list of bound or default handlers. This is to prevent infinite recursion when a handler signals the same condition that it is handling, as in the following simplistic example:

```
(condition-bind ((error '(lambda (x) (ferror "foo"))))
  (ferror "foo"))
```

If you want recursion, the handler should bind its own condition.

64.1.2 Signalling Simple Conditions

If a simple condition or a debugger condition not based on **error** is signalled, the signalling mechanism searches for a handler on the bound handler and default handler lists. When it finds one, it invokes it. Otherwise, the signalling mechanism checks for an interactive handler, invoking the first one it finds. If there are no interactive handlers, the first restart handler for that condition is invoked. If no restart handler for the condition is found, **signal** returns **nil**; **error** enters the Debugger.

Normally, there is only one interactive handler. This handler calls the debugger if the condition is a debugger condition and not a simple condition. See the section "Interactive Handlers", page 511.

64.1.3 Signalling Errors

In practice, if the **signal** function is applied to an error condition object, **signal** is very unlikely to return **nil**, because most processes contain a restart handler that handles all error conditions. The function at the base of the stack of most processes contains a **catch-error-restart** form that handles **error** and **sys:abort**. Thus, if you are in the Debugger as a result of an error, you can always use ABORT. The restart handler at the base of the stack always handles **sys:abort** and either terminates or restarts the process.

64.1.4 Restriction Due to Scope

A condition must be signalled only in the environment in which the event that it represents took place, to insure that handlers run in the proper dynamic environment. Therefore, you cannot signal a condition object that has already been signalled once. In particular, when you are writing a handler, you cannot have that handler signal its condition argument. Similarly, if a condition object is returned by some program (such as the **open** function given **nil** for the **:error** keyword), you cannot signal that object.

It is not correct to pass on the condition by signalling the handler's condition argument. This is incorrect:

```
(defun condition-handler (condition)
   (if something (*throw ...) (signal condition)))
```

Instead you should do this:

```
(defun condition-handler (condition)
   (if something (*throw ...) nil))
```

or this:

```
(defun condition-handler (condition)
   (if something (*throw ...) (signal 'some-other-condition)))
```

64.2 Reference Material

signal *flavor-name* &rest *init-options* *Function*

signal is the primitive function for signalling a condition. The argument *flavor-name* is a condition flavor symbol. The *init-options* are the init options when the **condition-object** is created; they are passed in the **:init** message to the instance. (See the function **make-instance**, page 430.) **signal** creates a new condition object of the specified flavor, and signals it. If no handler handles the condition and the object is not an error object, **signal** returns **nil**. If no handler handles the condition and the object is an error object, the Debugger assumes control.

In a more advanced form of **signal**, *flavor-name* can be a condition object that has been created with **make-condition** but not yet signalled. In this case, *init-options* is ignored.

error *flavor-name* &rest *init-options* *Function*

error is the function for signalling a condition that is not proceedable. The argument *flavor-name* is a condition flavor symbol or an error object, created by **make-condition**. The *init-options* are the init options specified when the error object is created; they are passed in the **:init** message. **error** is similar to **signal** but restricted in the following ways:

- **error** sets the proceed types of the error object to **nil** so that it cannot be proceeded.
- If no handler exists, the Debugger assumes control, whether or not the object is an error object.
- **error** never returns to its caller.

In a more advanced form of **error**, *flavor-name* can be a condition object that has been created with **make-condition** but not yet signalled. In this case, *init-options* is ignored.

For compatibility with the old Maclisp **error** function, **error** tries to determine that it has been called with Maclisp-style arguments and turns into an **fsignal** or **ferror** as appropriate. If *flavor-name* is a string or a symbol that is not the name of a flavor, and **error** has no more than three arguments, **error** assumes it was called with Maclisp-style arguments.

fsignal *format-string* &rest *format-args* *Function*

fsignal is a simple function for signalling when you do not care to use a particular condition. **fsignal** signals **dbg:proceedable-ferror**. (See the flavor **dbg:proceedable-ferror**, page 532.) The arguments are passed as the **:format-string** and **:format-args** init keywords to the error object.

ferror *format-string* &rest *format-args* *Function*

ferror is a simple function for signalling when you do not care what the condition is. **ferror** signals **ferror**. (See the flavor **ferror**, page 532.) The arguments are passed as the **:format-string** and **:format-args** init keywords to the error object.

The old **(ferror nil ...)** syntax continues to be accepted for compatibility reasons indefinitely; the **nil** is ignored. An error is signalled if the first argument is a symbol other than **nil**; the first argument must be **nil** or a string.

parse-ferror *format-string* &rest *format-args* *Function*

Signals an error of flavor **sys:parse-ferror**. *format-string* and *format-args* are passed as the **:format-string** and **:format-args** init options to the error object.

See the flavor **parse-ferror**, page 543.

errorp *object* *Function*

errorp returns **t** if *object* is an error object, and **nil** otherwise. That is:

```
(errorp x) <=> (typep x 'error)
```

make-condition *flavor-name* &rest *init options* *Function*

make-condition creates a condition object of the specified flavor with the specified init-options. This object can then be signalled by passing it to **signal** or **error**. Note that you are not supposed to design functions that indicate errors by *returning* error objects; functions should always indicate errors by *signalling* error objects. This function makes it possible to build complex systems that use subroutines to generate condition objects so that their callers can signal them.

check-arg *var-name predicate description* *Macro*

The **check-arg** form is useful for checking arguments to make sure that they are valid. A simple example is:

```
(check-arg foo stringp "a string")
```

foo is the name of an argument whose value should be a string. **stringp** is a predicate of one argument, which returns **t** if the argument is a string. **"a string"** is an English description of the correct type for the variable.

The general form of **check-arg** is

```
(check-arg var-name
           predicate
           description)
```

var-name is the name of the variable whose value is of the wrong type. If the error is proceeded this variable is **setq**'ed to a replacement value. *predicate* is a test for whether the variable is of the correct type. It can be either a symbol whose function definition takes one argument and returns non-**nil** if the type is correct, or it can be a nonatomic form which is evaluated to check the type, and presumably contains a reference to the variable *var-name*. *description* is a string which expresses *predicate* in English, to be used in error messages.

The *predicate* is usually a symbol such as **fixp, stringp, listp,** or **closurep,** but when there isn't any convenient predefined predicate, or when the condition is complex, it can be a form. For example:

```
(defun test1 (a)
  (check-arg a
             (and (numberp a) (≤ a 10.) (> a 0.))
             "a number from one to ten")
  ...)
```

If **test1** is called with an argument of 17, the following message is printed:

```
The argument A to TEST1, 17, was of the wrong type.
The function expected a number from one to ten.
```

In general, what constitutes a valid argument is specified in two ways in a **check-arg**. *description* is human-understandable and *predicate* is executable. It is up to the user to ensure that these two specifications agree.

check-arg uses *predicate* to determine whether the value of the variable is of the correct type. If it is not, **check-arg** signals the **sys:wrong-type-argument** condition. See the flavor **sys:wrong-type-argument**, page 539.

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-type foo :number)
```

foo is the name of an argument whose value should be a number. **:number** is a value which is passed as a second argument to **typep**; that is, it is a symbol that specifies a data type. The English form of the type name, which gets put into the error message, is found automatically.

The general form of **check-arg-type** is:

```
(check-arg-type var-name
                type-name
                description)
```

var-name is the name of the variable whose value is of the wrong type. If the error is proceeded this variable is **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* is provided correctly. If it is omitted and *type-name* describes some other data type, then the description is the word "a" followed by the printed representation of *type-name* in lowercase.

argument-typecase *arg-name* &body *clauses* *Special Form*

argument-typecase is a hybrid of **typecase** and **check-arg-type**. Its clauses look like clauses to **typecase**. **argument-typecase** automatically generates an **otherwise** clause which signals an error. The proceed types to this error are similar to those from **check-arg**; that is, you can supply a new value that replaces the argument that caused the error.

For example, this:

```
(defun foo (x)
 (argument-typecase x
   (:symbol (print 'symbol))
   (:number (print 'number))))
```

is the same as this:

```
(defun foo (x)
  (check-arg x
    (typecase x
      (:symbol (print 'symbol) t)
      (:number (print 'number) t)
      (otherwise nil))
   "a symbol or a number"))
```

65. Default Handlers and Complex Modularity

When more than one handler exists for a condition, which one should be invoked? The signalling mechanism has an elaborate rule, but in practice, it usually invokes the innermost handler. See the section "Finding a Handler", page 501. "Innermost" is defined dynamically and thus means "the most recently bound handler".

This decision is made on the basic principle of modularity and referential transparency: a function should behave the same way, regardless of what calls it. Therefore, whether a handler bound by a function gets invoked should not depend on what is going on with that function's callers.

For example, suppose function **a** sets up a handler to deal with the **fs:file-not-found** condition, and then calls procedure **b** to perform some service for it. Now, unbeknownst to **a**, **b** sometimes opens a file, and **b** has a condition handler for **fs:file-not-found**. If **b**'s file is not found, **b**'s handler handles the error rather than **a**'s. This is as it should be, because it should not be visible to **a** that **b** uses a file (this is a hidden implementation detail of **b**). **a**'s unrelated condition handler should not meddle with **b**'s internal functioning. Therefore, the signalling mechanism follows a basic inside-to-outside searching rule.

Sometimes a function needs to signal a condition but still handle the condition itself if none of its callers handles it. On first encounter, this need seems to require an outside-to-inside searching rule instead of the inside-to-outside searching rule mandated by modularity considerations. How can you circumvent the rules to allow a function to handle something only if no outer function handles it?

Several strategies are available for dealing with this. Because of our lack of experience with the condition signalling system, we are not yet sure which of these are better than others. We are providing several mechanisms in order to allow experimentation and flexibility.

- The simplest solution is to provide a proceed type for proceeding from the Debugger. That is, your program signals an error to allow callers to handle the condition. If none of them handles it, the Debugger assumes control. Provided that the user decides to use the proceed type, your program then gets to handle the condition. If what your program wanted to do was to prompt the user anyway, this might be the right thing. This is most likely true if you think that a program error is probably happening and the user might want to be able to trace and manipulate the stack using the Debugger.

- Another simple solution is to signal a condition that is not an error. **signal** returns **nil** when no handler is found, and your program can take appropriate action.

- Use **condition-bind-default** to create a handler on the default handler list. The signalling mechanism searches this list only after searching through all regular bound handlers. One drawback of this scheme is that it works only to one level. If you have three nested functions, you cannot get outside-to-inside modularity for all three, because only two lists exist, the bound list and the default list. This facility is probably good enough for some applications however.

- Use **dbg:condition-handled-p** to determine whether a handler has been bound for a particular flavor. This has the advantage that it works for any number of levels of nested handler, instead of only two. One disadvantage is that it can return **:maybe**, which is ambiguous.

The simple solutions work only if your program is doing the signalling. If some other program is signalling a condition, you cannot control whether the condition is an error condition or whether it has any proceed types; you can only write handlers.

65.1 Reference Material

dbg:condition-handled-p *condition-flavor* *Function*

dbg:condition-handled-p searches the bound handler list and the default handler list to see whether a handler exists for *condition-flavor*. This function should be called only from a **condition-bind** handler function. It starts looking from the point in the lists from which the current handler was invoked and proceeds to look outwards through the bound handler list and the default handler list. It returns a value to indicate what it found:

Value	*Meaning*
:maybe	**condition-bind** handlers for the flavor exist. These handlers are permitted to decline to handle the condition. You cannot determine what would happen without actually running the handler.
nil	No handler exists.
t	A handler exists.

66. Interactive Handlers

The interactive handler list contains one element: a handler that invokes the Debugger if the condition is built on **dbg:debugger-condition** and declines to handle the condition if it is not. No standard procedure exists for changing the contents of this list.

One of the original design goals of the condition signalling mechanism was to support building complex applications that could take over the function of the Debugger and provide their own. The exact definition of the problem is not completely clear however. We are not sure whether the current system provides this functionality.

If you are writing an application that needs to take over error handling completely, you might be able to create a **condition-bind** handler that handles **error**, to prevent invocation of the Debugger. This strategy might have problems that we have not anticipated. If you really need to get the Debugger out of the way, you might try changing the interactive handler list. We have not defined a way to do this; read the code for complete details. We cannot guarantee that whatever you do will work in future releases. However, we encourage your experimentation. Please contact us so that we can help you if possible.

Briefly, the variable holding the list is named **dbg:*interactive-handlers***, which holds an interactive handler object. The list is reset to hold the standard Debugger when you warm boot the machine.

An interactive handler object must handle the following messages:

:handle-condition-p *cond* *Message*

:handle-condition-p examines *cond* which is a condition object. It returns **nil** it if declines to handle the condition and something other than **nil** when it is prepared to handle the condition.

:handle-condition *cond ignore* *Message*

cond is a condition object. You should handle this condition, ignoring the second argument. **:handle-condition** can return values or throw in the same way that **condition-bind** handlers can.

67. Restart Handlers

One way to handle an error is to restart at some earlier point the program that got the error. A program can specify points where it is safe or convenient for it to be restarted should a condition signal occur during processing a function. The basic special form for doing this is called **error-restart**. The following example is taken from the system code:

```
(defun connect (address contact-name
                &optional (window-size default-window-size)
                (timeout (* 10. 60.))
                &aux conn real-address (try 0))
  (error-restart (connection-error
                   "Retry connection to ~A at ~S with longer timeout"
                   address contact-name)
    forms...))
```

This code fragment evaluates *forms* and returns the final value(s) if successful. If the Debugger assumes control as a result of a **chaos:connection-error** condition, the user is given the opportunity of restarting the program. The Debugger's prompt message would be something like this:

```
s-A: "Retry connection to SCRC at FILE 1 with longer timeout"
```

If the user were to press s-A at this point, the forms implementing the connection would be evaluated again. That is, the body of the **error-restart** would be started again from the beginning.

Two variations on this basic paradigm are provided. **error-restart-loop** is an infinite loop version of **error-restart**. It always starts over regardless of whether a condition has been signalled. **catch-error-restart** never restarts, even when a condition is signalled. Instead it always returns, returning either the values from the body (if successful) or **nil** if a condition signal occurred.

catch-error-restart is the most primitive version of this control structure. The other two are built from it. It too has a conditional variant, **catch-error-restart-if**, for binding a restart handler conditionally.

A common paradigm is to use one of these forms in the command loop of an interactive program, with *condition-flavor* being **(error sys:abort)**. This way, if an unhandled error occurs, the user is offered the option of returning to the command loop, and the ABORT key returns to the command loop. Which form you use depends on the nature of your command loop.

67.1 Reference Material

The use of "error-" in the names of these functions has no real significance. They could have been called **condition-restart, condition-restart-loop**, and so on, because they apply to all conditions.

error-restart *(condition-flavor format-string format-arg...) body...* *Special Form*

This form establishes a restart handler for *condition-flavor* and then evaluates the body. If the handler is not invoked, **error-restart** returns the values produced by the last form in the body and the restart handler disappears. When the restart handler is invoked, control is thrown back to the dynamic environment inside the **error-restart** form and execution of the body starts all over again. The format is:

```
(error-restart (condition-flavor format-string . format-args)
  form-1
  form-2
  ...)
```

condition-flavor is either a condition or a list of conditions that can be handled. *format-string* and *format-args* are a control string and a list of arguments (respectively) to be passed to **format** to construct a meaningful description of what would happen if the user were to invoke the handler. *format-args* are evaluated when the handler is bound. The Debugger uses these values to create a message explaining the intent of the restart handler.

error-restart-loop *(condition-flavor format-string format-args...) body...* *Special Form*

error-restart-loop establishes a restart handler for *condition-flavor* and then evaluates the body. If the handler is not invoked, **error-restart-loop** evaluates the body again and again, in an infinite loop. Use the **return** function to leave the loop. This mechanism is useful for interactive top levels.

If a condition is signalled during the execution of the body and the restart handler is invoked, control is thrown back to the dynamic environment inside the **error-restart-loop** form and execution of the body is started all over again. The format is:

```
(error-restart-loop (condition-flavor format-string . format-args)
  form-1
  form-2
  ...)
```

condition-flavor is either a condition or a list of conditions that can be handled. *format-string* and *format-args* are a control string and a list of arguments (respectively) to be passed to **format** to construct a meaningful

description of what would happen if the user were to invoke the handler. The Debugger uses these values to create a message explaining the intent of the restart handler.

catch-error-restart *(condition-flavor format-string . format-args)* *Special Form*
body...

catch-error-restart establishes a restart handler for *condition-flavor* and then evaluates the body. If the handler is not invoked, **catch-error-restart** returns the values produced by the last form in the body, and the restart handler disappears. If a condition is signalled during the execution of the body and the restart handler is invoked, control is thrown back to the dynamic environment of the **catch-error-restart** form. In this case, **catch-error-restart** also returns **nil** as its first value and something other than **nil** as its second value. Its format is:

```
(catch-error-restart (condition-flavor format-string . format-args)
  form-1
  form-2
  ...)
```

condition-flavor is either a condition or a list of conditions that can be handled. *format-string* and *format-args* are a control string and a list of arguments (respectively) to be passed to **format** to construct a meaningful description of what would happen if the user were to invoke the handler. The Debugger uses these values to create a message explaining the intent of the restart handler.

catch-error-restart-if *cond-form (condition-flavor format-string* *Special Form*
format-args) body...

catch-error-restart-if establishes its restart handler conditionally. In all other respects, it is the same as **catch-error-restart**. Its format is:

```
(catch-error-restart-if cond-form
    (condition-flavor format-string . format-args)
  form-1
  form-2
  ...)
```

catch-error-restart-if first evaluates *cond-form*. If the result is **nil**, it evaluates the body as if it were a **progn** but does not establish any handlers. If the result is not **nil**, it continues just like **catch-error-restart**, establishing the handlers and executing the body.

67.2 Invoking Restart Handlers Manually

dbg:invoke-restart-handlers *object* &key *flavors* *Function*

dbg:invoke-restart-handlers searches the list of restart handlers to find a restart handler for *object*. The *flavors* argument controls which restart handlers are examined. *flavors* is a list of condition names. When *flavors* is omitted, the function examines every restart handler. When *flavors* is provided, the function examines only those restart handlers that handle at least one of the conditions on the list.

The first restart handler that it finds to handle the condition is invoked and given *object*. It returns **nil** if no appropriate restart handler is found.

dbg:invoke-restart-handlers can be called by handlers set up by **condition-bind** or **condition-bind-default**. The *object* argument should be the condition object passed to the handler. The handler calls this function to bypass the interactive handlers list, letting the innermost restart handler handle the condition. A program that wants to attempt to continue with a computation in the presence of errors might find this useful. For example, it could be used to support batch-mode compilation, with the user away from the console.

68. Proceeding

In some situations, execution can proceed past the point at which a condition was signalled. Events for which this is the case are called *proceedable conditions*. Some external agent makes the decision about whether it is reasonable to proceed after repairing the original problem. The agent is either a **condition-bind** handler or the user operating the Debugger.

In general, many different ways are available to proceed from a particular condition. Each way is identified by a *proceed type*, which is represented as a symbol.

68.1 Protocol for Proceeding

For proceeding to work, two conceptual agents must agree:

- The programmer who wrote the program that signals the condition;
- The programmer who wrote the **condition-bind** handler that decided to proceed from the condition, or else the user who told the Debugger to proceed.

The signaller signals the condition and provides the various proceed types. The handler chooses from among the proceed types to make execution proceed.

Each agent has certain responsibilities to the other; each must follow the protocol described below to make sure that any handler interacts correctly with any signaller. The following description should be considered a two-part protocol that each agent must follow in order to communicate correctly with the other.

In very simple cases, the signaller can use **fsignal**, which does not require any new flavor definitions.

In all other cases, the signaller signals the condition using **signal** or **signal-proceed-case**. The signaller also defines a condition flavor with at least one method to handle a proceed type. The way to define a method that creates a new proceed type is somewhat unusual in that it uses a style of method combination called **:case** combination. Here's an example from the system:

```
(defmethod (sys:subscript-out-of-bounds :case :proceed :new-subscript)
           (&optional (sub (prompt-and-read :number
                                            "Subscript to use instead: ")))
  "Supply a different subscript."
  (values :new-subscript sub))
```

This code fragment creates a proceed type called **:new-subscript** for the condition flavor **sys:subscript-out-of-bounds**. New proceed types are always defined by

adding a **:case :proceed** method handler to the condition flavor. The method must always return values rather than throwing.

In **:case** method combination, the first argument to the **:proceed** message is like a subsidiary message name, causing a further dispatch just as the original message name caused a primary dispatch. The method from the example is invoked whenever an object of this flavor receives a **:proceed** message like this:

```
(send obj :proceed :new-subscript new-sub)
```

The variables in the lambda list for the method come from the rest of the arguments of the **send**.

All of the arguments to a **:proceed** method must be optional arguments. The **:proceed** method should provide default values for all its arguments. One useful way of doing this is to prompt a user for the arguments using the **query-io** stream. The example uses **prompt-and-read**. If all the optional arguments were supplied, the **:proceed** method must not do any input or output using **query-io**.

This facility has been defined assuming that **condition-bind** handlers would supply all the arguments for the method themselves. The Debugger runs this method and does not supply arguments, relying on the method to prompt the user for the arguments.

As in the example, the method should have a documentation string as the first form in its body. The **:document-proceed-type** message to a proceedable condition object displays the string. This string is used by the Debugger as a prompt to describe the proceed type. For example, the subscript example might result in the following Debugger prompt:

```
s-A: Supply a different subscript
```

The string should be phrased as a one-line description of the effects of proceeding from the condition. It should not have any leading or trailing newlines. (You can use the messages that the Debugger prints out to describe the effects of the `s-` commands as models if you are interested in stylistic consistency.)

Sometimes a simple fixed string is not adequate. You can provide a piece of Lisp code to compute the documentation text at run time by providing your own method for **:case :document-proceed-type**. This method definition takes the following form:

```
(defmethod (condition-flavor :case :document-proceed-type proceed-type)
           (stream)
  body...)
```

The body of the method should print documentation for *proceed-type* of *condition-flavor* onto *stream*.

The body of the **:proceed** method can do anything it wants. In general, it tries to repair the state of things so that execution can proceed past the point at which the condition was signalled. It can have side-effects on the state of the environment, it

can return values so that the function that called **signal** can try to fix things up, or it can do both. Its operation is invisible to the handler; the signaller is free to divide the work between the function that calls **signal** and the **:proceed** method as it sees fit. When the **:proceed** method returns, **signal** returns all of those values to its caller. That caller can examine them and take action accordingly.

The meaning of these returned values is strictly a matter of convention between the **:proceed** method and the function calling **signal**. It is completely internal to the signaller and invisible to the handler. By convention, the first value is often the name of a proceed type. See the section "Signallers", page 520.

A **:proceed** method can return a first value of **nil** if it declines to proceed from the condition. If a **nil** returned by a **:proceed** method becomes the return value for a **condition-bind** handler, this signifies that the handler has declined to handle the condition, and the condition continues to be signalled. When the **:proceed** message was sent by the Debugger, the Debugger prints a message saying that the condition was not proceeded, and it returns to its command level. This might be used by an interactive **:proceed** method that gives the user the opportunity either to proceed or to abort; if the user aborts, the method returns **nil**. Returning **nil** from a **:proceed** method should not be used as a substitute for detecting earlier (such as when the condition object is created) that the proceed type is inappropriate for that condition.

68.2 Proceed Type Messages

By default, condition objects have to handle all proceed types defined for the condition flavor. Condition objects can be created that handle only some of the proceed types for their condition flavor. When the signaller creates the condition object (with **signal** or **make-condition**), it can use the **:proceed-types** init option to specify which proceed types the object accepts. The value of the init option is a list of keyword symbols naming the proceed types.

```
(signal 'my-condition :proceed-types '(:abc))
```

The **:proceed-types** message to a condition object returns a list of keywords for the proceed types that the object is prepared to handle. (See the method **(:method condition :proceed-types)**, page 530.)

The **:proceed-type-p** message examines the list of valid proceed types to see whether it contains a particular proceed type. (See the method **(:method condition :proceed-type-p)**, page 530.)

A condition flavor might also have an **:init** daemon that could modify its **dbg:proceed-types** instance variable.

68.3 Proceeding with condition-bind Handlers

Suppose the handler is a **condition-bind** handler function. Just to review, when the condition is signalled, the handler function is called with one argument, the condition object. The handler function can throw to some tag, return **nil** to say that it doesn't want to handle the condition, or try to proceed the condition.

The handler must not attempt to proceed using an invalid proceed type. It must determine which proceed types are valid for any particular condition object. It must do this at run-time because condition objects can be created that do not handle all of the proceed types for their condition flavor. (See the init option **(:method condition :proceed-types)**, page 530.) In addition, condition objects created with **error** instead of **signal** do not have any proceed types. The handler can use the **:proceed-types** and **:proceed-type-p** messages to determine which proceed types are available.

To proceed from a condition, a handler function sends the condition object a **:proceed** message with one or more arguments. The first argument is the proceed type (a keyword symbol) and the rest are the arguments for that proceed type. All of the standard proceed types are documented with their condition flavors. Thus the programmer writing the handler function can determine the meanings of the arguments. The handler function must always pass all of the arguments, even though they are optional.

Sending the **:proceed** message should be the last thing the handler does. It should then return immediately, propagating the values from the **:proceed** method back to its caller. Determining the meaning of the returned values is the business of the signaller only; the handler should not look at or do anything with these values.

68.4 Proceed Type Names

Any symbol can be used as the name of a proceed type, although using keyword symbols is conventional. The symbols **:which-operations** and **:case-documentation** are not valid names for proceed types because they are treated specially by the **:case** flavor combination. Do not use either of these symbols as the name of a proceed type when you create a new condition flavor.

68.5 Signallers

Signallers can use the **signal-proceed-case** special form to signal a proceedable condition. **signal-proceed-case** assumes that the first value returned by every proceed type is the keyword symbol for that proceed type. This convention is not currently enforced.

68.6 Reference Material

signal-proceed-case *Special Form*

signal-proceed-case signals a proceedable condition. It has a clause to handle each proceed type of the condition. It has a slightly more complicated syntax than most special forms: you provide some variables, some argument forms, and some clauses:

```
(signal-proceed-case ((var1 var2 ...) arg1 arg2 ...)
   (proceed-type-1 body1...)
   (proceed-type-2 body2...)
   ...)
```

The first thing this form does is to call **signal**, evaluating each *arg* form to pass as an argument to **signal**. In addition to the arguments you supply, **signal-proceed-case** also specifies the **:proceed-types** init option, which it builds based on the *proceed-type-i* clauses.

When **signal** returns, **signal-proceed-case** treats the first returned value as the symbol for a proceed type. It then picks a *proceed-type-i* clause to run, based on that value. It works in the style of **selectq**: each clause starts with a proceed type (a keyword symbol), or a list of proceed types, and the rest of the clause is a list of forms to be evaluated. **signal-proceed-case** returns the values produced by the last form.

var1, *var2*, and so on, are bound to successive values returned from **signal** for use in the body of the *proceed-type-i* clause selected.

One *proceed-type-i* can be **nil**. If **signal** returns **nil**, meaning that the condition was not handled, **signal-proceed-case** runs the **nil** clause if one exists, or simply returns **nil** itself if no **nil** clause exists. Unlike **selectq**, no otherwise clause is available for **signal-proceed-case**.

The value passed as the **:proceed-types** option to **signal** lists the various proceed types in the same order as the clauses, so that the Debugger displays them in that order to the user and the RESUME command runs the first one.

69. Issues for Interactive Use

69.1 Tracing Conditions

trace-conditions *Variable*

The value of this variable is a condition or a list of conditions. It can also be **t**, meaning all conditions, or **nil**, meaning none.

If any condition is signalled that is built on the specified flavor (or flavors), the Debugger immediately assumes control, before any handlers are searched or called.

If the user proceeds, by using RESUME, signalling continues as usual. This might in fact revert control to the Debugger again. This variable is provided for debugging purposes only. It lets you trace the signalling of any condition so that you can figure out what conditions are being signalled and by what function. You can set this variable to **error** to trace all error conditions, for example, or you can be more specific.

This variable replaces the **errset** variable from earlier releases.

69.2 Breakpoints

The functions **breakon** and **unbreakon** can be used to set breakpoints in a program. They use the encapsulation mechanism like **trace** and **advise** to force the function to signal a condition when it is called. See the section "Encapsulations", page 325.

breakon &optional *function-spec condition-form* *Function*

With no arguments, **breakon** returns a list of all functions with breakpoints set by **breakon**.

breakon sets a breakpoint for the *function-spec*. Whenever *function-spec* is called, the condition **sys:call-trap** is signalled, and the Debugger assumes control. At this point, you can inspect the state of the Lisp environment and the stack. Proceeding from the condition then causes the program to continue to run.

The first argument can be any function spec, so that you can trace methods and other functions not named by symbols. See the section "Function Specs", page 297.

condition-form can be used for making a conditional breakpoint. *condition-form* should be a Lisp form. It is evaluated when the function is

called. If it returns **nil**, the function call proceeds without signalling anything. *condition-form* arguments from multiple calls to **breakon** accumulate and are treated as an **or** condition. Thus, when any of the forms becomes true, the breakpoint "goes off". *condition-form* is evaluated in the dynamic environment of the function call. You can inspect the arguments of *function-spec* by looking at the variable **arglist**.

unbreakon &optional *function-spec condition-form* *Function*
Turns off a breakpoint set by **breakon**. If *function-spec* is not provided, all breakpoints set by **breakon** are turned off. If *condition-form* is provided, it turns off only that condition, leaving any others. If *condition-form* is not provided, the entire breakpoint is turned off for that function.

Calling a function for which a breakpoint is set signals a condition with the following message:

```
Break on entry to function name
```

It provides a **:no-action** proceed type, which allows the function entry to proceed. The "trap on exit" bit is set in the stack frame of the function call, so that when the function returns or is thrown through another condition is signalled. Similarly, the "Break on exit from marked frame" message and the **:no-action** proceed type are provided, allowing the function return to proceed.

69.3 Debugger Bug Reports

The c-M command in the Debugger sends a bug report, creating a new process and running the **bug** function in that process. By default, the first argument to **bug** is the symbol **lispm**, so that the report is sent to the **BUG-LISPM** mailing list. Also by default, the mail-sending text buffer initially contains a standard set of information dumped by the Debugger. You can control this behavior for your own condition flavors. You can control the mailing list to which the bug report is sent by defining your own primary method for the following message.

:bug-report-recipient-system *Message*
This message is sent by the c-M command in the Debugger to find the mailing list to which to send the bug report mail. The default method (the one in the **condition** flavor) returns **lispm**, and this is passed as the first argument to the **bug** function.

You can control the initial contents of the mail-sending buffer by altering the handling of the following message, either by providing your own primary method to replace the default message, or by defining a **:before** or **:after** daemon to add your own specialized information before or after the default text.

:bug-report-description *stream number* *Message*

This message is sent by the c-M command in the Debugger to print out the text that is the initial contents of the mail-sending buffer. The handler should simply print whatever information it considers appropriate onto *stream*. *number* is the numeric argument given to c-M. The Debugger interprets *number* as the number of frames from the backtrace to include in the initial mail buffer. A *number* of **nil** means all frames.

69.4 Debugger Special Commands

When the Debugger assumes control because an error condition was signalled and not handled, it offers the user various ways to proceed or to restart. Sometimes you want to offer the user other kinds of options. In the system, the most common example of this occurs when you forget to type a package prefix. It signals a **sys:unbound-symbol** error and offers to let you use the symbol from the right package instead. This is neither a proceed type nor a restart-handler; it is a Debugger special command.

You can add one or more special commands to any condition flavor. For any particular instance, you can control whether to offer the special command. For example, the package-guessing service is not offered unless some other symbol with the same print name exists in a different package. Special commands are called only by the Debugger; **condition-bind** handler functions never see them.

Special commands provide the same kind of functionality that a **condition-bind** handler does. There is no reason, for example, that the package-prefix service could not have been provided by **condition-bind**. It is only a matter of convenience to have it in a special command.

To add special commands to your condition flavor, you must mix in the flavor **dbg:special-commands-mixin**, which provides both the instance variable **dbg:special-commands** and several method combinations. Each special command to a particular flavor is identified by a keyword symbol, just the same way that proceed types are identified. You can then create handlers for any of the following messages:

:special-command *command-type* *Message*

:special-command is sent when the user invokes the special command. It uses **:case** method-combination and dispatches on the name of the special command. No arguments are passed. The syntax is:

```
(defmethod (my-flavor :case :special-command my-command-keyword) ()
  "documentation"
  body...)
```

Any communication with the user should take place over the **query-io**

stream. The method can return **nil** to return control to the Debugger or it can return the same thing that any of the **:proceed** methods would have returned in order to proceed in that manner.

:document-special-command *command-type stream* *Message*

:document-special-command prints the documentation of *command-type* onto *stream*. If you don't handle this message explicitly, the default handler uses the documentation string from the **:special-command** method. You can, however, handle this message in order to print a prompt string that has to be computed at run-time. This is analogous to **:document-proceed-type**. The syntax is:

```
(defmethod (my-flavor :case :document-special-command my-command-keyword)
           (stream)
  body...)
```

:initialize-special-commands *Message*

The Debugger sends **:initialize-special-commands** after it prints the error message. The methods are combined with **:progn** combination, so that each one can do some initialization. In particular, the methods for this message can remove items from the list **dbg:special-commands** in order to decide not to offer these special commands.

69.5 Special Keys

The system normally handles the ABORT and SUSPEND keys so that ABORT aborts what you are doing and SUSPEND enters a breakpoint. Without a CONTROL modifier, a keystroke command takes effect only when the process reads the character from the keyboard; with the CONTROL modifier, a keystroke command takes effect immediately. The META modifier means "do it more strongly"; M-ABORT resets the process entirely, and M-SUSPEND enters the Debugger instead of entering a simple read-eval-print loop.

A complete and accurate description of what these keys do requires a discussion of conditions and the Debugger.

With no CONTROL modifier, ABORT and SUSPEND are detected when your process tries to do input from the keyboard (typically by doing an input stream operation such as **:tyi** on a window). Therefore, if your process is computing or waiting for something else, the effects of the keystrokes are deferred until your process tries to do input.

With a CONTROL modifier, ABORT and SUSPEND are intercepted immediately by the Keyboard Process, which sends your process an **:interrupt** message. Thus, it performs the specified function immediately, even if it is computing or waiting.

ABORT — Prints the following string on the **terminal-io** stream, unless it suspects that output on that stream might not work.

```
[Abort]
```

It then signals **sys:abort**, which is a simple condition. Programs can set up bound handlers for **sys:abort**, although most do not. Many programs set up restart handlers for **sys:abort**; most interactive programs have such a handler in their command loops. Therefore, ABORT usually restarts your program at the innermost command loop. Inside the Debugger, ABORT has a special meaning.

m-ABORT — Prints the following string on the **terminal-io** stream, unless it suspects that output on that stream might not work.

```
[Abort all]
```

It then sends your process a **:reset** message, with the argument **:always**. This has nothing to do with condition signalling. It just resets the process completely, unwinding its entire stack. What the process does after that depends on what kind of process it is and how it was created: it might start over from its initial function, or it might disappear. See the section "Processes" in *Internals, Processes, and Storage Management.*

SUSPEND — Calls the **break** function with the argument **break**. This has nothing to do with condition signalling. See the special form **break** in *User's Guide to Symbolics Computers.*

m-SUSPEND — Causes the Debugger to assume control without signalling any condition. The Debugger normally expects to be invoked because of some condition object, though, which it needs to interact properly with proceeding and restarting. Therefore, a condition object of flavor **break** is created in order to give the Debugger something to work with. **break** is not an error flavor; it is built on **condition**. It has no proceed types, but RESUME in the Debugger causes the Debugger to return and the process to resume what it was doing.

Several techniques are available for overriding the standard operation of ABORT and SUSPEND when they are being used with modifier keys.

- For using these keys with the CONTROL modifier, use the asynchronous character facility. See the section "Asynchronous Characters" in *Programming the User Interface.*

- Defining your own hook function and binding **tv:kbd-tyi-hook** to it also overrides the interception of these characters with no CONTROL modifier. See the section "Windows as Input Streams" in *Programming the User Interface.*

At the Debugger command loop, ABORT is the same as the Debugger c-Z command. It throws directly to the innermost restart handler that is appropriate for either the current error or the **sys:abort** condition.

When the Debugger assumes control, it displays a list of commands appropriate to the current condition, along with key assignments for each. Proceed types come first, followed by special commands, followed by restart handlers. One alphabetic key with the SUPER modifier is assigned to each command on the list. In addition, ABORT is always assigned to the innermost restart handler that handles **sys:abort** or the condition that was signalled; RESUME is always assigned to the first proceed type in the **:proceed-types** list. See the section "Proceed Type Messages", page 519.

If RESUME is not otherwise used, it invokes the first error restart that does not handle **abort**. When you enter the Debugger with m-SUSPEND, RESUME resumes the process.

You can customize the Debugger, assigning certain keystrokes to certain proceed types or special commands, by setting these variables in your init file:

dbg:*proceed-type-special-keys* *Variable*

The value of this variable should be an alist associating proceed types with characters. When an error supplies any of these proceed types, the Debugger assigns that proceed type to the specified key. For example, this is the mechanism by which the **:store-new-value** proceed type is offered on the m-C keystroke.

dbg:*special-command-special-keys* *Variable*

The value of this variable should be an alist associating names of special commands with characters. When an error supplies any of these special commands, the Debugger assigns that special command to the specified key. For example, this is the mechanism by which the **:package-dwim** special command is offered on the c-sh-P keystroke.

70. Condition Flavors Reference

A condition object is an instance of any flavor built out of the **condition** flavor. An error object is an instance of any flavor built out of the **error** flavor. The **error** flavor is built out of the **dbg:debugger-condition** flavor, which is built out of the **condition** flavor. Thus, all error objects are also condition objects.

Every flavor of condition that is instantiated must handle the **:report** message. (Flavors that just define sets of conditions need not handle it). This message takes a stream as its argument and prints out a textual message describing the condition on that stream. The printed representation of a condition object is like the default printed representation of any instance when slashifying is turned on. However, when slashifying is turned off (by **princ** or the **~A format** directive), the printed representation of a condition object is its **:report** message. Example:

```
(condition-case (co)
    (open "f:>a>b.c")
  (fs:file-not-found
    (prin1 co)))    prints out  #<QFILE-NOT-FOUND 33712233>

(condition-case (co)
    (open "f:>a>b.c")
  (fs:file-not-found
    (princ co)))    prints out  The file was not found
                                For F:>a>b.c
```

70.1 Messages and Init Options

These messages can be sent to any condition object. They are handled by the basic **condition** flavor, on which all condition objects are built. Some particular condition flavors handle other messages; those are documented along with the particular condition flavors in another section. See the section "Standard Conditions", page 531.

:document-proceed-type *proceed-type stream* of **condition** *Method*

Prints out a description of what it means to proceed, using the given *proceed-type*, from this condition, on *stream*. This is used mainly by the Debugger to create its prompt messages. Phrase such a message as an imperative sentence, without any leading or trailing **return** characters. This sentence is for the human users of the machine who read this when they have just been dumped unexpectedly into the Debugger. It should be composed so that it makes sense to a person to issue that sentence as a command to the system.

:proceed-type-p *proceed-type* of **condition** *Method*

Returns **t** if *proceed-type* is one of the valid proceed types of this condition object. Otherwise, returns **nil**.

:proceed-types of **condition** *Method*

Returns a list of all the valid proceed types for this condition.

:set-proceed-types *new-proceed-types* of **condition** *Method*

Sets the list of valid proceed types for this condition to *new-proceed-types*.

:proceed-types *proceed-types* (for **condition**) *Init Option*

Defines the set of proceed types to be handled by this instance. *proceed-types* is a list of proceed types (symbols); it must be a subset of the set of proceed types understood by this flavor. If this option is omitted, the instance is able to handle all of the proceed types understood by this flavor in general, but by passing this option explicitly, a subset of acceptable proceed types can be established. This is used by **signal-proceed-case**.

If only one way to proceed exists, *proceed-types* can be a single symbol instead of a list.

If you pass a symbol that is not an understood proceed type, it is ignored. It does not signal an error because the proceed type might become understood later when a new **defmethod** is evaluated; if not, the problem is caught later.

The order in which the proceed types occur in the list controls the order in which the Debugger displays them in its list. Sometimes you might want to select an order that makes more sense for the user, although usually this is not important. The most important thing is that the RESUME command in the Debugger is assigned to the first proceed type in the list.

:special-commands of **condition** *Method*

Returns a list of all Debugger special commands for this condition. See the section "Debugger Special Commands", page 525.

:special-command-p *command-type* of **condition** *Method*

Returns **t** if *command-type* is a valid Debugger special command for this condition object; otherwise, returns **nil**.

:report *stream* of **condition** *Method*

Prints the text message associated with this object onto *stream*. The **condition** flavor does not support this itself, but it is a required message, and any flavor built on **condition** that is instantiated must support this message.

:report-string of **condition** *Method*

Returns a string containing the report message associated with this object. It works by sending **:report** to the object.

70.2 Standard Conditions

This section presents the standard condition flavors provided by the system. Some of these flavors are the flavors of actual condition objects that get instantiated in response to certain conditions. Others never actually get instantiated, but are used to build other flavors.

In some cases, the flavor that the system uses to signal an error is not exactly the one listed here, but rather a flavor built on the one listed here. This often comes up when the same error can be reported by different programs that implement a generic protocol. For example, the condition signalled by a remote file-system stream when a file is not found is different from the one signalled by a local file-system stream; however, only the generic **fs:file-not-found** condition should ever be handled by programs, so that a program works regardless of what kind of file-system stream it is using. The exact flavors signalled by each file system are considered to be internal system names, subject to change without notice and not documented herein.

Do not look at system source code to figure out the names of error flavors without being careful to choose the right level of flavor! Furthermore, take care to choose a flavor that can be instantiated if you try to signal a system-defined condition. For example, you can not signal a condition object of type **fs:file-not-found** because this is really a set of errors and this flavor does not handle the **:report** message. If you were to implement your own file system and wanted to signal an error when a file cannot be found, it should probably have its own flavor built on **fs:file-not-found** and other flavors.

Choosing the appropriate condition to handle is a difficult problem. In general you do not want to choose a condition on the basis of the apparent semantics of its name. Rather you should choose it according to the appropriate level of the condition flavor hierarchy. This holds particularly for file-related errors. See the section "File-system Errors", page 544.

70.2.1 Fundamental Conditions

These conditions are basic to the functionality of the condition mechanism, rather than having anything to do with particular system errors.

condition *Flavor*

This is the basic flavor on which all condition objects are built.

dbg:debugger-condition *Flavor*

This flavor is built on **condition**. It is used for entering the Debugger without necessarily classifying the event as an error. This is intended primarily for system use; users should normally build on **error** instead.

error *Flavor*

This flavor is built on **dbg:debugger-condition**. All flavors that represent errors, as opposed to debugger conditions or simple conditions, are built on this flavor.

ferror *Flavor*

This is a simple error flavor for the **ferror** function. Use it when you do not want to invent a new error flavor for a certain condition. Its only state information is an error message, normally created by the call to the **ferror** function. It has two gettable and initable instance variables **format-string** and **format-args**. The **format** function is applied to these values to produce the **:report** message.

dbg:proceedable-ferror *Flavor*

This is a simple error flavor for the **fsignal** function. Use it when you do not want to invent a new error flavor for a certain condition, but you want the condition to be proceedable. Its only state information is an error message, created by the call to the **fsignal** function. Its only proceed type is **:no-action**. Proceeding in this way does nothing and causes **fsignal** (or **signal**) to return the symbol **:no-action**.

sys:no-action-mixin *Flavor*

This flavor can be mixed into any condition flavor to define a proceed type called **:no-action**. Proceeding in this way causes the computation to proceed as if no error check had occurred. The signaller might try the action again or might simply go on doing what it would have done. For example, **proceedable-ferror** is just **ferror** with this mixin.

sys:abort *Flavor*

The ABORT key on the keyboard was pressed. This is a simple condition. When **sys:abort** is signalled, control is thrown straight to a restart handler without entering the Debugger. See the section "Special Keys", page 526.

break *Flavor*

This is the flavor of the condition object passed to the Debugger as a result of the M-BREAK command. It is never actually signalled; rather, it is a convention to ensure that the Debugger always has a condition when it assumes control. This is based on **dbg:debugger-condition**. See the section "Special Keys", page 526.

70.2.2 Lisp Errors

This section describes the conditions signalled for basic Lisp errors. All of the conditions in this section are based on the **error** flavor unless otherwise indicated.

70.2.2.1 Base Flavor: sys:cell-contents-error

sys:cell-contents-error *Flavor*

All of the kinds of errors resulting from finding invalid contents in a cell of virtual memory are built on this flavor. This represents a set of errors including the various kinds of unbound-variable errors, the undefined-function error, and the bad data-type error.

Proceed type	*Action*
:new-value	Takes one argument, a new value to be used instead of the contents of the cell.
:store-new-value	Takes one argument, a new value to replace the contents of the cell.
:no-action	If you have intervened and stored something into the cell, the contents of the cell can be reread.

sys:unbound-variable *Flavor*

All of the kinds of errors resulting from unbound variables are built on this flavor. Because these are a subset of the "cell contents" errors, this flavor is built on **sys:cell-contents-error**. The **:variable-name** message returns the name of the variable that was unbound (a symbol).

sys:unbound-symbol *Flavor*

An unbound symbol (special variable) was evaluated. Some instances of this flavor provide the **:package-dwim** special command, which takes no arguments and asks whether you want to examine the value of various other symbols with the same print name in other packages. This proceed type is provided only if any such symbols exist in any other packages. (See also **dbg:*defer-package-dwim***.) This flavor is built on **sys:unbound-variable**. The proceed types from **sys:cell-contents-error** are provided, as is the **:variable-name** message from **sys:unbound-variable**.

sys:unbound-closure-variable *Flavor*

An unbound closure variable was evaluated. This flavor is built on **sys:unbound-variable**. The proceed types from **cell-contents-error** are provided, as is the **:variable-name** message from **sys:unbound-variable**.

sys:unbound-instance-variable *Flavor*

An unbound instance variable was evaluated. The **:instance** message returns the instance in which the unbound variable was found. The proceed types from **cell-contents-error** are provided, as is the **:variable-name** message from **sys:unbound-variable**.

sys:undefined-function *Flavor*

An undefined function was invoked; that is, an unbound function cell was referenced. This flavor is built on **sys:cell-contents-error** and provides all of its proceed types. The **:function-name** message returns the name of the function that was undefined (a function spec). This also provides **:package-dwim** service, like **sys:unbound-symbol**.

sys:bad-data-type-in-memory *Flavor*

A word with an invalid type code was read from memory. This flavor is built on **sys:cell-contents-error** and provides all of its proceed types.

Message	*Value returned*
:address	virtual address, as a locative pointer, from which the word was read
:data-type	numeric value of the data-type tag field of the word
:pointer	numeric value of the pointer field of the word

70.2.2.2 Location Errors

sys:unknown-setf-reference *Flavor*

setf did not find a **setf** property on the car of the form. The **:form** message returns the form that **setf** tried to operate on. This error is signalled when the **setf** macro is expanded.

sys:unknown-locf-reference *Flavor*

locf did not find a **locf** property on the car of the form. The **:form** message returns the form that **locf** tried to operate on. This error is signalled when the **locf** macro is expanded.

70.2.2.3 Base Flavor: sys:arithmetic-error

sys:arithmetic-error *Flavor*

Represents the set of all arithmetic errors. No condition objects of this flavor are actually created; any arithmetic error signals a more specific condition, built on this one. This flavor is provided to make it easy to handle any arithmetic error.

All arithmetic errors handle the **:operands** message. This returns a list of the operands in the operation that caused the error.

sys:divide-by-zero *Flavor*

Division by zero was attempted. This flavor is built on **sys:arithmetic-error**. The **:function** message returns the function that did the division.

sys:non-positive-log *Flavor*

Computation of the logarithm of a nonpositive number was attempted. This flavor is built on **sys:arithmetic-error**. The **:number** message returns the nonpositive number.

math:singular-matrix *Flavor*

A singular matrix was given to a matrix operation such as inversion, taking of the determinant, or computation of the LU decomposition. This flavor is built on **sys:arithmetic-error**.

70.2.2.4 Base Flavor: sys:floating-point-exception

sys:floating-point-exception and the condition flavors based on it are designed to support IEEE floating-point standards. See the section "Numbers", page 87. By default, all IEEE traps are enabled, except for the inexact-result trap. See the function **without-floating-underflow-traps**, page 94.

The trap handlers that signal these conditions from the system all cause pressing the RESUME key to mean "return the result that would have been returned if the trap had been disabled". For example, pressing RESUME on an overflow returns the appropriately signed infinity as the result. On an underflow it returns the denormalized (possibly zero) result.

sys:floating-point-exception *Flavor*

This is the base flavor for floating-point exceptional conditions. No condition objects of this flavor are actually created. This flavor is provided to make it easy to handle any floating-point exception. It is built on **sys:arithmetic-error**.

Message	*Value returned*
:operation	A symbol indicating the operation that caused the exception.
:operands	The list of operands to the operation.
:non-trap-result	The result that would have been returned if this trap had been disabled.
:saved-float-operation-status	The value of **sys:float-operation-status** at the time of the exception.

Proceed type	*Action*
:new-value	Takes one argument and uses this value as the result of the operation.

sys:float-divide-by-zero *Flavor*

A floating-point division by zero was attempted. This flavor is built on **sys:divide-by-zero** and **sys:floating-point-exception**.

sys:floating-exponent-overflow *Flavor*

Overflow of an exponent occurred during floating-point arithmetic. This flavor is built on **sys:floating-point-exception**. The **:function** message returns the function that got the overflow, if it is known, and **nil** if it is not known. The **:new-value** proceed type is provided with one argument, a floating-point number to use instead.

sys:floating-exponent-underflow *Flavor*

Underflow of an exponent occurred during floating-point arithmetic. This flavor is built on **sys:floating-point-exception**. The **:function** message returns the function that got the underflow, if it is known, and **nil** if it is not known. The **:use-zero** proceed type is provided with no arguments; a **0.0** is used instead.

sys:float-inexact-result *Flavor*

A floating-point result does not exactly represent the operation's result, due to the fixed precision of floating-point representation. Since most floating-point calculations are inexact, the inexact-result trap is disabled by default. This flavor is built on **sys:floating-point-exception**.

sys:float-invalid-operation *Flavor*

An invalid floating-point operation was attempted, such as dividing infinity by infinity. This flavor is built on **sys:floating-point-exception**.

sys:float-invalid-compare-operation *Flavor*

This is built on and is identical to **sys:float-invalid-operation**, except that it does not expect a numeric result. This flavor is raised for any arithmetic comparison ($<$, $>$, $\leq$, $\geq$, $=$, $\neq$) in which at least one of the operands is a NaN (IEEE not-a-number object).

For example:

```
(< (// 0.0 0.0) 0.0)
```

signals **sys:float-invalid-compare-operation** if you "proceed" from the invalid division by zero operation.

sys:negative-sqrt *Flavor*

Computing the square root of a negative number was attempted. This flavor is built on **sys:float-invalid-operation**.

sys:float-divide-zero-by-zero *Flavor*

A floating-point division of zero by zero was attempted. This flavor is built on **sys:float-invalid-operation** and **sys:float-divide-by-zero**. Most programs handle not this condition itself, but rather one of the component condition flavors.

70.2.2.5 Miscellaneous System Errors Not Categorized by Base Flavor

sys:end-of-file *Flavor*

A function doing input from a stream attempted to read past the end-of-file. The **:stream** message returns the stream.

sys:stream-closed *Flavor*

An operation that required a stream to be open was attempted on a closed

stream. **sys:stream-closed** accepts the following messages and has corresponding required init keywords:

:attempt Returns a string briefly describing the attempted action on **stream**, for example, "read from"

:stream Returns the stream

Example:

```
(error 'sys:stream-closed :attempt "write to" :stream self)
```

sys:wrong-stack-group-state *Flavor*

A stack group was in the wrong state to be resumed. The **:stack-group** message returns the stack group.

sys:draw-off-end-of-screen *Flavor*

Drawing graphics past the edge of the screen was attempted.

sys:draw-on-unprepared-sheet *Flavor*

A drawing primitive (such as **tv:%draw-line**) was used on a screen array not inside a **tv:prepare-sheet** special form. The **:sheet** message returns the sheet (window) that should have been prepared.

sys:bitblt-destination-too-small *Flavor*

The destination array of a **bitblt** was too small.

sys:bitblt-array-fractional-word-width *Flavor*

An array passed to **bitblt** does not have a first dimension that is a multiple of 32 bits. The **:array** message returns the array.

sys:write-in-read-only *Flavor*

Writing into a read-only portion of memory was attempted. The **:address** message returns the address at which the write was attempted.

sys:pdl-overflow *Flavor*

A stack (pdl) overflowed. The **:pdl-name** message returns the name of the stack (a string, such as "regular" or "special"). The **:grow-pdl** proceed type is provided, with no arguments; it increases the size of the stack. This is based on **dbg:debugger-condition**, not on **error**.

sys:area-overflow *Flavor*

This is signalled when the maximum-size (**:size** argument to **make-area**) is exceeded.

sys:virtual-memory-overflow *Flavor*

This is an irrecoverable error that is signalled when you run out of virtual memory.

sys:region-table-overflow *Flavor*

This is an irrecoverable error that is signalled when you run out of regions.

sys:cons-in-fixed-area *Flavor*

Allocation of storage from a fixed area of memory was attempted.

Message	*Value returned*
:area	name of the area
:region	region number

sys:throw-tag-not-seen *Flavor*

***throw** or **throw** was called, but no matching ***catch** or **catch** was found.

Message	*Value returned*
:tag	Catch-tag that was being thrown to.
:values	List of the values that were being thrown. If ***throw** was called, this is always a list of two elements, the value being thrown and the tag; if the **throw** special form of Common Lisp is used, the list can be of any length.

The **:new-tag** proceed type is provided with one argument, a new tag (a symbol) to try instead of the original.

sys:instance-variable-zero-referenced *Flavor*

Referencing instance variable 0 of an instance was attempted. This usually means that some method is referring to an instance variable that was deleted by a later evaluation of a **defflavor** form.

sys:instance-variable-pointer-out-of-range *Flavor*

Referencing an instance variable that does not exist was attempted. This usually means that some method is using an obsolete instance because a **defflavor** form got evaluated again and changed the flavor incompatibly.

sys:disk-error *Flavor*

An error was reported by the disk software or controller. The **:retry-disk-operation** proceed type is provided; it takes no arguments.

sys:redefinition *Flavor*

This is a simple condition rather than an error condition. It signals an attempt to redefine something by some other file than the one that originally defined it. The **:definition-type** argument specifies the kind of definition: it might be **defun** if the function cell is being defined, **defstruct** if a structure is being defined, and so on.

Message	*Value returned*
:name	symbol (or function spec) being redefined
:old-pathname	pathname that originally defined it
:new-pathname	pathname that is now trying to define it

Either pathname is **nil** if the definition was from inside the Lisp environment rather than from loading a file.

The following proceed types are provided:

Message	*Action*
:proceed	Redefinition should go ahead; in the future no warnings should be signalled for this pair of pathnames.
:inhibit-definition	Definition is not changed and execution proceeds.
:no-action	Function should be redefined as if no warning had occurred.

Note: if this condition is not handled, the action is controlled by the value of **fs:inhibit-fdefine-warnings**.

70.2.2.6 Function-calling Errors

sys:zero-args-to-select-method *Flavor*

A select method was applied to no arguments.

sys:too-few-arguments *Flavor*

A function was called with too few arguments.

Message	*Value returned*
:function	the function
:nargs	number of arguments supplied
:argument-list	list of the arguments passed

The **:additional-arguments** proceed type is provided with one argument, a list of additional argument values to which the function should be applied. If the error is proceeded, these new arguments are appended to the old arguments and the function is called with this new argument list.

sys:too-many-arguments *Flavor*

A function was called with too many arguments.

Message	*Value returned*
:function	the function
:nargs	number of arguments supplied
:argument-list	list of the arguments passed

The **:fewer-arguments** proceed type is provided with one argument, the new number of arguments with which the function should be called. In proceeding from this error, the function is called with the first *n* arguments only, where *n* is the number specified.

sys:wrong-type-argument *Flavor*

A function was called with at least one argument of invalid type.

Message	*Value returned*
:function	function with invalid argument(s)
:old-value	invalid value
:description	description of valid value
:arg-name	name of the argument
:arg-number	number of the argument (the first argument to a function is **0**, and so on) or **nil** if this does not apply

:description, **:arg-name**, and **:arg-number** are valid messages only when the error was signalled by **check-arg**, **check-arg-type**, or **argument-typecase**. Check to be sure that the message is valid before sending it (remember **:operation-handled-p**).

Proceed type	*Action*
:argument-value	Takes one argument, the new value to use for the argument.
:store-argument-value	Takes one argument, the new value to use and to store back into the local variable in which it was found.

70.2.2.7 Array Errors

dbg:bad-array-mixin *Flavor*

Errors involving an array that seems to be the wrong object include this flavor. It provides the **:array** message, which returns the array.

Proceed type	*Action*
:new-array	Takes one argument, an array to use instead of the old one.
:store-new-array	Takes one argument, an array to use instead of the old one and to store back into the local variable in which it was found.

sys:bad-array-type *Flavor*

A meaningless array type code was found in virtual memory, indicating a system bug. The **:type** message returns the numeric type code.

sys:array-has-no-leader *Flavor*

Using the leader of an array that has no array leader was attempted. The **:array** message returns the array. This includes the **bad-array-mixin** flavor.

sys:fill-pointer-not-fixnum *Flavor*

The fill pointer of an array was not a fixnum. The **:array** message returns the array. This includes the **bad-array-mixin** flavor.

sys:array-wrong-number-of-dimensions *Flavor*

The rank of the array provided was wrong; the array is in error and the subscripts are correct.

Message	*Value returned*
:dimensions-given	number of subscripts presented
:dimensions-expected	number that should have been given
:array	the array

This includes the **bad-array-mixin** flavor.

sys:array-wrong-number-of-subscripts *Flavor*

This assumes that the array is correct and that the user/application caused the error by providing the incorrect number of subscripts.

Message	*Value returned*
:array	The array
:subscripts-given	A list of the subscripts given
:number-of-subscripts-given	The number of subscripts given
:number-of-subscripts-expected	The rank of the array

The following example signals **sys:array-wrong-number-of-subscripts**:

```
(array-in-bounds-p some-3-dimensional-array 2 3)
```

sys:number-array-not-allowed *Flavor*

A number array (such as an **art-4b** or **art-16b**) was used in a context in which number arrays are not valid, such as an attempt to make a pointer to an element with **aloc** or **locf**. This includes the **bad-array-mixin** flavor.

sys:subscript-out-of-bounds *Flavor*

An attempt was made to reference an array using out-of-bounds subscripts, an out-of-bounds array leader element, or an out-of-bounds instance variable.

Message	*Value returned*
:object	the object (an array or instance) if it is known, and **nil** otherwise
:function	function that did the reference, or **nil** if it is not known
:subscript-used	the subscript that was actually used
:subscript-limit	the limit that it passed

The individual subscripts are reported for the **:subscript-used** and **:subscript-limit** messages. These values are fixnums; if a multidimensional array was used, they are computed products.

Proceed type	*Action*
:new-subscript	Takes an arbitrary number of arguments, the new subscripts for the array reference.
:store-new-subscript	Takes the same arguments as **:new-subscript** and stores them back into the local variables in which they were found.

70.2.2.8 Eval Errors

sys:invalid-function *Flavor*

The evaluator attempted to apply an object that is not a function or a symbol whose definition is an object that is not a function. The **:function** message returns the object that was applied. **sys:invalid-function** is signalled but is not proceedable.

sys:undefined-keyword-argument *Flavor*

The evaluator attempted to pass a keyword to a function that does not recognize that keyword.

Message	*Value returned*
:keyword	Unrecognized keyword
:value	The value passed with it

Proceed type	*Action*
:no-action	The keyword and its value are ignored.
:new-keyword	Specifies a new keyword to use instead. Its one argument is the new keyword.

sys:unclaimed-message *Flavor*

This flavor is built on **error**. The flavor system signals this error when it finds a message for which no method is available.

Message	*Value returned*
:object	the object
:message	the message-name
:arguments	the arguments of the message

The object can be an instance or a select method.

70.2.2.9 Interning Errors Based on sys:package-error

sys:package-error *Flavor*

All package-related error conditions are built on **sys:package-error**.

sys:package-not-found *Flavor*

A package-name lookup did not find any package by the specified name.

The **:name** message returns the name. The **:relative-to** message returns **nil** if only absolute names are being searched, or else the package whose relative names are also searched.

The **:no-action** proceed type can be used to try again. The **:new-name** proceed type can be used to specify a different name or package. The **:create-package** proceed type creates the package with default characteristics.

sys:package-locked *Flavor*

There was an attempt to intern a symbol in a locked package.

The **:symbol** message returns the symbol. The **:package** message returns the package.

The **:no-action** proceed type interns the symbol just as if the package had not been locked. Other proceed types are also available when interning the symbol would cause a name conflict.

70.2.2.10 Errors Involving Lisp Printed Representations

si:*print-object-error-message* *Variable*

Controls what happens when errors are signalled inside the Lisp printer. When **nil** (the default), the error is not handled. Otherwise, the value should be a string. If an error is signalled during the printing of an object, that string is sent to the stream instead of the printed representation of the object, and the printing function immediately returns to its caller. This applies to all functions that are entries to the Lisp printer, including **print**, **princ**, and **prinl**.

Example:

```
(let ((si:*print-object-error-message* "[Error printing object]"))
  (format t "foo: ~S, bar: ~S" foo bar))
```

This is useful because **bar** is printed even if the printing of **foo** causes an error.

sys:print-not-readable *Flavor*

The Lisp printer encountered an object that it cannot print in a way that the Lisp reader can understand. The printer signals this condition only if **si:print-readably** is not **nil** (it is normally **nil**). The **:object** message returns the object. The **:no-action** proceed type is provided; proceeding this way causes the object to be printed as if **si:print-readably** were **nil**.

sys:parse-error *Flavor*

This flavor is built on **error** and is the type of error caught by the input editor. This flavor accepts the init keyword **:correct-input**. If the value is **t**, which is the default, the input editor prints "Type RUBOUT to correct your input" and does not erase the message until a non-self-inserting character is typed. If the value is **nil**, no message is printed, and any typeout from the read function is erased immediately after the next character is typed. Syntax errors signalled by read functions should be built on top of this flavor.

parse-ferror *Flavor*

This flavor is built on **sys:parse-error** and **ferror**. It accepts the init keywords **:format-string** and **:format-args** as well as **:correct-input**. This flavor exists for read functions that do not have a special flavor of error defined for them.

sys:read-error *Flavor*

This flavor, built on **sys:parse-error**, includes errors encountered by the Lisp reader.

sys:read-premature-end-of-symbol *Flavor*

This is a new error flavor based on **sys:read-error**. It can be used for signalling when some read function finishes reading in the middle of a string that was supposed to contain a single expression.

Message	*Value returned*
:short-symbol	the symbol that was read
:original-string	the string that it was reading from when it finished in the middle

An example of the use of **sys:read-premature-end-of-symbol** is in **zwei:symbol-from-string**.

sys:read-end-of-file *Flavor*

The Lisp reader encountered an end-of-file while in the middle of a string or list. This flavor is built on **sys:read-error** and **sys:end-of-file**.

sys:read-list-end-of-file *Flavor*

The Lisp reader attempted to read past the end-of-file while it was in the middle of reading a list. This is built on **sys:read-end-of-file**. The **:list** message returns the list that was being built.

sys:read-string-end-of-file *Flavor*

The Lisp reader attempted to read past the end-of-file while it was in the middle of reading a string. This is built on **sys:read-end-of-file**. The **:string** message returns the string that was being built.

70.2.3 File-system Errors

The following condition flavors are part of the Symbolics Lisp Machine's generic file system interface. These flavors work for all file systems, whether local Lisp Machine file systems, remote Lisp Machine file systems (accessed over a network), or remote file systems of other kinds, such as UNIX or TOPS-20. All of them report errors uniformly.

Some of these condition flavors describe situations that can occur during any file system operation. These include not only the most basic flavors, such as **fs:file-request-failure** and **fs:data-error**, but also flavors such as **fs:file-not-found** and **fs:directory-not-found**. Other file system condition flavors describe failures related to specific file system operations, such as **fs:rename-failure**, and **fs:delete-failure**. Given all these choices, you have to determine what condition is appropriate to handle, for example in checking for success of a rename operation. Would **fs:rename-failure** include cases where, say, the directory of the file being renamed is not found?

The answer to this question is that you should handle **fs:file-operation-failure**. **fs:rename-failure** and all other conditions at that level are signalled only for errors that relate specifically to the semantics of the operation involved. If you cannot delete a file because the file is not found, **fs:file-not-found** would be signalled. Suppose you cannot delete the file because its "don't delete switch" is set, which is an error relating specifically to deletion. **fs:delete-failure** would be signalled. Therefore, since you cannot know whether a condition flavor related to an operation requested or some more general error is signalled, you usually want to handle one of the most general flavors of file system error.

Under normal conditions, you would bind only for **fs:file-request-failure** or **fs:file-operation-failure** rather than for the more specific condition flavors described in this section. Some guidelines for using the different classes of errors:

error Any error at all. It is not wise in general to attempt to handle this, because it catches program and operating system bugs as well as file-related bugs, thus "hiding" knowledge of the system problems from you.

fs:file-error Any file related error at all. This includes **fs:file-operation-failure** as well as **fs:file-request-failure**. Condition objects of flavor **fs:file-request-failure** usually indicate that the file system, host operating system, or network did not operate properly. If your program is attempting to handle file-related errors, it should not handle these: it is usually better to allow the program to enter the debugger. Thus it is very rare that one would want to handle **fs:file-error**.

fs:file-operation-failure
This includes almost all predictable file-related errors, whether they are related to the semantics of a specific operation, or are capable of occurring during many kinds of operations. Therefore, **fs:file-operation-failure** is usually the appropriate condition to handle.

Specific conditions It is appropriate and correct to handle specific conditions, like **fs:delete-failure**, if your program assigns specific meaning to (or has specific actions associated with) specific occurrences, such as a nonexistent directory or an attempt to delete a protected file. If you do not "care" about specific conditions, but you wish to handle predictable file-related errors, you should handle **fs:file-operation-failure**. You should *not* attempt to handle, say, **fs:delete-failure** to test for any error occurring during deletion; it does not mean that.

fs:file-error *Flavor*

This set includes errors encountered during file operations. This flavor is built on **error**.

Message	*Value returned*
:pathname	pathname that was being operated on or **nil**
:operation	name of the operation that was being done: this is a keyword symbol such as **:open**, **:close**, **:delete**, or **:change-properties**, and it might be **nil** if the signaller does not know what the operation was or if no specific operation was in progress

In a few cases, the **:retry-file-error** proceed type is provided, with no arguments; it retries the file system request. All flavors in this section accept these messages and might provide this proceed type.

fs:file-request-failure *Flavor*

This set includes all file-system errors in which the request did not manage to get to the file system.

fs:file-operation-failure *Flavor*

This set includes all file-system errors in which the request was delivered to the file system, and the file system decided that it was an error.

Note: every file-system error is either a request failure or an operation failure, and the rules given above explain the distinction. However, these rules are slightly unclear in some cases. If you want to be sure whether a certain error is a request failure or an operation failure, consult the detailed descriptions in the rest of this section.

70.2.3.1 Request Failures Based on fs:file-request-failure

fs:data-error *Flavor*

Bad data are in the file system. This might mean data errors detected by hardware or inconsistent data inside the file system. This flavor is built on **fs:file-request-failure**. The **:retry-file-operation** proceed type from **fs:file-error** is provided in some cases; send a **:proceed-types** message to find out.

fs:host-not-available *Flavor*

The file server or file system is intentionally denying service to users. This does *not* mean that the network connection failed; it means that the file system explicitly does not care to be available. This flavor is built on **fs:file-request-failure**.

fs:no-file-system *Flavor*

The file system is not available. For example, this host does not have any file system, or this host's file system cannot be initialized for some reason. This flavor is built on **fs:file-request-failure**.

fs:network-lossage *Flavor*

The file server had some sort of trouble trying to create a new data connection and was unable to do so. This flavor is built on **fs:file-request-failure**.

fs:not-enough-resources *Flavor*

Some resource was not available in sufficient supply. Retrying the operation might work if you wait for some other users to go away or if you close some of your own files. This flavor is built on **fs:file-request-failure**.

fs:unknown-operation *Flavor*

An unsupported file-system operation was attempted. This flavor is built on **fs:file-request-failure**.

70.2.3.2 Login Problems

Some login problems are correctable and some are not. To handle any correctable login problem, you set up a handler for **fs:login-required** rather than handling the individual conditions.

The correctable login problem conditions work in a special way. The Symbolics Lisp Machine's generic file system interface, in the user-end of the remote file protocol, always handles these errors with its own condition handler; it then signals the **fs:login-required** condition. Therefore to handle one of these problems, you set up a handler for **fs:login-required**. The condition object for the correctable login problem can be obtained from the condition object for **fs:login-required** by sending it an **:original-condition** message.

fs:login-problems *Flavor*

This set includes all problems encountered while trying to log in to the file system. Currently, none of these ever happen when you use a local file system. This flavor is built on **fs:file-request-failure**.

fs:correctable-login-problems *Flavor*

This set includes all correctable problems encountered while trying to log in to the foreign host. None of these ever happen when you use a local file system. This flavor is built on **fs:login-problems**.

fs:unknown-user *Flavor*

The specified user name is unknown at this host. The **:user-id** message returns the user name that was used. This flavor is built on **fs:correctable-login-problems**.

fs:invalid-password *Flavor*

The specified password was invalid. This flavor is built on **fs:correctable-login-problems**.

fs:not-logged-in *Flavor*

A file operation was attempted before logging in. Normally the file system interface always logs in before doing any operation, but this problem can come up in certain unusual cases in which logging in has been aborted. This flavor is built on **fs:correctable-login-problems**.

fs:login-required *Flavor*

This is a simple condition built on **condition**. It is signalled by the file-system interface whenever one of the correctable login problems happens.

Message	*Value returned*
(send (send error :access-path) :host)	the foreign host
:host-user-id	user name that would be the default for this host
:original-condition	condition object of the correctable login problem

The **:password** proceed type is provided with two arguments, a new user name and a new password, both of which should be strings. If the condition is not handled by any handler, the file system prompts the user for a new user name and password, using the **query-io** stream.

70.2.3.3 File Lookup

fs:file-lookup-error *Flavor*

This set includes all file-name lookup errors. This flavor is built on **fs:file-operation-failure**.

fs:file-not-found *Flavor*

The file was not found in the containing directory. The TOPS-20 and TENEX "no such file type" and "no such file version" errors also signal this condition. This flavor is built on **fs:file-lookup-error**.

fs:multiple-file-not-found *Flavor*

None of a number of possible files was found. This flavor is built on **fs:file-lookup-error**. It is signalled when **load** is not given a specific file type but cannot find either a source or a binary file to load.

The flavor allows three init keywords of its own. These are also the names of messages that return the following:

:operation	The operation that failed
:pathname	The pathname given to the operation
:pathnames	A list of pathnames that were sought unsuccessfully

The condition has a **:new-pathname** proceed type to prompt for a new pathname.

fs:directory-not-found *Flavor*

The directory of the file was not found or does not exist. This means that the containing directory was not found. If you are trying to open a directory, and the actual directory you are trying to open is not found, **fs:file-not-found** is signalled. This flavor is built on **fs:file-lookup-error**.

This flavor has two Debugger special commands: **:create-directory**, to create only the lowest level of directory, and **:create-directories-recursively**, to create any missing superiors as well.

Errors of this flavor support the **:directory-pathname** message. This message, which can be sent to any such error, returns (when possible) a "pathname as directory" for the actual directory which was not found.

Example:

Assume the directory x:>a>b exists, but has no inferiors. The following produces an error instance to which **:pathname** produces #<LMFS-PATHNAME x:>a>b>c>d>thing.lisp> and **:directory-pathname** produces #<LMFS-PATHNAME x:>a>b>c> >.

```
(open "x:>a>b>c>d>thing.lisp")
```

Note: Not all hosts and access media can provide this information (currently, only local LMFS and LMFS accessed via New File can). When a host does not return this information, **:directory-pathname** returns the same as **:pathname**, whose value is a pathname as directory for the best approximation known to the identity of the missing directory.

fs:device-not-found *Flavor*

The device of the file was not found or does not exist. This flavor is built on **fs:file-lookup-error**.

fs:link-target-not-found *Flavor*

The target of the link that was opened did not exist. This flavor is built on **fs:file-lookup-error**.

70.2.3.4 fs:access-error

fs:access-error *Flavor*

This set includes all protection-violation errors. This flavor is built on **fs:file-operation-failure**.

fs:incorrect-access-to-file *Flavor*

Incorrect access to the file in the directory was attempted. This flavor is built on **fs:access-error**.

fs:incorrect-access-to-directory *Flavor*

Incorrect access to some directory containing the file was attempted. This flavor is built on **fs:access-error**.

70.2.3.5 fs:invalid-pathname-syntax

fs:invalid-pathname-syntax *Flavor*

This set includes all invalid pathname syntax errors. This is not the same as **fs:parse-pathname-error**. (See the flavor **fs:parse-pathname-error**, page 553.) These errors occur when a successfully parsed pathname object is given to the file system, but something is wrong with it. See the specific conditions that follow. This flavor is built on **fs:file-operation-failure**.

fs:invalid-wildcard *Flavor*

The pathname is not a valid wildcard pathname. This flavor is built on **fs:invalid-pathname-syntax**.

fs:wildcard-not-allowed *Flavor*

A wildcard pathname was presented in a context that does not allow wildcards. This flavor is built on **fs:invalid-pathname-syntax**.

70.2.3.6 fs:wrong-kind-of-file

fs:wrong-kind-of-file *Flavor*

This set includes errors in which an invalid operation for a file, directory, or link was attempted.

fs:invalid-operation-for-link *Flavor*

The specified operation is invalid for links, and this pathname is the name of a link. This flavor is built on **fs:wrong-kind-of-file**.

fs:invalid-operation-for-directory *Flavor*

The specified operation is invalid for directories, and this pathname is the name of a directory. This flavor is built on **fs:wrong-kind-of-file**.

70.2.3.7 fs:creation-failure

fs:creation-failure *Flavor*

This set includes errors related to attempts to create a file, directory, or link. This flavor is built on **fs:file-operation-failure**.

fs:file-already-exists *Flavor*

A file of this name already exists. This flavor is built on **fs:creation-failure**.

fs:create-directory-failure *Flavor*

This set includes errors related to attempts to create a directory. This flavor is built on **fs:creation-failure**.

fs:directory-already-exists *Flavor*

A directory or file of this name already exists. This flavor is built on **fs:creation-directory-failure**.

fs:create-link-failure *Flavor*

This set includes errors related to attempts to create a link. This flavor is built on **fs:creation-failure**.

70.2.3.8 fs:rename-failure

fs:rename-failure *Flavor*

This set includes errors related to attempts to rename a file. The **:new-pathname** message returns the target pathname of the rename operation. This flavor is built on **fs:file-operation-failure**.

fs:rename-to-existing-file *Flavor*

The target name of a rename operation is the name of a file that already exists. This flavor is built on **fs:rename-failure**.

fs:rename-across-directories *Flavor*

The devices or directories of the initial and target pathnames are not the same, but on this file system they are required to be. This flavor is built on **fs:rename-failure**.

fs:rename-across-hosts *Flavor*

The hosts of the initial and target pathnames are not the same. This flavor is built on **fs:rename-failure**.

70.2.3.9 fs:change-property-failure

fs:change-property-failure *Flavor*

This set includes errors related to attempts to change properties of a file. This might mean that you tried to set a property that only the file system is allowed to set. For file systems without user-defined properties, it might mean that no such property exists. This flavor is built on **fs:file-operation-failure**.

fs:unknown-property *Flavor*

The property is unknown. This flavor is built on **fs:change-property-failure**.

fs:invalid-property-value *Flavor*

The new value provided for the property is invalid. This flavor is built on **fs:change-property-failure**.

70.2.3.10 fs:delete-failure

fs:delete-failure *Flavor*

This set includes errors related to attempts to delete a file. It applies to cases where the file server reports that it cannot delete a file. The exact events involved depend on what the host file server has received from the host. This flavor is built on **fs:file-operation-failure**.

fs:directory-not-empty *Flavor*

An invalid deletion of a nonempty directory was attempted. This flavor is built on **fs:delete-failure**.

fs:dont-delete-flag-set *Flavor*

Deleting a file with a "don't delete" flag was attempted. This flavor is built on **fs:delete-failure**.

70.2.3.11 Miscellaneous Operations Failures

fs:circular-link *Flavor*

The pathname is a link that eventually gets linked back to itself. This flavor is built on **fs:file-operation-failure**.

fs:unimplemented-option *Flavor*

This set includes errors in which an option to a command is not implemented. This flavor is built on **fs:file-operation-failure**.

fs:inconsistent-options *Flavor*

Some of the options given in this operation are inconsistent with others. This flavor is built on **fs:file-operation-failure**.

fs:invalid-byte-size *Flavor*

The value of the "byte size" option was not valid. This flavor is built on **fs:unimplemented-option**.

fs:no-more-room *Flavor*

The file system is out of room. This can mean any of several things:

- The entire file system might be full
- The particular volume that you are using might be full
- Your directory might be full
- You might have run out of your allocated quota
- Other system-dependent things

This flavor is built on **fs:file-operation-failure**. The **:retry-file-operation** proceed type from **fs:file-error** is sometimes provided.

fs:filepos-out-of-range *Flavor*

Setting the file pointer past the end-of-file position or to a negative position was attempted. This flavor is built on **fs:file-operation-failure**.

fs:file-locked *Flavor*

The file is locked. It cannot be accessed, possibly because it is in use by some other process. Different file systems can have this problem in various system-dependent ways. This flavor is built on **fs:file-operation-failure**.

fs:file-open-for-output *Flavor*

Opening a file that was already opened for output was attempted. This flavor is built on **fs:file-operation-failure**. Note: ITS, TOPS-20, and TENEX file servers do not use this condition; they signal **fs:file-locked** instead.

fs:not-available *Flavor*

The file or device exists but is not available. Typically, the disk pack is not mounted on a drive, the drive is broken, or the like. Probably operator intervention is required to fix the problem, but retrying the operation is likely to work after the problem is solved. This flavor is built on **fs:file-operation-failure**. Do not confuse this with **fs:host-not-available**.

70.2.4 Pathname Errors

fs:pathname-error *Flavor*

This set includes errors related to pathnames. This is built on the **error** flavor. The following flavors are built on this one.

fs:parse-pathname-error *Flavor*

A problem occurred in attempting to parse a pathname.

fs:invalid-pathname-component *Flavor*

Attempt to create a pathname with an invalid component.

Message	*Value returned*
:pathname	the pathname
:component-value	the invalid value
:component	the name of the component (a keyword symbol such as **:name** or **:directory**)
:component-description	a "pretty name" for the component (such as file name or directory)

The **:new-component** proceed type is defined with one argument, a component value to use instead.

At the time this is signalled, a pathname object with the invalid component has actually been created; this is what the **:pathname** message returns. The error is signalled just after the pathname object is created before it goes in the pathname hash table.

fs:unknown-pathname-host *Flavor*

The function **fs:get-pathname-host** was given a name that is not the name of any known file computer. The **:name** message returns the name (a string).

fs:undefined-logical-pathname-translation *Flavor*

A logical pathname was referenced but is not defined. The **:logical-pathname** message returns the logical pathname. This flavor has a **:define-directory** proceed type, which prompts for a physical pathname whose directory component is the translation of the logical directory on the given host.

70.2.5 Network Errors

sys:network-error *Flavor*

This set includes errors signalled by networks. These are generic network errors that are used uniformly for any supported networks. This flavor is built on **error**.

70.2.5.1 Local Network Problems

sys:local-network-error *Flavor*

This set includes network errors related to problems with one's own Symbolics Lisp Machine rather than with the network or the foreign host. This flavor is built on **sys:network-error**.

sys:network-resources-exhausted *Flavor*

The local network control program exhausted some resource; for example, its connection table is full. This flavor is built on **sys:local-network-error**.

sys:unknown-address *Flavor*

The network control program was given an address that is not understood. The **:address** message returns the address. This flavor is built on **sys:local-network-error**.

sys:unknown-host-name *Flavor*

The host parser (**si:parse-host**) was given a name that is not the name of any known host. The **:name** message returns the name. This flavor is built on **sys:local-network-error**.

70.2.5.2 Remote Network Problems

sys:remote-network-error *Flavor*

This set includes network errors related to problems with the network or the foreign host, rather than with one's own Symbolics Lisp Machine.

Message	*Value returned*
:foreign-host	the remote host
:connection	the connection or **nil** if no particular connection is involved

This flavor is built on **sys:network-error**.

sys:bad-connection-state *Flavor*

This set includes remote errors in which a connection enters a bad state. This flavor is built on **sys:remote-network-error**. It actually can happen due to local causes, however. In particular, if your Symbolics Lisp Machine stays inside a **without-interrupts** for a long time, the network control program might decide that a host is not answering periodic status requests and put its connections into a closed state.

sys:connection-error *Flavor*

This set includes remote errors that occur while trying to establish a new network connection. The **:contact-name** message to any error object in this set returns the contact name that you were trying to connect to. This flavor is built on **sys:remote-network-error**.

sys:host-not-responding *Flavor*

This set includes errors in which the host is not responding, whether during initial connection or in the middle of a connection. This flavor is built on **sys:remote-network-error**.

70.2.5.3 Connection Problems

sys:host-not-responding-during-connection *Flavor*

The network control program timed out while trying to establish a new connection to a host. The host might be down, or the network might be down. This flavor is built on **sys:host-not-responding** and **sys:connection-error**.

sys:host-stopped-responding *Flavor*

A host stopped responding during an established network connection. The host or the network might have crashed. This flavor is built on **sys:host-not-responding** and **sys:bad-connection-state**.

sys:connection-refused *Flavor*

The foreign host explicitly refused to accept the connection. The **:reason** message returns a text string from the foreign host containing its explanation, or **nil** if it had none. This flavor is built on **sys:connection-error**.

sys:connection-closed *Flavor*

An established connection became closed. The **:reason** message returns a text string from the foreign host containing its explanation, or **nil** if it had none. This flavor is built on **sys:bad-connection-state**.

sys:connection-closed-locally *Flavor*

The local host closed the connection and then tried to use it. This flavor is built on **sys:bad-connection-state**.

sys:connection-lost *Flavor*

The foreign host reported a problem with an established connection and that connection can no longer be used. The **:reason** message returns a text string from the foreign host containing its explanation, or **nil** if it had none. This flavor is built on **sys:bad-connection-state**.

sys:connection-no-more-data *Flavor*

No more data remain on this connection. This flavor is built on **sys:bad-connection-state**.

sys:network-stream-closed *Flavor*

This is a combination of **sys:network-error** and **sys:stream-closed** and is usually used as a base flavor by network implementations (for example, Chaos and TCP).

70.2.6 Tape Errors

tape:tape-error *Flavor*

This set includes all tape errors. This flavor is built on **error**.

tape:mount-error *Flavor*

A set of errors signalled because a tape could not be mounted. This includes problems such as no ring and drive not ready. Normally, **tape:make-stream** handles these errors and manages mount retry. This flavor is built on **tape:tape-error**.

tape:tape-device-error *Flavor*

A hardware data error, such as a parity error, controller error, or interface error, occurred. This flavor has **tape:tape-error** as a **:required-flavor**.

tape:end-of-tape *Flavor*

The end of the tape was encountered. When this happens on writing, the tape usually has a few more feet left, in which the program is expected to finish up and write two end-of-file marks. Normally, closing the stream does this automatically. Whether or not this error is ever seen on input depends on the tape controller. Most systems do not see the end of tape on reading, and rely on the software that wrote the tape to have cleanly terminated its data, with EOFs.

This flavor is built on **tape:tape-device-error** and **tape:tape-error**.

PART XII.

Packages

71. The Need for Packages

A Lisp program is a collection of function definitions. The functions are known by their names, and so each must have its own name to identify it. Clearly a programmer must not use the same name for two different functions.

The Symbolics Lisp Machine consists of a huge Lisp environment, in which many programs must coexist. All of the "operating system", the compiler, the editor, and a wide variety of programs are provided in the initial environment. Furthermore, every program that you use during his session must be loaded into the same environment. Each of these programs is composed of a group of functions; each function must have its own distinct name to avoid conflicts. For example, if the compiler had a function named **pull**, and you loaded a program that had its own function named **pull**, the compiler's **pull** would be redefined, probably breaking the compiler.

It would not really be possible to prevent these conflicts, since the programs are written by many different people who could never get together to hash out who gets the privilege of using a specific name such as **pull**.

Now, if two programs are to coexist in the Lisp world, each with its own function **pull**, then each program must have its own symbol named "**pull**", because one symbol cannot have two function definitions. The same reasoning applies to any other use of symbols to name things. Not only functions but variables, flavors, and many other things are named with symbols, and hence require isolation between symbols belonging to different programs.

A *package* is a mapping from names to symbols. When two programs are not closely related and hence are likely to have conflicts over the use of names, the programs can use separate packages to enable each program to have a different mapping from names to symbols. In the example above, the compiler can use a package that maps the name **pull** into a symbol whose function definition is the compiler's **pull** function. Your program can use a different package that maps the name **pull** into a different symbol whose function definition is your function. When your program is loaded, the compiler's **pull** function is not redefined, because it is attached to a symbol that is not affected by your program. The compiler does not break.

The word "package" is used to refer to a mapping from names to symbols because a number of related symbols are packaged together into a single entity. Since the substance of a program (such as its function definitions and variables) consists of attributes of symbols, a package also packages together the parts of a program. The package system allows the author of a group of closely related programs that should share the same symbols to define a single package for those programs.

It is important to understand the distinction between a name and a symbol. A name is a sequence of characters that appears on paper (or on a screen or in a file).

This is often called a *printed representation.* A symbol is a Lisp object inside the machine. You should keep in mind how Lisp reading and loading work. When a source file is read into the Symbolics Lisp Machine, or a compiled binary file is loaded in, the file itself obviously cannot contain Lisp objects; it contains printed representations of those objects. When the reader encounters a printed representation of a symbol, it uses a package to map that printed representation (a name) into the symbol itself. The loader does the same thing. The package system arranges to use the correct package whenever a file is read or loaded. (For a detailed explanation of this process: See the section "Specifying Packages in Programs", page 579.

Example of the Need for Packages

Suppose there are two programs named **chaos** and **arpa**, for handling the Chaosnet and Arpanet respectively. The author of each program wants to have a function called **get-packet**, which reads in a packet from the network. Also, each wants to have a function called **allocate-pbuf**, which allocates the packet buffer. Each "get" routine first allocates a packet buffer, and then reads bits into the buffer; therefore, each version of **get-packet** should call the respective version of **allocate-pbuf**.

Without the package system, the two programs could not coexist in the same Lisp environment. But the package system can be used to provide a separate space of names for each program. What is required is to define a package named **chaos** to contain the Chaosnet program, and another package **arpa** to hold the Arpanet program. When the Chaosnet program is read into the machine, the names it uses are translated into symbols via the **chaos** package. So when the Chaosnet program's **get-packet** refers to **allocate-pbuf**, the **allocate-pbuf** in the **chaos** package is found, which is the **allocate-pbuf** of the Chaosnet program — the right one. Similarly, the Arpanet program's **get-packet** would be read in using the **arpa** package and would refer to the Arpanet program's **allocate-pbuf**.

72. Symbols

72.1 The Value Cell of a Symbol

Each symbol has associated with it a *value cell*, which refers to one Lisp object. This object is called the symbol's *binding* or *value*, since it is what you get when you evaluate the symbol. The binding of symbols to values allows symbols to be used as the implementation of *variables* in programs.

The value cell can also be *empty*, referring to *no* Lisp object, in which case the symbol is said to be *unbound*. This is the initial state of a symbol when it is created. An attempt to evaluate an unbound symbol causes an error.

Symbols are often used as special variables. See the section "Namespace System Variables" in *Networks*. The symbols **nil** and **t** are always bound to themselves; they cannot 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.

set *symbol value* *Function*

set is the primitive for assignment of symbols. The *symbol*'s value is changed to *value*; *value* can be any Lisp object. **set** returns *value*. Example:

```
(set (cond ((eq a b) 'c)
           (t 'd))
     'foo)
```

either sets **c** to **foo** or sets **d** to **foo**.

set does not work on local variables.

set-globally *var value* *Function*

Works like **set** but sets the global value regardless of any bindings currently in effect.

set-globally operates on the *global value* of a special variable; it bypasses any bindings of the variable in the current stack group. It resides in the global package.

set-globally does not work on local variables.

symeval *sym* *Function*

symeval is the basic primitive for retrieving a symbol's value. **(symeval** *sym***)** returns *sym*'s current binding. This is the function called by **eval** when it is given a symbol to evaluate. If the symbol is unbound, then **symeval** causes an error.

symeval-globally *var* *Function*

Works like **symeval** but returns the global value regardless of any bindings currently in effect.

symeval-globally operates on the *global value* of a special variable; it bypasses any bindings of the variable in the current stack group. It resides in the global package.

symeval-globally does not work on local variables.

makunbound *sym* *Function*

makunbound causes *sym* to become unbound. Example:

```
(setq a 1)
a => 1
(makunbound 'a)
a => causes an error.
```

makunbound returns its argument.

makunbound-globally *var* *Function*

Works like **makunbound** but sets the global value regardless of any bindings currently in effect.

makunbound-globally operates on the *global value* of a special variable; it bypasses any bindings of the variable in the current stack group. It resides in the global package.

makunbound-globally does not work on local variables.

boundp *sym* *Function*

Returns **t** if *sym* is bound; otherwise, it returns **nil**.

variable-boundp *variable* *Special Form*

Returns **t** if the variable is bound and **nil** if the variable is not bound. *variable* should be any kind of variable (it is not evaluated): local, special, or instance. Note: local variables are always bound; if *variable* is local, the compiler issues a warning and replaces this form with **t**.

If **a** is a special variable, **(boundp 'a)** is the same as **(variable-boundp a)**.

variable-makunbound *variable* *Special Form*

Makes the variable be unbound and returns *variable*. *variable* should be any kind of variable (it is not evaluated): local, special, or instance. Note: since local variables are always bound, they cannot be made unbound; if *variable* is local, the compiler issues a warning.

If **a** is a special variable, **(makunbound 'a)** is the same as **(variable-makunbound a)**.

value-cell-location *sym* *Function*

This function is obsolete on local and instance variables; use **variable-location** instead.

value-cell-location returns a locative pointer to *sym*'s internal value cell. See the section "Locatives", page 83. It is preferable to write:

```
(locf (symeval sym))
```

instead of calling this function explicitly.

(value-cell-location 'a) is still useful when **a** is a special variable. It behaves slightly differently from the form **(variable-location a)**, in the case that **a** is a variable "closed over" by some closure. See the section "Dynamic Closures", page 331. **value-cell-location** returns a locative pointer to the internal value cell of the symbol (the one that holds the invisible pointer, which is the real value cell of the symbol), whereas **variable-location** returns a locative pointer to the external value cell of the symbol (the one pointed to by the invisible pointer, which holds the actual value of the variable).

variable-location *variable* *Special Form*

Returns a locative pointer to the memory cell that holds the value of the variable. *variable* can be any kind of variable (it is not evaluated): local, special, or instance.

variable-location should be used in almost all cases instead of **value-cell-location**; **value-cell-location** should only be used when referring to the internal value cell. For more information on internal value cells: See the section "What is a Dynamic Closure?", page 331.

You can also use **locf** on variables. **(locf a)** expands into **(variable-location a)**.

72.2 The Function Cell of a Symbol

Every symbol also has associated with it a *function cell*. The *function* cell is similar to the *value* cell; it refers to a Lisp object. When a function is referred to by name, that is, when a symbol is *applied* or appears as the car of a form to be evaluated, that symbol's function cell is used to find its *definition*, the functional object that is to be applied. For example, when evaluating **(+ 5 6)**, the evaluator looks in +'s function cell to find the definition of +, in this case a compiled function, to apply to 5 and 6.

Like the value cell, a function cell can be empty, and it can be bound or assigned. (However, to bind a function cell you must use the **bind** subprimitive.) The following functions are analogous to the similar value-cell-related functions. See the section "The Value Cell of a Symbol", page 561.

fsymeval *sym* *Function*

Returns *sym*'s definition, the contents of its function cell. If the function cell is empty, **fsymeval** causes an error.

fset *sym definition* *Function*

Stores *definition*, which can be any Lisp object, into *sym*'s function cell. It returns *definition*.

fboundp *sym* *Function*

Returns **nil** if *sym*'s function cell is empty, that is, *sym* is undefined. Otherwise it returns **t**.

fmakunbound *sym* *Function*

Causes *sym* to be undefined, that is, its function cell to be empty. It returns *sym*.

function-cell-location *sym* *Function*

Returns a locative pointer to *sym*'s function cell. See the section "Locatives", page 83. It is preferable to write:

```
(locf (fsymeval sym))
```

rather than calling this function explicitly.

Since functions are the basic building block of Lisp programs, the system provides a variety of facilities for dealing with functions. See the section "Functions", page 297.

72.3 The Property List of a Symbol

Every symbol has an associated property list. See the section "Property Lists", page 67. When a symbol is created, its property list is initially empty.

The Lisp language itself does not use a symbol's property list for anything. (This was not true in older Lisp implementations, where the print-name, value-cell, and function-cell of a symbol were kept on its property list.) However, various system programs use the property list to associate information with the symbol. For instance, the editor uses the property list of a symbol that is the name of a function to remember where it has the source code for that function, and the compiler uses the property list of a symbol which is the name of a special form to remember how to compile that special form.

Because of the existence of print-name, value, function, and package cells, none of the Maclisp system property names (**expr**, **fexpr**, **macro**, **array**, **subr**, **lsubr**, **fsubr**, and in former times **value** and **pname**) exist in Symbolics-Lisp.

plist *sym* *Function*

Returns the list that represents the property list of *sym*. Note that this is not the property list itself; you cannot do **get** on it.

setplist *sym list* *Function*

Sets the list that represents the property list of *sym* to *list*. Use **setplist** with extreme caution, since property lists sometimes contain internal system properties, which are used by many useful system functions. Also, it is inadvisable to have the property lists of two different symbols be **eq**, since the shared list structure causes unexpected effects on one symbol if **putprop** or **remprop** is done to the other.

property-cell-location *sym* *Function*

Returns a locative pointer to the location of *sym*'s property-list cell. This locative pointer is as valid as *sym* itself as a handle on *sym*'s property list.

72.4 The Print Name of a Symbol

Every symbol has an associated string called the *print-name*, or *pname* for short. This string is used as the external representation of the symbol: if the string is typed in to **read**, it is read as a reference to that symbol (if it is interned), and if the symbol is printed, **print** types out the print-name. More information about the *reader* and the *printer* can be found elsewhere. See the section "What the Reader Recognizes", page 20. See the section "What the Printer Produces", page 14.

get-pname *sym* *Function*

Returns the print-name of the symbol *sym*. Example:

```
(get-pname 'xyz) => "xyz"
```

samepnamep *sym1 sym2* *Function*

Returns **t** if the two symbols *sym1* and *sym2* have **string=** print-names, that is, if their printed representation is 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 for being in different packages. Examples:

```
(samepnamep 'xyz (maknam '(x y z)) => t

(samepnamep 'xyz (maknam '(w x y)) => nil

(samepnamep 'xyz "xyz") => t
```

This is the same function as **string=**. **samepnamep** is provided mainly so that you can write programs that work in Maclisp as well as Symbolics-Lisp; in new programs, you should just use **string=**.

72.5 The Package Cell of a Symbol

Every symbol has a *package cell* that is used, for interned symbols, to point to the package to which the symbol belongs. For an uninterned symbol, the package cell contains **nil**.

72.6 Creating Symbols

The functions in this section are primitives for creating symbols. However, before discussing them, it is important to point out that most symbols are created by a higher-level mechanism, namely the reader and the **intern** function. Nearly all symbols in Lisp are created by virtue of the reader's having seen a sequence of input characters that looked like the printed representation of a symbol. When the reader sees such a p.r., it calls **intern**, which looks up the sequence of characters in a big table and sees whether any symbol with this print-name already exists. If it does, **read** uses the existing symbol. If it does not exist, then **intern** creates a new symbol and puts it into the table, and **read** uses that new symbol.

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 Symbolics-Lisp, interned symbols are handled by the *package* system.

An *uninterned* symbol is a symbol used simply as a data object, with no special cataloging. An uninterned symbol prints the same as an interned symbol with the same print-name, but cannot be read back in.

The following functions can be used to create uninterned symbols explicitly.

make-symbol *pname* &optional *permanent-p* *Function*

Creates a new uninterned symbol whose print-name is the string *pname*. The value and function bindings are unbound and the property list is empty. If *permanent-p* is specified, it is assumed that the symbol is going to be interned and probably kept around forever; in this case it and its pname are put in the proper areas. If *permanent-p* is **nil** (the default), the symbol goes in the default area and the pname is not copied. *permanent-p* is mostly for the use of **intern** itself.

Examples:

```
(make-symbol "FOO") => FOO
(make-symbol "Foo") => |Foo|
```

Note that the symbol is *not* interned; it is simply created and returned.

If a symbol has lowercase characters in its print-name, the printer quotes the name using slashes or vertical bars. The vertical bars inhibit the Lisp reader's normal action, which is to convert a symbol to uppercase upon reading it. See the section "What the Printer Produces", page 14.

Example:

```
(setq a (make-symbol "Hello"))     ; => |Hello|
(princ a)                          ; prints out Hello
```

copysymbol *sym copy-props* *Function*

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 are the same as those of *sym*, and the property list of the new symbol is a copy of *sym*'s. If *copy-props* is **nil**, then the new symbol is unbound and undefined, and its property list is empty.

gensym &optional *x* *Function*

Invents a print-name, and creates a new symbol with that print-name. It returns the new, uninterned symbol.

The invented print-name is a character prefix (the value of **si:*gensym-prefix**) followed by the decimal representation of a number (the value of **si:*gensym-counter**), for example, "g0001". The number is increased by 1 every time **gensym** is called.

If the argument *x* is present and is a fixnum, then **si:*gensym-counter** is set to *x*. If *x* is a string or a symbol, then **si:*gensym-prefix** is set to the first character of the string or of the symbol's print-name. After handling the argument, **gensym** creates a symbol as it would with no argument. Examples:

```
if  (gensym) => g0007
then        (gensym 'foo) => f0008
        (gensym 32.) => f0032
        (gensym) => f0033
```

Note that the number is in decimal and always has four digits, and the prefix is always one character.

gensym is usually used to create a symbol that should not normally be seen by the user, and whose print-name is unimportant, except to allow easy distinction by eye between two such symbols. The optional argument is rarely supplied. The name comes from "generate symbol", and the symbols produced by it are often called "gensyms".

73. Sharing of Symbols Among Packages

How the Package System Allows Symbol Sharing

Besides keeping programs isolated by giving each program its own set of symbols, the package system must also provide controlled sharing of symbols among packages. It would not be adequate for each package's set of symbols to be completely disjoint from the symbols of every other package. For example, almost every package ought to include the whole Lisp language: **car**, **cdr**, **format**, and so on should be available to every program.

There is a critical tension between these two goals of the package system. On the one hand, we want to keep the packages isolated, to avoid the need to think about conflicts between programs when we choose names for things. On the other hand, we want to provide connections among packages so that the facilities of one program can be made available to other programs. All the complexity of the package system arises from this tension. Almost all of the package system's features exist to provide easy ways to control the sharing of symbols among packages, while avoiding accidental unwanted sharing of symbols. Unexpected sharing of a symbol between packages, when the authors of the programs in those packages expected to have private symbols of their own, is a *name conflict* and can cause programs to go awry. See the section "Package Name-conflict Errors", page 593.

Note that sharing symbols is not as simple as merely making the symbols defined by the Lisp language available in every package. A very important feature of the Symbolics Lisp Machine is *shared programs*; if one person writes a function to, say, print numbers in Roman numerals, any other function can call it to print Roman numerals. This contrasts sharply with many other systems, where many different programs have been written to accomplish the same thing.

For example, the routines to manipulate a robot arm might be a separate program, residing in its own package. A second program called **blocks** (the blocks world, of course) wants to manipulate the arm, so it would want to call functions from the arm package. This means that the blocks program must have a way to name those robot arm functions. One way to do this is to arrange for the name-to-symbol mapping of the blocks package to map the names of those functions into the same identical symbols as the name-to-symbol mapping of the arm package. These symbols would then be shared between the two packages.

This sharing must be done with great care. The symbols to be shared between the two packages constitute an interface between two modules. The names to be shared must be agreed upon by the authors of both programs, or at least known to them. They cannot simply make every symbol in the arm program available to the blocks program. Instead they must define some subset of the symbols used by the arm program as its *interface* and make only those symbols available. Typically each name

in the interface is carefully chosen (more carefully than names that are only used internally). The arm program comes with documentation listing the symbols that constitute its interface and describing what each is used for. This tells the author of the blocks program not only that a particular symbol is being used as the name of a function in the arms program (and thus cannot be used for a function elsewhere), but also what that function does (move the arm, for instance) when it is called.

The package system provides for several styles of interface between modules. For several examples of how the blocks program and the arm program might communicate: See the section "Examples of Symbol Sharing Among Packages", page 589.

An important aspect of the package system is that it makes it necessary to clarify the modularity of programs and the interfaces between them. The package system provides some tools to allow the interface to be explicitly defined and to check that everyone agrees on the interface.

73.1 External Symbols

The name-to-symbol mappings of a package are divided into two classes, *external* and *internal*. We refer to the symbols accessible via these mappings as being *external* and *internal* symbols of the package in question, though really it is the mappings that are different and not the symbols themselves. Within a given package, a name refers to one symbol or to none; if it does refer to a symbol, that symbol is either external or internal in that package, but not both.

External symbols are part of the package's public interface to other packages. These are supposed to be chosen with some care and are advertised to outside users of the package. Internal symbols are for internal use only, and these symbols are normally hidden from other packages. Most symbols are created as internal symbols; they become external only if they are explicitly *exported* from a package.

A symbol can appear in many packages. It can be external in one package and internal in another. It is valid for a symbol to be internal in more than one package, and for a symbol to be external in more than one package. A name can refer to different symbols in different packages. However, a symbol always has the same name no matter where it appears. This restriction is imposed both for conceptual simplicity and for ease of implementation.

73.2 Package Inheritance

Some name-to-symbol mappings are established by the package itself, while others are inherited from other packages. When package A inherits mappings from package B, package A is said to *use* package B. A symbol is said to be *accessible* in a package if its name maps to it in that package, whether directly or by inheritance. A symbol is said to be *present* in a package if its name maps to it directly (not by inheritance). If a symbol is accessible to a package, then it can be referenced by a program that is read into that package. Inheritance allows a package to be built up by combining symbols from a number of other packages.

Package inheritance interacts with the distinction between internal and external symbols. When one package uses another, it inherits only the external symbols of that package. This is necessary in order to provide a well-defined interface and avoid accidental name conflicts. The external symbols are the ones that are carefully chosen and advertised. If internal symbols were inherited, it would be hard to predict just which symbols were shared between packages.

A package can use any number of other packages; it inherits the external symbols of all of them. If two of these external symbols had the same name it would be unpredictable which one would be inherited, so this is considered to be a name-conflict error. Consequently the order of the used packages is immaterial and does not affect what symbols are accessible.

Only symbols that are present in a package can be external symbols of that package. However, the package system hides this restriction by copying an inherited mapping directly into a package if you request that the symbol be exported. Note: When package A uses package B, it inherits the external symbols of B. But these do not become external symbols of A, and are not inherited by package C that uses package A. A symbol becomes an external symbol of A only by an explicit request to export it from A.

A package can be made to use another package by the **:use** option to **defpackage** or **make-package** or by calling the **use-package** function.

73.3 The global Package

Almost every package should have the basic symbols of the Lisp language accessible to it. This includes:

- Symbols that are names of useful functions, such as **cdr**, **cons**, and **print**
- Symbols that are names of special forms, such as **cond** and **selectq**
- Symbols that are names of useful variables, such as **base**, **standard-output**, and *****

- Symbols that are names of useful constants, such as **lambda-list-keywords** and **%%kbd-control-meta**

- Symbols that are used by the language as symbols in their own right, such as **&optional**, **t**, **nil**, and **special**

Rather than providing an explicit interface between every program and the Lisp language, listing explicitly the particular symbols from the Lisp language that that program intends to use, it is more convenient to make all the Lisp symbols accessible. Unless otherwise specified, every package inherits from the **global** package. The external symbols of **global** are all the symbols of the Lisp language, including all the symbols documented without a colon (:) in their name. The **global** package has no internal symbols.

All programs share the global symbols, and cannot use them for private purposes. For example, the symbol **delete** is the name of a Lisp function and thus is in the **global** package. Even if a program does not use the **delete** function, it inherits the global symbol named **delete** and therefore cannot define its own function with that name to do something different. Furthermore, if two programs each want to use the symbol **delete** as a property list indicator, they can bump into each other because they do not have private symbols. You can use a mechanism called *shadowing* to declare that a private symbol is desired rather than inheriting the global symbol. See the section "Shadowing Symbols", page 574. You can also use the **where-is** function and the Where Is Symbol (m-X) editor command to determine whether a symbol is private or shared when writing a program.

Similar to the **global** package is the **system** package, which contains all the symbols that are part of the "operating system" interface or the machine architecture, but not regarded as part of the Lisp language. The **system** package is not inherited unless specifically requested.

Here is how package inheritance works in the example of the two network programs. (See the section "Example of the Need for Packages", page 560.) When the Chaosnet program is read into the Lisp world, the current package is the **chaos** package. Thus all of the names in the Chaosnet program are mapped into symbols by the **chaos** package. If there is a reference to some well-known global symbol such as **append**, it is found by inheritance from the **global** package, assuming no symbol by that name is present in the **chaos** package. If, however, there is a reference to a symbol that you created, a new symbol is created in the **chaos** package. Suppose the name **get-packet** is referenced for the first time. No symbol by this name is directly present in the **chaos** package, nor is such a symbol inherited from **global**. Therefore the reader (actually the **intern** function) creates a new symbol named **get-packet** and makes it present in the **chaos** package. When **get-packet** is referred to later in the Chaosnet program, that symbol is found.

When the Arpanet program is read in, the current package is **arpa** instead of **chaos**. When the Arpanet program refers to **append**, it gets the **global** one; that

is, it shares the same symbol that the Chaosnet program got. However, if it refers to **get-packet**, it does *not* get the same symbol the Chaosnet program got, because the **chaos** package is not being searched. Rather, the **arpa** and **global** packages are searched. A new symbol named **get-packet** is created and made present in the **arpa** package.

So what has happened is that there are two **get-packet**s: one for **chaos** and one for **arpa**. The two programs are loaded together without name conflicts.

73.4 Home Package of a Symbol

Every symbol has a *home package*. When a new symbol is created by the reader and made present in the current package, its home package is set to the current package. The home package of a symbol can be obtained with the **symbol-package** function.

Most symbols are present only in their home package; however, it is possible to make a symbol be present in any number of packages. Only one of those packages can be distinguished as the home package; normally this is the first package in which the symbol was present. The package system tries to ensure that a symbol *is* present in its home package. When a symbol is first created by the reader (actually by the **intern** function), it is guaranteed to be present in its home package. If the symbol is removed from its home package (by the **remob** function), the home package of the symbol is set to **nil**, even if the symbol is still present in some other package.

Some symbols are not present in any package; they are said to be *uninterned*. See the section "Mapping Names to Symbols", page 604. The **make-symbol** function can be used to create such a symbol. An uninterned symbol has no home package; the **symbol-package** function returns **nil** given such a symbol.

When a symbol is printed, for example, with **prinl**, the printer produces a printed representation that the reader turns back into the same symbol. If the symbol is not accessible to the current package, a qualified name is printed. See the section "Qualified Package Names", page 584. The symbol's home package is used as the prefix in the qualified name.

73.5 Importing and Exporting Symbols

A symbol can be made accessible to packages other than its home package in two ways, *importing* and *exporting*.

Any symbol can be made present in a package by *importing* it into that package. This is how a symbol can be present in more than one package at the same time. After importing a symbol into the current package, it can be referred to directly with

an unqualified name. Importing a symbol does not change its home package, and does not change its status in any other packages in which it is present.

When a symbol is imported, if another symbol with the same name is already accessible to the package, a name-conflict error is signalled. The *shadowing-import* operation is a combination of shadowing (described in the next section) and importing; it resolves a name conflict by getting rid of any existing symbol accessible to the package.

Any number of symbols can be *exported* from a package. This declares them to be *external* symbols of that package and makes them accessible in any other packages that *use* the first package. To use a package means to inherit its external symbols.

When a symbol is exported, the package system makes sure that no name conflict is caused in any of the packages that inherit the newly exported symbol.

A symbol can be imported by using the **:import**, **:import-from**, or **:shadowing-import** option to **defpackage** and **make-package**, or by calling the **import** or **shadowing-import** function. A symbol can be exported by using the **:export** option to **defpackage** or **make-package**, or by calling the **export** function. See the section "Defining a Package", page 598. See the section "Functions That Import, Export, and Shadow Symbols", page 611.

73.6 Shadowing Symbols

You can avoid inheriting unwanted symbols by *shadowing* them. To shadow a symbol that would otherwise be inherited, you create a new symbol with the same name and make it present in the package. The new symbol is put on the package's list of shadowing symbols, to tell the package system that it is not an accident that there are two symbols with the same name. A shadowing symbol takes precedence over any other symbol of the same name that would otherwise be accessible to the package. Shadowing allows the creator of a package to avoid name conflicts that are anticipated in advance.

As an example of shadowing, suppose you want to define a function named **nth** that is different from the normal **nth** function. (Perhaps you want **nth** to be compatible with the Interlisp function of that name.) Simply writing **(defun nth ...)** in your program would redefine the system-provided **nth** function, probably breaking other programs that use it. (The system detects this and queries you before proceeding with the redefinition.)

The way to resolve this conflict is to put the program (call it snail) that needs the incompatible definition of **nth** in its own package and to make the **snail** package shadow the symbol **nth**.

Now there are two symbols named **nth**, so defining **snail**'s **nth** to be an Interlisp-compatible function does not affect the definition of the global **nth**. Inside the snail

program, the global symbol **nth** cannot be seen, which is why we say that it is shadowed. If some reason arises to refer to the global symbol **nth** inside the snail program, the qualified name **global:nth** can be used.

A shadowing symbol can be established by the **:shadow** or **:shadowing-import** option to **defpackage** or **make-package**, or by calling the **shadow** or **shadowing-import** function. See the section "Functions That Import, Export, and Shadow Symbols", page 611.

73.7 Introduction to Keywords

The Lisp reader is not context-sensitive; it reads the same printed representation as the same symbol regardless of whether the symbol is being used as the name of a function, the name of a variable, a quoted constant, a syntactic word in a special form, or anything else. The consistency and simplicity afforded by this lack of context sensitivity are very important to Lisp's interchangeability of programs and data, but they do cause a problem in connection with packages. If a certain function is to be shared between two packages, then the symbol that names that function has to be shared for all contexts, not just for functional context. This can accidentally cause a variable, or a property list indicator, or some other use of a symbol, to be shared between two packages when not desired. Consequently, it is important to minimize the number of symbols that are shared between packages, since every such symbol becomes a "reserved word" that cannot be used without thinking about the implications. Furthermore, the set of symbols shared among all the packages in the world is not legitimately user-extensible, because adding a new shared symbol could cause a name conflict between unrelated programs that use symbols by that name for their own private purposes.

On the other hand, there are many important applications for which the package system just gets in the way and one would really like to have *all* symbols shared between packages. Typically this occurs when symbols are used as objects in their own right, rather than just as names for things.

This dilemma is partially resolved by the introduction of *keywords* into the language. Keywords are a set of symbols that is disjoint from all other symbols and exist as a completely independent set of names. There is no separation of packages as far as keywords are concerned; all keywords are available to all packages and two distinct keywords cannot have the same name. Of course, a keyword can have the same name as one or more ordinary symbols. To distinguish keywords from ordinary symbols, the printed representation of a keyword starts with a colon (:) character.

Since keywords are disjoint from ordinary symbols, the sharing of keywords among all packages does not affect the separation of ordinary symbols into private symbols of each package. The set of keywords is user-extensible; simply reading the printed representation of a new keyword is enough to create it.

Keywords are implemented as symbols whose home package is the **keyword** package, which has the empty string as a nickname. See the section "Package Names", page 581. Hence the printed representation of a keyword, a symbol preceded by a colon, is actually just a qualified name. As a matter of style, keywords are never imported into other packages and the **keyword** package is never inherited (used) by another package.

As a syntactic convenience, every keyword is a constant that evaluates to itself (just like numbers and strings). This eliminates the need to write a lot of " ' " marks when calling a function that takes **&key** arguments, but makes it impossible to have a variable whose name is a keyword. However, there is no desire to use keywords as names of variables (or of functions), because the colon would look ugly. In fact, no syntactic words of the Lisp language are keywords. Names of special forms, the **otherwise** that goes at the end of a **selectq**, the **lambda** that identifies an interpreted function, names of declarations such as **special** and **arglist**, all are not keywords.

The only aspects of symbols significant to keywords are name and property list; otherwise, keywords could just as easily be some other data type. (Note that keywords are referred to as enumeration types in some other languages.)

Using Keywords

Keywords can be used as symbolic names for elements of a finite set. For example, when opening a file with the **open** function you must specify a direction. The various directions are named with keywords, such as **:input** and **:output**.

One of the most common uses of keywords is to name arguments to functions that take a large number of optional arguments and therefore are inconvenient to call with arguments identified positionally. Each argument is preceded by a keyword that tells the function how to use that argument. When the function is called, it compares each keyword that was passed to it against each of the keywords it knows, using **eq**.

Another common use for keywords is as names for messages that are passed to active objects such as instances. When an instance receives a message, it compares its first argument against all the message names it knows, using **eq**.

See the section "Generic Operations on Objects", page 421.

Since two distinct keywords cannot have the same name, keywords are not used for applications in which name conflicts can arise. For example, suppose a program stores data on the property lists of symbols. The data are internal to the program but the symbols can be global. An example of this would be a program-understanding program that puts some information about each Lisp function and special form on the symbol that names that function or special form. The indicator used should not be a keyword, because some other program might choose the same keyword to store its own internal data on the same symbol, causing a name conflict.

It is permissible, and in fact quite common, to use the same keyword for two different purposes when the two purposes are always separable by context. For instance, the use of keywords to name arguments to functions does not permit the possibility of a name conflict if you always know what function you are calling.

To see why keywords are used to name **&key** arguments, consider the function **make-array**, which takes one required argument followed by any number of keyword arguments. For example, the following specifies, after the first required argument, two options with names **:leader-length** and **:type** and values **10** and **art-string**.

```
(make-array 100 :leader-length 10 :type 'art-string)
```

The file containing **make-array**'s definition is in the **system-internals** package, but the function is accessible to everyone without the use of a qualified name because the symbol **make-array** is itself inherited from **global**. But all the keyword names, such as **type**, are short and should not have to exist in **global** where they would either cause name conflicts or use up all the "good" names by turning them into reserved words. However, if all callers of **make-array** had to specify the options using long-winded qualified names such as **system-internals:leader-length** and **system-internals:type** (or even **si:leader-length** and **si:type**) the point of making **make-array** global so that one can write **make-array** rather than **system-internals:make-array** would be lost. Furthermore, by rights one should not have to know about internal symbols of another package in order to use its documented external interface. By using keywords to name the arguments, we avoid this problem while not increasing the number of characters in the program, since we trade a "'" for a ":".

The data type names used with the **typep** function and the **typecase** and **check-arg-type** special forms are sometimes keywords and sometimes not keywords. The names of data types that are built into the machine, such as **:symbol**, **:list**, **:fixnum**, and **:compiled-function**, are keywords. On the other hand, the names of data types that are defined as flavors or structures, such as **package** or **tv:window**, are not keywords. This unfortunate anomaly exists for historical reasons and is removed by Common Lisp, where names of data types, like names of functions, are never keywords.

When in doubt as to whether or not a symbol of the language is supposed to be a keyword, check to see whether it is documented with a colon at the front of its name.

74. Specifying Packages in Programs

If you are an inexperienced user, you need never be aware of the existence of packages when writing programs. The **user** package is selected by default as the package for reading expressions typed at the Lisp Listener. Files are read in the **user** package if no package is specified. Since all the functions that users are likely to need are provided in the **global** package, which is used by **user**, they are all accessible. In the documentation, functions that are not in the **global** package are documented with colons in their names, so typing the name the way it is documented works. Keywords, of course, must be typed with a prefix colon, but since that is the way they are documented it is possible to regard the colon as just part of the name, not as anything having to do with packages.

The current package is the value of the variable **package**. The current package in the "selected" process is displayed in the status line. This allows you to tell how forms you type in are read.

If you are writing a program that you expect others to use, you should put it in some package other than **user**, so that its internal functions do not conflict with names other users use. For whatever reason, if you are loading your programs into packages other than **user**, you need to know about special constructs including **defpackage**, qualified names, and file attribute lists. See the section "Defining a Package", page 598. See the section "Qualified Package Names", page 584.

Obviously, every file must be loaded into the right package to serve its purpose. It might not be so obvious that every file must be compiled in the right package, but it is just as true. Any time the names of symbols appearing in the file must be converted to the actual symbols, the conversion must take place relative to a package.

The system usually decides which package to use for a file by looking at the file's *attribute list*. See the section "File Attribute Lists" in *Reference Guide to Streams, Files, and I/O*. The package can also be selected by **make-system**. A compiled file remembers the name of the package it was compiled in, and loads into the same package. In the absence of any of these specifications, the package defaults to the current value of **package**, which is usually the **user** package unless you change it.

The file attribute list of a character file is the line at the front of the file that looks something like:

```
;;; -*- Mode:Lisp; Package:System-Internals -*-
```

This specifies that the package whose name or nickname is **system-internals** is to be used. Alphabetic case does not matter in these specifications. Relative package names are not used, since there is no meaningful package to which the name could be relative. See the section "Relative Package Names", page 582.

If the package attribute contains parentheses, then the package is automatically created if it is not found. This is useful when a single file is in its own package, not shared with any other files, and no special options are required to set up that package. The valid forms of package attribute are:

-*- **Package:** *Name* -*-
: Signal an error if the package is not found, allowing you to load the package's definition from another file, specify the name of an existing package to use instead, or create the package with default characteristics.

-*- **Package:** (*Name*) -*-
: If the package is not found, create it with the specified name and default characteristics. It uses **global** so that it inherits the Lisp language symbols.

-*- **Package:** (*Name use*) -*-
: If the package is not found, create it with the specified name and make it use *use*, which can be the name of a package or a list of names of packages.

-*- **Package:** (*Name use size*) -*-
: If the package is not found, create it with the specified name and make it use *use*, which can be the name of a package or a list of names of packages. *size* is a decimal number, the number of symbols that expected to be present in the package.

-*- **Package:** (*Name keyword value keyword value...*) -*-
: If the package is not found, create it with the specified name. The rest of the list supplies the keyword arguments to **make-package**. In the event of an ambiguity between this form and the previous one, the previous one is preferred. You can avoid ambiguity by specifying more than one keyword.

Binary files have similar file attribute lists. The compiler always puts in a **:package** attribute to cause the binary file to be loaded into the same package it was compiled in, unless this attribute is overridden by arguments to **load**.

75. Package Names

75.1 Introduction to Package Names

Each package has a name and perhaps some nicknames. These are assigned when the package is created, though they can be changed later. A package's name should be something long and self-explanatory like **editor**; there might be a nickname that is shorter and easier to type, like **ed**. Typically the name of a package is also the name of the program that resides in that package.

There is a single namespace for packages. Instead of setting up a second-level package system to isolate names of packages from each other, we simply say that package name conflicts are to be resolved by using long explanatory names. There are sufficiently few packages in the world that a mechanism to allow two packages to have the same name does not seem necessary. Note that for the most frequent use of package names, qualified names of symbols, name clashes between packages can be alleviated using relative names.

The syntax conventions for package names are the same as for symbols. When the reader sees a package name (as part of a qualified symbol name), alphabetic characters in the package name are converted to uppercase unless preceded by the "/" escape character or unless the package name is surrounded by "|" characters. When a package name is printed by the printer, if it does not consist of all uppercase alphabetics and non-delimiter characters, the "/" and "|" escape characters are used.

Package name lookup is currently case-insensitive, but it might be changed in the future to be case-sensitive. In any case you should not make two packages whose names differ only in alphabetic case.

Internally names of packages are strings, but the functions that require a package-name argument from the user accept either a symbol or a string. If you supply a symbol, its print-name is used, which has already undergone case conversion by the usual rules. If you supply a string, you must be careful to capitalize the string in the same way that the package's name is capitalized.

Note that **|Foo|:|Bar|** refers to a symbol whose name is **"Bar"** in a package whose name is **"Foo"**. By contrast, **|Foo:Bar|** refers to a 7-character symbol with a colon in its name, and is interned in the current package. Following the convention used in the documentation for symbols, we show package names as being in lowercase, even though the name string is really in uppercase.

Invisible Packages

In addition to normal packages, there can be *invisible* packages. An invisible package has a name, but it is not entered into the system's table that maps package names to packages. An invisible package cannot be referenced via a qualified name (unless you set up a relative name for it) and cannot be used in such contexts as the **:use** keyword to **defpackage** and **make-package** (unless you pass the package object itself, rather than its name). Invisible packages are useful if you simply want a package to use as a data structure, rather than as the package in which to write a program.

75.2 Relative Package Names

See the section "Introduction to Package Names", page 581. In addition to the absolute package names (and nicknames) described there, packages can have *relative* names. If **p** is a relative name for package B, relative to package A, then in contexts where relative names are allowed and A is the contextually relevant package the name **p** can be used instead of **b**. The relative name mapping *belongs to* package A and defines a new name (**p**) *for* package B. It is important not to confuse the package that the name is relative to with the package that is named.

Relative names are established with the **:relative-names** and **:relative-names-for-me** options to **defpackage** and **make-package**. You can also use the **pkg-add-relative-name** function to establish a relative name. The **pkg-delete-relative-name** function removes a relative name.

There are two important differences between relative names and absolute names: relative names are recognized only in certain contexts, and relative names can "shadow" absolute names. One application for relative names is to replace one package by another. Thus if a program residing in package A normally refers to the **thermodynamics** package, but for testing purposes we would like it to use the **phlogiston** package instead, we can give A a relative name mapping from the name **thermodynamics** to the **phlogiston** package. This relative name shadows the absolute name **thermodynamics**.

Another application for relative names is to ease the establishment of a family of mutually dependent packages. For example, if you have three packages named **algebra**, **rings**, and **polynomials**, these packages might refer to each other so frequently that you would like to use the nicknames **a**, **r**, and **p** rather than spelling out the full names each time. It would obviously be bad to use up these one-letter names in the system-wide space of package names; what if someone else has a program with two packages named **reasoning** and **truth-maintenance**, and would like to use the nicknames **r** and **t**? The solution to this name conflict is to make the abbreviated names be relative names defined in the **algebra**, **rings**, and **polynomials** packages. These abbreviations are seen by references emanating from

those packages, and there is no conflict with other abbreviations defined by other packages.

An extension of the shadowing application for relative names is to set up a complete family of packages parallel to the normal one, such as **experimental-global** and **experimental-user**. Within this family of packages you establish relative name mappings so that the usual names such as **global** and **user** can be used. Certain system utility programs work this way.

When package A uses package B, in addition to inheriting package B's external symbols, any relative name mappings established by package B are inherited. In the event of a name conflict between relative names defined directly by A and inherited relative names, the inherited name is ignored. The results if two relative name mappings inherited from two different packages conflict are unpredictable.

The Lisp system does not itself use relative names, so a freshly booted Lisp Machine contains no relative-name mappings.

Relative names are recognized in the following contexts:

- Qualified symbol names — The package name before the colon is relative to the package in which the symbol is being read (the value of the variable **package**). The printer prefers a relative package name to an absolute package name when it prints a qualified symbol name.

- Package references in package-manipulating functions — For example, the package names in the **:use** option to **defpackage** and in the first argument to **use-package** can be relative names. All such relative names are relative to the value of the variable **package**.

- Package arguments that default to the current package — The functions **intern, intern-local, intern-soft, intern-local-soft, remob, export, unexport, import, shadow, shadowing-import, use-package,** and **unuse-package** all take an optional second argument that defaults (except in the case of **remob**) to the current package. If supplied, this argument can be a package, an absolute name of a package, or a relative name of a package. All such relative names are relative to the value of the variable **package**.

Relative names are not recognized in "global" contexts, where there is no obvious contextual package to be relative to, such as:

- File attribute lists ("-*-" lines)

- Package names requested from you as part of error recovery, or in commands such as the Set Package (m-X) editor command.

- The **pkg-find-package** function (unless its optional third argument is specified).

- Package arguments to the **mapatoms**, **pkg-goto**, **describe-package**, and **pkg-kill** functions.

- Package specifiers in the **do-symbols**, **do-local-symbols**, and **do-external-symbols** special forms, and the **interned-symbols** and **local-interned-symbols loop** iteration paths.

When a package object is printed, if it has a relative name (relative to the value of **package**) that differs from its absolute name, both names are printed.

75.3 Qualified Package Names

75.3.1 Introduction to Qualified Package Names

Often it is desirable to refer to an external symbol in some package other than the current one. You do this through the use of a *qualified name*, consisting of a package name, then a colon, then the name of the symbol. This causes the symbol's name to be looked up in the specified package, rather than in the current one. For example, **editor:buffer** refers to the external symbol named **buffer** of the package named **editor**, regardless of whether there is a symbol named **buffer** in the current package. If there is no package named **editor**, or if no symbol named **buffer** is present in **editor** or if **buffer** is an internal symbol of **editor**, an error is signalled.

On rare occasions, you might need to refer to an *internal* symbol of some package other than the current one. It is invalid to do this with the colon qualifier, since accessing an internal symbol of some other package is usually a mistake. See the section "Specifying Internal and External Symbols in Packages", page 585. However, this operation is valid if you use "::" as the separator in place of the usual colon. If the reader sees **editor::buffer**, the effect is exactly the same as reading **buffer** with **package** temporarily rebound to the package whose name is **editor**. This special-purpose qualifier should be used with caution.

Qualified names are implemented in the Lisp reader by treating the colon character (:) specially. When the reader sees one or two colons preceded by the name of a package, it reads in the next Lisp object with **package** bound to that package. Note that the next Lisp object need not be a symbol; the printed representation of any Lisp object can follow a package prefix. If the object is a list, the effect is exactly as if every symbol in that list had been written as a qualified name, using the prefix that appears in front of the list. When a qualified name is among the elements of the list, the package name in the second package prefix is taken relative to the package selected by the first package prefix. The internal/external mode is controlled entirely by the innermost package prefix in effect.

75.3.2 Specifying Internal and External Symbols in Packages

To ease the transition for people whose programs are not yet organized according to the distinction between internal and external symbols, a package can be set up so that the ":" type of qualified name does the same thing as the "::" type. This is controlled by the package that appears before the colon, not by the package in which the whole expression is being read. To set this attribute of a package, use the **:colon-mode** keyword to **defpackage** and **make-package**. **:external** causes ":" to behave as described in another section, accessing only external symbols. See the section "Qualified Package Names as Interfaces", page 585. **:internal** causes ":" to behave the same as "::", accessing all symbols. Note that **:internal** mode is compatible with **:external** mode except in cases where an error would be signalled. The default mode is **:internal** and all predefined system packages are created with this mode. In Common Lisp the default mode is **:external**.

75.3.3 Qualified Package Names as Interfaces

See the section "How the Package System Allows Symbol Sharing", page 569. In the example of the blocks world and the robot arm, a program in the **blocks** package could call a function named **go-up** defined in the **arm** package by calling **arm:go-up**. **go-up** would be listed among the external symbols of **arm**, using **:export** in its **defpackage**, since it is part of the interface allowing the outside world to operate the arm. If the **blocks** program uses qualified names to refer to functions in the **arm** program, rather than sharing symbols as in the original example, then the possibility of name conflicts between the two programs is eliminated.

See the section "Example of the Need for Packages", page 560. Similarly, if the **chaos** program wanted to refer to the **arpa** program's **allocate-pbuf** function, it would simply call **arpa:allocate-pbuf**, assuming that function had been exported. If it was not exported (because **arpa** thought no one from the outside had any business calling it), the **chaos** program would call **arpa::allocate-pbuf**.

75.3.4 Qualified Names of Symbols

The printer uses qualified names when necessary. (The **princ** function, however, never prints qualified names for symbols.) The goal of the printer (for example, the **prin1** function) when printing a symbol is to produce a printed representation that the reader turns back into the same symbol. When a symbol that is accessible in the current package (the value of **package**) is printed, a qualified name is not used, regardless of whether the symbol is present in the package. This happens for one of three reasons: because this is its home package, is present because it was imported, or is not present but was inherited. When an inaccessible symbol is printed, a qualified name is used. The printer chooses whether to use ":" or "::" based on whether the symbol is internal or external and the **:colon-mode** of its home package. The qualified name used by the printer can be read back in and yields the

same symbol. If the inaccessible symbol were printed without qualification, the reader would translate that printed representation into a different symbol, probably an internal symbol of the current package.

The qualified name used by the printer is based on the symbol's home package, not on the path by which it was originally read (which of course cannot be known). Suppose **foo** is an internal symbol of package A, has been imported into package B, and has then been exported from package B. If it is printed while **package** is neither A nor B, nor a package that uses B, the name printed is **a::foo**, not **b:foo**, because **foo**'s home package is A. This is an unlikely case, of course.

In addition to the simplest printed representation of a symbol, its name standing by itself, there are four forms of qualified name for a symbol. These are accepted by the reader and are printed by the printer when necessary; except when printing an uninterned symbol, the printer prints some printed representation that yields the same symbol when read. The following table shows the four forms of qualified name, assuming that the **foo** package specifies **:colon-mode :external**. If **foo** specifies **:colon-mode :internal**, as is currently the default, the first and second forms are equivalent.

foo:bar	When read, looks up **bar** among the external symbols of the package named **foo**. Printed the when symbol **bar** is external in its home package **foo** and is not accessible in the current package.
foo::bar	When read, interprets **bar** as if **foo** were the current package. Printed when the symbol **bar** is internal in its home package **foo** and is not accessible in the current package.
:bar	When read, interprets **bar** as an external symbol in the **keyword** package. Printed when the home package of the symbol **bar** is **keyword**.
#:bar	When read, creates a new uninterned symbol named **bar**. Printed when the symbol named **bar** has no home package.

75.3.5 Multilevel Qualified Package Names

Due to shadowing by relative names, a given package might sometimes be inaccessible. In this case a multilevel qualified name, containing more than one package prefix, can be used.

Suppose packages **moe**, **larry**, **curly**, and **shemp** exist. For its own reasons, the **moe** package uses **curly** as a relative name for the **shemp** package. Thus, when the current package is **larry** the printed representation **curly:hair** designates a symbol in the **curly** package, but when the current package is **moe** the same printed representation designates a symbol in the **shemp** package.

If the **moe** package is current and the symbol **hair** in the **curly** package needs to be read or printed, the printed representation **curly:hair** cannot be used since it

refers to a different symbol. If **curly** had a nickname that is not also shadowed by a relative name it would be used, but suppose there is no nickname. In this case the only possible way to refer to that symbol is with a multilevel qualified name. **larry:curly:hair** would work, since the **larry:** escapes from the scope of **moe**'s relative name. The printer actually prefers to print **global:curly:hair** because of the way it searches for a usable qualified name.

76. Examples of Symbol Sharing Among Packages

See the section "How the Package System Allows Symbol Sharing", page 569.
Consider again the example of the robot arm in the blocks world. Two separate programs, written by different people, interact with each other in a single Lisp environment. The arm-control program resides in a package named **arm**, and the blocks-world program resides in a package named **blocks**. The operation of the two programs requires them to interact. For example, to move a block from one place to another the **blocks** program calls functions in the **arm** program with names like **raise-arm**, **move-arm**, and **grasp**. To find the edges of the table, the **arm** program accesses variables of the **blocks** program.

Communication between the two programs requires that they both know about certain objects. Usually these objects are the sort that have names (for example, functions or variables). The names are symbols. Thus each program must be able to name some symbols and to know that the other program is naming the same symbols.

Let us consider the case of the function **grasp** in the arm-control program, which the blocks-world program must call in order to pick up a block with the arm. The **grasp** function is named by the symbol **grasp** in the **arm** package. Assume that we are not going to use either of the mechanisms (keywords and the **global** package) that make symbols available to *all* packages; we only want **grasp** to be shared between the two specific packages that need it. There are basically three ways provided by the package system for a symbol to be known by two separate programs in two separate packages.

1. If the **blocks** package *imports* the symbol **grasp** from the **arm** package, then both packages map the name **grasp** into the same symbol. The **blocks** package could be defined by:

```
(defpackage blocks
       (:import-from arm grasp))
```

2. The **arm** package can *export* the symbol **grasp**, along with whatever other symbols constitute its interface to the outside world. If the **blocks** package *uses* the **arm** package, then both packages again map the name **grasp** into the same symbol. The package definitions would look like:

```
(defpackage arm
       (:export grasp move-arm raise-arm ...))

(defpackage blocks
       (:use arm global))
```

Note that the **blocks** package must explicitly mention that it is using the **global** package as well as the **arm** package, since it is not letting its **:use** clause default.

The difference between this method (the export method) and the first method (the import method) is that the list of symbols that is to constitute the interface is associated with the **arm** package, that is, the package that *provides* the interface, not the package that *uses* the interface.

3. In the third method, we do not have the two packages map the same name into the same symbol. Instead we use a different, longer name for the symbol in the blocks program than the name used by the arm program. This makes it clear, when reading the text of the blocks program, which symbol references are connected with the interface between the two programs. These longer names are called *qualified names*. Again, the **arm** package defines the interface:

```
(defpackage arm
       (:export grasp move-arm raise-arm ...))
```

A fragment of the blocks-world program might look like

```
(defun pick-up (block)
  (clear-top block)
  (arm:grasp (block-coordinates block :top))
  (arm:raise-arm))
```

arm:grasp and **arm:raise-arm** are qualified names. **pick-up**, **block**, **clear-top**, and **block-coordinates** are internal symbols of the blocks-world program. **defun** is inherited from the **global** package. **:top** is a keyword. Note that although the two programs do not use the same names to refer to the same symbol, the names they use are related in an obvious way, avoiding confusion. The package system makes no provision for the same symbol to be named by two completely arbitrary names.

77. Consistency Rules for Packages

Package-related bugs can be very subtle and confusing: the program is not using the same symbols as you think it is using. The package system is designed with a number of safety features to prevent most of the common bugs that would otherwise occur in normal use. This might seem overprotective, but experience with earlier package systems has shown that such safety features are needed.

In dealing with the package system, it is useful to keep in mind the following consistency rules, which remain in force as long as the value of **package** is not changed by you or your code:

- *Read-Read consistency:* Reading the same print name always gets you the same (**eq**) symbol.
- *Print-Read consistency:* An interned symbol always prints as a sequence of characters that, when read back in, yields the same (**eq**) symbol.
- *Print-Print consistency:* If two interned symbols are not **eq**, then their printed representations are not be the same sequence of characters.

These consistency rules remain true in spite of any amount of implicit interning caused by typing in Lisp forms, loading files, and so on. This has the important implication that results are reproducible regardless of the order of either loading files or typing in symbols. The rules can only be violated by explicit action: changing the value of **package**, forcing some action by continuing from an error, or calling a function that makes explicit modifications to the package structure (**remob**, for example).

To ensure that the consistency rules are obeyed, the system ensures that certain aspects of the package structure are chosen by conscious decision of the programmer, not by accidents such as which symbols happen to be typed in by a user. External symbols, the symbols that are shared between packages without being explicitly listed by the "accepting" package, must be explicitly listed by the "providing" package. No reference to a package can be made before it has been explicitly defined.

78. Package Name-conflict Errors

78.1 Introduction to Package Name-conflict Errors

A fundamental invariant of the package system is that within one package any particular name can refer to only one symbol. A *name conflict* is said to occur when more than one candidate symbol exists and it is not obvious which one to choose. If the system does not always choose the same way, the read-read consistency rule would be violated. For example, some programs or data might have been read in under a certain mapping of the name to a symbol. If the mapping changes to a different symbol, then additional programs or data are read, the two programs do not access the same symbol even though they use the same name. Even if the system did always choose the same way, a name conflict is likely to result in a different mapping from names to symbols than you expected, causing programs to execute incorrectly. Therefore, any time a name conflict occurs, an error is signalled. You can continue from the error and tell the package system how to resolve the conflict.

Note that if the same symbol is accessible to a package through more than one path, for instance as an external of more than one package, or both through inheritance and through direct presence in the package, there is no name conflict. Name conflicts only occur between distinct symbols with the same name.

See the section "Shadowing Symbols", page 574. As discussed there, the creator of a package can tell the system in advance how to resolve a name conflict through the use of *shadowing*. Every package has a list of shadowing symbols. A shadowing symbol takes precedence over any other symbol of the same name that would otherwise be accessible to the package. A name conflict involving a shadowing symbol is always resolved in favor of the shadowing symbol, without signalling an error (except for one exception involving **import** described below). The **:shadow** and **:shadowing-import** options to **defpackage** and **make-package** can be used to declare shadowing symbols. The functions **shadow** and **shadowing-import** can also be used.

78.2 Checking for Package Name-conflict Errors

Name conflicts are detected when they become possible, that is, when the package structure is altered. There is no need to check for name conflicts during every name lookup. The functions **use-package**, **import**, and **export** check for name conflicts.

Using a package makes the external symbols of the package being used accessible to

the using package; each of these symbols is checked for name conflicts with the symbols already accessible.

Importing a symbol adds it to the internals of a package, checking for a name conflict with an existing symbol either present in the package or accessible to it. **import** signals an error even if there is a name conflict with a shadowing symbol, because two explicit directives from you are inconsistent.

Exporting a symbol makes it accessible to all the packages that use the package from which the symbol is exported. All of these packages are checked for name conflicts. **(export** *s p***)** does **(intern-soft** *s q***)** for each package *q* in **(package-used-by-list** *p***)**. Note that in the usual case of exporting symbols only during the initial definition of a package, there are no users of the package yet and the name-conflict checking takes no time.

intern does not need to do any name-conflict checking, because it never creates a new symbol if there is already an accessible symbol with the name given.

Note that the function **intern-local** can create a new symbol with the same name as an already accessible symbol. Nevertheless, **intern-local** does not check for name conflicts. This function is considered to be a low-level primitive and indiscriminate use of it can cause undetected name conflicts. Use **import**, **shadow**, or **shadowing-import** for normal purposes.

shadow and **shadowing-import** never signal a name-conflict error, because by calling these functions the user has specified how any possible conflict is to be resolved. **shadow** does name-conflict checking to the extent that it checks whether a distinct existing symbol with the specified name is accessible, and if so whether it is directly present in the package or inherited; in the latter case a new symbol is created to shadow it. **shadowing-import** does name-conflict checking to the extent that it checks whether a distinct existing symbol with the same name is accessible; if so it is shadowed by the new symbol, which implies that it must be **remob**ed if it was directly present in the package.

unuse-package, **unexport**, and **remob** (when the symbol being **remob**ed is not a shadowing symbol) do not need to do any name-conflict checking, because they only remove symbols from a package; they do not make any new symbols accessible.

remob of a shadowing symbol can uncover a name conflict that had previously been resolved by the shadowing. If package A uses packages B and C, A contains a shadowing symbol **x**, and B and C each contain external symbols named **x**, then **remob**ing **x** from A reveals a name conflict between **b:x** and **c:x** if those two symbols are distinct. In this case **remob** signals an error.

78.3 Resolving Package Name-conflict Errors

Aborting from a name-conflict error leaves the original symbol accessible. Package functions always signal name-conflict errors before making any change to the package structure. Note: when multiple changes are to be made, for example when exporting a list of symbols, it is valid for each change to be processed separately, so that aborting from a name conflict caused by the second symbol in the list does not unexport the first symbol in the list. However, aborting from a name-conflict error caused by exporting a single symbol does not leave that symbol accessible to some packages and inaccessible to others; exporting appears as an atomic operation.

Continuing from a name-conflict error offers you a chance to resolve the name conflict in favor of either of the candidates. This can involve shadowing or **remob**ing. Another possibility that is offered to you is to merge together the conflicting symbols' values, function definitions, and property lists in the same way as **globalize**. This is useful when the conflicting symbols are not being used as objects, but only as names for functions (or variables, or flavors, for example). You are also offered the choice of simply skipping the particular package operation that would have caused a name conflict.

A name conflict in **use-package** between a symbol directly present in the using package and an external symbol of the used package can be resolved in favor of the first symbol by making it a shadowing symbol, or in favor of the second symbol by **remob**ing the first symbol from the using package. The latter resolution is dangerous if the symbol to be **remob**ed is an external symbol of the using package, since it ceases to be an external symbol.

A name conflict in **use-package** between two external symbols inherited by the using package from other packages can be resolved in favor of either symbol by importing it into the using package and making it a shadowing symbol.

A name conflict in **export** between the symbol being exported and a symbol already present in a package that would inherit the newly exported symbol can be resolved in favor of the exported symbol by **remob**ing the other one, or in favor of the already present symbol by making it a shadowing symbol.

A name conflict in **export** or **remob** due to a package inheriting two distinct symbols with the same name from two other packages can be resolved in favor of either symbol by importing it into the using package and making it a shadowing symbol, just as with **use-package**.

A name conflict in **import** between the symbol being imported and a symbol inherited from some other package can be resolved in favor of the symbol being imported by making it a shadowing symbol, or in favor of the symbol already accessible by not doing the **import**. A name conflict in **import** with a symbol already present in the package can be resolved by **remob**ing that symbol, or by not doing the **import**.

Good user-interface style dictates that **use-package** and **export**, which can cause many name conflicts simultaneously, first check for all of the name conflicts before presenting any of them to you. You can then choose to resolve all of them wholesale, or to resolve each of them individually, requiring considerable interaction but permitting different conflicts to be resolved different ways.

132. Package Functions, Special Forms, and Variables

Packages are represented as Lisp objects. A package is a structure that contains various fields and a hash table that maps from names to symbols. Most of the structure field accessor functions for packages are only used internally by the package system and are not documented.

The **typep** function with one argument returns the symbol **package** if given a package object. **(typep** *obj* **'package)** is a predicate that is true if *obj* is a package object.

Many of the functions that operate on packages accept either an actual package or the name of a package. A package name can be either a string or a symbol.

Many of the functions and variables associated with packages have names that begin with "**pkg-**". This naming convention is considered obsolescent and will eventually be phased out in favor of the Common Lisp-compatible naming convention that uses a prefix of "**package-**" on names that do not already contain the word **package**. Currently, however, only "**pkg-**" is valid.

132.1 The Current Package

package *Variable*

The value of **package** is the current package; many functions that take packages as optional arguments default to the value of **package**, including **intern** and related functions. The reader and the printer deal with printed representations that depend on the value of **package**. Hence the current package is part of the user interface and is displayed in the status line at the bottom of the screen.

It is often useful to bind **package** to a package around some code that deals with that package. The operations of loading, compiling, and editing a file all bind **package** to the package associated with the file.

pkg-goto &optional *pkg globally* *Function*

pkg can be a package or the name of a package. *pkg* is made the current package; in other words, the variable **package** is set to the package named by *pkg*. **pkg-goto** can be useful to "put the keyboard inside" a package when you are debugging.

pkg defaults to the **user** package.

If *globally* is specified non-**nil**, then **package** is set with **setq-globally** instead of **setq**. This is useful mainly in an init file, where you want to

change the default package for user interaction, and a simple **setq** of **package** does not work because it is bound by **load** when it loads the init file.

pkg-bind *pkg body...* *Macro*

pkg can be a package or a package name. The forms of the *body* are evaluated with the variable **package** bound to the package named by *pkg*. The values of the last form are returned.

Example:

```
(pkg-bind "zwei"
  (read-from-string function-name))
```

The difference between **pkg-bind** and a simple **let** of the variable **package** is that **pkg-bind** ensures that the new value for **package** is actually a package; it coerces package names (strings or symbols) into actual package objects.

79.2 Defining a Package

The **defpackage** special form is the preferred way to create a package. A **defpackage** form is treated as a *definition* form by the editor; hence the Edit Definition (M-.) command can find package definitions.

Typically you put a **defpackage** form in its own file, separate from the rest of a program's source code. The reason to use a separate file is that a package must be defined before it can be used. In order to compile, load, or edit your program, the package in which its symbols are to be read must already be defined. Typically the file containing the **defpackage** is read in the **user** package, while all the rest of the files of your program are read in your own private package.

When a large program consisting of multiple source files is maintained with the system system, one source file typically contains nothing but a **defpackage** form and a **defsystem** form. (Occasionally a few other housekeeping forms are present.) This file is called the *system declaration file*. The packages and systems built into the initial Lisp system are defined in two files: sys:sys;pkgdcl defines all the packages while sys:sys;sysdcl defines all the systems. See the section "Maintaining Large Programs" in *Program Development Utilities*.

In the simplest cases, where no nontrivial **defpackage** options are required, the **defpackage** form can be omitted and no separate file is required. All the information required to create your package is contained in the file attribute list of the file containing your program. See the section "Specifying Packages in Programs", page 579.

The **make-package** function is available as the primitive way to create package objects.

defpackage *name options...* *Special Form*

Define a package named *name*; the name must be a symbol so that the source file name of the package can be recorded and the editor can correctly sectionize the definition. If no package by that name already exists, a new package is created according to the specified options. If a package by that name already exists, its characteristics are altered according to the options specified. If any characteristic cannot be altered, an error is signalled. If the existing package was defined by a different file, you are queried before it is changed, as with any other type of definition.

Each *option* is a keyword or a list of a keyword and arguments. A keyword by itself is equivalent to a list of that keyword and one argument, **t**; this syntax really only makes sense for the **:external-only** and **:hash-inherited-symbols** keywords.

Wherever an argument is said to be a name or a package, it can be either a symbol or a string. Usually symbols are preferred, because the reader standardizes their alphabetic case and because readability is increased by not cluttering up the **defpackage** form with string quote (") characters.

None of the arguments are evaluated. The keywords arguments, most of which are identical to **make-package**'s, are:

(:nicknames *name name...***)**
The package is given these nicknames, in addition to its primary name.

(:prefix-name *name***)**
This name is used when printing a qualified name for a symbol in this package. The specified name should be one of the nicknames of the package or its primary name. If **:prefix-name** is not specified, it defaults to the shortest of the package's names (the primary name plus the nicknames).

(:use *package package...***)**
External symbols and relative name mappings of the specified packages are inherited. If this option is not specified, it defaults to **(:use global)**. To inherit nothing, specify **(:use)**.

(:shadow *name name...***)**
Symbols with the specified names are created in this package and declared to be shadowing.

(:export *name name...***)**
Symbols with the specified names are created in this package, or inherited from the packages it uses, and declared to be external.

(:import *symbol symbol...***)**
The specified symbols are imported into the package. Note that unlike **:export**, **:import** requires symbols, not names; it matters in which package this argument is read.

(:shadowing-import *symbol symbol...***)**
The same as **:import** but no name conflicts are possible; the symbols are declared to be shadowing.

(:import-from *package name name...***)**
The specified symbols are imported into the package. The symbols to be imported are obtained by looking up each *name* in *package*.

(**defpackage** only) This option exists primarily for system bootstrapping, since the same thing can normally be done by **:import**. The difference between **:import** and **:import-from** can be visible if the file containing a **defpackage** is compiled; when **:import** is used the symbols are looked up at compile time, but when **:import-from** is used the symbols are looked up at load time. If the package structure has been changed between the time the file was compiled and the time it is loaded, there might be a difference.

(:relative-names (*name package*) (*name package*)...**)**
Declare relative names by which this package can refer to other packages. The package being created cannot be one of the *packages*, since it has not been created yet.

(:relative-names-for-me (*package name*) (*package name*)...**)**
Declare relative names by which other packages can refer to this package.

(**defpackage** only) It is valid to use the name of the package being created as a *package* here; this is useful when a package has a relative name for itself.

(:size *number***)**
The number of symbols expected to be present in the package. This controls the initial size of the package's hash table. The **:size** specification can be an underestimate; the hash table is expanded as necessary.

(:hash-inherited-symbols *boolean***)**
If true, inherited symbols are entered into the package's hash table to speed up symbol lookup. If false (the default), looking up a symbol in this package searches the hash table of each package it uses.

(:external-only *boolean***)**
If true, all symbols in this package are external and the package is locked. This feature is only used to simulate the old package system that was used before Release 5.0. See the section "External-only Packages and Locking", page 626.

(:include *package package...***)**
Any package that uses this package also uses the specified packages. Note that if the **:include** list is changed, the change is not propagated to users of this package. This feature is used only to simulate the old package system that was used before Release 5.0.

(:new-symbol-function *function***)**
function is called when a new symbol is to be made present in the package. The default is **si:pkg-new-symbol** unless **:external-only** is specified. Do not specify this option unless you understand the internal details of the package system.

(:colon-mode *mode***)**
If *mode* is **:external**, qualified names mentioning this package behave differently depending on whether ":" or "::" is used, as in Common Lisp. ":" names access only external symbols. If *mode* is **:internal**, ":" names access all symbols. **:internal** is the default currently. See the section "Specifying Internal and External Symbols in Packages", page 585.

(:prefix-intern-function *function***)**
The function to call to convert a qualified name referencing this package with ":" (rather than "::") to a symbol. The default is **intern** unless **(:colon-mode :external)** is specified. Do not specify this option unless you understand the internal details of the package system.

make-package *name* &key ... *Function*

make-package is the primitive subroutine called by **defpackage**. **make-package** makes a new package and returns it. An error is signalled if the package name or nickname conflicts with an existing package. **make-package** takes the same arguments as **defpackage** except that standard **&key** syntax is used, and there is one additional keyword, **:invisible**.

When an argument is called a *name*, it can be either a symbol or a string. When an argument is called a *package*, it can be the name of the package as a symbol or a string, or the package itself.

The keyword arguments, most of which are identical to **defpackage**'s, are:

:nicknames '(*name name...***)**
The package is given these nicknames, in addition to its primary name.

:prefix-name *name*
This name is used when printing a qualified name for a symbol in this package. The specified name should be one of the nicknames of the package or its primary name. If **:prefix-name** is not specified, it defaults to the shortest of the package's names (the primary name plus the nicknames).

:invisible *boolean*
If true, the package is not entered into the system's table of packages, and therefore cannot be referenced via a qualified name.

This is useful if you simply want a package to use as a data structure, rather than as the package in which to write a program.

:use '(*package package...*)
External symbols and relative name mappings of the specified packages are inherited. If only a single package is to be used, the name rather than a list of the name can be passed. If no package is to be used, specify **nil**. The default value for **:use** is **global**.

:shadow '(*name name...*)
Symbols with the specified names are created in this package and declared to be shadowing.

:export '(*name name...*)
Symbols with the specified names are created in this package, or inherited from the packages it uses, and declared to be external.

:import '(*symbol symbol...*)
The specified symbols are imported into the package. Note that unlike **:export**, **:import** requires symbols, not names; it matters in which package this argument is read.

:shadowing-import '(*symbol symbol...*)
The same as **:import** but no name conflicts are possible; the symbols are declared to be shadowing.

:import-from '(*package name name...*)
The specified symbols are imported into the package. The symbols to be imported are obtained by looking up each *name* in *package*.

(**defpackage** only) This option exists primarily for system bootstrapping, since the same thing can normally be done by **:import**. The difference between **:import** and **:import-from** can be visible if the file containing a **defpackage** is compiled; when **:import** is used the symbols are looked up at compile time, but when **:import-from** is used the symbols are looked up at load time. If the package structure has been changed between the time the file was compiled and the time it is loaded, there might be a difference.

:relative-names '((*name package*) (*name package*)...)
Declare relative names by which this package can refer to other packages. The package being created cannot be one of the *packages*, since it has not been created yet.

:relative-names-for-me '((*package name*) (*package name*)...)
Declare relative names by which other packages can refer to this package.

(**defpackage** only) It is valid to use the name of the package being created as a *package* here; this is useful when a package has a relative name for itself.

:size *number*
The number of symbols expected to be present in the package. This controls the initial size of the package's hash table. The **:size** specification can be an underestimate; the hash table is expanded as necessary.

:hash-inherited-symbols *boolean*
If true, inherited symbols are entered into the package's hash table to speed up symbol lookup. If false (the default), looking up a symbol in this package searches the hash table of each package it uses.

:external-only *boolean*
If true, all symbols in this package are external and the package is locked. This feature is only used to simulate the old package system that was used before Release 5.0. See the section "External-only Packages and Locking", page 626.

:include '(*package package...***)**
Any package that uses this package also uses the specified packages. Note that if the **:include** list is changed, the change is not propagated to users of this package. This feature is used only to simulate the old package system that was used before Release 5.0.

:new-symbol-function *function*
function is called when a new symbol is to be made present in the package. The default is **si:pkg-new-symbol** unless **:external-only** is specified. Do not specify this option unless you understand the internal details of the package system.

:colon-mode *mode*
If *mode* is **:external**, qualified names mentioning this package behave differently depending on whether ":" or "::" is used, as in Common Lisp. ":" names access only external symbols. If *mode* is **:internal**, ":" names access all symbols. **:internal** is the default currently. See the section "Specifying Internal and External Symbols in Packages", page 585.

:prefix-intern-function *function*
The function to call to convert a qualified name referencing this package with ":" (rather than "::") to a symbol. The default is **intern** unless **(:colon-mode :external)** is specified. Do not specify this option unless you understand the internal details of the package system.

pkg-kill *package* *Function*
Kill *package* by removing it from all package system data structures. The name and nicknames of *package* cease to be recognized package names. If *package* is used by other packages, it is un-used, causing its external symbols to stop being accessible to those packages. If other packages have relative names for *package*, the names are deleted.

Any symbols in *package* still exist and their home package is not changed. If this is undesirable, evaluate **(mapatoms #'remob** *package* **nil)** first.

package can be a package or the name of a package.

79.3 Mapping Names to Symbols

The name of a symbol is a string, corresponding to the printed representation of that symbol with quoting characters removed. Mapping the name of a symbol into the symbol itself is called *interning*, for historical reasons. Interning is only meaningful with respect to a particular package, since packages are name-to-symbol mappings. Unless a package is explicitly specified, the current package is assumed.

There are four functions for interning: **intern**, **intern-soft**, **intern-local**, and **intern-local-soft**. Each function takes two arguments and returns two values. The arguments are a name and a package. The name can be a string or a symbol. The package argument can be a package, the name of a package as a string or a symbol, or **nil** or unsupplied, in which case the current package (the value of **package**) is used by default.

The **-soft** functions do not create new symbols, but only find existing symbols. The other two functions add a new symbol to the package if no existing symbol with the specified name is found. When adding a new symbol, if the name argument is a string, a new symbol is created and its home package is made to be the specified package. If the name argument is a symbol, that symbol is used as the new symbol. If it has a home package, it is not changed, but if it does not have a home package its home package is set to the package to which it was just added.

The **-local** functions only look for symbols present in the package; they do not search through inherited symbols. The other two functions see all accessible symbols.

The first value is the symbol that was found or created, or **nil** if no symbol was found and a **-soft** function was called. The second value is a flag that takes on one of the following values:

nil No preexisting symbol was found. If the function called was not a **-soft** version, a new internal symbol was added to the package.

:internal An existing internal symbol was found to be present in the package.

:external An existing external symbol was found to be present in the package.

:inherited An existing symbol was found to be inherited by the package. This symbol is necessarily external in the package from which it was inherited, and cannot be external in the package being searched.

Note that the first value should not be used as a flag to detect whether or not a symbol was found, since the *false* value, **nil**, is a symbol. The second value must be used for this purpose. The **-soft** functions return both values **nil** if they do not find a symbol.

Note: interning is sensitive to case; that is, it considers two character strings different even if the only difference is one of uppercase versus lowercase (unlike most string comparisons elsewhere in the Symbolics Lisp Machine system). Symbols are converted to uppercase when you type them in because the reader converts the case of characters in the printed representation of symbols; the characters are converted to uppercase before **intern** is ever called. So if you call **intern** with a lowercase "**foo**" and then with an uppercase "**FOO**", you do not get the same symbol.

79.3.1 Functions That Map Names to Symbols

intern *string* &optional *(pkg* **package***)* *Function*

Find or create a symbol named *string* accessible to *pkg*, either directly present in *pkg* or inherited from a package it uses.

If *string* is not a string but a symbol, **intern** searches for a symbol with the same name. If it does not 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.

For more information: See the section "Mapping Names to Symbols", page 604.

intern-local *string* &optional *(pkg* **package***)* *Function*

Find or create a symbol named *string* directly present in *pkg*. Symbols inherited by *pkg* from packages it uses are not considered, thus **intern-local** can cause a name conflict. **intern-local** is considered to be a low-level primitive and indiscriminate use of it can cause undetected name conflicts. Use **import**, **shadow**, or **shadowing-import** for normal purposes.

If *string* is not a string but a symbol, and no symbol with that print name is already interned in *pkg*, **intern-local** interns *string* — rather than a newly created symbol — in *pkg* (even if it is also interned in some other package) and returns it.

For more information: See the section "Mapping Names to Symbols", page 604.

intern-soft *string* &optional *(pkg* **package***)* *Function*

Find a symbol named *string* accessible to *pkg*, either directly present in *pkg* or inherited from a package it uses. If no symbol is found, the two values **nil nil** are returned.

intern-local-soft *string* &optional *(pkg* **package)** *Function*

Find a symbol named *string* directly present in *pkg*. Symbols inherited by *pkg* from packages it uses are not considered. If no symbol is found, the two values **nil nil** are returned.

intern-local-soft is a good low-level primitive for when you want complete control of what packages to search and when to add new symbols.

For more information: See the section "Mapping Names to Symbols", page 604.

find-all-symbols *string* *Function*

Search all packages for symbols named *string* and return a list of them. Duplicates are removed from the list; if a symbol is present in more than one package, it only appears once in the list. The **global** package is searched first, and so global symbols appear earlier in the list than symbols that shadow them. In general packages are searched in the order that they were created.

string can be a symbol, in which case its name is used. This is primarily for user convenience when calling **find-all-symbols** directly from the read-eval-print loop.

Invisible packages are not searched.

The **where-is** function is a more user-oriented version of **find-all-symbols**; it returns information about *string*, rather than just a list.

For more information: See the section "Mapping Names to Symbols", page 604.

remob *symbol* &optional *package* *Function*

remob removes *symbol* from *package* (the name is historical and means "REMove from OBlist"). *symbol* itself is unaffected, but **intern** no longer finds it in *package*. Removing a symbol from its home package sets its home package to **nil**; removing a symbol from a package different from its home package leaves the symbol's home package unchanged.

remob returns **t** if the symbol was found and removed, or **nil** if it was not found.

remob is always "local", in that it removes only from the specified package and not from any other packages. Thus **remob** has no effect unless the symbol is present in the specified package, even if it is accessible from that package via inheritance.

If *package* is unspecified it defaults to the symbol's home package. Note this exception well: the default value of **remob**'s *package* argument is *not* the current package.

79.4 Functions That Find the Home Package of a Symbol

symbol-package *symbol* *Function*

Returns the contents of *symbol*'s package cell, which is the package that owns *symbol*, or **nil** if *symbol* is uninterned.

package-cell-location *symbol* *Function*

Returns a locative pointer to *symbol*'s package cell. It is preferable to write the following, rather than calling this function explicitly.

```
(locf (symbol-package symbol))
```

keywordp *object* *Function*

A predicate that is true if *object* is a symbol and its home package is the keyword package, and false otherwise.

79.5 Mapping Between Names and Packages

pkg-name *package* *Function*

Get the (primary) name of a package. The name is a string.

It is an error if *package* is not a package object. (The phrase "it is an error" has special significance in Common Lisp. See the Common Lisp manual, not available from Symbolics, for more information.) Note that **pkg-name** is a structure-accessing function and does not check that its argument is a package object, only that it is some kind of an array with a leader.

pkg-find-package *x* &optional *(create-p* **:error***) (relative-to* **nil***)* *Function*

pkg-find-package tries to interpret *x* as a package. Most of the functions whose descriptions say "... can be either a package or the name of a package" call **pkg-find-package** to interpret their package argument.

If *x* is a package, **pkg-find-package** returns it.

If *x* is a symbol or a string, it is interpreted as the name of a package. If *relative-to* is specified and non-**nil**, then it must be a package or the name of a package. If *relative-to* or one of the packages it uses has a relative name of *x*, the package named by that relative name is used. If the relative name search fails, or if no relative name search is called for (that is, *relative-to* is **nil**, which is the default), then if a package with a primary name or nickname of *x* exists it is returned.

If *x* is a list, it is presumed to have come from a file attribute line. **pkg-find-package** is done on the car of the list. If that fails, a new package is created with that name, according to the specifications in the rest of the list. See the section "Specifying Packages in Programs", page 579.

If no package is found, the *create-p* argument controls what happens. Note that this can only happen if *x* is a symbol or a string. The possible values for *create-p* are:

:error or **nil**	An error is signalled. The error can be continued by defining the package manually, creating it automatically with default attributes, or using a different package name instead. **:error** is the default. **nil** is accepted as a synonym for **:error** for backwards compatibility.
:find	Just return **nil**.
:ask	Ask the user whether to create it.
t	Create a package with the specified name with default attributes. It does inherit from **global** but not from any other packages.

The package name search is independent of alphabetic case. However, this might be changed in the future for Common Lisp compatibility and should not be depended upon. In any event it is not considered good style to have two distinct packages whose names differ only in alphabetic case.

79.6 Package Iteration

mapatoms *function* &optional *(package* **package***)* *(inherited-p* **t***)* *Function*

function should be a function of one argument. **mapatoms** applies *function* to each of the symbols in *package*. If *inherited-p* is **t**, this is all symbols accessible to *package*, including symbols it inherits from other packages. If *inherited-p* is **nil**, *function* only sees the symbols that are directly present in *package*.

Note that when *inherited-p* is **t** symbols that are shadowed but otherwise would have been inherited are seen; this slight blemish is for the sake of efficiency. If this is a problem, *function* can try **intern** in *package* on each symbol it gets, and ignore the symbol if it is not **eq** to the result of **intern**; this measure is rarely needed.

mapatoms-all *function* *Function*

function should be a function of one argument. **mapatoms-all** applies *function* to all of the symbols in all of the packages in existence, except for invisible packages. Note that symbols that are present in more than one package are seen more than once.

Example:

```
(mapatoms-all
  (function
    (lambda (x)
      (and (alphalessp 'z x)
           (print x)))))
```

do-symbols (*variable* &optional *package result*) *body...* *Special Form*

Evaluate the *body* forms repeatedly with *variable* bound to each symbol accessible in *package*. *package* can be a package object or a string or symbol that is the name of a package, or it can be omitted, in which case the value of **package** is used by default.

When the iteration terminates, *result* is evaluated and its values are returned. The value of *variable* is **nil** during the evaluation of *result*. If *result* is not specified, the value returned is **nil**.

The **return** special form can be used to cause a premature exit from the iteration.

do-local-symbols (*variable* &optional *package result*) *body...* *Special Form*

Evaluate the *body* forms repeatedly with *variable* bound to each symbol present in *package*. *package* can be a package object or a string or symbol that is the name of a package, or it can be omitted, in which case the value of **package** is used by default.

When the iteration terminates, *result* is evaluated and its values are returned. The value of *variable* is **nil** during the evaluation of *result*. If *result* is not specified, the value returned is **nil**.

The **return** special form can be used to cause a premature exit from the iteration.

do-external-symbols (*variable* &optional *package result*) *body...* *Special Form*

Evaluate the *body* forms repeatedly with *variable* bound to each external symbol exported by *package*. *package* can be a package object or a string or symbol that is the name of a package, or it can be omitted, in which case the value of **package** is used by default.

When the iteration terminates, *result* is evaluated and its values are returned. The value of *variable* is **nil** during the evaluation of *result*. If *result* is not specified, the value returned is **nil**.

The **return** special form can be used to cause a premature exit from the iteration.

do-all-symbols (*variable* &optional *result*) *body...* *Special Form*

Evaluate the *body* forms repeatedly with *variable* bound to each symbol present in any package (excluding invisible packages).

When the iteration terminates, *result* is evaluated and its values are

returned. The value of *variable* is **nil** during the evaluation of *result*. If *result* is not specified, the value returned is **nil**.

The **return** special form can be used to cause a premature exit from the iteration.

See the section "Iteration Paths", page 222. This section contains a discussion of the **interned-symbols** and **local-interned-symbols loop** iteration paths.

79.7 Interpackage Relations

pkg-add-relative-name *from-package name to-package* *Function*

Add a relative name named *name*, a string or a symbol, that refers to *to-package*. From now on, qualified names using *name* as a prefix, when the current package is *from-package* or a package that uses *from-package*, refer to *to-package*.

from-package and *to-package* can be packages or names of packages.

It is an error if *from-package* already defines *name* as a relative name for a package different from *to-package*.

pkg-delete-relative-name *from-package name* *Function*

If *from-package* defines *name* as a relative name, it is removed. *from-package* can be a package or the name of a package. *name* can be a symbol or a string. It is not an error if *from-package* does not define *name* as a relative name.

package-use-list *package* *Function*

The list of other packages used by the argument package. *package* can be a package object or the name of a package (a symbol or a string). The elements of the list returned are package objects.

package-used-by-list *package* *Function*

The list of other packages that use the argument package. *package* can be a package object or the name of a package (a symbol or a string). The elements of the list returned are package objects.

use-package *packages-to-use* &optional *package* *Function*

The *packages-to-use* argument should be a list of packages or package names, or a single package or package name. These packages are added to the use-list of *package* if they are not there already. All external symbols in the packages to use become accessible in *package*. *package* can be a package object or the name of a package (a symbol or a string). If unspecified, *package* defaults to the value of **package**. Returns **t**.

unuse-package *packages-to-unuse* &optional *package* *Function*

The *packages-to-unuse* argument should be a list of packages or package names, or a single package or package name. These packages are removed from the use-list of *package* and their external symbols are no longer accessible, unless they are accessible through another path. *package* can be a package object or the name of a package (a symbol or a string). If unspecified, *package* defaults to the value of **package**. Returns **t**.

79.8 Functions That Import, Export, and Shadow Symbols

export *symbols* &optional *package* *Function*

The *symbols* argument should be a list of symbols or a single symbol. If *symbols* is **nil**, it is treated like an empty list. These symbols become available as external symbols in *package*. *package* can be a package object or the name of a package (a symbol or a string). If unspecified, *package* defaults to the value of **package**. Returns **t**. The **:export** option to **defpackage** and **make-package** is equivalent.

unexport *symbols* &optional *package* *Function*

The *symbols* argument should be a list of symbols or a single symbol. If *symbols* is **nil**, it is treated like an empty list. These symbols become internal symbols in *package*. *package* can be a package object or the name of a package (a symbol or a string). If unspecified, *package* defaults to the value of **package**. Returns **t**.

package-external-symbols *package* *Function*

A list of all the external symbols exported by *package*. *package* can be a package object or the name of a package (a symbol or a string).

import *symbols* &optional *package* *Function*

The *symbols* argument should be a list of symbols or a single symbol. If *symbols* is **nil**, it is treated like an empty list. These symbols become internal symbols in *package*, and can therefore be referred to without a colon qualifier. **import** signals a correctable error if any of the imported symbols has the same name as some distinct symbol already available in the package.

package can be a package object or the name of a package (a symbol or a string). If unspecified, *package* defaults to the value of **package**. Returns **t**.

shadowing-import *symbols* &optional *package* *Function*

This is like **import**, but it does not signal an error even if the importation of a symbol would shadow some symbol already available in the package. If a distinct symbol with the same name is already present in the package, it is removed (using **remob**). The imported symbol is placed on the shadowing-symbols list of *package*.

The *symbols* argument should be a list of symbols or a single symbol. If *symbols* is **nil**, it is treated like an empty list. *package* can be a package object or the name of a package (a symbol or a string). If unspecified, *package* defaults to the value of **package**. Returns **t**.

shadowing-import should be used with caution. It changes the state of the package system in such a way that the consistency rules do not hold across the change.

shadow *symbols* &optional *package* *Function*

The *symbols* argument should be a list of symbols or a single symbol. If *symbols* is **nil**, it is treated like an empty list. The name of each symbol is extracted, and *package* is searched for a symbol of that name. If no such symbol is present in this package (directly, not by inheritance), a new symbol is created with this name and inserted in *package* as an internal symbol. The symbol is also placed on the shadowing-symbols list of *package*.

package can be a package object or the name of a package (a symbol or a string). If unspecified, *package* defaults to the value of **package**. Returns **t**.

shadow should be used with caution. It changes the state of the package system in such a way that the consistency rules do not hold across the change.

package-shadowing-symbols *package* *Function*

The list of symbols that have been declared as shadowing symbols in this package by **shadow** or **shadowing-import**. All symbols on this list are present in the specified package. *package* can be a package object or the name of a package (a symbol or a string).

79.9 Package "Commands"

describe-package *package* *Function*

Print a description of *package*'s attributes and the size of its hash table of symbols on **standard-output**. *package* can be a package or the name of a package. The **describe** function calls **describe-package** when its argument is a package.

where-is *pname* *Function*

Finds all symbols named *pname* and prints on **standard-output** a description of each symbol. The symbol's home package and name are printed. If the symbol is present in a different package than its home package (that is, it has been imported), that fact is printed. A list of the packages from which the symbol is accessible is printed, in alphabetical order. **where-is** searches all packages that exist, except for invisible packages.

If *pname* is a string it is converted to uppercase, since most symbols' names use uppercase letters. If *pname* is a symbol, its exact name is used.

where-is returns a list of the symbols it found.

The **find-all-symbols** function is the primitive that does what **where-is** does without printing anything.

globalize *name* &optional *package* *Function*

Establish a symbol named *name* in *package* and export it. If this causes any name conflicts with symbols with the same name in packages that use *package*, instead of signalling an error make an attempt to resolve the name conflict automatically. Print an explanation of what is being done on **error-output**.

globalize is useful for patching up an existing package structure. For example, if a new function is added to the Lisp language **globalize** can be used to add its name to the **global** package and hence make it accessible to all packages. Symbols with the desired name might already exist, either by coincidence or because the function was already defined or already called. **globalize** makes all such symbols have the new function as their definition.

package can be a package or the name of a package, as a symbol or a string. It defaults to the **global** package. **globalize** is the only function that does not care whether *package* is locked.

name can be a symbol or a string. If *package* already contains a symbol by that name, that symbol is chosen. Otherwise, if *name* is a symbol, it is chosen. If *name* is a string and any of the packages that use *package* contains a nonshadowing symbol by that name, one such symbol is chosen. Otherwise, a new symbol named *name* is created. Whichever symbol is chosen this way is made present in *package* and exported from it. If the home package of the chosen symbol is a package that uses *package*, then the home package is set to *package*; in other words, the symbol is "promoted" to a "higher" package. If the home package of the chosen symbol is some other package, it is not changed. This case typically occurs when the chosen symbol is inherited by *package* from some package it uses.

The above rules for choosing a symbol to export ensure that no name conflict occurs if at all possible. If any nonshadowing symbols exist named *name* but that are distinct from the chosen symbol present in the packages that use *package*, then a name conflict occurs. **globalize** does its best to resolve the name conflict by merging together the values, function definitions, and properties of all the symbols involved. After merging, all the symbols have the same value, the same function definition, and the same properties. The value cells, function cells, and property list cells of all the symbols are forwarded to the corresponding cells of the chosen symbol, using **dtp-one-q-forward**. This ensures that any future change to one of the symbols is reflected by all of the symbols.

The merging operation simply consists of making sure that there are no conflicts. If more than one of the symbols has a value (is **boundp**), all the values must be **eql** or an error is signalled. Similarly, all the function definitions of symbols that are **fboundp** must be **eql** and all the properties with any particular indicator must be **eql**. If an error occurs you must manually resolve it by removing the unwanted value, definition, or property (using **makunbound**, **fmakunbound**, or **remprop**) then try again.

Note that if *name* is a symbol, **globalize** attempts to use that symbol, but there is no guarantee that it will not use some other symbol. If *name* is in a package that does not use *package*, and **globalize** does not use *name* as the symbol (because another symbol by that name already exists in *package* or in some package that uses *package*), then *name* is not merged with the chosen symbol. It is generally more predictable to use a string, rather than a symbol, for *name*.

Of course, **globalize** cannot cause two distinct symbols to become **eq**. Its conflict resolution techniques are useful only for symbols that are used as names for things like functions and variables, not for symbols that are used for their own sake. You can sometimes get the desired effect by using one of the conflicting symbols as the first argument to **globalize**, rather than using a string.

For example, suppose a program in the **color** package deals with colors by symbolic names, perhaps using **selectq** to test for such symbols as **red**, **green**, and **yellow**. Suppose there is also a function named **red** in the **math** package and someone decides that this function is generally useful and should be made global. Doing **(globalize 'color:red)** ensures that the exported symbol is the one that the color program is looking for; this means that every package except the **math** package sees the right symbol to use if it wants to call the color program. Programs that call the **red** function do not care which of the two symbols they use as the name of the function, since both symbols have the same definition. Usually the situation described in this example would not arise, because standard programming style dictates that the color program should have been using keywords for this application.

globalize returns two values. The first is the chosen symbol and the second is a (possibly empty) list of all the symbols whose value, function, and property cells were forwarded to the cells of the chosen symbol.

To disable the messages printed by **globalize**, bind **error-output** to a null stream (one that throws away all output). For example:

```
(let ((error-output 'si:null-stream))
  (globalize 'rumpelstiltskin))
```

There is a subtle pitfall in the interaction between **globalize** and the binary files output by the compiler. Because of this it is best to use a string, rather than a

symbol, as the argument to **globalize** in files that are to be compiled. Suppose a file contains the following form at top level:

```
(eval-when (compile load eval)
  (globalize 'si:rumpelstiltskin))
```

If the file is loaded without being compiled, the form is read and evaluated in the obvious fashion. **rumpelstiltskin** is read as the symbol by that name in the **si** package, that symbol is passed to the **globalize** function, and the symbol is moved to the **global** package. Now suppose the file is compiled. Again **rumpelstiltskin** is read as the symbol by that name in the **si** package. The **eval-when** causes the compiler first to evaluate the **globalize** form, and then to place a representation of the form into its output file. But at the time the output file is being generated, the symbol **rumpelstiltskin** is global; the compiler no longer has any way to know that it came from the **si** package. When the binary file is loaded, it globalizes the symbol **rumpelstiltskin** in the current package, not the one in the **si** package as the programmer intended. Furthermore, if at compile time there was a **rumpelstiltskin** symbol in the current package, the compile-time **globalize** turns that symbol into a shadowing symbol. When the binary file is loaded, it tries to refer to the symbol **rumpelstiltskin** in the **global** package, which gets an error since the **global** package is locked. The same pitfall can arise without the use of **eval-when** if the file being compiled was previously loaded into the Lisp that compiled it, perhaps for test purposes.

79.10 System Packages

The following are some of the packages initially present in the Lisp world. New packages will be added to this list from time to time. The list is presented in "logical" order, with the most important or interesting packages first. A number of packages that are not of general interest have been omitted from the list for the sake of brevity.

global Contains the global symbols of the Lisp language, including function names, variable names, special form names, and so on. All symbols in **global** are supposed to be documented. **global** does not inherit symbols from any other package.

keyword Contains keyword symbols. **keyword** has a blank nickname so that keywords print as **:foo** rather than **keyword:foo**. **keyword** does not inherit symbols from any other package.

user The default package for user programs that do not have their own package. When first booted the Symbolics Lisp Machine uses the **user** package to read expressions typed in by the user.

sys or **system** Contains symbols shared among various system programs. **system** is for symbols global to the Symbolics Lisp Machine

"operating system", while **global** is for symbols global to the Lisp language.

si or **system-internals**
Most of the programs that implement the Lisp language and operating system are in the **system-internals** package. **system-internals** is one of the packages that uses **system**. The externally advertised symbols of these programs are in **system** or **global**. **system-internals** would not exist as a separate package from **system** if the system took advantage of the distinction between internal symbols and external symbols, but it does not yet do so.

compiler Contains the compiler. **compiler** is one of the packages that use **system**.

dbg or **debugger** Contains the condition system and the debugger. **debugger** is one of the packages that use **system**.

zwei Contains the editor and Zmail.

tv Contains the window system. **tv** is one of the packages that use **system**.

fs or **file-system** Contains pathnames and the generic file access system. **file-system** is one of the packages that use **system**.

lmfs Contains the Symbolics Lisp Machine file storage system. **lmfs** is one of the packages that use **system**.

format Contains the function **format** and its associated subfunctions.

net or **network** Contains the external interfaces to the generic network system. **network** is one of the packages that uses **system**. Each network implementation and network-related program has its own package, which uses **network**.

neti or **network-internals**
Contains the programs that implement the generic network system. **network-internals** is one of the packages that use **network** and **system**.

chaos Contains the Chaosnet control program. **chaos** is one of the packages that use **network** and **system**.

cl or **common-lisp-global**
Contains the global symbols of the Common Lisp Compatibility Package. Inside of Common Lisp this package is called **lisp**. **common-lisp-global** does not use **global**.

fonts Contains the names of all fonts. **fonts** does not inherit symbols from any other package.

The following variables have the most important packages as their values.

pkg-global-package *Variable*
The **global** package.

pkg-keyword-package *Variable*
The **keyword** package.

pkg-system-package *Variable*
The **system** package.

80. Package-related Conditions

This section documents the most basic package-related conditions. There are other conditions built on these, but most programmers should not need to deal with them.

sys:package-error *Flavor*

All package-related error conditions are built on **sys:package-error**.

sys:package-not-found *Flavor*

A package-name lookup did not find any package by the specified name.

The **:name** message returns the name. The **:relative-to** message returns **nil** if only absolute names are being searched, or else the package whose relative names are also searched.

The **:no-action** proceed type can be used to try again. The **:new-name** proceed type can be used to specify a different name or package. The **:create-package** proceed type creates the package with default characteristics.

sys:external-symbol-not-found *Flavor*

A ":" qualified name referenced a name that had not been exported from the specified package.

The **:string** message returns the name being referenced (no symbol by this name exists yet). The **:package** message returns the package.

The **:export** proceed type exports a symbol by that name and uses it.

sys:package-locked *Flavor*

There was an attempt to intern a symbol in a locked package.

The **:symbol** message returns the symbol. The **:package** message returns the package.

The **:no-action** proceed type interns the symbol just as if the package had not been locked. Other proceed types are also available when interning the symbol would cause a name conflict.

sys:name-conflict *Flavor*

Any sort of name conflict occurred (there are specific flavors, built on **sys:name-conflict**, for each possible type of name conflict.) The following proceed types might be available, depending on the particular error:

The **:skip** proceed type skips the operation that would cause a name conflict.

The **:shadow** proceed type prefers the symbols already present in a package to conflicting symbols that would be inherited. The preferred symbols are added to the package's shadowing-symbols list.

The **:export** proceed type prefers the symbols being exported (or being inherited due to a **use-package**) to other symbols. The conflicting symbols are **remob**'ed if they are directly present, or shadowed if they are inherited.

The **:unintern** proceed type removes the conflicting symbol (with **remob**).

The **:shadowing-import** proceed type imports one of the conflicting symbols and makes it shadow the others. The symbol to be imported is an optional argument.

The **:share** proceed type causes the conflicting symbols to share value, function, and property cells. It as if **globalize** were called.

The **:choose** proceed type pops up a window in which the user can choose between the above proceed types individually for each conflict.

81. Multipackage Programs

Usually, each independent program occupies one package. But large programs, such as MACSYMA, are usually made up of a number of subprograms, and each subprogram might be maintained by a different person or group of people. We would like each subprogram to have its own namespace, since the program as a whole has too many names for anyone to remember. The package system can provide the same benefits to programs that are part of the same superprogram as it does to programs that are completely independent.

Putting each subprogram into its own package is easy enough, but it is likely that a fair number of functions and symbols should be shared by all of MACSYMA's subprograms. These would be internal interfaces between the different subprograms.

A package named **macsyma** can be defined and each of the internal interface symbols can be exported from it. Each subprogram of MACSYMA has its own package, which uses the **macsyma** package in addition to any other packages it uses. Thus the interface symbols are accessible to all subprograms, through package inheritance. These interface symbols typically get their function definitions, variable values, and other properties from various subprograms read into the various internal MACSYMA packages, although there is nothing wrong with also putting a subprogram directly into the **macsyma** package. This is similar to the way the Lisp system works; the **global** package exports a large number of symbols, which get their values, definitions, and so on from programs residing in other packages that use **global**, such as **system-internals** or **compiler**.

It is also often convenient for the **macsyma** package to supply relative names that can be used by the various subprograms to refer to each other's packages. This allows package name abbreviations to be used internally to MACSYMA without contaminating the external environment.

The system declaration file for MACSYMA would then look something like the following:

```
;Contains the interfaces between the various subprograms
(defpackage macsyma
            (:export meval mprint ptimes ...)
            (:colon-mode :external))    ;error-checking in qualified names

;The integration package based on the Risch algorithm
(defpackage risch
            (:use macsyma global))
```

```
;The integration package based on pattern matching
(defpackage sin
            (:use macsyma global))

;Interface to the operating system.  This uses the SYSTEM package
;because it needs many system-dependent functions and constants.
;This package also has a local nickname because its primary name
;is so long.
(defpackage macsyma-system-interface
            (:relative-names-for-me (macsyma sysi))
            (:use macsyma system global))
```

You can break the interface symbols down into separate categories. For instance, you might want to separate internal symbols used only inside MACSYMA from symbols that are also useful to the outside world. The latter symbols clearly should be externals of the **macsyma** package. You could create an additional package named **macsyma-internals** that exports all the symbols that are interfaces between different subprograms of MACSYMA but are not for use by the outside world. In this case we would have:

```
(defpackage risch
            (:use macsyma-internals macsyma global))
```

A program in the outside world that needed to use parts of MACSYMA would either use qualified names such as **macsyma:solve** or would include **macsyma** in the **:use** option in its package definition.

The interface symbols can be broken down into even more categories. Each subpackage can have its own list of exported symbols, and can use whichever other subpackages it depends on. The subset of these exported symbols that are also useful to the outside world can be exported from the **macsyma** package as well. In this case our example system declaration file would look something like:

```
;Contains the interfaces between the various subprograms
(defpackage macsyma
            (:export solve integrate ...)
            (:colon-mode :external))    ;error-checking in qualified names

;The rational function package
(defpackage rat
            (:export ptimes ...)
            (:use macsyma global))

;The integration package
(defpackage risch
            (:export integrate)
            (:use rat macsyma global))
```

```
;The macsyma interpreter
(defpackage meval
            (:export meval mprint ...))
```

The symbol **integrate** exported by the **macsyma** package and the symbol **integrate** exported by the **risch** package are the same symbol, because **risch** inherits it from **macsyma**.

Sometimes you can get involved in forward references when setting up this sort of package structure. In the above example, **risch** needs to use **rat**, hence **rat** was defined first. If **rat** also needed to use **risch**, there would be no way to write the package definitions using only **defpackage**. In this case you can explicitly call **use-package** after both packages have been defined. For example:

```
;The rational function package
(defpackage rat
            (:export ptimes ...)
            (:use macsyma global))      ;also uses risch

;The integration package
(defpackage risch
            (:export integrate)
            (:use rat macsyma global))

;Now complete the forward references
(use-package 'risch 'rat)
```

An analogous issue arises when using **:import**.

Now, the **risch** program and the **sin** program both do integration, and so it would be natural for each to have a function called **integrate**. From inside **sin**, **sin**'s **integrate** would be referred to as **integrate** (no prefix needed), while **risch**'s would be referred to as **risch::integrate** or as **risch:integrate** if **risch** exported it (which is likely). Similarly, from inside **risch**, **risch**'s own **integrate** would be called **integrate**, whereas **sin**'s would be referred to as **sin::integrate** or **sin:integrate**.

If **sin**'s **integrate** were a recursive function, you would refer to it from within **sin** itself, and would not have to type **sin:integrate** every time; you would just say **integrate**.

If the names **sin** and **risch** are considered to be too short to use up in the general space of package names, they can be made local abbreviations within MACSYMA's family of package through local names. The package definitions would be

```
;Contains the interfaces between the various subprograms
(defpackage macsyma
            (:export meval mprint ptimes ...)
            (:colon-mode :external))    ;error-checking in qualified names
```

```
;The integration package based on the Risch algorithm
(defpackage macsyma-risch-integration
            (:relative-names-for-me (macsyma risch))
            (:use macsyma global))

;The integration package based on pattern matching
(defpackage macsyma-pattern-integration
            (:relative-names-for-me (macsyma sin))
            (:use macsyma global))
```

From inside the **macsyma** package or any package that uses it the two integration functions would be referred to as **sin:integrate** and as **risch:integrate**. From anywhere else in the hierarchy, they could be called **macsyma:sin:integrate** and **macsyma:risch:integrate**, or **macsyma-pattern-integration:integrate** and **macsyma-risch-integration:integrate**.

82. Compatibility with the Pre-release 5.0 Package System

The package system used before Release 5.0 used a hierarchical arrangement of packages and used **package-declare** rather than **defpackage** to create packages. Most users will not see any change between the old and new package systems, since the same function names continue to work and most of the old functionality can be simulated. All programs do need to be recompiled, however, because old assumptions built into the compiled code — such as where keyword symbols reside and what the indices of fields in the package structure are — are no longer valid.

If **pack1** was a subpackage of **pack2** in the hierarchical package system, then in the current system **pack1** should use **pack2** and **pack2** should be declared external-only so that all of its symbols will be inherited by **pack1**. Relative names follow the package use relations just as refnames used to follow the subpackage relations.

package-declare *name superior size* &optional *file-alist clause...* *Special Form*

This special form exists only for compatibility with the pre-Release 5.0 package system. **defpackage** should be used instead.

name is the name of the package to be created. It must be a string or a symbol (a list is no longer acceptable).

superior is used as the **:use** option to **defpackage**.

size is used as the **:size** option to **defpackage**.

file-alist must be **nil**; this feature has been obsolete for several years. It can be omitted if there are no clauses.

Each *clause* is a list whose first element is one of the following symbols and whose remaining elements are "arguments". It makes no difference in what package the symbols are read, since only their names are used.

borrow Used as the **:import-from** option to **defpackage**.

intern Used as the **:export** option to **defpackage**.

shadow Used as the **:shadow** option to **defpackage**.

refname Used as an element of the **:relative-names** option to **defpackage**. Note that this clause is usually unnecessary in the current package system, since package naming works more rationally.

myrefname Used as an element of the **:relative-names-for-me** option to **defpackage**, unless the first argument is **global**, in which case it is used as an element of the **:nicknames**

option. Note that this clause is usually unnecessary in the current package system, since package naming works more rationally.

The **use, external, advertise, forward, forward-alias, indirect, indirect-alias, keyword,** and **subpackage** clauses that **package-declare** used to accept cannot be simulated and are no longer allowed. None of these were documented and some of them did not work.

pkg-create-package *name* &optional *superior size* *Function*

This function exists only for compatibility with the pre-Release 5.0 package system. **make-package** should be used instead.

name must be a symbol or a string; lists are no longer accepted. *superior* is used as the **:use** argument to **make-package**. If *superior* is **nil** then **:invisible t** is specified. *size* is used as the **:size** argument to **make-package**.

The *dont-lock-superior* argument no longer exists. Package locking is now controlled explicitly by the **:external-only** option to **defpackage** and **make-package**.

The global functions **pkg-contained-in, pkg-debug-copy, pkg-load, pkg-refname-alist,** and **pkg-super-package** no longer exist. The first three of these were not documented.

The functions **intern, intern-local, intern-soft,** and **intern-local-soft** no longer return three values. Now only two values are returned. The second value is different but upward-compatible.

The functions **mapatoms-all** and **where-is** no longer take an optional argument defaulting to the **global** package. They now always process all packages that are not invisible. The function **package-used-by-list** can help if you need to process only the subset of all packages that use some particular package.

82.1 External-only Packages and Locking

The facilities described in this section are primarily for compatibility with the old, hierarchical package system used before Release 5.0. Full use of these facilities requires knowing about functions that are in the **si** package.

A package can be *locked*, which means that any attempt to add a new symbol to it signals an error. Continuing from the error adds the symbol.

When reading from an interactive stream, such as a window, the error for adding a new symbol to a locked package does not go into the Debugger. Instead it asks you to correct your input, using the input editor. You cannot add a new symbol to a

locked package just by typing its name; you must explicitly call **intern**, **export**, or **globalize**.

A package can be declared *external-only*. This causes any symbol added to the package to be exported automatically. Since exporting of symbols should be a conscious decision, when you create an external-only package it is automatically locked. Any attempt to add a new symbol to an external-only package signals an error because it is locked. If adding the symbol would cause a name conflict in some package that uses the package to which the symbol is being added, the error message mentions that fact. Continuing from the error adds the symbol anyway. In the event of name conflicts, appropriate proceed types for resolving name conflicts are offered.

To set up an external-only package, it can be temporarily unlocked and then the desired set of symbols can be interned in it. Unlocking an external-only package disables name-conflict checking, since the system (perhaps erroneously) assumes you know what you are doing. The **global** package is external-only and locked. Its contents are initialized when the system is built by reading files containing the desired symbols with **package** bound to the **global** package object, which is temporarily unlocked. The **system** package is external-only, locked, and initialized the same way.

PART XIII.

Symbolics Common Lisp

83. Introduction to Symbolics Common Lisp

Symbolics Common Lisp (SCL) is an enhanced version of Common Lisp that contains all of the useful features of Zetalisp.

SCL is built on top of the normal Lisp Machine system, known as Zetalisp. SCL enables you to write programs that can be transported between the 3600-family machines and other machines that run Common Lisp implementations. In a future release Symbolics Common Lisp will become the standard language and Zetalisp will continue to be supported by means of a compatibility package.

Zetalisp is gradually being modified to make it more compatible with Common Lisp, while at the same time, SCL is being enhanced. The changes to Zetalisp and the availability of SCL provide an environment for you to easily convert from Zetalisp to SCL.

Common Lisp programs running under SCL and Zetalisp programs can call each other freely. The two languages use the same data structures, with one important exception, strings. See the section "SCL and Strings", page 634.

SCL and Zetalisp share the same interpreter, compiler, and other tools. Both the interpreter and the compiler use lexical scoping.

Syntactic differences between Common Lisp and Zetalisp are handled by the reader/printer control variables, such as **ibase**, **base**, **readtable**, and **package**. In Common Lisp programs these variables appear under the names ***read-base***, ***print-base***, ***readtable***, and ***package***. The binding of these variables is controlled automatically by the system.

Most Zetalisp functions, special forms, and facilities are available in SCL. Some of them, such as the **defstruct** macro, have been modified to make them compatible with Common Lisp. When the SCL documentation refers to the Zetalisp implementation, you should refer to the Symbolics documentation for more information.

For information about using Symbolics Common Lisp: See the section "Using SCL", page 633..

For a description of the differences between SCL and Common Lisp as described in the Digital Press edition of the *Common Lisp* manual (*CLM*) by Guy Steele: See the section "SCL and Common Lisp Differences", page 637.

84. Using SCL

This section describes how to use Symbolics Common Lisp (SCL). SCL can be used on 3600s, 3670s, and 3640s. Unless stated otherwise, the information applies to both interpreted and compiled code.

84.1 SCL Packages

SCL provides a separate set of packages for Common Lisp. When the two languages have a feature in common, some of the symbols in these packages are identical to symbols in Zetalisp. Other symbols are specific to Common Lisp.

The symbols in SCL can be found in the **common-lisp** and **symbolics-common-lisp** packages. The **common-lisp** package contains all the symbols defined in the Common Lisp language, while the **symbolics-common-lisp** package contains those symbols plus the symbols that are Symbolics extensions to Common Lisp.

The following packages are provided by SCL:

common-lisp This package exports all symbols defined by the Common Lisp language, other than keywords. It is also known by the names **common-lisp-global**, **lisp**, and **cl**. All Common Lisp packages inherit from the **common-lisp** package. The Common Lisp name for this package is **lisp**.

symbolics-common-lisp
This package exports all the symbols that are either in the Common Lisp language or are Symbolics extensions to Common Lisp. Most of the internals used by SCL are in this package. It is also known by the name **scl**.

common-lisp-user
This is the default package for user programs. It is also known by the names **user** and **cl-user**.

common-lisp-user inherits from **symbolics-common-lisp**. User programs should be placed in the **common-lisp-user** package, rather than the **common-lisp** package, to insulate them from the internal symbols of SCL. The Common Lisp name for this package is **user**.

common-lisp-system
This package exports a variety of 3600-specific architectural and implementational symbols. It is also known by the name **cl-sys**. In Zetalisp, some of these symbols are in **global** and some are in

	system. **common-lisp-user** does not inherit from **common-lisp-system**. The Common Lisp names for this package are **system** and **sys**.
gprint	This package contains portions of SCL concerned with the printing of Lisp expressions. It is not a standard Common Lisp package.
language-tools	This package contains portions of SCL concerned with Lisp code analysis and construction. It has the nickname **lt**. It is not a standard Common Lisp package.
zl	The name **zl** can be used in a Common Lisp program to refer to Zetalisp's **global** package. The name **zetalisp** is synonymous with **zl**.
zl-user	The name **zl-user** can be used in a Common Lisp program to refer to Zetalisp's **user** package.

SCL and Zetalisp share the same keyword package.

Common Lisp packages can be referred to by their Common Lisp names from Common Lisp programs, but not from Zetalisp programs. These names are relative names defined by the **common-lisp** package.

All Zetalisp packages can be referred to from a Common Lisp program. Those packages that have the same name as a Common Lisp package, such as **system** and **user**, can be referenced with a multilevel package prefix, for example, **zl:user:foo**. **zl-user:foo** is synonymous with **zl:user:foo**.

Packages can be used to shadow Common Lisp global symbols. For example, if you have a program in which you would like to use **merge** as the name of a function, you put the program in its own package (separate from **cl-user**), specify **:shadow merge** in the **defpackage**, and use **lisp:merge** to refer to the SCL **merge** function.

84.2 SCL and Strings

Zetalisp uses integers to represent characters, while Common Lisp has a special data type for characters. Because a string is an array of characters, Zetalisp and Common Lisp have incompatible strings. This is the only data incompatibility you encounter when using SCL. It should not be an issue for strict Common Lisp programs, only for programs that use Zetalisp facilities such as the window system.

In general an error occurs if you pass a Common Lisp string to a Zetalisp function or a Zetalisp string to a Common Lisp function. Certain Zetalisp functions such as **format** have been made to accept Common Lisp strings for convenience in programming. The SCL **string** function converts a Zetalisp string to a Common

Lisp string. Likewise, the Zetalisp **string** function converts a Common Lisp string to a Zetalisp string.

Each language prints its own strings surrounded by double quote (") characters and prints the strings of the other language preceded by #" and followed by ". This syntax is not accepted on input and is regarded as a temporary measure to aid SCL users in dealing with the string incompatibility, not as a permanent language feature.

Because of the current incompatibility of characters and strings, Common Lisp programs must do I/O operations using the Common Lisp functions, not by sending messages directly to streams. For example, you should call the **read-char** function or the **read-byte** function rather than sending the **:tyi** message, and call the **write-string** function rather than sending the **:string-out** message. In a future release, Zetalisp will be modified to be compatible with Common Lisp and this incompatibility will vanish.

84.3 SCL and Symbolics Common Lisp Extensions

Most of the language features of Zetalisp that are not in Common Lisp are provided by SCL in the **symbolics-common-lisp** package. This includes such things as processes, **loop**, and flavors. In some cases (**string-append**, for example) these Zetalisp features have been modified to make them implementationally or philosophically compatible with Common Lisp. In most cases you can refer to the documentation for information about these features.

84.4 SCL and Optimization

Some of the optimizations that are described in Steele's *Common Lisp* manual *(CLM)* and other useful optimizations have not yet been implemented in SCL. This document does not describe any specific optimizations or the lack of them. Some of these optimizations will be implemented in the future and others might not. Additionally, other optimizations that are not suggested in the *CLM* might be added to SCL in the future.

84.5 SCL and Common Lisp Files

The file attribute line of a Common Lisp file should be used to tell the editor, the compiler, and other programs that the file contains a Common Lisp program. The following file attributes are relevant:

Syntax — The value of this attribute can be Common-Lisp or Zetalisp. It controls the binding of the Zetalisp variable **readtable**, which is known as ***readtable*** in Common Lisp.

Package — **user** is the package most commonly used for Common Lisp programs. You can also create your own package. Note that the Package file attribute accepts relative package names, which means that you can specify **user** rather than **cl-user**.

The following example shows the attributes that should be in an SCL file's attribute line:

```
;;; -*- Mode:Lisp; Syntax:Common-Lisp; Package:USER -*-
```

84.6 SCL Documentation

The SCL documentation is based on the Guy Steele's *Common Lisp* manual *(CLM)* and on the Release 6.0 documentation. In addition, the Release 5.2 and Release 6.0 Beta Test Notes provide additional information about changes to Zetalisp.

For a list of all the differences between the *CLM* and SCL in the same order and by the same topic name as in the *CLM*: See the section "SCL and Common Lisp Differences", page 637. This information is subject to change as the development of SCL continues.

Additionally, all Lisp objects referred to in the SCL documentation are indexed; thus, you can find information either by looking up the topic that corresponds to the *CLM* or by looking up the names of objects.

85. SCL and Common Lisp Differences

The sections in this chapter describe the differences between SCL and the language specification presented in Steele's *Common Lisp* manual *(CLM)*. The sections in this chapter are arranged in the same order as are those in the *CLM*. Where it is appropriate to do so, the same information is provided in more than one place.

This list of differences will change in the future. Many of the items that are currently marked as incompatible will be changed to be compatible with Common Lisp.

Chapter 2 - Data Types

All atoms (non-lists) that are not symbols are self-evaluating, although the *CLM* only requires that bit-vectors, numbers, characters, and strings be so.

This difference is compatible.

Section 2.2.5 - String Characters

The type string is implemented to be a subtype of the type common. The type string-char is not a subtype of the type common.

Section 2.5.2 - Strings

Each language prints its own strings surrounded by double quote (") characters and prints the strings of the other language preceded by #" and followed by ". This syntax is not accepted on input and is regarded as a temporary measure to aid SCL users in dealing with the string incompatibility, not as a permanent language feature. In a future major release, there will be only one kind of string.

For example, an array of fixnums prints as #"**foo**" rather than "**foo**". The variable **zl:si:*flag-wrong-type-strings*** has been set to **t** to enable this. However, in the future, the #" syntax will be used to mean something else.

This difference is compatible.

Section 4.8 - Type Conversion Function

Some uses of the SCL **coerce** function generate errors. This is not an incompatibility with the *CLM*; the following examples are provided only for clarification.

Some examples that generate errors and the reasons for the errors are shown below.

The following is an error because the length is specified.

(coerce '(1 2 3) '(vector t 3))

The following is an error because the length is specified and does not match the number of elements for *object*.

(coerce '(1 2 3) '(vector t 4))

The following is an error because *object* is not a subtype of the type sequence.

(coerce x '(array single-float 2))

The following is an error because *result-type* is a subtype of the type character.

(coerce #\A 'string-char)

The following is an error because *object* is not an element of *result-type*.

(coerce #\c-A 'string-char)

The following is an error because a range is specified.

(coerce 22/7 '(float 0 10))

Section 5.1.3 - Special Forms

SCL does not provide some of the equivalent macro definitions for special forms described in *CLM* as macros but implemented in SCL as special forms.

This difference is incompatible.

Section 5.2.2 - Lambda-Expressions

The arguments for **&rest** parameters have dynamic extent. Furthermore, these arguments should not be modified with the **rplaca** or **rplacd** functions.

Section 6.2.2 - Specific Data Type Predicates

SCL includes the predicate **equal-typep**, which returns either **t** or **nil**. The arglist is *type1 type2*; both arguments must be recognized types.

The predicate **functionp** returns **nil** if its argument is a symbol that is not defined as a function. Similarly, **functionp** returns **nil** if the argument is a Zetalisp special form. Thus, **functionp** is not true of all values returned by the **function** special form; for example, **functionp** returns **nil** if the result of evaluating **(function setq)** is used as an argument. It is not clear whether this difference actually constitutes a discrepancy from the *CLM* specification; this is being checked.

Section 7.1.1 - Reference

The **function** special form can return objects that are not functions; for example, **function** can return the contents of the function cell of a macro.

Section 7.2 - Generalized Variables

In the complex version of **defsetf**, optional arguments are not fully implemented. The arguments can have defaults, but the defaults do not necessarily depend on the arguments to the left, because at the point where they are processed the "lambda variables" have not yet been bound. **supplied-p** variables are ignored completely.

The **documentation** function cannot be used for the *place* argument for **setf**.

These differences are incompatible.

Section 7.4 - Simple Sequencing

In compiled code, the macros **prog1** and **prog2** do not enforce the restriction that they must always return exactly one value. They can return no values, one value, or several values, such as in the case where *form** is empty.

The macro **progv** binds excess variables to **nil** if more variables than values are supplied.

These differences are incompatible.

Section 7.7 - Blocks and Exits

The **return** and **return-from** special forms return no values rather than returning **nil** when the *result* form is omitted. **return** and **return-from** allow multiple subforms.

This is a incompatible difference.

Section 7.8.5 - The "Program Feature"

The **go** special form does not accept integers as tags in the compiler.

This difference is incompatible.

Section 7.9.2 - Rules Governing the Passing of Multiple Values

The restriction that compiled and interpreted code always return a single value in a singleton **cond** clause is not enforced. For example, the following expression returns the values 1 and 2.

```
(cond ((values 1 2)))
```

This difference is incompatible.

Chapter 8.1 - Macro Definition

defmacro treats the following form:

```
(defmacro foo x ...)
```

as equivalent to the following:

```
(defmacro foo (& rest x) ...)
```

Likewise, the form **(defmacro foo (& rest x) ...)** is equivalent to **(defmacro foo x ...)**.

Chapter 9 - Declarations

declare forms at the top level are handled according to the Zetalisp and Maclisp rules rather than following the Common Lisp rules that they are an error. Declarations that are embedded inside a form, where allowed by Common Lisp, are evaluated according to Zetalisp rules. Thus, **special** declarations embedded inside a form are usually compatible with Common Lisp.

Section 9.1 - Declaration Syntax

The compiler ignores the *value-type* argument in the **the** special form. This difference is compatible.

The **proclaim** function is implemented so that its forms at the top level are evaluated unconditionally at compile time. The CLM does not state specifically how this evaluation should be done.

Section 9.2 - Declaration Specifiers

The declarations **type**, **ftype**, **inline**, **notinline**, **optimize**, and **declaration** are not implemented. In general all Common Lisp declarations other than **special** are ignored.

This is a minor incompatible difference.

Section 9.3 - Type Declaration for Forms

The *value-type* argument in **the** is ignored by the compiler.

This difference is compatible.

Section 11.3 - Translating Strings to Symbols

Package-name lookup is not case-sensitive.

This difference is incompatible.

Section 11.8 - Modules

Using one argument with **require** is the equivalent of using **(make-system** *module-name* **:noconfirm)** if the module is not already loaded. Thus, the "registry" of module names consists of the files in the site directory.

This difference is compatible.

Section 13.2 - Predicates on Characters

standard-char-p does not signal an error if given a non-character.

This difference is compatible.

Section 14.3 - Modifying Sequences

The **substitute**, **substitute-if**, and **substitute-if-not** functions are not optimized for detecting the case in which they can return just their argument. This difference is compatible.

delete-duplicates supports the use of the **:replace** keyword. In addition to removing duplicates from the front of the list, the

element that stays is moved up to the position of the element that is deleted. **:replace** is not meaningful if **:from-end t** is also used. This is an SCL extension to Common Lisp.

Section 14.4 - Searching Sequences for Items

The function **mismatch** erroneously returns **nil** in some cases. This will be changed.

Chapter 14 - Sequences, Chapter 15 - Lists

All sequence and list functions that take a two-argument predicate (such as **:test** and **:not-test**) always keep the order of arguments to the predicate consistent with the order of arguments to the sequence or list function. Thus, when there are two sequences and the predicate is called with one item of each, the first argument to the predicate is an element of the first sequence. When there is an item and a sequence, the first argument to the predicate is the item. When there is one sequence and two elements of it are compared, they are always compared in the order they appear in the sequence.

This is not a discrepancy from the *CLM*; this information is provided for clarification.

Section 15.2 - Lists

The macros **push** and **pushnew** take the keyword **:area**. This is an SCL extension to Common Lisp.

Chapter 16 - Hash Tables

When the **:test** argument to **make-hash-table** is **eq** or **#'eq**, **:rehash-threshold** cannot be used. When **:test** is any other value, **:rehash-size** cannot be an integer.

This difference is incompatible.

Chapter 17 - Arrays

This section describes the *CLM* and SCL differences about arrays in general.

- Multidimensional arrays currently use column-major order. This difference is incompatible.

- Except in one case, all arrays are adjustable. The exception is that indirect arrays that are created shorter than a certain length cannot be made larger than that size.

- When using arrays, you cannot use the circular-structure labelling feature; thus, you cannot access an object before it has been created. For example, the following form signals an error when **#1#** is read.

```
;;;this signals an error because the object
;;;is being accessed before it is created
#( #1= #( #1# ) )
```

However, it is possible to access the object within the same form when using lists, as shown in the following example:

```
( #1= ( #1# ) )
```

This works for lists because they are built up out of conses, which is not true of arrays.

This difference is incompatible.

Section 17.3 - Array Information

The function **array-row-major-index** is not implemented.

This difference is incompatible.

Chapter 18 - Strings

Each language prints its own strings surrounded by double quote (") characters and prints the strings of the other language preceded by #" and followed by ". This syntax is not accepted on input and is regarded as a temporary measure to aid SCL users in dealing with the string incompatibility, not as a permanent language feature. In a future major release, there will be only one kind of string.

For example, an array of fixnums prints as #"**foo**" rather than "**foo**". The variable **zl:si:*flag-wrong-type-strings*** has been set to **t** to enable this. However, in the future, the #" syntax will be used to mean something else.

This difference is compatible.

Chapter 19 - Structures

The default printing of structures does not use **#S**.

This difference is incompatible.

Section 19.5 - Defstruct Options

The expression **(defstruct (foo (:type vector) :named) ...)** makes a named-structure. The *CLM* specifies that this should be a vector whose first element is the type symbol. This variation is incompatible only in the printed representation and in the returned value from the **type-of** function.

Section 19.6 - By-position Constructor Functions

The constructor does not evaluate **defstruct** slot initializations in the appropriate lexical environment.

Section 20.2 - The Top-Level Loop

In SCL, a top-level form that returns no values does not set the variable *. The variable * remains unchanged.

This difference is incompatible.

Section 21.2 - Creating New Streams

The functions **make-concatenated-stream** and **make-echo-stream** are not implemented.

This difference is incompatible.

Section 22.2.1 - Input from Character Streams

The **read-char** function echoes the character read from the input stream if it is the terminal.

The second value returned by **read-from-string** is at most the length of the string; it is never one greater than the length of the string.

These differences are compatible.

Section 22.1.1 - What the Read Function Accepts

SCL uses the Zetalisp rules for vertical bars. It supports a maximum of two vertical bars (|) in a token. The vertical bars are boundaries for the token; all characters must be contained between the two vertical bars. Any character outside the pair of vertical bars is treated as a separate symbol. Thus, **a|b|c** is treated as three symbols and **|abc|** is treated as one symbol.

This difference is incompatible.

Section 22.1.2 - Parsing of Numbers and Symbols

Setting the value of ***read-base*** greater than 10 causes tokens to fail to be interpreted as numbers rather than symbols. For example, if ***read-base*** is set to 16 (hexadecimal radix), variables with names such as **a**, **b**, and **face** are interpreted as symbols rather than numbers. You can set the values of the variables ***read-extended-ibase-signed-numbers*** and ***read-extended-ibase-unsigned-numbers*** to **t** to cause the tokens to always be interpreted as a number. This difference is incompatible.

The variable ***read-suppress*** is not implemented.

The **set-syntax-from-char** function can copy most character attributes rather than being limited to the standard character syntax types shown in Table 22-1, Standard Character Syntax Types.

SCL does not implement the requirements in Table 22-3, Standard Constituent Character Attributes, about *illegal* character attributes. Changing the syntactic type of space, tab, backspace, newline (also called return), linefeed, page, or rubout to *constituent* or *non-terminating macro* type does not signal an error in SCL.

Section 22.1.3 - Macro Characters

#P is used for printing pathnames and is followed by a string in double quotes.

This difference is compatible.

Section 22.1.4 - Standard Dispatching Macro Character Syntax

Symbols in the ***features*** list must be keywords for the reader macros #+ and #- to work with them. The #+ and #- reader macros read the *feature* that follows them in the keyword package, not in the package that is currently in effect.

It is not clear whether this difference actually constitutes a discrepancy from the *CLM* specification; this is being checked.

Section 22.1.5 - The Readtable

The **get-macro-character** function returns **nil** for built-in macros, as they are not defined with **set-macro-character**. This causes the example for the **read-delimited-list** function in the *CLM* under **read-delimited-list** to not work.

This difference is incompatible.

Section 22.1.6 - What the Print Function Produces

SCL uses the Zetalisp names (the names on the keyboard), rather than the names shown in the *CLM*, for the printing of characters. This is not completely compatible with Common Lisp.

Slashification is controlled by which tokens the SCL reader interprets as numbers. Only symbols whose printed representations are actual numbers get slashified on printing. A symbol whose printed representation is a potential number and not an actual number does not get slashified. Potential numbers are described in Section 22.1.2, *Parsing of Numbers and Symbols*, in the *CLM*. This difference is incompatible.

Section 22.3.3 - Formatted Output to Character Streams

The ~G directive is not implemented for the **format** function. The ~E directive is the Zetalisp implementation.

SCL implementation of the ~T directive does not know the column position when the output is directed to a file.

These differences are incompatible.

Section 23.1.1 - Pathnames

Pathname components of **:unspecific** for the device, directory, type, and version components are allowed in some circumstances.

Pathname hosts are instances; they are not strings or lists of strings. The host component of a pathname should be considered to be a structured component. This difference is incompatible and will not be changed.

Section 23.1.2 - Pathname Functions

The **parse-namestring** function uses the new (Release 5.0) Zetalisp rules for **fs:parse-pathname** regarding what a non-null host means, rather than the rules shown in the *CLM*. Thus, when *host* is not **nil**, *thing* should not contain a host name. This difference is compatible.

The **:junk-allowed** keyword for **parse-namestring** is not implemented to accept **t** as an argument. This difference is incompatible.

The variable ***default-pathname-defaults*** is a defaults alist, not a pathname. This difference is incompatible and will be changed.

Section 23.5 - Accessing Directories

The **directory** function returns **nil** if no files matching *pathname* are found, but still signals an error for other file lookup errors, such as not finding the directory.

This difference is compatible.

Section 25.1 - The Compiler

The **compile-file** function accepts the keyword **:load**. This is provided for compatibility with Spice Lisp.

This difference is compatible.

Section 25.2 - Documentation

The **documentation** function is the Zetalisp implementation. This difference is incompatible.

Section 25.3 - Debugging Tools

The **describe** function is the Zetalisp **describe**; it returns its argument after describing the object. This difference is incompatible.

The function **dribble** calls **zl:dribble-start** and **zl:dribble-end**. This means that **dribble** does not return until the dribbling has been completed, because it creates a new command loop to do the dribbling. This difference is compatible.

Section 25.4.1 - Time Functions

The *time-zone* argument for the **decode-universal-time** function currently suppresses checking for daylight savings time, just as the **encode-universal-time** function does. It is not clear whether this difference actually constitutes a discrepancy from the *CLM* specification; this is being checked. However, the compatibility note in the *CLM* about *time-zone* in Zetalisp is obsolete. **get-decoded-time** now returns *time-zone*.

Section 25.4.2 - Other Environment Inquiries

Symbols in the ***features*** list must be keywords for the reader

macros #+ and #- to work with them. The reader macros #+ and #- read the *feature* that follows them in the keyword package, not in the package that is currently in effect.

The feature name **ieee-floating-point** is not yet implemented for the ***features*** variable.

Index

λ λ λ
Lambda (λ) character 32

γ γ γ
Gamma (γ) character 32

δ δ δ
Delta (δ) character 32

↑ ↑ ↑
Up-Arrow (↑) character 32

± ± ±
Plus-Minus (±) character 32

⊕ ⊕ ⊕
Circle-plus (⊕) character 32

⊗ ⊗ ⊗
Circle-X (⊗) character 25

≠ ≠ ≠
≠ function 100

≤ ≤ ≤
≤ function 99

≥ ≥ ≥
≥ function 99

# # #
#◊ Reader Macro 30
Sharp-sign (#) macro character 26
#: package qualifier 584
#: Reader Macro 29
#- Reader Macro 30

SCL and #- Reader Macro 644, 645
reader macros 27
#" syntax 637, 642
#' Reader Macro 28
#' special form 162
#+ Reader Macro 29
SCL and #+ Reader Macro 644, 645
#, Reader Macro 28
#. Reader Macro 29
#/ character identifier 277
#\ or #/ Reader Macro 27
#< Reader Macro 30
#b Reader Macro 29
#m Reader Macro 29
#n Reader Macro 29
#o Reader Macro 29
#P Reader Macro 643
#q Reader Macro 29
#r Reader Macro 29
SCL and #S macro form 642
#x Reader Macro 29
#\ character identifier 277
#\ or #/ Reader Macro 27
#^ Reader Macro 28
#| Reader Macro 30

&
& keywords 151, 309

'
Single quote (') 161
Quote (') macro character 26

*
SCL and * variable 642
* function 102

+
+$ function 100
+ function 100

,
,. 347
Comma character (,) in backquote facility 345
Comma (,) macro character 26
,@ 345

-

-
-$ function 101
- function 101

-

/

/
//$ function 103
// function 102

/

1

1
1-$ function 105
1+$ function 104
1+ function 104
sys: %1d-aloc function 246
sys: %1d-aref function 246
sys: %1d-aset function 246
1- function 104

1

3

3
%32-bit-difference function 119
%32-bit-plus function 119
32-bit Numbers 119
Addition of 32-bit numbers 119
Subtraction of 32-bit numbers 119

3

:

:
: character as keyword identifier 576
: package qualifier 585, 601, 603

:

;

;
Semicolon (;) macro character 26

;

<

<
<= function 99
< function 99

<

=

=
= function 98

=

>

>
>= function 98
> function 98

>

A A A

Function abbreviation 28
ABORT 532
Abort character 32
sys: **abort** flavor 502, 526, 532
ABORT key 502, 526
Getting Information About an Array 246
above loop keyword 207, 225
abs function 101
Absolute value 101
Combining abstract types 431
:abstract-flavor Option for **defflavor** 441, 450
Abstract-operation functions 418
Abstract types 418, 425
&-Keywords Accepted by **defmacro** 373
Clm: What the Read Function Accepts 643
fs: **access-error** 549
fs: **access-error** flavor 549
Access functions 147
Clm: Accessing Directories 645
Accessing Multidimensional Arrays as One-dimensional 245
Accessor functions 379, 401
:accessor-prefix Option for **defflavor** 441, 448
Active elements in arrays 238, 247
Active frame 494
Interesting active frame 494
Actual parameters 151
add1 function 104
Adding new methods 438
Adding new symbols to locked packages 626
Adding to the End of an Array 251
Addition 100, 104
Addition of 32-bit numbers 119
adjust-array-size function 249
SCL and adjustable arrays 641
advise special form 302
Character Comparisons Affected by Case, Style, and Bits 270
String Comparisons Affected by Case, Style, and Bits 282
String Searching Affected by Case, Style, and Bits 286
:after method type 455, 457
After-daemon methods 431
Aggregated Boolean Tests 215
Aid for Debugging Macros 369
Aids for Defining Macros 343
Alist 59
si:encapsulated-definition debugging info alist element 325
Debugging info alist functions 302, 323, 325
all-flavor-names variable 429
Funargs and Lexical Closure Allocation 139, 140, 141, 312, 313
Storage allocation 250
Storage allocation error 538
Memory allocation of conses 56
&allow-other-keys Lambda-list Keyword 310, 373
How the Package System Allows Symbol Sharing 569
aloc function 244
alpha-char-p function 270
si: **alphabetic** syntax description 34

alphabetic-case-affects-string-comparison variable 273
alphalessp function 292
alphanumericp function 270
Using the Constructor and Alterant Macros 395
:alterant option for **defstruct** 385, 386
Alterant Macros 397
Alteration of List Structure 54
Altering slot values of structures 395
Altmode character 32
always loop keyword 215
Examples of Symbol Sharing Among Packages 589
Sharing of Symbols Among Packages 569
Logical **and** function 178
and loop keyword 207, 210, 216, 222
:and method combination type 455
:and method type 455, 458
and special form 178
ap-1 function 244
ap-2 function 245
ap-leader function 245
append loop keyword 212
:append method combination type 455, 456
append function 51, 56
appending loop keyword 212
Application: Handlers Examining the Stack 494
Reference Material: Application: Handlers Examining the Stack 495
How Applications Programs Treat Conditions 481
apply function 8, 151, 159
Applying functions to list items 201
ar-1 function 244
ar-2 function 244
Arctangent 106, 107
Printed Representation of Arrays That Are Named Structures 16
Printed Representation of Arrays That Are Not Named Structures 17
:area init option for **si:eq-hash-table** 71
:area keyword for **make-list** 49
:area option for **make-array** 241
sys: **area-overflow** flavor 537
aref 255
aref function 235, 244, 277
%%arg-desc-interpreted numeric argument descriptor field 324
%arg-desc-interpreted numeric argument descriptor field 324
%%arg-desc-max-args numeric argument descriptor field 324
%%arg-desc-min-args numeric argument descriptor field 324
%%arg-desc-quoted numeric argument descriptor field 324
%%arg-desc-rest-arg numeric argument descriptor field 324
arg function 165
arglist declaration 312
arglist function 302, 323
arglist variable 325
%args-info function 325

args-info function 324
SCL and **documentation** as **setf** argument 639
Numeric argument descriptor 324
%%arg-desc-interpreted numeric argument descriptor field 324
%arg-desc-interpreted numeric argument descriptor field 324
%%arg-desc-max-args numeric argument descriptor field 324
%%arg-desc-min-args numeric argument descriptor field 324
%%arg-desc-quoted numeric argument descriptor field 324
%%arg-desc-rest-arg numeric argument descriptor field 324
environment argument for macro-expander functions 138
SCL and *value-type* argument for **the** special form 640
Keywords in argument lists 189
:macro argument to **setsyntax** 36
nil argument to **setsyntax** 36
:single argument to **setsyntax** 36
:splicing argument to **setsyntax** 36
:atom argument to **typep** 9
:fix argument to **typep** 9
:float argument to **typep** 9
:instance argument to **typep** 9
:list-or-nil argument to **typep** 9
:non-complex-number argument to **typep** 9
:null argument to **typep** 9
:number argument to **typep** 9
Binding Parameters to Arguments 153
Checking for valid arguments 505
Functional arguments 139
Lexical Environment Objects and Arguments 138
Safety of **&rest** Arguments 157
Arguments to functions 323
argument-typecase special form 506
Arithmetic 100
Performing arithmetic operations on characters in SCL 265
Base Flavor: **sys:** **arithmetic-error** 534
sys: **arithmetic-error** flavor 534
Arithmetic errors 534
Adding to the End of an Array 251
art-fat-string array 277
art-string array 277
Changing the Size of an Array 249
Copying an Array 252
Copying From and to the Same Array 253
Getting Information About an Array 246
Indirect array 240, 241, 248
Named structure array 238, 241
Maclisp Array Compatibility 262
The Array Data Type 5
Bit size of array elements 236
Character strings as array elements 236
Integers as array elements 236
Returning array elements 244
Storing into array elements 244
Basic Array Functions 241
Array header information 239
Clm: Array Information 642
Array Register Restrictions 258
Hints for Using Array Registers 257
Array Registers and Performance 255

Array Representation Tools 237
:array returned by **typep** 9
Multidimensional array subscripts 245
art-fat-string Array Type 236
art-q-list Array Tpe 236
art-16b array type 236
art-1b array type 236
art-2b array type 236
art-4b array type 236
art-8b array type 236
art-boolean Array Type 237
art-*N*b Array Type 236
art-q Array Type 236
art-string Array Type 236
array-#-dims function 247
array-active-length function 247
array-bits-per-element function 238
array-bits-per-element variable 238
array-column-major-index function 248
array-dimension-n function 247
array-displaced-p function 248
array-element-size function 238
array-elements-per-q function 238
array-elements-per-q variable 238
array-has-leader-p function 248
sys: **array-has-no-leader** flavor 540
array-in-bounds-p function 248
array-indexed-p function 248
array-indirect-p function 248
array-leader-length function 249
array-push-extend function 251
array-push-portion-extend function 251
sys: **array-register-1d** 245
sys: **array-register-1d** declaration 312
SCL and **array-row-major-index** function 642
sys: **array-wrong-number-of-dimensions** flavor 540
sys: **array-wrong-number-of-subscripts** flavor 541
arraycall function 262, 263
Array decoding 255
Array dimensions 235, 247
array-dimensions function 247
arraydims function 247
array-element loop iteration path 225
array-elements loop iteration path 225, 257
Array Errors 540
***array** function 263
array-grow function 249
Array initialization 241
Array leader 238, 241, 248
array-leader function 245
Array Leaders 238
array-length function 246
array macro 263
arrayp function 8, 69
array-pop function 252
array-push function 251
sys: **array-register** 255
sys: **array-register** declaration 312

Array Registers 255
Array-register variable 255
Arrays 235, 303
Active elements in arrays 238, 247
Clm: Arrays 641
Dead arrays 262
Displaced Arrays 239, 241, 248
Extra Features of Arrays 238
Flonum arrays 262
Grouped Arrays 401
Indirect Arrays 240
Integer arrays 262
Multics external arrays 262
Multidimensional Arrays 237
Numeric arrays 5
SCL and adjustable arrays 641
SCL and column-major order for arrays 641
SCL and multidimensional arrays 641
Sorting arrays 79
Storage of arrays 235
Un-garbage-collected arrays 262
SCL arrays and circular-structure labelling restriction 641
Sorting Arrays and Lists 79
Arrays as functions 235
Arrays as lists 236
Accessing Multidimensional Arrays as One-dimensional 245
Arrays Overlaid with Lists 250
Printed Representation of Arrays That Are Named Structures 16
Printed Representation of Arrays That Are Not Named Structures 17
Arrays used as functions 305
Arrays, Characters, and Strings 233
Array subscripts 235, 248
array-type function 246
Array Types 235, 241
array-types function 237
array-types variable 237
art-fat-string Array Type 236
art-fat-string array 277
art-q-list Array Type 236
art-16b array type 236
art-1b array type 236
art-2b array type 236
art-4b array type 236
art-8b array type 236
art-boolean Array Type 237
art-*N*b Array Type 236
art-q Array Type 236
art-string Array Type 236
art-string array 277
as loop keyword 207
as-1 function 244
as-2 function 244
ascii-to-char function 274
ascii-to-string function 290
ASCII Characters 274
ascii-code function 274
ascii function 293
ASCII Strings 290

aset 255
aset function 235, 244
ash function 115
ass function 65
assoc function 65
Symbol associated with property list 67
Association list 238
Association lists 59, 64, 67
assq function 64
atan2 function 107
atan function 107
Atom 3
:atom argument to **typep** 9
atom function 7
Atomic symbol 3
Self-evaluating atoms and SCL 637
Attribute 67
Package attribute 579
SCL and *constituent* character attribute 643
SCL and *non-terminating macro* character attribute 643
File attribute list 579
SCL and *illegal* character attributes 643
SCL and Standard Constituent Character Attributes Table 643
&aux keyword for **defmacro** 373
&aux Lambda-list Keyword 310
Specifying Aux-variables in Lambda Lists 157

B B B

Back-Next character 32
Backquote 345
Backquote character (`) 345
Comma character (,) in backquote facility 345
Backquote (`) macro character 26
Backspace character 32
dbg: **bad-array-mixin** flavor 540
sys: **bad-array-type** flavor 540
sys: **bad-connection-state** flavor 555
sys: **bad-data-type-in-memory** flavor 534
SCL and vertical bars for quoting 643
Miscellaneous System Errors Not Categorized by Base Flavor 536
Base Flavor: **sys:arithmetic-error** 534
Base Flavor: **sys:cell-contents-error** 533
Base Flavor: **sys:floating-point-exception** 535
Request Failures Based on **fs:file-request-failure** 546
Interning Errors Based on **sys:package-error** 542
Base flavor 451
Reading Integers in Bases Greater Than 10 21
base variable 14
Basic Array Functions 241
Basic definition of the function spec 297, 325
Basic List Operations 46
Basic String Operations 278
Basic Objects 1
:before method type 455, 457
Before-daemon methods 431
being loop keyword 207, 222
below loop keyword 207, 225

:bignum returned by **typep** 9
Bignums 4, 8, 89
bigp function 8
Read rational number in binary 29
Binary files 613
Binary integers 89
Condition Bind Recursion 502
Binding 125, 331, 561
Binding condition handlers 481
Binding local and special variables 126
Binding of a symbol 3
Binding Parameters to Arguments 153
Special Forms for Binding Variables 128
Binding handlers 487
Bindings 210
Bindings in loops 210, 220
Binding Variables 125, 331
Trap on exit bit 523
Bit size of array elements 236
sys: **bitblt-array-fractional-word-width** flavor 537
sys: **bitblt-destination-too-small** flavor 537
bitblt errors 537
bitblt function 254
Bit manipulation 114
Character Comparisons Affected by Case, Style, and Bits 270
Character Comparisons Ignoring Case, Style, and Bits 271
Least bits 115
Rotate bits 115
Shift bits 114
Significant bits 115
String Comparisons Affected by Case, Style, and Bits 282
String Comparisons Ignoring Case, Style, and Bits 283
String Searching Affected by Case, Style, and Bits 286
String Searching Ignoring Case, Style, and Bits 287
bit-test function 114
block special form 183
Blocks 67
Blocks and Exits 183
Clm: Blocks and Exits 639
&body keyword for **defmacro** 373
&body Lambda-list Keyword 310
Aggregated Boolean Tests 215
Boolean operations 113
Truth table for the Boolean operations 114
boole function 114
both-case-p function 270
Bound handlers 488, 501, 502
boundp-in-closure function 336
boundp function 562
Break on exit from marked frame message 523
si: **break** syntax description 34
Break character 32
break flavor 532
breakon function 523
Breakpoints 523
Conditional breakpoints 523
BS character 32
Debugger Bug Reports 524

:bug-report-description message 525
:bug-report-recipient-system message 524
bug function 524
BUG-LISPM mailing list 524
:but-first option for **defstruct** 385, 390
butlast function 53
by loop keyword 207, 225
Clm: By-position Constructor Functions 642
By-position Constructor Macros 396
Byte 115
Using Byte Fields and **defstruct** 399
Byte Manipulation Functions 115
Create a byte specifier 116
Extract position field of a byte specifier 116
Extract size field of a byte specifier 116
%%ch- byte specifiers and file characters 268
%%kbd- byte specifiers and keyboard characters 268
byte function 116
byte-position function 116
byte-size function 116
Byte specifiers 115, 399

C C C

c-sh-A command 302
c-sh-D command 302
caaaar function 42
caaaar function 43
caaadr function 42
caaadr function 43
caaar function 42, 43
caadar function 42
caadar function 44
caaddr function 42
caaddr function 44
caadr function 42, 43
caar function 42, 43
cadaar function 42
cadaar function 44
cadadr function 42
cadadr function 44
cadar function 42, 43
caddar function 42
caddar function 44
cadddr function 42
cadddr function 44
caddr function 42, 43
cadr function 42, 43
:callable-accessors option for **defstruct** 385, 391
Call character 32
call function 161
sys: **call-trap** flavor 523
Operations the User Can Perform on Functions 302
:proceed Can Return **nil** 519
Captured free reference 126
Car 41, 43, 44
car function 42, 83
car-location function 45

Composition of Cars and Cdrs 42
SCL case checking in package-name lookup 640
:case flavor combination 520
:case method combination 517
:case method combination type 455, 457
Case sensitivity of interning 604
Character Comparisons Affected by Case, Style, and Bits 270
Character Comparisons Ignoring Case, Style, and Bits 271
String Comparisons Affected by Case, Style, and Bits 282
String Comparisons Ignoring Case, Style, and Bits 283
String Searching Affected by Case, Style, and Bits 286
String Searching Ignoring Case, Style, and Bits 287
:case-documentation symbol 520
caseq special form 182
Catch 197
***catch** special form 169, 200
catch special form 169, 197
catch-error-restart special form 501, 502, 513, 515
catch-error-restart-if special form 513, 515
Miscellaneous System Errors Not Categorized by Base Flavor 536
cdaaar function 42
cdaaar function 44
cdaadr function 42
cdaadr function 44
cdaar function 42
cdaar function 43
cdadar function 42
cdadar function 44
cdaddr function 42
cdaddr function 44
cdadr function 42
cdadr function 43
cdar function 42, 43
cddaar function 42
cddaar function 44
cddadr function 42
cddadr function 44
cddar function 42
cddar function 43
cdddar function 42
cdddar function 44
cddddr function 42
cddddr function 44
cdddr function 42
cdddr function 43
cddr function 42, 43
Cdr 41, 43, 44
Cdr storing functions 297
Cdr-code field 56
Cdr-coding 41, 56
Cdr-coding and Locatives 83
cdr function 42, 83
Cdr-next 56
Cdr-nil 56
Cdr-normal 56
Composition of Cars and Cdrs 42
ceiling function 109
Cell 83

External value cell 331
Function cell 3
Internal value cell 331
Value cell 3, 83, 331, 561, 563
Memory cell as property list 67
Invalid contents in cell errors 533
Package cell of a symbol 3
The Function Cell of a Symbol 563
The Package Cell of a Symbol 566
The Value Cell of a Symbol 561
Base Flavor: **sys:** **cell-contents-error** 533
sys: **cell-contents-error** flavor 533
Cells and Locatives 83
%%ch- byte specifiers and file characters 268
Functions That Change Character Syntax 34
Functions That Change Characters Into Macro Characters 35
Change current package 597
change-instance-flavor function 436
fs: **change-property-failure** 551
fs: **change-property-failure** flavor 551
Changing a Flavor 468
Changing the Size of an Array 249
Changing the Value of a Variable 125
chaos package 615
char≠ function 271
char≤ function 271
char≥ function 271
char-device-font function 269
char-mouse-button function 273
char-mouse-n-clicks function 274
char-not-equal function 271
char-not-greaterp function 272
char-not-lessp function 272
char-to-ascii function 274
char< function 271
char= function 270
char> function 271
Abort character 32
Back-Next character 32
Backquote (`) macro character 26
Backspace character 32
Break character 32
BS character 32
Call character 32
Circle-plus (⊕) character 32
Circle-X (⊗) character 25
Clear character 32
Clear-Input character 32
Clear-Screen character 32
Comma (,) macro character 26
CR character 32
Delta (δ) character 32
End character 32
ESC character 32
Form character 32
Gamma (γ) character 32
Hand-Down character 32
Hand-Left character 32

Hand-Right character 32
Help character 32
Hold-Output character 32
Integral (∫) character 32
Lambda (λ) character 32
LF character 32
Line character 32
Macro character 32
Network character 32
Overstrike character 32
Page character 32
Quote character 32
Quote (') macro character 26
Resume character 32
Return character 32
Roman-I character 32
Roman-II character 32
Roman-III character 32
Roman-IV character 32
Rubout character 32
Semicolon (;) macro character 26
Sharp-sign (#) macro character 26
SP character 32
Space character 32
Stop-Output character 32
System character 32
Tab character 32
Terminal character 32
Up-Arrow (↑) character 32
Altmode character 32
Hand-Up character 32
Macro character 26
Plus-Minus (±) character 32
Status character 32
Comma character (,) in backquote facility 345
: character as keyword identifier 576
SCL and *constituent* character attribute 643
SCL and *non-terminating macro* character attribute 643
SCL and *illegal* character attributes 643
SCL and Standard Constituent Character Attributes Table 643
Character code for nonprinting characters 27
Reading octal character codes 25
Character Comparisons Affected by Case, Style, and Bits 270
Character Comparisons Ignoring Case, Style, and Bits 271
#/ character identifier 277
#\ character identifier 277
Zetalisp and SCL character incompatibilities 265
SCL and printed character names 644
Special Character Names 32
Character Object Details 265
Font numbers and character objects 267
Printed Representation of Common Lisp Character Objects 16
Character objects code field 265
Character objects font field 265
Character objects style field 265
Expanded character set 277

Support for Nonstandard Character Sets 275
Character Sets and Character Styles 267
Clm: Formatted Output to Character Streams 644
Clm: Input From Character Streams 643
Character strings as array elements 236
Character Sets and Character Styles 267
Clm: Standard Dispatching Macro Character Syntax 644
Functions That Change Character Syntax 34
SCL and Standard Character Syntax Types Table 643
Backquote character (') 345
Character codes 27, 265
Character Comparisons 270
Character constants 27
Character Conversions 272
Character Fields 269
character function 272
Character Names 273
Character Objects 265
Character Predicates 270
Characters 265
ASCII Characters 274
%%ch- byte specifiers and file characters 268
Character code for nonprinting characters 27
Clm: Macro Characters 643
Clm: Predicates on Characters 640
Clm: String Characters 637
Derived fields for characters 268
Floating-point Exponent Characters 22
Functions That Change Characters Into Macro Characters 35
How the Reader Recognizes Macro Characters 26
%%kbd- byte specifiers and keyboard characters 268
Mouse Characters 273
Special characters 32
Two Kinds of Characters 268
Performing arithmetic operations on characters in SCL 265
Functions That Change Characters Into Macro Characters 35
Displaying characters on output devices 268
Arrays, Characters, and Strings 233
Character sets 267
Character strings 277
char-bit function 269
char-bits function 269
char-code function 269
char-downcase function 272
char-equal function 271
char-flipcase function 273
char-greaterp function 272
char-int function 272
char-lessp function 271
char-name function 273
char-standard function 275
char-subindex function 269
char-upcase function 272
check-arg-type special form 576
check-arg-type macro 506
check-arg macro 505
Checking for Package Name-conflict Errors 593
Checking for valid arguments 505

SCL case checking in package-name lookup 640
:choose proceed type 619
si: **circlecross** syntax description 34
Circle-plus (⊕) character 32
Circle-X (⊗) character 25
fs: **circular-link** flavor 552
Circular list 41
circular-list function 50
SCL arrays and circular-structure labelling restriction 641
Circumflex (^) in integer syntax 20
cis function 107
cl:*read-default-float-format* variable 23
Classes of Handlers 487
Clauses 206
Iteration-driving Clauses 207, 221
Miscellaneous Other Clauses 218
cl:double-float format 23
Cleanup handler 197
:clear method of **si:heap** 78
Clear character 32
:clear-hash message 73
Clear-Input character 32
Clear-Screen character 32
cl:long-float format 23
Clm: Accessing Directories 645
Clm: Array Information 642
Clm: Blocks and Exits 639
Clm: By-position Constructor Functions 642
Clm: Creating New Streams 643
Clm: Data Types 637
Clm: Debugging Tools 645
Clm: Declaration Specifiers 640
Clm: Declaration Syntax 640
Clm: **defstruct** Options 642
Clm: Formatted Output to Character Streams 644
Clm: Generalized Variables 639
Clm: Hash Tables 641
Clm: Input From Character Streams 643
Clm: Macro Characters 643
Clm: Macro Definition 639
Clm: Modifying Sequences 640
Clm: Other Environment Inquiries 645
Clm: Parsing of Numbers and Symbols 643
Clm: Pathname Functions 644
Clm: Predicates on Characters 640
Clm: Rules Governing the Passing of Multiple Values 639
Clm: Searching Sequences for Items 641
Clm: Sequences, Lists 641
Clm: Simple Sequencing 639
Clm: Special Forms 638
Clm: Specific Data Type Predicates 638
Clm: Standard Dispatching Macro Character Syntax 644
Clm: String Characters 637
Clm: the "Program Feature" 639
Clm: the Compiler 645
Clm: the Readtable 644

Clm: the Top-level Loop 642
Clm: Time Functions 645
Clm: Translating Strings to Symbols 640
Clm: Type Conversion Function 637
Clm: Type Declaration for Forms 640
Clm: What the Print Function Produces 644
Clm: What the Read Function Accepts 643
Clm: Arrays 641
Clm: Declarations 640
Clm: Documentation 645
Clm: Lambda-expressions 638
Clm: Lists 641
Clm: Modules 640
Clm: Pathnames 644
Clm: Reference 638
Clm: Strings 637, 642
Clm: Structures 642
Closed subroutine 351
Funargs and Lexical Closure Allocation 139, 140, 141, 312, 313
Closure and flavors 423
:closure returned by **typep** 9
Unbound closure variable error 533
What is a Dynamic Closure? 331
closure-alist function 335
closure function 305, 335
closure-function function 335
Dynamic Closure-manipulating Functions 335
closurep function 9, 335
Dynamic closures 303, 305, 331
Examples of the Use of Dynamic Closures 333
Functions and Dynamic Closures 295
Lexical closures 137, 303, 305, 331
Stack-allocated closures 139
closure-variables function 336
cl package 615, 633
clrhash-equal function 74
clrhash function 74
cl:short-float format 23
cl:single-float format 23
c-M Debugger command 524
Error logging code 494
Loop exit code 205, 212
Loop initialization code 205, 212
Macros That Surround Code 362
Invalid type code error 534
Character objects code field 265
Character code for nonprinting characters 27
code-char function 270
In-line coded functions 351
Open coded functions 351
Character codes 27, 265
Reading octal character codes 25
SCL **coerce** function 637
Coercion Rules for Numbers 90
Coercion rules 90
collect loop keyword 212
collecting loop keyword 212
Hash Tables and the Garbage Collector 75

:external colon mode 601, 603
:internal colon mode 601, 603
:colon-mode Option for **defpackage** and **make-package** 601, 603
:colon-mode option for **make-package** 585
:colon-mode option for **defpackage** 585
SCL and column-major order for arrays 641
Column-major ordering 248
:case flavor combination 520
:case method combination 517
Method Combination 455
Combination Method Types 459
:and method combination type 455
:append method combination type 455, 456
:case method combination type 455, 457
:daemon method combination type 455
:daemon-with-and method combination type 455, 456
:daemon-with-or method combination type 455, 456
:daemon-with-override method combination type 455, 457
:inverse-list method combination type 455, 456
:list method combination type 455
:nconc method combination type 455, 456
:or method combination type 455
:pass-on method combination type 455, 456
:progn method combination type 455
Instance variables of combined flavors 431
:combined method type 455, 458
Combined methods 438
Zmacs Command: Edit Combined Methods 472
Zmacs Command: List Combined Methods 471
Combined method 431
Combining abstract types 431
Combining flavors 431, 461
Combining methods 461
Comma (,) macro character 26
Comma character (,) in backquote facility 345
c-sh-A command 302
c-sh-D command 302
c-M Debugger command 524
:create-directories-recursively Debugger command 549
:create-directory Debugger command 549
m-sh-D command 302
m-BREAK Debugger command 532
Where Is Symbol (m-X) Zmacs command 571
Zmacs Command: Describe Flavor 471
Zmacs Command: Edit Combined Methods 472
Zmacs Command: Edit Methods 471
Zmacs Command: List Combined Methods 471
Zmacs Command: List Methods 471
Zmacs Command: m-. 471
Debugger Special Commands 525
Package commands 612
Zmacs Commands for Flavors 471
comment special form 164
Comments in macros 26
SCL and type common 637
Greatest common denominator 105
Greatest common divisor 105

Introduction to Symbolics Common Lisp 631
Symbolics Common Lisp 629
Printed Representation of Common Lisp Character Objects 16
Common Lisp Compatibility Package 615
SCL and Common Lisp Differences 637
SCL and Symbolics Common Lisp Extensions 635
SCL and Common Lisp Files 635
Common Lisp readtable 27
Interaction of Some Common Special Forms with Multiple Values 169
common-lisp-global package 615
common-lisp-system package 633
common-lisp-user package 633
common-lisp package 633
Compact lists 56
Sorting compact lists 79
Character Comparisons 270
Cons comparisons 11
Keyword comparisons 227
Numeric Comparisons 98
Object comparisons 10
String comparisons 11, 282
Character Comparisons Affected by Case, Style, and Bits 270
String Comparisons Affected by Case, Style, and Bits 282
Character Comparisons Ignoring Case, Style, and Bits 271
String Comparisons Ignoring Case, Style, and Bits 283
Maclisp compatibility 503
Maclisp Array Compatibility 262
Readtable Functions for Maclisp Compatibility 36
Common Lisp Compatibility Package 615
Functions for Compatibility with Maclisp Lexprs 165
Compatibility with the Pre-release 5.0 Package System 625
compile-flavor-methods macro 438, 485
The Compiled Function Data Type 4
Compiled file 579
:compiled-function returned by **typep** 9
Compiled functions 303, 305
SCL and **compile-file** function 645
Clm: the Compiler 645
Lexically scoped compiler 137
Numbers in the Compiler 90
Effect of compiler on variables 126
compiler-let special form 129
compiler package 615
Complement logical operation 114
Default Handlers and Complex Modularity 509
Reference Material: Default Handlers and Complex Modularity 510
How the Reader Recognizes Complex Numbers 24
Printed Representation of Complex Numbers 15
:complex returned by **typep** 9
complex function 108
Complex magnitude 101
Complex numbers 89, 95
complexp function 8
SCL and pathname components 644
Composition of Cars and Cdrs 42
String concatenation 279
:conc-name option for **defstruct** 385, 387

SCL and returned values from **cond** macro 639
cond special form 177
cond-every special form 178
Condition 479
:document-proceed-type method of **condition** 529
:proceed-type-p method of **condition** 530
:proceed-types init option for **condition** 530
:proceed-types method of **condition** 530
:report method of **condition** 530
:report-string method of **condition** 531
:set-proceed-types method of **condition** 530
Signalling a condition 479
:special-command-p method of **condition** 530
:special-commands method of **condition** 530
sys:wrong-type-argument condition 505
Condition Bind Recursion 502
Condition flavor hierarchy 482
Creating a Set of Condition Flavors 486
Condition Flavors Reference 529
Proceedable condition functions 521
Binding condition handlers 481
Creating condition objects 505
Condition system package 615
condition-bind-default special form 489, 501, 509
condition-bind-default-if special form 489
condition-bind-if special form 489
condition-call and **:no-error** 493
condition-call-if special form 493
condition-case-if special form 491
dbg: **condition-handled-p** function 509, 510
Conditional breakpoints 523
Conditional construct 175
Conditionalization 216
Read-time conditionalization facility 29
Conditionals 177
Proceeding with **condition-bind** Handlers 520
condition-bind special form 488, 501, 517
condition-call special form 492, 501
condition-case special form 490, 501
condition flavor 482, 529, 531
Condition Flavors 482
Condition hierarchies 486
Condition objects 479, 481, 519, 529
Conditions 477
Creating New Conditions 485
Error conditions 482
Fundamental Conditions 531
How Applications Programs Treat Conditions 481
Introduction to Signalling and Handling Conditions 479
Package-related Conditions 619
Proceedable conditions 517
Reference Material: Signalling Conditions 503
Signalling Conditions 501
Signalling proceedable conditions 521
Signalling Simple Conditions 502
Simple conditions 482
Standard Conditions 531
Tracing Conditions 523

Using the RESUME key with floating-point conditions 535
Debugger conditions 482
Conditions as instances of flavors 479
Name Conflicts 357, 613
Package name conflicts 581
prog-Context Conflicts 359
Symbol name conflicts 574
Conformal Indirection 240
conjugate function 101
sys: **connection-closed-locally** flavor 555
sys: **connection-no-more-data** flavor 556
sys: **connection-closed** flavor 555
sys: **connection-error** flavor 555
sys: **connection-lost** flavor 556
Connection Problems 555
Network connection problems 555
sys: **connection-refused** flavor 555
Cons 41
Cons as property list 67
The Cons Data Type 4
:cons option for **defstruct-define-type** 410
cons-in-area function 45
sys: **cons-in-fixed-area** flavor 538
Cons comparisons 11
Conses 42
How the Reader Recognizes Conses 25
Memory allocation of conses 56
Printed Representation of Conses 17
Conses represented as pointers 56
cons function 45, 56
Hash table considerations while using multiprocessing 69
Print-Print consistency 591
Print-Read consistency 591
Read-Read consistency 591
Consistency Rules for Packages 591
Special Form for Declaring a Named Constant 135
Functions and Special Forms for Constant Values 161
Character constants 27
SCL and *constituent* character attribute 643
SCL and Standard Constituent Character Attributes Table 643
Conditional construct 175
Using the Constructor and Alterant Macros 395
Clm: By-position Constructor Functions 642
By-position Constructor Macros 396
:constructor option for **defstruct** 385, 386, 396
Constructor macros 379, 395
Expressions in loop constructs 206
Invalid contents in cell errors 533
continue-whopper function 463
Contracts 418
Flow of Control 173, 175
Introduction to Flow of Control 175
Program control 175
Transfer of Control 187
Exit control structures 175
Nonlocal exit control structures 175
Controlling the Printed Representation of an Object 19

Naming convention 7
Clm: Type Conversion Function 637
Conversion of numbers 90
Character Conversions 272
Numeric Type Conversions 107
String Conversions 285
:copier option for **defstruct** 385, 392
:copier option for **defstruct-define-type** 410, 412
copy-array-contents-and-leader function 253
copy-array-contents function 253
copy-array-portion function 253
copyalist function 50
copy-closure function 336
Copying an Array 252
Copying From and to the Same Array 253
Copying Instances 465
copylist* function 50
copylist function 50, 56
copy-readtable function 33
copysymbol function 567
copytree function 50
copytree-share function 50
Coroutine 423
fs: **correctable-login-problems** flavor 547
:correct-input 504, 543
cosd function 107
cos function 106
cosh function 107
Cosine 106
count loop keyword 212
counting loop keyword 212
CR character 32
Create a byte specifier 116
Functions That Create New Readtables 33
:create-directories-recursively Debugger command 549
fs: **create-directory-failure** flavor 550
fs: **create-link-failure** flavor 551
:create-directory Debugger command 549
:create-package proceed type 542, 619
Creating a Set of Condition Flavors 486
Creating condition objects 505
Creating data types 379
Functions for Creating Flavors 428
Creating Hash Tables 71
Creating instances of flavors 430
Creating instances of structures 395
Creating New Conditions 485
Clm: Creating New Streams 643
Creating flavors 428
Creating methods 429
Creating Symbols 566
File creation errors 550
fs: **creation-failure** 550
fs: **creation-failure** flavor 550
Current package 597
Change current package 597
The Current Package 597

Customizing Debugger keystrokes 526

D D D

D exponential representation 94
:daemon method combination type 455
:daemon-with-and method combination type 455, 456
:daemon-with-or method combination type 455, 456
:daemon-with-override method combination type 455, 457
Daemon methods 431
dtp-instance-header data type 467
dtp-select-method data type 467
dtp-instance data type 467
The Array Data Type 5
The Compiled Function Data Type 4
The Cons Data Type 4
The List Data Type 5
The Locative Data Type 5
The Symbol Data Type 3
Data type names 576
Clm: Specific Data Type Predicates 638
Data Types 3
Clm: Data Types 637
Creating data types 379
Numeric Data Types 4
Printed Representation of Miscellaneous Data Types 18
Data Types Recognized by **loop** 219
fs: **data-error** flavor 546
fixnum loop data-type keyword 219
flonum loop data-type keyword 219
integer loop data-type keyword 219
notype loop data-type keyword 219
number loop data-type keyword 219
dbg:*interactive-handlers* variable 511
dbg:*proceed-type-special-keys* variable 528
dbg:*special-command-special-keys* variable 528
dbg:bad-array-mixin flavor 540
dbg:condition-handled-p function 509, 510
dbg:debugger-condition flavor 501, 511, 529, 532
dbg:frame-active-p function 497
dbg:frame-arg-value function 498
dbg:frame-local-value function 498
dbg:frame-next-active-frame function 496
dbg:frame-next-interesting-active-frame function 496
dbg:frame-next-nth-active-frame function 496
dbg:frame-next-nth-interesting-active-frame function 496
dbg:frame-next-nth-open-frame function 497
dbg:frame-next-open-frame function 496
dbg:frame-number-of-locals function 498
dbg:frame-number-of-spread-args function 497
dbg:frame-out-to-interesting-active-frame function 497
dbg:frame-previous-active-frame function 496
dbg:frame-previous-interesting-active-frame

function 496
dbg:frame-previous-open-frame function 496
dbg:frame-real-function function 497
dbg:frame-real-value-disposition function 498
dbg:frame-self-value function 498
dbg:frame-total-number-of-args function 497
dbg:get-frame-function-and-args function 495
dbg:invoke-restart-handlers function 516
dbg package 615
dbg:print-frame-locals function 499
dbg:print-function-and-args function 499
dbg:proceedable-ferror flavor 532
dbg:special-commands-mixin flavor 525
dbg:with-erring-frame macro 494, 495
Dead arrays 262
Debugger 502, 517, 523
Entering Debugger 532
Debugger Bug Reports 524
c-M Debugger command 524
:create-directories-recursively Debugger command 549
:create-directory Debugger command 549
m-BREAK Debugger command 532
Debugger conditions 482
Customizing Debugger keystrokes 526
Debugger Special Commands 525
dbg: **debugger-condition** flavor 501, 511, 529, 532
debugger package 615
si:encapsulated-definition debugging info alist element 325
Debugging info alist functions 302, 323, 325
Aid for Debugging Macros 369
Clm: Debugging Tools 645
debugging-info function 302, 323
decf macro 149
Trailing decimal point 14
arglist declaration 312
def declaration 312
SCL and **declaration** declaration 640
SCL and **ftype** declaration 640
SCL and **inline** declaration 640
SCL and **notinline** declaration 640
SCL and **optimize** declaration 640
special declaration 312
sys:array-register-1d declaration 312
sys:array-register declaration 312
sys:downward-funarg declaration 139, 141, 313
sys:downward-function declaration 139, 140, 312
sys:function-parent declaration 312
unspecial declaration 312
Using the **sys:function-parent** Declaration 319
values declaration 312
SCL and **declaration** declaration 640
System declaration file 598, 621
Clm: Type Declaration for Forms 640
Clm: Declaration Specifiers 640
Clm: Declaration Syntax 640
Declarations 311
Clm: Declarations 640
The **sys:downward-function** and **sys:downward-funarg**

Declarations 140
declare special form 126, 311
SCL and **declare** special form 640
Special Form for Declaring a Named Constant 135
SCL and **decode-universal-time** function 645
Array decoding 255
math: **decompose** function 259
Decrementing generalized variables 149
def special form 307
Specifying Default Forms in Lambda Lists 155
Default forms of lambda-list parameters 151
Default Handlers and Complex Modularity 509
Reference Material: Default Handlers and Complex Modularity 510
:default method type 455, 457
Default values for instance variables 431
:default-init-plist Option for **defflavor** 441, 442
SCL and ***default-pathname-defaults*** variable 644
:default-value option for **make-plane** 261
:default-handler Option for **defflavor** 441, 447
Default handlers 501, 502
:default-pointer option for **defstruct** 385, 386
defconst special form 126, 135
defconstant special form 135
def declaration 312
deff special form 307
:abstract-flavor Option for **defflavor** 441, 450
:accessor-prefix Option for **defflavor** 441, 448
:default-init-plist Option for **defflavor** 441, 442
:default-handler Option for **defflavor** 441, 447
:documentation Option for **defflavor** 441, 449
:export-instance-variables Option for **defflavor** 441, 448
:gettable-instance-variables Option for **defflavor** 441, 485
:included-flavors Option for **defflavor** 441, 444
:initable-instance-variables Option for **defflavor** 441, 485
:init-keywords Option for **defflavor** 441, 442
:method-combination Option for **defflavor** 441, 449
:method-order Option for **defflavor** 441, 449
:mixture Option for **defflavor** 441, 445
:no-vanilla-flavor Option for **defflavor** 441, 445
:ordered-instance-variables Option for **defflavor** 441, 447
:outside-accessible-instance-variables Option for **defflavor** 441, 447
:required-init-keywords Option for **defflavor** 441, 442
:required-instance-variables Option for **defflavor** 441, 442
:required-flavors Option for **defflavor** 441, 443
:required-methods Option for **defflavor** 441, 442
:settable-instance-variables Option for **defflavor** 441
:special-instance-variables Option for **defflavor** 441, 448
defflavor macro 428
defflavor Options 441
deffunction special form 356
define-loop-macro macro 218
define-loop-path macro 227
define-loop-sequence-path macro 225
si: **define-simple-method-combination** macro 459
define-symbol-macro special form 353
@define macro 308
Defining a Package 598
Defining function specs 297

Defining Iteration Paths 227
Aids for Defining Macros 343
Defining special forms 362
Defining special variables 134
Special Forms for Defining Special Variables 134
Defining flavors 428
Defining functions 297, 300, 305
Defining methods 429
An Example Path Definition 229
Clm: Macro Definition 639
Symbol definition 3, 563
Basic definition of the function spec 297, 325
How Programs Manipulate Definitions 316
Selective evaluation in macro definitions 345
Simple Function Definitions 300
SCL and equivalent macro definitions for special forms 638
Definitions of functions 316
Overview and Definitions of Signalling and Handling 479
Definition types 319
deflambda-macro-displace special form 356
deflambda-macro special form 356
defmacro 343
&aux keyword for **defmacro** 373
&body keyword for **defmacro** 373
&-Keywords Accepted by **defmacro** 373
&list-of keyword for **defmacro** 373
&optional keyword for **defmacro** 373
&rest keyword for **defmacro** 373
defmacro and lexical scoping 138
defmacro special form 305
defmacro-displace macro 372
defmacro macro 344
defmethod special form 297
defmethod macro 429
:colon-mode option for **defpackage** 585
:export option for **defpackage** 573
:import option for **defpackage** 573
:import-from option for **defpackage** 573
:relative-names-for-me option for **defpackage** 582
:relative-names option for **defpackage** 582
:shadow option for **defpackage** 574
:shadowing-import option for **defpackage** 573, 574
:use option for **defpackage** 571, 599
:colon-mode Option for **defpackage** and **make-package** 601, 603
:export Option for **defpackage** and **make-package** 599, 602
:external-only Option for **defpackage** and **make-package** 600, 603
:hash-inherited-symbols Option for **defpackage** and **make-package** 600, 603
:import Option for **defpackage** and **make-package** 599, 602
:import-from Option for **defpackage** and **make-package** 600, 602
:include Option for **defpackage** and **make-package** 600, 603
:new-symbol-function Option for **defpackage** and **make-package** 601, 603
:nicknames Option for **defpackage** and **make-package** 599, 601
:prefix-intern-function Option for **defpackage** and **make-package** 601, 603
:prefix-name Option for **defpackage** and **make-package** 599, 601
:relative-names Option for **defpackage** and **make-package** 600, 602
:relative-names-for-me Option for **defpackage** and **make-package** 600, 602
:shadow Option for **defpackage** and **make-package** 599, 602
:shadowing-import Option for **defpackage** and **make-package** 600, 602

:size Option for **defpackage** and **make-package** 600, 602
defpackage special form 598, 599
defprop special form 69
defselect special form 297, 305, 308
defselect-method special form 437
SCL and **defsetf** function 639
:alterant option for **defstruct** 385, 386
:but-first option for **defstruct** 385, 390
:callable-accessors option for **defstruct** 385, 391
:conc-name option for **defstruct** 385, 387
:constructor option for **defstruct** 385, 386, 396
:copier option for **defstruct** 385, 392
:default-pointer option for **defstruct** 385, 386
:eval-when option for **defstruct** 385, 391
:export option for **defstruct** 385, 386
Extensions to **defstruct** 409
:include option for **defstruct** 385, 387
:initial-offset option for **defstruct** 385, 390
:make-array option for **defstruct** 385, 389, 395
:named option for **defstruct** 385, 389
Options to **defstruct** 385
:predicate option for **defstruct** 385, 392
:print option for **defstruct** 385, 391
:property option for **defstruct** 385, 391
SCL and **:named** option for **defstruct** 642
SCL and **:type** option for **defstruct** 642
:size-macro option for **defstruct** 385, 390
:size-symbol option for **defstruct** 385, 390
:times option for **defstruct** 385, 390, 395
:type option for **defstruct** 385
Using **defstruct** 383
Using Byte Fields and **defstruct** 399
:array option for **defstruct :type** 385
:array-leader option for **defstruct :type** 385
:fixnum option for **defstruct :type** 385, 386
:grouped-array option for **defstruct :type** 385, 386
:list option for **defstruct :type** 385
:named-array-leader option for **defstruct :type** 385
:named-array option for **defstruct :type** 385
:named-list option for **defstruct :type** 385
:tree option for **defstruct :type** 385, 386
defstruct Internal Structures 407
SCL and **defstruct** macro 642
:defstruct option for **defstruct-define-type** 410, 412
Clm: **defstruct** Options 642
SCL and **defstruct** slot initializations 642
An Example of **defstruct-define-type** 409
:cons option for **defstruct-define-type** 410
:copier option for **defstruct-define-type** 410, 412
:defstruct option for **defstruct-define-type** 410, 412
:keywords option for **defstruct-define-type** 410, 412
:named option for **defstruct-define-type** 410, 411
Options to **defstruct-define-type** 410
:overhead option for **defstruct-define-type** 410, 411
:predicate option for **defstruct-define-type** 410, 412
:ref option for **defstruct-define-type** 410
defstruct-define-type macro 409
si: **defstruct-description** property 407

defstruct macro 345, 383
defsubst 351
defsubst special form 305, 351
sys: **defsubst-with-parent** macro 319
defun special form 297, 300, 305
Removing a **defun-method** 437
defun-method special form 436, 437
defunp macro 302, 305
defvar special form 126, 134
defwhopper special form 463
defwhopper-subst macro 464
defwrapper macro 461
Degrees in trigonometric functions 106
del-if-not function 64
:delete-by-item method of **si:heap** 78
:delete-by-key method of **si:heap** 78
SCL and **:replace** keyword for **delete-duplicates** function 640
fs: **delete-failure** 551
fs: **delete-failure** flavor 551
delete function 61
File deletion errors 551
del function 62
del-if function 64
delq function 61
Delta (δ) character 32
Greatest common denominator 105
Integer denominator 93
denominator function 108
deposit-byte function 117
deposit-field function 117
Depth of recursion of printing lists 16
The Device-font and Subindex Derived Fields 268
Derived fields for characters 268
Zmacs Command: Describe Flavor 471
:describe keyword for **named-structure-invoke** 403, 404
:describe method of **si:heap** 78
describe-defstruct function 384
describe-flavor function 440
describe function 69
SCL and **describe** function 645
:describe message 453
describe-package function 612
si:alphabetic syntax description 34
si:break syntax description 34
si:circlecross syntax description 34
si:doublequote syntax description 34
si:macro syntax description 34
si:single syntax description 34
si:slash syntax description 34
si:verticalbar syntax description 34
si:whitespace syntax description 34
Slot description 407
Instance descriptor 467
Numeric argument descriptor 324
%%arg-desc-interpreted numeric argument descriptor field 324
%arg-desc-interpreted numeric argument descriptor field 324
%%arg-desc-max-args numeric argument descriptor field 324

%%arg-desc-min-args numeric argument descriptor field 324
%%arg-desc-quoted numeric argument descriptor field 324
%%arg-desc-rest-arg numeric argument descriptor field 324
desetq special form 133
Destructuring 220
destructuring-bind special form 132
Character Object Details 265
math: **determinant** function 259
fs: **device-not-found** flavor 549
The Device-font and Subindex Derived Fields 268
Displaying characters on output devices 268
dfloat function 108
difference function 101
SCL and Common Lisp Differences 637
si: **digested-lambda** functions 297, 300, 304, 305
digit-char-p function 270
digit-char function 273
Array dimensions 235, 247
SCL and ~E directive 644
SCL and ~G directive 644
SCL and ~T directive 644
SCL and **format** function directives 644
Clm: Accessing Directories 645
SCL and **directory** function 645
fs: **directory-already-exists** flavor 550
fs: **directory-not-empty** flavor 552
fs: **directory-not-found** flavor 549
:directory-pathname message 549
Enabling and disabling of floating-point traps 535
disassemble function 302
Disembodied property list 67
Disk error 538
sys: **disk-error** flavor 538
dispatch special form 182
Clm: Standard Dispatching Macro Character Syntax 644
Displaced macro expansions 371
:displaced-conformally option for **make-array** 241
:displaced-index-offset option for **make-array** 240, 241
:displaced-to option for **make-array** 239, 241
Displaced Arrays 239, 241, 248
displace function 371
Displacing Macros 371
Displaying characters on output devices 268
sys: **divide-by-zero** flavor 479, 534
Division 102, 103
Integer division 93
Division by zero error 534
Greatest common divisor 105
dlet* special form 133
dlet special form 133
do loop keyword 212
do special form 167, 189, 359
do* special form 191
do-all-symbols special form 609
do-external-symbols special form 609
do-local-symbols special form 609
:document-proceed-type method of **condition** 529

:document-proceed-type message 517
:document-special-command message 526
Clm: Documentation 645
SCL Documentation 636
SCL and **documentation** as **setf** argument 639
:documentation Option for **defflavor** 441, 449
Documentation string functions 302, 322
documentation function 302, 322
SCL and **documentation** function 645
doing loop keyword 212
dolist special form 193
do*-named special form 192
do-named special form 192
fs: **dont-delete-flag-set** flavor 552
do-symbols special form 609
Dot (.) in symbols 357
dotimes special form 192
Dotted list 25, 41
sys: **double-float-p** function 8
Double-precision floating-point numbers 89
Double-float 22
:double-float returned by **typep** 9
cl: **double-float** format 23
Double-floats 4, 89
si: **doublequote** syntax description 34
downfrom loop keyword 207, 225
downto loop keyword 207, 225
The **sys:downward-function** and **sys:** **downward-funarg** Declarations 140
The **sys:** **downward-function** and **sys:downward-funarg** Declarations 140
sys: **downward-funarg** declaration 139, 141, 313
Downward funargs 139
sys: **downward-function** declaration 139, 140, 312
dpb function 117
sys: **draw-off-end-of-screen** flavor 537
sys: **draw-on-unprepared-sheet** flavor 537
Drawing on unprepared sheet error 537
Drawing past edge of screen error 537
SCL and **dribble** function 645
dtp-instance-header data type 467
dtp-select-method data type 467
dtp-instance data type 467
Restriction Due to Scope 503
dumparrays Maclisp function 262
Dumping Hash Tables to Files 75
Functions Used During Expansion 368
What is a Dynamic Closure? 331
Dynamic Closure-manipulating Functions 335
Dynamic closures 303, 305, 331
Examples of the Use of Dynamic Closures 333
Functions and Dynamic Closures 295
Dynamic scope 126

E

Drawing past edge of screen error 537
Zmacs Command: Edit Combined Methods 472
Zmacs Command: Edit Methods 471
Editor package 615
Effect of compiler on variables 126
Side Effects 212
Effects of Slashification on Printing 13
si:encapsulated-definition debugging info alist element 325
Bit size of array elements 236
Character strings as array elements 236
Integers as array elements 236
List elements 41
Returning array elements 244
Storing into array elements 244
Removing elements from list 150
Active elements in arrays 238, 247
Inserting elements into list 150
Maximum number of list elements to be printed 18
Elements out-of-bounds 541
else loop keyword 216
:empty-p method of **si:heap** 78
Enabling and disabling of floating-point traps 535
si: **encapsulated-definition** debugging info alist element 325
si: **encapsulated-function** variable 325
si: **encapsulate** macro 326
si: **encapsulation-standard-order** variable 327
Encapsulation mechanism 523
Encapsulations 297, 325
Rename-within Encapsulations 329
Adding to the End of an Array 251
Read past the end-of-file error 536
sys: **end-of-file** flavor 536
tape: **end-of-tape** flavor 556
End character 32
Endtests 214, 221
Entering Debugger 532
Entrance and Exit 212
MDL programming environment 205
environment argument for macro-expander functions 138
Clm: Other Environment Inquiries 645
&environment Lambda-list Keyword 311, 374
Lexical Environment Objects and Arguments 138
Loop epilogue 205, 212
Hashing on **eq** 75
eq versus **equal** 10
eq versus **equal** 11
:area init option for **si:** **eq-hash-table** 71
:growth-factor init option for **si:** **eq-hash-table** 71
:rehash-before-cold init option for **si:** **eq-hash-table** 71
:size init option for **si:** **eq-hash-table** 71
si: **eq-hash-table** flavor 71
eq function 10, 89

eql function 10
eq versus **equal** 10
Hashing on **equal** 75
eq versus **equal** 11
:rehash-threshold init option for **si:** **equal-hash-table** 72
si: **equal-hash-table** flavor 72
equal function 11
si: **equal-hash** function 75
SCL and **equal-typep** function 638
Matrices and Systems of Linear Equations 258
Simultaneous linear equations 258
SCL and equivalent macro definitions for special forms 638
Disk error 538
Division by zero error 534
Drawing on unprepared sheet error 537
Drawing past edge of screen error 537
Exponent overflow error 535
Exponent underflow error 536
Illegal redefinition error 538
Invalid type code error 534
Logarithm of nonpositive number error 534
Read past the end-of-file error 536
Read-only error 537
Singular matrix operation error 535
Square root of a negative number error 536
Stack group state error 537
Stack overflow error 537
Storage allocation error 538
Throw tag error 538
Unbound closure variable error 533
Unbound instance variable error 533
Undefined function error 534
Error logging code 494
error-restart-loop special form 501, 513, 514
Error conditions 482
error flavor 482, 529, 532
error function 482, 503
Error object 529
errorp function 9, 504
error-restart special form 501, 513, 514
Errors 479
Arithmetic errors 534
Array Errors 540
bitblt errors 537
Checking for Package Name-conflict Errors 593
Eval Errors 542
Evaluator errors 542
File creation errors 550
File deletion errors 551
File lookup errors 548
File property errors 551
File rename errors 551
File-system Errors 544
Function-calling Errors 539
Instance variable errors 538
Introduction to Package Name-conflict Errors 593
Invalid contents in cell errors 533
Invalid file operation errors 550

Invalid pathname syntax errors 550
Lisp Errors 533
Location Errors 534
Login errors 547
Network Errors 554
Package Name-conflict Errors 593
Pathname Errors 553
Protection-violation errors 549
Resolving Package Name-conflict Errors 595
SCL and file lookup errors 645
Signalling Errors 502
Tape Errors 556
Unbound variable errors 533
Interning Errors Based on **sys:package-error** 542
Syntax errors in read functions 543
Errors inside Lisp printer 543
Errors Involving Lisp Printed Representations 543
Miscellaneous System Errors Not Categorized by Base Flavor 536
ESC character 32
Establishing Handlers 487
Reference Material: Establishing Handlers 488
&eval Lambda-list Keyword 310
:eval-inside-yourself message 454
Eval Errors 542
eval function 159
evalhook 123, 159
Evaluating a Function Form 151
Evaluation 121
Function for Evaluation 159
Introduction to Evaluation 123
Multiple and Out-of-order Evaluation 363
Evaluation 123
Evaluation in loops 212
Selective evaluation in macro definitions 345
Evaluation of special functions 303
Evaluator 303
Evaluator errors 542
:eval-when option for **defstruct** 385, 391
Testing for even number 97
evenp function 97
Event 479
every function 64
How Programs Examine Functions 322
Flavor Examiner 475
Application: Handlers Examining the Stack 494
Reference Material: Application: Handlers Examining the Stack 495
Example of a Handler 481
An Example of **defstruct-define-type** 409
Example of the Need for Packages 560
An Example Path Definition 229
Examples of Simple Lambda Lists 154
Examples of Symbol Sharing Among Packages 589
Examples of the Use of Dynamic Closures 333
Exclusive or 113
Entrance and Exit 212
Nonlocal exit 197
Trap on exit bit 523
Loop exit code 205, 212

Exit control structures 175
Nonlocal exit control structures 175
Break on exit from marked frame message 523
Blocks and Exits 183
Clm: Blocks and Exits 639
Nonlocal Exits 197
Functions to Expand Macros 375
Expanded character set 277
Macros expanded to Lisp functions 355
Expander function 144
Macros Expanding Into Many Forms 360
Functions Used During Expansion 368
Displaced macro expansions 371
exp function 106
Floating-point Exponent Characters 22
Exponent overflow error 535
Exponent underflow error 536
D exponential representation 94
E exponential representation 94
E exponential representation 14
Exponential notation 14
Exponentiation 105
Exponent overflow 94
Exponent underflow 94
:export option for **defpackage** 573
:export Option for **defpackage** and **make-package** 599, 602
:export option for **defstruct** 385, 386
:export option for **make-package** 573
:export proceed type 619
Functions That Import, Export, and Shadow Symbols 611
:export-instance-variables Option for **defflavor** 441, 448
export function 593, 611
Exporting symbols 573, 589, 593, 611, 613
Importing and Exporting Symbols 573
expr Maclisp type 305
Reader macro for infix expressions 30
Expressions in loop constructs 206
expt function 105
:extension option for **make-plane** 261
SCL and Symbolics Common Lisp Extensions 635
Extensions to **defstruct** 409
Multics external arrays 262
:external colon mode 601, 603
Specifying Internal and External Symbols in Packages 585
External value cell 331
:external-only Option for **defpackage** and **make-package** 600, 603
:package message to **sys:** **external-symbol-not-found** 619
:string message to **sys:** **external-symbol-not-found** 619
sys: **external-symbol-not-found** flavor 619
External-only Packages and Locking 626
External Symbols 570, 571, 573
Extra Features of Arrays 238
Extract position field of a byte specifier 116
Extract size field of a byte specifier 116

F F F

Hash table facilities 59
Comma character (,) in backquote facility 345
Read-time conditionalization facility 29
FOR facility in Interlisp 205
Hasharray facility of Interlisp 69
Miscellaneous file operations failures 552
Miscellaneous Operations Failures 552
Request Failures Based on **fs:file-request-failure** 546
false function 164
Flavor Families 451
:fasd-form message 465
fast-aref instruction 255
fast-aset instruction 255
fboundp function 564
sys: **fceiling** function 112
sys: **fdefine-file-pathname** variable 319
fdefinedp function 321
fdefine function 297, 316
fdefinition function 322
sys: **fdefinition-location** function 322
SCL and **ieee-floating-point** feature name 645
Clm: the "Program Feature" 639
Extra Features of Arrays 238
SCL and keywords in the ***features*** list 644
SCL and ***features*** variable 645
ferror flavor 532
ferror function 482, 504
fexpr Maclisp type 305
sys: **ffloor** function 111
%%arg-desc-interpreted numeric argument descriptor field 324
%arg-desc-interpreted numeric argument descriptor field 324
%%arg-desc-max-args numeric argument descriptor field 324
%%arg-desc-min-args numeric argument descriptor field 324
%%arg-desc-quoted numeric argument descriptor field 324
%%arg-desc-rest-arg numeric argument descriptor field 324
Character objects code field 265
Character objects font field 265
Character objects style field 265
Cdr-code field 56
Extract position field of a byte specifier 116
Extract size field of a byte specifier 116
Character Fields 269
The Device-font and Subindex Derived Fields 268
Using Byte Fields and **defstruct** 399
Derived fields for characters 268
fifth function 46
Compiled file 579
Sysdcl file 598
System declaration file 598, 621
File attribute list 579
%%ch- byte specifiers and file characters 268
File creation errors 550
File deletion errors 551
File lookup errors 548
SCL and file lookup errors 645
Invalid file operation errors 550

Miscellaneous file operations failures 552
File property errors 551
File rename errors 551
fs: **file-already-exists** flavor 550
fs: **file-lookup-error** flavor 548
fs: **file-not-found** flavor 548
fs: **file-open-for-output** flavor 553
fs: **file-operation-failure** flavor 481, 546
Request Failures Based on **fs:** **file-request-failure** 546
fs: **file-request-failure** flavor 546
fs: **file-error** flavor 545
fs: **file-locked** flavor 552
File Lookup 548
fs: **filepos-out-of-range** flavor 552
Binary files 613
Dumping Hash Tables to Files 75
SCL and Common Lisp Files 635
File-system Errors 544
file-system package 615
math: **fill-2d-array** function 260
:fill-pointer option for **make-array** 241
sys: **fill-pointer-not-fixnum** flavor 540
fillarray function 252
:filled-elements message 73
Fill pointer 238, 247
fill-pointer function 245
finally loop keyword 212
Functions That Find the Home Package of a Symbol 607
find-all-symbols function 606
:find-by-item method of **si:heap** 78
:find-by-key method of **si:heap** 78
find-position-in-list-equal function 61
find-position-in-list function 60
Finding a Handler 501
first loop keyword 207
first function 46
firstn function 53
:fix argument to **typep** 9
Fixed-point numbers 7
fix function 107
fixnum loop data-type keyword 219
:fixnum returned by **typep** 9
fixnump function 8
Fixnums 4, 89
fixp function 7
fixr function 108
SCL and **zl:si:** ***flag-wrong-type-strings*** 637, 642
Base flavor 451
break flavor 532
Changing a Flavor 468
condition flavor 482, 529, 531
dbg:bad-array-mixin flavor 540
dbg:debugger-condition flavor 501, 511, 529, 532
dbg:proceedable-ferror flavor 532
dbg:special-commands-mixin flavor 525
error flavor 482, 529, 532
ferror flavor 532
fs:access-error flavor 549

fs:change-property-failure flavor 551
fs:circular-link flavor 552
fs:correctable-login-problems flavor 547
fs:create-directory-failure flavor 550
fs:create-link-failure flavor 551
fs:creation-failure flavor 550
fs:data-error flavor 546
fs:delete-failure flavor 551
fs:device-not-found flavor 549
fs:directory-already-exists flavor 550
fs:directory-not-empty flavor 552
fs:directory-not-found flavor 549
fs:dont-delete-flag-set flavor 552
fs:file-already-exists flavor 550
fs:file-lookup-error flavor 548
fs:file-not-found flavor 548
fs:file-open-for-output flavor 553
fs:file-operation-failure flavor 481, 546
fs:file-request-failure flavor 546
fs:file-error flavor 545
fs:file-locked flavor 552
fs:filepos-out-of-range flavor 552
fs:host-not-available flavor 546
fs:inconsistent-options flavor 552
fs:incorrect-access-to-directory flavor 549
fs:incorrect-access-to-file flavor 549
fs:invalid-byte-size flavor 552
fs:invalid-operation-for-directory flavor 550
fs:invalid-operation-for-link flavor 550
fs:invalid-pathname-component flavor 553
fs:invalid-pathname-syntax flavor 550
fs:invalid-property-value flavor 551
fs:invalid-password flavor 547
fs:invalid-wildcard flavor 550
fs:link-target-not-found flavor 549
fs:login-problems flavor 547
fs:login-required flavor 548
fs:multiple-file-not-found flavor 548
fs:network-lossage flavor 547
fs:no-file-system flavor 546
fs:no-more-room flavor 552
fs:not-enough-resources flavor 547
fs:not-logged-in flavor 548
fs:not-available flavor 553
fs:parse-pathname-error flavor 553
fs:pathname-error flavor 553
fs:rename-across-directories flavor 551
fs:rename-across-hosts flavor 551
fs:rename-to-existing-file flavor 551
fs:rename-failure flavor 551
fs:undefined-logical-pathname-translation flavor 554
fs:unimplemented-option flavor 552
fs:unknown-pathname-host flavor 553
fs:unknown-operation flavor 547
fs:unknown-property flavor 551
fs:unknown-user flavor 547
fs:wildcard-not-allowed flavor 550
fs:wrong-kind-of-file flavor 550

math:singular-matrix flavor 535
Message to an object of some flavor 461
Miscellaneous System Errors Not Categorized by Base Flavor 536
Mixin flavor 451
parse-ferror flavor 543
Removing flavor 437
si:eq-hash-table flavor 71
si:equal-hash-table flavor 72
si:property-list-mixin flavor 473
si:vanilla-flavor flavor 453
sys:abort flavor 502, 526, 532
sys:area-overflow flavor 537
sys:arithmetic-error flavor 534
sys:array-has-no-leader flavor 540
sys:array-wrong-number-of-dimensions flavor 540
sys:array-wrong-number-of-subscripts flavor 541
sys:bad-array-type flavor 540
sys:bad-connection-state flavor 555
sys:bad-data-type-in-memory flavor 534
sys:bitblt-array-fractional-word-width flavor 537
sys:bitblt-destination-too-small flavor 537
sys:call-trap flavor 523
sys:cell-contents-error flavor 533
sys:connection-closed-locally flavor 555
sys:connection-no-more-data flavor 556
sys:connection-closed flavor 555
sys:connection-error flavor 555
sys:connection-lost flavor 556
sys:connection-refused flavor 555
sys:cons-in-fixed-area flavor 538
sys:disk-error flavor 538
sys:divide-by-zero flavor 479, 534
sys:draw-off-end-of-screen flavor 537
sys:draw-on-unprepared-sheet flavor 537
sys:end-of-file flavor 536
sys:external-symbol-not-found flavor 619
sys:fill-pointer-not-fixnum flavor 540
sys:float-divide-by-zero flavor 535
sys:float-divide-zero-by-zero flavor 536
sys:float-inexact-result flavor 536
sys:float-invalid-compare-operation flavor 536
sys:float-invalid-operation flavor 536
sys:floating-exponent-overflow flavor 535
sys:floating-exponent-underflow flavor 536
sys:floating-point-exception flavor 535
sys:host-not-responding flavor 555
sys:host-not-responding-during-connection flavor 555
sys:host-stopped-responding flavor 555
sys:instance-variable-pointer-out-of-range flavor 538
sys:instance-variable-zero-referenced flavor 538
sys:invalid-function flavor 542
sys:local-network-error flavor 554
sys:name-conflict flavor 619
sys:negative-sqrt flavor 536
sys:network-resources-exhausted flavor 554
sys:network-stream-closed flavor 556
sys:network-error flavor 554

sys:no-action-mixin flavor 532
sys:non-positive-log flavor 534
sys:number-array-not-allowed flavor 541
sys:package-not-found flavor 542, 619
sys:package-error flavor 542, 619
sys:package-locked flavor 543, 619
sys:parse-error flavor 543
sys:pdl-overflow flavor 537
sys:print-not-readable flavor 543
sys:read-end-of-file flavor 544
sys:read-list-end-of-file flavor 544
sys:read-premature-end-of-symbol flavor 544
sys:read-string-end-of-file flavor 544
sys:read-error flavor 544
sys:redefinition flavor 538
sys:region-table-overflow flavor 538
sys:remote-network-error flavor 554
sys:stream-closed flavor 536
sys:subscript-out-of-bounds flavor 541
sys:throw-tag-not-seen flavor 538
sys:too-few-arguments flavor 539
sys:too-many-arguments flavor 539
sys:unbound-closure-variable flavor 533
sys:unbound-instance-variable flavor 533
sys:unbound-symbol flavor 533
sys:unbound-variable flavor 533
sys:unclaimed-message flavor 542
sys:undefined-keyword-argument flavor 542
sys:undefined-function flavor 534
sys:unknown-host-name flavor 554
sys:unknown-locf-reference flavor 534
sys:unknown-setf-reference flavor 534
sys:unknown-address flavor 554
sys:virtual-memory-overflow flavor 537
sys:write-in-read-only flavor 537
sys:wrong-stack-group-state flavor 537
sys:wrong-type-argument flavor 539
sys:zero-args-to-select-method flavor 539
tape:end-of-tape flavor 556
tape:mount-error flavor 556
tape:tape-device-error flavor 556
tape:tape-error flavor 556
Vanilla Flavor 453
Zmacs Command: Describe Flavor 471
:case flavor combination 520
Condition flavor hierarchy 482
Flavor inheritance mechanism 482
Introduction to the Flavor System 417
Message Passing in the Flavor System 423
Objects and the Flavor System 417
Using the Flavor System 425
Flavor system messages 453
Flavor system storing functions 297
si: **flavor-allowed-init-keywords** function 439
flavor-allows-init-keyword-p function 439
si: ***flavor-compile-trace*** variable 440
si: **flavor-default-init-get** function 440
si: **flavor-default-init-putprop** function 440

si: **flavor-default-init-remprop** function 440
Base Flavor: **sys:arithmetic-error** 534
Base Flavor: **sys:cell-contents-error** 533
Base Flavor: **sys:floating-point-exception** 535
si: ***flavor-compilations*** variable 440
Flavor Examiner 475
Flavor Families 451
Flavor Functions 435
Flavors 415, 417
Closure and flavors 423
Combining flavors 431, 461
Condition Flavors 482
Conditions as instances of flavors 479
Creating flavors 428
Creating a Set of Condition Flavors 486
Creating instances of flavors 430
Defining flavors 428
Functions for Creating Flavors 428
Implementation of Flavors 467
Instance variables of combined flavors 431
Mixing Flavors 431
Modifying flavors 438, 468
Simple Use of Flavors 425
Tree of flavors 431
Zmacs Commands for Flavors 471
Flavors and SCL 635
Condition Flavors Reference 529
Ordering Flavors, Methods, and Wrappers 467
flet special form 142
flet, labels, and **macrolet** Special Forms 142
:float argument to **typep** 9
sys: **float-divide-by-zero** flavor 535
sys: **float-divide-zero-by-zero** flavor 536
sys: **float-inexact-result** flavor 536
sys: **float-invalid-compare-operation** flavor 536
sys: **float-invalid-operation** flavor 536
float function 108
sys: **floating-exponent-overflow** flavor 535
sys: **floating-exponent-underflow** flavor 536
Double-precision floating-point numbers 89
Base Flavor: **sys:** **floating-point-exception** 535
sys: **floating-point-exception** flavor 535
Using the RESUME key with floating-point conditions 535
Floating-point Exponent Characters 22
How the Reader Recognizes Floating-point Numbers 22
Printed Representation of Floating-point Numbers 14
Single-precision floating-point numbers 89
IEEE Floating-point Representation 94
Enabling and disabling of floating-point traps 535
Floating-point infinity 14, 22
Floating-point numbers 4, 7, 94
floatp function 7
flonum loop data-type keyword 219
Flonum arrays 262
flonump function 8
floor function 108
Flow of Control 173, 175
Introduction to Flow of Control 175

fmakunbound function 564
Character objects font field 265
Font numbers and character objects 267
Font information 236, 277
fonts package 615
for loop keyword 207
#' special form 162
advise special form 302
and special form 178
argument-typecase special form 506
block special form 183
caseq special form 182
***catch** special form 169, 200
catch special form 169, 197
catch-error-restart special form 501, 502, 513, 515
catch-error-restart-if special form 513, 515
check-arg-type special form 576
comment special form 164
compiler-let special form 129
cond special form 177
cond-every special form 178
condition-bind-default special form 489, 501, 509
condition-bind-default-if special form 489
condition-bind-if special form 489
condition-call-if special form 493
condition-case-if special form 491
condition-bind special form 488, 501, 517
condition-call special form 492, 501
condition-case special form 490, 501
declare special form 126, 311
def special form 307
defconst special form 126, 135
defconstant special form 135
deff special form 307
deffunction special form 356
define-symbol-macro special form 353
deflambda-macro-displace special form 356
deflambda-macro special form 356
defmacro special form 305
defmethod special form 297
defpackage special form 598, 599
defprop special form 69
defselect special form 297, 305, 308
defselect-method special form 437
defsubst special form 305, 351
defun special form 297, 300, 305
defun-method special form 436, 437
defvar special form 126, 134
defwhopper special form 463
desetq special form 133
destructuring-bind special form 132
dispatch special form 182
dlet* special form 133
dlet special form 133
do special form 167, 189, 359
do* special form 191
do-all-symbols special form 609
do-external-symbols special form 609

do-local-symbols special form 609
dolist special form 193
do*-named special form 192
do-named special form 192
do-symbols special form 609
dotimes special form 192
error-restart-loop special form 501, 513, 514
error-restart special form 501, 513, 514
Evaluating a Function Form 151
flet special form 142
function special form 162
go special form 187, 189
if special form 177
ignore-errors special form 493
keyword-extract special form 193
labels special form 144
lambda special form 163
lambda-macro special form 355
let* special form 129
let special form 128, 142
let-globally-if special form 131
let-closed special form 305, 335
letf* special form 130
letf special form 130
let-globally special form 131
let-if special form 130
local-declare special form 126, 315
macro special form 305, 339, 355
macrolet special form 144
multiple-value-bind special form 168
multiple-value-call special form 168
multiple-value-list special form 168
multiple-value-prog1 special form 169
multiple-value special form 168
or special form 179
package-declare special form 625
prog special form 167, 194, 359
prog* special form 196
prog1 special form 165
prog2 special form 165
progn special form 164
progv special form 131
progw special form 132
psetq special form 128
quote special form 161
return special form 167, 169, 185, 189
return-from special form 167, 184, 218
SCL and **#S** macro form 642
SCL and **declare** special form 640
SCL and **function** special form 638
SCL and **go** special form 639
SCL and **the** special form 640
SCL and *value-type* argument for **the** special form 640
select special form 180
selector special form 181
selectq special form 179
selectq-every special form 182
setf special form 147

setq special form 125, 128
signal-proceed-case special form 517, 521
signp special form 97
tagbody special form 187
throw special form 198
trace special form 302
typecase special form 181, 576
undefun-method special form 437
unwind-protect special form 169, 198
variable-boundp special form 562
variable-location special form 563
variable-makunbound special form 562
with-input-from-string special form 290
with-output-to-string special form 291
without-floating-underflow-traps special form 94
Special Form for Declaring a Named Constant 135
Formal parameters 151
cl:double-float format 23
cl:long-float format 23
cl:short-float format 23
cl:single-float format 23
SCL and **format** function directives 644
:format-args 504
format package 615
:format-string 504
Clm: Formatted Output to Character Streams 644
Form character 32
Clm: Special Forms 638
Clm: Type Declaration for Forms 640
Defining special forms 362
flet, labels, and **macrolet** Special Forms 142
Function-defining Special Forms 305
Macros Expanding Into Many Forms 360
SCL and equivalent macro definitions for special forms 638
Some Functions and Special Forms 159
Special forms 303
Special Forms for Binding Variables 128
Functions and Special Forms for Constant Values 161
Special Forms for Defining Special Variables 134
Special Forms for Receiving Multiple Values 167
Special Forms for Sequencing 164
Special Forms for Setting Variables 128
Specifying Default Forms in Lambda Lists 155
Default forms of lambda-list parameters 151
Forms of qualified names 585
Interaction of Some Common Special Forms with Multiple Values 169
Package Functions, Special Forms, and Variables 597
fourth function 46
Active frame 494
Interesting active frame 494
Next frame 494
Open frame 494
Previous frame 494
Stack frame 494
Break on exit from marked frame message 523
dbg: **frame-active-p** function 497
dbg: **frame-arg-value** function 498
dbg: **frame-local-value** function 498

dbg: **frame-next-active-frame** function 496
dbg: **frame-next-interesting-active-frame** function 496
dbg: **frame-next-nth-active-frame** function 496
dbg: **frame-next-nth-interesting-active-frame** function 496
dbg: **frame-next-nth-open-frame** function 497
dbg: **frame-next-open-frame** function 496
dbg: **frame-number-of-locals** function 498
dbg: **frame-number-of-spread-args** function 497
dbg: **frame-out-to-interesting-active-frame** function 497
dbg: **frame-previous-active-frame** function 496
dbg: **frame-previous-interesting-active-frame** function 496
dbg: **frame-previous-open-frame** function 496
dbg: **frame-real-function** function 497
dbg: **frame-real-value-disposition** function 498
dbg: **frame-self-value** function 498
dbg: **frame-total-number-of-args** function 497
Frame-manipulating functions 494
Frame pointer 494
The Iteration Framework 221
Free reference 126
Captured free reference 126
Copying From and to the Same Array 253
Clm: Input From Character Streams 643
from loop keyword 207, 225
sys: **fround** function 113
fs:access-error 549
fs:access-error flavor 549
fs:change-property-failure 551
fs:change-property-failure flavor 551
fs:circular-link flavor 552
fs:correctable-login-problems flavor 547
fs:create-directory-failure flavor 550
fs:create-link-failure flavor 551
fs:creation-failure 550
fs:creation-failure flavor 550
fs:data-error flavor 546
fs:delete-failure 551
fs:delete-failure flavor 551
fs:device-not-found flavor 549
fs:directory-already-exists flavor 550
fs:directory-not-empty flavor 552
fs:directory-not-found flavor 549
fs:dont-delete-flag-set flavor 552
fset-carefully function 321
fset function 3, 564
fs:file-already-exists flavor 550
fs:file-lookup-error flavor 548
fs:file-not-found flavor 548
fs:file-open-for-output flavor 553
fs:file-operation-failure flavor 481, 546
Request Failures Based on **fs:file-request-failure** 546
fs:file-request-failure flavor 546
fs:file-error flavor 545
fs:file-locked flavor 552
fs:filepos-out-of-range flavor 552
fs:host-not-available flavor 546

fsignal function 482, 504, 517
fs:inconsistent-options flavor 552
fs:incorrect-access-to-directory flavor 549
fs:incorrect-access-to-file flavor 549
fs:invalid-byte-size flavor 552
fs:invalid-operation-for-directory flavor 550
fs:invalid-operation-for-link flavor 550
fs:invalid-pathname-component flavor 553
fs:invalid-pathname-syntax 550
fs:invalid-pathname-syntax flavor 550
fs:invalid-property-value flavor 551
fs:invalid-password flavor 547
fs:invalid-wildcard flavor 550
fs:link-target-not-found flavor 549
fs:login-problems flavor 547
fs:login-required flavor 548
fs:multiple-file-not-found flavor 548
fs:network-lossage flavor 547
fs:no-file-system flavor 546
fs:no-more-room flavor 552
fs:not-enough-resources flavor 547
fs:not-logged-in flavor 548
fs:not-available flavor 553
fs package 615
fs:parse-pathname-error flavor 553
fs:pathname-error flavor 553
fs:rename-across-directories flavor 551
fs:rename-across-hosts flavor 551
fs:rename-to-existing-file flavor 551
fs:rename-failure 551
fs:rename-failure flavor 551
fs:undefined-logical-pathname-translation flavor 554
fs:unimplemented-option flavor 552
fs:unknown-pathname-host flavor 553
fs:unknown-operation flavor 547
fs:unknown-property flavor 551
fs:unknown-user flavor 547
fs:wildcard-not-allowed flavor 550
fs:wrong-kind-of-file 550
fs:wrong-kind-of-file flavor 550
fsymeval function 3, 564
sys: **ftruncate** function 112
SCL and **ftype** declaration 640
Downward funargs 139
Upward funargs 139
Funargs and Lexical Closure Allocation 139, 140, 141, 312, 313
:funcall-inside-yourself message 454
funcall function 160
≠ function 100
≤ function 99
≥ function 99
***$** function 102
***** function 102
+ function 100
+$ function 100
- function 101

-$ function 101
// function 102
//$ function 103
1- function 104
1-$ function 105
1+ function 104
1+$ function 104
%32-bit-difference function 119
%32-bit-plus function 119
< function 99
<= function 99
= function 98
> function 98
>= function 98
abs function 101
add1 function 104
adjust-array-size function 249
aloc function 244
alpha-char-p function 270
alphalessp function 292
alphanumericp function 270
ap-1 function 244
ap-2 function 245
ap-leader function 245
append function 51, 56
apply function 8, 151, 159
ar-1 function 244
ar-2 function 244
aref function 235, 244, 277
arg function 165
arglist function 302, 323
%args-info function 325
args-info function 324
***array** function 263
array-#-dims function 247
array-active-length function 247
array-bits-per-element function 238
array-column-major-index function 248
array-dimension-n function 247
array-displaced-p function 248
array-element-size function 238
array-elements-per-q function 238
array-has-leader-p function 248
array-in-bounds-p function 248
array-indexed-p function 248
array-indirect-p function 248
array-leader-length function 249
array-push-extend function 251
array-push-portion-extend function 251
arraycall function 262, 263
array-dimensions function 247
arraydims function 247
array-grow function 249
array-leader function 245
array-length function 246
arrayp function 8, 69
array-pop function 252
array-push function 251

array-type function 246
array-types function 237
as-1 function 244
as-2 function 244
ascii function 293
ascii-to-char function 274
ascii-to-string function 290
ascii-code function 274
aset function 235, 244
ash function 115
ass function 65
assoc function 65
assq function 64
atan function 107
atan2 function 107
atom function 7
bigp function 8
bitblt function 254
bit-test function 114
boole function 114
both-case-p function 270
boundp function 562
boundp-in-closure function 336
breakon function 523
bug function 524
butlast function 53
byte function 116
byte-position function 116
byte-size function 116
caaaar function 43
caaadr function 43
caaar function 42, 43
caadar function 44
caaddr function 44
caadr function 42, 43
caar function 42, 43
cadaar function 44
cadadr function 44
cadar function 42, 43
caddar function 44
cadddr function 44
caddr function 42, 43
cadr function 42, 43
call function 161
car function 42, 83
car-location function 45
cdaaar function 44
cdaadr function 44
cdaar function 43
cdadar function 44
cdaddr function 44
cdadr function 43
cdar function 42, 43
cddaar function 44
cddadr function 44
cddar function 43
cdddar function 44
cddddr function 44

cdddr function 43
cddr function 42, 43
cdr function 42, 83
ceiling function 109
change-instance-flavor function 436
char≠ function 271
char≤ function 271
char≥ function 271
char-device-font function 269
char-mouse-button function 273
char-mouse-n-clicks function 274
char-not-equal function 271
char-not-greaterp function 272
char-not-lessp function 272
char-to-ascii function 274
char< function 271
char= function 270
char> function 271
character function 272
char-bit function 269
char-bits function 269
char-code function 269
char-downcase function 272
char-equal function 271
char-flipcase function 273
char-greaterp function 272
char-int function 272
char-lessp function 271
char-name function 273
char-standard function 275
char-subindex function 269
char-upcase function 272
circular-list function 50
cis function 107
Clm: Type Conversion Function 637
closure function 305, 335
closure-alist function 335
closure-function function 335
closurep function 9, 335
closure-variables function 336
clrhash function 74
clrhash-equal function 74
code-char function 270
complex function 108
complexp function 8
conjugate function 101
cons function 45, 56
cons-in-area function 45
continue-whopper function 463
copy-array-contents function 253
copy-array-contents-and-leader function 253
copy-array-portion function 253
copyalist function 50
copy-closure function 336
copylist function 50, 56
copylist* function 50
copy-readtable function 33
copysymbol function 567

copytree function 50
copytree-share function 50
cos function 106
cosd function 107
cosh function 107
dbg:condition-handled-p function 509, 510
dbg:frame-active-p function 497
dbg:frame-arg-value function 498
dbg:frame-local-value function 498
dbg:frame-next-active-frame function 496
dbg:frame-next-interesting-active-frame function 496
dbg:frame-next-nth-active-frame function 496
dbg:frame-next-nth-interesting-active-frame function 496
dbg:frame-next-nth-open-frame function 497
dbg:frame-next-open-frame function 496
dbg:frame-number-of-locals function 498
dbg:frame-number-of-spread-args function 497
dbg:frame-out-to-interesting-active-frame function 497
dbg:frame-previous-active-frame function 496
dbg:frame-previous-interesting-active-frame function 496
dbg:frame-previous-open-frame function 496
dbg:frame-real-function function 497
dbg:frame-real-value-disposition function 498
dbg:frame-self-value function 498
dbg:frame-total-number-of-args function 497
dbg:get-frame-function-and-args function 495
dbg:invoke-restart-handlers function 516
dbg:print-frame-locals function 499
dbg:print-function-and-args function 499
debugging-info function 302, 323
del function 62
del-if-not function 64
delete function 61
del-if function 64
delq function 61
denominator function 108
deposit-byte function 117
deposit-field function 117
describe function 69
describe-defstruct function 384
describe-flavor function 440
describe-package function 612
dfloat function 108
difference function 101
digit-char-p function 270
digit-char function 273
disassemble function 302
displace function 371
documentation function 302, 322
dpb function 117
dumparrays Maclisp function 262
eq function 10, 89
eql function 10
equal function 11
error function 482, 503
errorp function 9, 504
eval function 159
evenp function 97

every function 64
exp function 106
Expander function 144
export function 593, 611
expt function 105
false function 164
fboundp function 564
fdefine function 297, 316
fdefinedp function 321
fdefinition function 322
ferror function 482, 504
fifth function 46
fillarray function 252
fill-pointer function 245
find-all-symbols function 606
find-position-in-list function 60
find-position-in-list-equal function 61
first function 46
firstn function 53
fix function 107
fixnump function 8
fixp function 7
fixr function 108
flavor-allows-init-keyword-p function 439
float function 108
floatp function 7
flonump function 8
floor function 108
fmakunbound function 564
fourth function 46
fset function 3, 564
fset-carefully function 321
fsignal function 482, 504, 517
fsymeval function 3, 564
funcall function 160
function-cell-location function 564
functionp function 8
fundefine function 322
g-l-p function 236, 250
gcd function 105
gensym function 357, 567
get function 68
get-flavor-handler-for function 439
get-handler-for function 439
getchar function 292
getcharn function 293
gethash function 74
gethash-equal function 74
getl function 68
get-pname function 565
globalize function 613
graphic-char-p function 270
greaterp function 98
grindef function 302
haipart function 115
haulong function 115
ignore function 164
imagpart function 108

implode function 293
import function 573, 593, 611
instantiate-flavor function 435
int-char function 272
intern function 571, 593, 605
intern-local-soft function 606
intern-local function 593, 605
intern-soft function 605
intersection function 62
isqrt function 106
keywordp function 607
last function 48
ldb function 116
ldb-test function 117
ldiff function 54
length function 46
lessp function 99
lexpr-continue-whopper function 463
lexpr-send-if-handles function 431
lexpr-funcall function 160
lexpr-send function 160, 430
list function 48, 56
list* function 49, 56
list-array-leader function 253
list*-in-area function 49
list-in-area function 49, 56
listarray function 252
listify function 166
listp function 7
loadarrays Maclisp function 262
load-byte function 117
locate-in-closure function 335
locate-in-instance function 440
location-boundp function 85
location-contents function 84
location-makunbound function 84
locativep function 9
log function 106
logand function 113
%logdpb function 118
Logical **and** function 178
Logical **or** function 179
logior function 113
%logldb function 117
lognot function 114
logxor function 113
lower-case-p function 270
lsh function 114
macroexpand function 375
macroexpand-1 function 375
make-array-into-named-structure function 405
make-equal-hash-table function 72
make-hash-table function 72
make-mouse-char function 274
make-array function 235, 241
make-char function 270
make-condition function 505
make-heap function 77

make-instance function 430
make-list function 49, 56
make-package function 601
make-plane function 261
make-symbol function 566, 573
maknam function 293
makunbound function 562
makunbound-in-closure function 336
makunbound-globally function 562
map function 202
mapatoms function 608
mapatoms-all function 608, 625
mapc function 202
mapcan function 203
mapcar function 202
mapcon function 203
maphash function 74
maphash-equal function 74
maplist function 202
mask-field function 117
math:decompose function 259
math:determinant function 259
math:fill-2d-array function 260
math:invert-matrix function 259
math:list-2d-array function 260
math:multiply-matrices function 258
math:solve function 259
math:transpose-matrix function 259
max function 100
mem function 60
memass function 65
member function 60
memq function 59
mexp function 369
min function 100
minus function 101
minusp function 97
mod function 104
mouse-char-p function 273
name-char function 273
named-structure-invoke function 405
named-structure-p function 405
named-structure-symbol function 405
nbutlast function 53, 252
nconc function 52, 56
ncons function 45, 56
ncons-in-area function 45
neq function 10
nintersection function 62
nleft function 53
nlistp function 7
not function 12
nreconc function 53
nreverse function 51, 56
nsublis function 56
nsubst function 55
nsubstring function 240, 278
nsymbolp function 7

nth function 47
nthcdr function 48
null function 12
number-into-array function 281
numberp function 7
numerator function 108
nunion function 62
oddp function 97
operation-handled-p function 431
package-cell-location function 607
package-external-symbols function 611
package-shadowing-symbols function 612
package-use-list function 610
package-used-by-list function 610, 625
pairlis function 66
parse-ferror function 504
parse-number function 281
phase function 107
pkg-add-relative-name function 582, 610
pkg-contained-in function 625
pkg-create-package function 626
pkg-debug-copy function 625
pkg-delete-relative-name function 582, 610
pkg-find-package function 607
pkg-refname-alist function 625
pkg-super-package function 625
pkg-goto function 597
pkg-kill function 603
pkg-load function 625
pkg-name function 607
plane-aref function 261
plane-aset function 262
plane-default function 261
plane-extension function 261
plane-origin function 261
plane-ref function 261
plane-store function 262
plist function 565
plus function 100
plusp function 97
princ function 585
property-cell-location function 565
puthash function 74
puthash-equal function 74
putprop function 68
quotient function 102
random function 118
rass function 66
rassoc function 66
rassq function 65
rational function 108
rationalp function 8
read function 13, 56
realpart function 108
***rearray** Maclisp function 262
recompile-flavor function 438
record-source-file-name function 317, 319
rem function 62

rem-if-not function 63
remainder function 103
remhash function 74
remhash-equal function 74
rem-if function 64
remob function 573, 593, 606
remove function 62
remprop function 69
remq function 62
rest1 function 47
rest2 function 47
rest3 function 47
rest4 function 47
Return from **typeof** function 642
return-array function 250
return-list function 185
reverse function 51
rot function 115
round function 111
rplaca function 54, 56, 83, 236
rplacd function 54, 56, 83, 236
samepnamep function 293, 565
sassoc function 66
sassq function 66
SCL and **array-row-major-index** function 642
SCL and **compile-file** function 645
SCL and **decode-universal-time** function 645
SCL and **defsetf** function 639
SCL and **describe** function 645
SCL and **directory** function 645
SCL and **documentation** function 645
SCL and **dribble** function 645
SCL and **equal-typep** function 638
SCL and **functionp** function 638
SCL and **get-macro-character** function 644
SCL and **make-concatenated-stream** function 643
SCL and **make-echo-stream** function 643
SCL and **mismatch** function 641
SCL and **parse-namestring** function 644
SCL and **proclaim** function 640
SCL and **push** function 641
SCL and **pushnew** function 641
SCL and **read-delimited-list** function 644
SCL and **read-from-string** function 643
SCL and **read-char** function 643
SCL and **:rehash-size** keyword to **make-hash-table** function 641
SCL and **:rehash-threshold** keyword to **make-hash-table** function 641
SCL and **:replace** keyword for **delete-duplicates** function 640
SCL and **require** function 640
SCL and **return** function 639
SCL and **return-from** function 639
SCL and **set-macro-character** function 644
SCL and **set-syntax-from-char** function 643
SCL and **standard-char-p** function 640
SCL and **substitute** function 640
SCL and **substitute-if-not** function 640
SCL and **substitute-if** function 640

SCL and :test keyword to **make-hash-table** function 641
SCL **coerce** function 637
second function 46
send function 160, 430
send-if-handles function 431
set function 561
set-char-bit function 269
set-character-translation function 34
set-in-closure function 335
set-in-instance function 439
set-syntax-#-macro-char function 27, 36
set-syntax-from-char function 34
set-syntax-from-description function 34
set-syntax-macro-char function 26, 35
setarg function 166
set-globally function 561
setplist function 565
setsyntax function 36
setsyntax-sharp-macro function 36
seventh function 47
shadow function 574, 593, 612
shadowing-import function 573, 574, 593, 611
si:equal-hash function 75
si:flavor-allowed-init-keywords function 439
si:flavor-default-init-get function 440
si:flavor-default-init-putprop function 440
si:flavor-default-init-remprop function 440
si:function-spec-get function 322
si:function-spec-putprop function 322
signal function 482, 503, 517
signum function 106
si:loop-named-variable function 229
si:loop-tassoc function 228
si:loop-tequal function 228
si:loop-tmember function 228
sin function 106
sind function 106
sinh function 107
si:random-create-array function 119
si:random-initialize function 119
si:read-recursive function 27
si:rename-within-new-definition-maybe function 329
si:unencapsulate-function-spec function 328
sixth function 47
some function 64
sort function 56, 79
sort-grouped-array function 81
sort-grouped-array-group-key function 81
sortcar function 81
sqrt function 106
stable-sort function 81
stable-sortcar function 81
store-array-leader function 245
string function 278
string≠ function 282
string≤ function 282
string≥ function 283
string-capitalize-words function 286

string-exact-compare function 283
string-left-trim function 280
string-nconc-portion function 279
string-not-equal function 284
string-not-greaterp function 284
string-not-lessp function 284
string-reverse-search function 288
string-reverse-search-char function 287
string-reverse-search-exact function 287
string-reverse-search-exact-char function 286
string-reverse-search-not-char function 288
string-reverse-search-not-exact-char function 287
string-reverse-search-not-set function 290
string-reverse-search-set function 289
string-right-trim function 280
string-search-char function 287
%string-search-char function 289
string-search-exact function 287
%string-search-exact-char function 287
string-search-exact-char function 286
string-search-not-char function 287
string-search-not-exact-char function 286
string-search-not-set function 289
string-search-set function 289
string-to-ascii function 290
string< function 282
string= function 282
%string= function 283
string> function 282
string-append function 279
string-compare function 285
string-downcase function 285
%string-equal function 284
string-equal function 283
string-flipcase function 286
string-greaterp function 284
string-length function 278
string-lessp function 284
string-nconc function 279
string-nreverse function 280
stringp function 8
string-pluralize function 280
string-reverse function 280
string-search function 288
string-trim function 280
string-upcase function 285
sub1 function 104
sublis function 55
subrp function 8
subset function 63
subset-not function 63
subst function 55
substring function 278
swaphash function 74
swaphash-equal function 74
sxhash function 59, 76
symbolp function 7
symbol-package function 573, 607

symeval function 3, 561
symeval-in-closure function 335
symeval-in-instance function 439
symeval-globally function 562
sys:%1d-aloc function 246
sys:%1d-aref function 246
sys:%1d-aset function 246
sys:double-float-p function 8
sys:fceiling function 112
sys:fdefinition-location function 322
sys:ffloor function 111
sys:fround function 113
sys:ftruncate function 112
sys:function-parent function 319
sys:single-float-p function 8
sys:%string-exact-compare function 283
sys:%string-compare function 285
tailp function 61
tan function 106
tand function 107
tanh function 107
third function 46
***throw** function 200
times function 102
true function 164
truncate function 110
typep function 9, 335, 403, 576, 597
unbreakon function 524
uncompile function 316
undefflavor function 437
undefun function 322
unexport function 593, 611
union function 62
unuse-package function 593, 611
upper-case-p function 270
use-package function 571, 593, 610
value-cell-location function 563
values function 167
values-list function 167
where-is function 571, 612, 625
xcons function 45, 56
xcons-in-area function 45
zerop function 97
**** function 103
**** function 105
^ function 105
^$ function 105
caaaar function 42
caaadr function 42
caadar function 42
caaddr function 42
cadaar function 42
cadadr function 42
caddar function 42
cadddr function 42
cdaaar function 42
cdaadr function 42
cdaar function 42

cdadar function 42
cdaddr function 42
cdadr function 42
cddaar function 42
cddadr function 42
cddar function 42
cdddar function 42
cddddr function 42
cdddr function 42
SCL and **parse-pathname** function 644
Clm: What the Read Function Accepts 643
The Function Cell of a Symbol 563
The Compiled Function Data Type 4
Simple Function Definitions 300
SCL and **format** function directives 644
Undefined function error 534
Function for Evaluation 159
Evaluating a Function Form 151
Functions for Function Invocation 159
Clm: What the Print Function Produces 644
Basic definition of the function spec 297, 325
:handler function spec type 297
:internal function spec type 297
:location function spec type 297
:method function spec type 297
:property function spec type 297
Symbol function spec type 297
:within function spec type 297
function special form 162
SCL and **function** special form 638
Defining function specs 297
:method function specs 316
function-cell-location function 564
si: **function-spec-get** function 322
si: **function-spec-putprop** function 322
What is a Function? 297
Function abbreviation 28
Functional arguments 139
Functional objects 423
Function-calling Errors 539
Function cell 3
Function-defining Special Forms 305
SCL and **functionp** function 638
sys: **function-parent** declaration 312
Using the **sys:** **function-parent** Declaration 319
sys: **function-parent** function 319
functionp function 8
Function renaming 329
Functions 297
Abstract-operation functions 418
Access functions 147
Accessor functions 379, 401
Arguments to functions 323
Arrays as functions 235
Arrays used as functions 305
Basic Array Functions 241
Byte Manipulation Functions 115
Cdr storing functions 297

Clm: By-position Constructor Functions 642
Clm: Pathname Functions 644
Clm: Time Functions 645
Compiled functions 303, 305
Debugging info alist functions 302, 323, 325
Defining functions 297, 300, 305
Definitions of functions 316
Degrees in trigonometric functions 106
Documentation string functions 302, 322
Dynamic Closure-manipulating Functions 335
environment argument for macro-expander functions 138
Evaluation of special functions 303
Flavor Functions 435
Flavor system storing functions 297
Frame-manipulating functions 494
Handler-list searching functions 510
Hash Table Functions 73
How Programs Examine Functions 322
In-line coded functions 351
Interpreted functions 303, 304
Kinds of Functions 303
Lambda functions 304
Locate functions 147
Maclisp-compatible String-manipulation Functions 292
Macro-expander functions 368
Macros expanded to Lisp functions 355
Names of functions 297
Numeric Functions 97
Open coded functions 351
Operations the User Can Perform on Functions 302
Other Kinds of Functions 305
Printed representation of functions 297
Proceedable condition functions 521
Purpose of functions 339
Radians in trigonometric functions 106
Redefining functions 318
Restart handler functions 514
Select-method functions 305
si:digested-lambda functions 297, 300, 304, 305
Signalling functions 482, 503
Special functions 303
Stack groups used as functions 305
subst functions 351
Substitutable Functions 351
Symbols used as functions 297
Syntax errors in read functions 543
Transcendental Functions 106
Trigonometric functions 106
Update functions 147
Functions and Dynamic Closures 295
Macro-expander functions and lexical scoping 138
Some Functions and Special Forms 159
Functions and Special Forms for Constant Values 161
SCL sequence and list functions and two-argument predicates 641
Functions for Compatibility with Maclisp Lexprs 165
Functions for Creating Flavors 428
Functions for Function Invocation 159

Readtable Functions for Maclisp Compatibility 36
Handler Functions for Named Structures 403
Functions for Passing Messages 430
Storing functions on property lists 297
Functions That Change Character Syntax 34
Functions That Change Characters Into Macro Characters 35
Functions That Create New Readtables 33
Functions That Find the Home Package of a Symbol 607
Functions That Import, Export, and Shadow Symbols 611
Functions That Map Names to Symbols 605
Functions That Operate on Locatives 84
Functions That Operate on Named Structures 405
Functions that return multiple values 167
Functions to Expand Macros 375
Applying functions to list items 201
Functions Used During Expansion 368
Package Functions, Special Forms, and Variables 597
Function Specs 297
Fundamental Conditions 531
fundefine function 322

G G G

g-l-p function 236, 250
Gamma (γ) character 32
Hash Tables and the Garbage Collector 75
gcd function 105
Generalized Variables 147
Clm: Generalized Variables 639
Decrementing generalized variables 149
Incrementing generalized variables 149
Locating generalized variables 147
Updating generalized variables 147
Generator 333
Seed for random number generator 118
Generic Operations on Objects 421
gensym function 357, 567
get-flavor-handler-for function 439
dbg: **get-frame-function-and-args** function 495
get-handler-for function 439
:get-handler-for message 454
SCL and **get-macro-character** function 644
getchar function 292
getcharn function 293
get function 68
gethash-equal function 74
gethash function 74
:get-hash message 72
getl function 68
:getl message 473
:get message 473
get-pname function 565
:gettable-instance-variables Option for **defflavor** 441, 485
Getting Information About an Array 246

The **global** Package 571
globalize function 613
global package 615
Global symbols 571
Global variables 126
go special form 187, 189
SCL and **go** special form 639
Goto-less programming 194
Clm: Rules Governing the Passing of Multiple Values 639
SCL **gprint** package 633
graphic-char-p function 270
Reading Integers in Bases Greater Than 10 21
greaterp function 98
Greatest common denominator 105
Greatest common divisor 105
grindef function 302
Stack group state error 537
Grouped Arrays 401
Stack groups used as functions 305
:growth-factor init option for **si:eq-hash-table** 71

H

halpart function 115
Hand-Down character 32
:handle-condition-p message 511
:handle-condition message 511
Hand-Left character 32
Handler 479
Cleanup handler 197
Example of a Handler 481
Finding a Handler 501
:handler function spec type 297
Restart handler functions 514
Handler Functions for Named Structures 403
Interactive handler object 511
What is a Handler? 487
Handler-list searching functions 510
Binding handlers 487
Binding condition handlers 481
Bound handlers 488, 501, 502
Classes of Handlers 487
Default handlers 501, 502
Establishing Handlers 487
Interactive handlers 501, 511
Proceeding with **condition-bind** Handlers 520
Reference Material: Establishing Handlers 488
Reference Material: Restart Handlers 514
Restart handlers 501, 513
Search rule for invoking handlers 501, 509
Default Handlers and Complex Modularity 509
Reference Material: Default Handlers and Complex Modularity 510
Application: Handlers Examining the Stack 494
Reference Material: Application: Handlers Examining the Stack 495
Invoking Restart Handlers Manually 516
Overview and Definitions of Signalling and Handling 479
Introduction to Signalling and Handling Conditions 479
Hand-Right character 32

Hand-Up character 32
Hash table considerations while using multiprocessing 69
Hash table facilities 59
Hash Table Functions 73
Hash table keys 69
Objects as hash table keys 69
Trees as hash table keys 69
Hash Table Messages 72
Clm: Hash Tables 641
Creating Hash Tables 71
Hash Tables and Loop Iteration 75
Hash Tables and the Garbage Collector 75
loop Iteration Over Hash Tables or Heaps 224
Dumping Hash Tables to Files 75
:hash-inherited-symbols Option for **defpackage** and **make-package** 600, 603
Hasharray facility of Interlisp 69
hash-elements loop iteration path 224
Hashing 75
Hashing on **eq** 75
Hashing on **equal** 75
Hash Primitive 75
Hash table 75
Hash Tables 69
haulong function 115
Array header information 239
:clear method of **si:** **heap** 78
:delete-by-item method of **si:** **heap** 78
:delete-by-key method of **si:** **heap** 78
:describe method of **si:** **heap** 78
:empty-p method of **si:** **heap** 78
:find-by-item method of **si:** **heap** 78
:find-by-key method of **si:** **heap** 78
:insert method of **si:** **heap** 79
:remove method of **si:** **heap** 79
:top method of **si:** **heap** 79
heap-elements loop iteration path 224
Heaps 77
loop Iteration Over Hash Tables or Heaps 224
Messages to Heaps 78
Heaps and Loop Iteration 79
Help character 32
her loop keyword 222
Condition hierarchies 486
Condition flavor hierarchy 482
Hints for Using Array Registers 257
Hints to Macro Writers 357
his loop keyword 222
Hold-Output character 32
Home Package of a Symbol 573
Functions That Find the Home Package of a Symbol 607
Home package 606
fs: **host-not-available** flavor 546
sys: **host-not-responding-during-connection** flavor 555
sys: **host-not-responding** flavor 555
sys: **host-stopped-responding** flavor 555
SCL and non-null hosts 644

How Applications Programs Treat Conditions 481
How Programs Examine Functions 322
How Programs Manipulate Definitions 316
How the Package System Allows Symbol Sharing 569
How the Printer Works 13
How the Reader Recognizes Complex Numbers 24
How the Reader Recognizes Conses 25
How the Reader Recognizes Floating-point Numbers 22
How the Reader Recognizes Integers 20
How the Reader Recognizes Macro Characters 26
How the Reader Recognizes Ratios 22
How the Reader Recognizes Strings 25
How the Reader Recognizes Symbols 24
How the Reader Works 19

SCL I/O operations and streams 634
I/O to Strings 290
ibase variable 20
#/ character identifier 277
#\ character identifier 277
: character as keyword identifier 576
IEEE Floating-point Representation 94
SCL and **ieee-floating-point** feature name 645
if loop keyword 216
if special form 177
ignore-errors special form 493
ignore function 164
Character Comparisons Ignoring Case, Style, and Bits 271
String Comparisons Ignoring Case, Style, and Bits 283
String Searching Ignoring Case, Style, and Bits 287
SCL and *illegal* character attributes 643
Illegal redefinition error 538
imagpart function 108
Implementation of Flavors 467
implode function 293
:import option for **defpackage** 573
:import Option for **defpackage** and **make-package** 599, 602
:import option for **make-package** 573
Functions That Import, Export, and Shadow Symbols 611
:import-from Option for **defpackage** and **make-package** 600, 602
:import-from option for **make-package** 573
:import-from option for **defpackage** 573
import function 573, 593, 611
Importing and Exporting Symbols 573
Importing symbols 573, 589, 593, 611
in loop keyword 207, 222, 225
incf macro 149
:include Option for **defpackage** and **make-package** 600, 603
:include option for **defstruct** 385, 387
:included-flavors Option for **defflavor** 441, 444
Inclusive **or** 179

Inclusive or 113
String incompatibilities 634
Zetalisp and SCL character incompatibilities 265
Zetalisp and SCL string incompatibilities 265
fs: **inconsistent-options** flavor 552
fs: **incorrect-access-to-directory** flavor 549
fs: **incorrect-access-to-file** flavor 549
Incrementing generalized variables 149
index loop keyword 222, 225
Index offset 240, 241
Indicator 67
Property list indicators 67
Indirect array 240, 241, 248
Indirect Arrays 240
Conformal Indirection 240
Inexact-result trap 535
Floating-point infinity 14, 22
Reader macro for infix expressions 30
si:encapsulated-definition debugging info alist element 325
Debugging info alist functions 302, 323, 325
Array header information 239
Clm: Array Information 642
Font information 236, 277
Getting Information About an Array 246
Package Inheritance 571, 599, 602
Flavor inheritance mechanism 482
inhibit-fdefine-warnings variable 317
:proceed-types init option 519
:proceed-types init option for **condition** 530
:area init option for **si:eq-hash-table** 71
:growth-factor init option for **si:eq-hash-table** 71
:rehash-before-cold init option for **si:eq-hash-table** 71
:size init option for **si:eq-hash-table** 71
:rehash-threshold init option for **si:equal-hash-table** 72
:property-list init option for **si:property-list-mixin** 474
Messages and Init Options 529
:initable-instance-variables Option for **defflavor** 441, 485
:initial-dimensions option for **make-plane** 261
:initial-origins option for **make-plane** 261
:initial-value keyword for **make-list** 49
:initial-value option for **make-array** 241
Array initialization 241
Loop initialization code 205, 212
Initialization in structures 395
Initialization keywords 425
SCL and **defstruct** slot initializations 642
:initialize-special-commands message 526
Initializing instance variables 425, 435, 455
initially loop keyword 212
:initial-offset option for **defstruct** 385, 390
si: **initial-readtable** variable 33
:init-keywords Option for **defflavor** 441, 442
Init options 425
Init-plist 425
In-line coded functions 351
SCL and **inline** declaration 640
Clm: Input From Character Streams 643

Tokens in the input stream 19
Clm: Other Environment Inquiries 645
:insert method of **si:heap** 79
Inserting elements into list 150
Errors inside Lisp printer 543
Instance 305
:instance argument to **typep** 9
Unbound instance variable error 533
Instance variable errors 538
Instance variable out-of-bounds 541
Default values for instance variables 431
Initializing instance variables 425, 435, 455
Value of instance variables 439
Instance variables of combined flavors 431
sys: **instance-variable-pointer-out-of-range** flavor 538
sys: **instance-variable-zero-referenced** flavor 538
Instance descriptor 467
Instances 303, 417
Copying Instances 465
Printed Representation of Instances 16
Conditions as instances of flavors 479
Creating instances of flavors 430
Creating instances of structures 395
Instance variables 126, 418, 485
instantiate-flavor function 435
fast-aref instruction 255
fast-aset instruction 255
int-char function 272
integer loop data-type keyword 219
Circumflex (^) in integer syntax 20
Underscore (_) in integer syntax 20
Integer arrays 262
Integer denominator 93
Integer division 93
Integer iteration 192
Integer numerator 93
Integer radix 14, 20
Integers 4, 89, 93
Binary integers 89
How the Reader Recognizes Integers 20
Printed Representation of Integers 14
String representation of integers 277
Integers as array elements 236
Reading Integers in Bases Greater Than 10 21
Integral (∫) character 32
Interaction of Some Common Special Forms with Multiple Values 169
Interactive handler object 511
Issues for Interactive Use 523
Interactive handlers 501, 511
dbg: ***interactive-handlers*** variable 511
Interesting active frame 494
Interface between two modules 569
Internal interfaces 621
Qualified Package Names as interfaces 585
FOR facility in Interlisp 205
Hasharray facility of Interlisp 69
intern-local-soft function 606

Specifying Internal and External Symbols in Packages 585
:internal colon mode 601, 603
:internal function spec type 297
defstruct Internal Structures 407
Internal value cell 331
Internal interfaces 621
Internal symbols 570
Interned symbol 566
interned-symbols loop iteration path 225
The **interned-symbols** Path 225
intern function 571, 593, 605
Interning 604
Case sensitivity of interning 604
Interning Errors Based on **sys:package-error** 542
intern-local function 593, 605
intern-soft function 605
Interpackage Relations 610
Interpreted functions 303, 304
Lexically scoped interpreter 137
intersection function 62
into loop keyword 212
Introduction to Evaluation 123
Introduction to Flow of Control 175
Introduction to Keywords 575
Introduction to **loop** 205
Introduction to Macros 339
Introduction to Named Structures 403
Introduction to Numbers 89
Introduction to Package Name-conflict Errors 593
Introduction to Package Names 581
Introduction to Qualified Package Names 584
Introduction to Signalling and Handling Conditions 479
Introduction to Structure Macros 379
Introduction to Symbolics Common Lisp 631
Introduction to the Flavor System 417
Invalid contents in cell errors 533
Invalid file operation errors 550
Invalid pathname syntax errors 550
Invalid type code error 534
fs: **invalid-byte-size** flavor 552
fs: **invalid-operation-for-directory** flavor 550
fs: **invalid-operation-for-link** flavor 550
fs: **invalid-pathname-component** flavor 553
fs: **invalid-pathname-syntax** 550
fs: **invalid-pathname-syntax** flavor 550
fs: **invalid-property-value** flavor 551
sys: **invalid-function** flavor 542
fs: **invalid-password** flavor 547
fs: **invalid-wildcard** flavor 550
:inverse-list method combination type 455, 456
math: **invert-matrix** function 259
:invisible Option for **make-package** 601
Invisible packages 581, 582
Invisible pointer 56
Functions for Function Invocation 159
dbg: **invoke-restart-handlers** function 516
Search rule for invoking handlers 501, 509

Invoking Restart Handlers Manually 516
Errors Involving Lisp Printed Representations 543
What is a Dynamic Closure? 331
What is a Function? 297
What is a Handler? 487
Where Is Symbol (m-X) Zmacs command 571
isqrt function 106
Issues for Interactive Use 523
it loop keyword 216
Applying functions to list items 201
Clm: Searching Sequences for Items 641
Iteration 175, 189, 205
Hash Tables and Loop Iteration 75
Heaps and Loop Iteration 79
Integer iteration 192
List iteration 193
loop iteration 221
Package Iteration 608
Sequence Iteration 225
Variable of iteration 207
The Iteration Framework 221
The **loop** Iteration Macro 205
loop Iteration Over Hash Tables or Heaps 224
array-element loop iteration path 225
array-elements loop iteration path 225, 257
hash-elements loop iteration path 224
heap-elements loop iteration path 224
interned-symbols loop iteration path 225
local-interned-symbols loop iteration path 225
Iteration Paths 222
Defining Iteration Paths 227
Predefined Iteration Paths 225
Iteration-driving Clauses 207, 221
Iteration variables 221
its loop keyword 207, 222

J J J

SCL and **:junk-allowed** keyword for **parse-namestring** 644

K K K

%%kbd- byte specifiers and keyboard characters 268
ABORT key 502, 526
RESUME key 526
SUPER key 526
&key Lambda-list Keyword 310, 373
Using the RESUME key with floating-point conditions 535
%%kbd- byte specifiers and keyboard characters 268
Hash table keys 69
Objects as hash table keys 69
Special Keys 526
Trees as hash table keys 69
Customizing Debugger keystrokes 526
above loop keyword 207, 225
&allow-other-keys Lambda-list Keyword 310, 373

always loop keyword 215
and loop keyword 207, 210, 216, 222
append loop keyword 212
appending loop keyword 212
as loop keyword 207
&aux Lambda-list Keyword 310
being loop keyword 207, 222
below loop keyword 207, 225
&body Lambda-list Keyword 310
by loop keyword 207, 225
collect loop keyword 212
collecting loop keyword 212
count loop keyword 212
counting loop keyword 212
do loop keyword 212
doing loop keyword 212
downfrom loop keyword 207, 225
downto loop keyword 207, 225
each loop keyword 207, 222
else loop keyword 216
&environment Lambda-list Keyword 311, 374
&eval Lambda-list Keyword 310
finally loop keyword 212
first loop keyword 207
fixnum loop data-type keyword 219
flonum loop data-type keyword 219
for loop keyword 207
from loop keyword 207, 225
her loop keyword 222
his loop keyword 222
if loop keyword 216
in loop keyword 207, 222, 225
index loop keyword 222, 225
initially loop keyword 212
integer loop data-type keyword 219
into loop keyword 212
it loop keyword 216
its loop keyword 207, 222
&key Lambda-list Keyword 310, 373
&list-of Lambda-list Keyword 310
&local Lambda-list Keyword 310
maximize loop keyword 212
minimize loop keyword 212
named loop keyword 218
nconc loop keyword 212
nconcing loop keyword 212
never loop keyword 215
nodeclare loop keyword 210
notype loop data-type keyword 219
number loop data-type keyword 219
of loop keyword 222, 225
on loop keyword 207
:optional keyword 161
&optional keyword 151
&optional Lambda-list Keyword 309
&quote Lambda-list Keyword 310
repeat loop keyword 207
&rest keyword 151

&rest Lambda-list Keyword 309
return loop keyword 216, 218
&special Lambda-list Keyword 310
:spread keyword 161
sum loop keyword 212
summing loop keyword 212
the loop keyword 207, 222
their loop keyword 222
then loop keyword 207
thereis loop keyword 215
to loop keyword 207, 225
unless loop keyword 216
until loop keyword 214
upfrom loop keyword 207
using loop keyword 222, 225
when loop keyword 216
while loop keyword 214
&whole Lambda-list Keyword 311, 374
with loop keyword 210
with-key loop keyword 224
&aux keyword for **defmacro** 373
&body keyword for **defmacro** 373
&list-of keyword for **defmacro** 373
&optional keyword for **defmacro** 373
&rest keyword for **defmacro** 373
SCL and **:replace** keyword for **delete-duplicates** function 640
:area keyword for **make-list** 49
:initial-value keyword for **make-list** 49
:describe keyword for **named-structure-invoke** 403, 404
:print-self keyword for **named-structure-invoke** 403, 404
:which-operations keyword for **named-structure-invoke** 403, 404
SCL and **:junk-allowed** keyword for **parse-namestring** 644
: character as keyword identifier 576
Specifying a Keyword Parameter's Symbol in Lambda Lists 156
SCL and **:rehash-size** keyword to **make-hash-table** function 641
SCL and **:rehash-threshold** keyword to **make-hash-table** function 641
SCL and **:test** keyword to **make-hash-table** function 641
Keyword comparisons 227
keyword-extract special form 193
keyword package 615
Keyword parameters 151
keywordp function 607
& keywords 151, 309
Initialization keywords 425
Introduction to Keywords 575
Lambda-list Keywords 309
Property list keywords 67
Using Keywords 576
Keywords in argument lists 189
SCL and keywords in the ***features*** list 644
:keywords option for **defstruct-define-type** 410, 412
Keyword symbols 151
Two Kinds of Characters 268
Kinds of Functions 303
Other Kinds of Functions 305
Kinds of Variables 126

L L L

SCL arrays and circular-structure
labelling restriction 641
labels special form 144
flet, **labels**, and **macrolet** Special Forms 142
Lambda (λ) character 32
Examples of Simple Lambda Lists 154
Specifying a Keyword Parameter's Symbol in Lambda Lists 156
Specifying Aux-variables in Lambda Lists 157
Specifying Default Forms in Lambda Lists 155
Variables in lambda lists 151
lambda special form 163
&allow-other-keys Lambda-list Keyword 310, 373
lambda-list-keywords symbol 151
lambda-list-keywords variable 309
Lambda-binding 125, 331
Lambda-expression 151
SCL and **&rest** parameters for lambda-expressions 638
Lambda functions 304
Lambda-list 304
Lambda list 151
lambda list 323
&aux Lambda-list Keyword 310
&body Lambda-list Keyword 310
&environment Lambda-list Keyword 311, 374
&eval Lambda-list Keyword 310
&key Lambda-list Keyword 310, 373
&list-of Lambda-list Keyword 310
Clm: Lambda-expressions 638
&local Lambda-list Keyword 310
&optional Lambda-list Keyword 309
&quote Lambda-list Keyword 310
&rest Lambda-list Keyword 309
&special Lambda-list Keyword 310
&whole Lambda-list Keyword 311, 374
Default forms of lambda-list parameters 151
Lambda-list Keywords 309
lambda-macro special form 355
Lambda Macros 355
Lambda symbol 151
Lisp language package 615
SCL **language-tools** package 633
last function 48
ldb function 116
ldb-test function 117
ldiff function 54
Array leader 238, 241, 248
:leader-length option for **make-array** 241
:leader-list option for **make-array** 241
Array Leaders 238
Least bits 115
length function 46
lessp function 99
let* special form 129
let special form 128, 142
let-globally-if special form 131
let-closed special form 305, 335
letf* special form 130
letf special form 130

let-globally special form 131
let-if special form 130
Funargs and Lexical Closure Allocation 139, 140, 141, 312, 313
Lexical Environment Objects and Arguments 138
Lexical Scoping 137, 142, 144
defmacro and lexical scoping 138
macro and lexical scoping 138
Macro-expander functions and lexical scoping 138
Lexical closures 137, 303, 305, 331
Lexically scoped compiler 137
Lexically scoped interpreter 137
Lexical scope 126
Lexpr Maclisp type 305
lexpr-continue-whopper function 463
lexpr-send-if-handles function 431
lexpr-funcall function 160
Lexprs 151, 159
Functions for Compatibility with Maclisp Lexprs 165
lexpr-send function 160, 430
LF character 32
Matrices and Systems of Linear Equations 258
Simultaneous linear equations 258
Line character 32
fs: **link-target-not-found** flavor 549
Introduction to Symbolics Common Lisp 631
Symbolics Common Lisp 629
Printed Representation of Common Lisp Character Objects 16
Common Lisp Compatibility Package 615
SCL and Common Lisp Differences 637
SCL and Symbolics Common Lisp Extensions 635
SCL and Common Lisp Files 635
Macros expanded to Lisp functions 355
Lisp language package 615
Errors Involving Lisp Printed Representations 543
Errors inside Lisp printer 543
Common Lisp readtable 27
Lisp Errors 533
Lisp macros 337
List 41
Association list 238
BUG-LISPM mailing list 524
Circular list 41
Cons as property list 67
Disembodied property list 67
Dotted list 25, 41
File attribute list 579
Inserting elements into list 150
Lambda list 151
lambda list 323
Memory cell as property list 67
Property list 67
Removing elements from list 150
SCL and keywords in the ***features*** list 644
Symbol associated with property list 67
Zmacs Command: List Combined Methods 471
The List Data Type 5
Maximum number of list elements to be printed 18
SCL sequence and list functions and two-argument predicates 641

Property list indicators 67
Applying functions to list items 201
Property list keywords 67
Property List Messages 473
:list method combination type 455
Zmacs Command: List Methods 471
Property list of a symbol 3
The Property List of a Symbol 564
Basic List Operations 46
:list returned by **typep** 9
Alteration of List Structure 54
Manipulating List Structure 41
Property list values 67
math: **list-2d-array** function 260
list-array-leader function 253
list*-in-area function 49
list-in-area function 49, 56
&list-of Lambda-list Keyword 310
:list-or-nil argument to **typep** 9
listarray function 252
List elements 41
list* function 49, 56
list function 48, 56
listify function 166
List iteration 193
&list-of keyword for **defmacro** 373
listp function 7
Lists 39
Arrays as lists 236
Arrays Overlaid with Lists 250
Association lists 59, 64, 67
Clm: Lists 641
Clm: Sequences, Lists 641
Compact lists 56
Depth of recursion of printing lists 16
Examples of Simple Lambda Lists 154
Keywords in argument lists 189
Printing nested lists 17
Property Lists 67
Sorting lists 79
Sorting Arrays and Lists 79
Sorting compact lists 79
Specifying a Keyword Parameter's Symbol in Lambda Lists 156
Specifying Aux-variables in Lambda Lists 157
Specifying Default Forms in Lambda Lists 155
Storing functions on property lists 297
Variables in lambda lists 151
Lists as Tables 59
Lists as templates 345
lmfs package 615
loadarrays Maclisp function 262
load-byte function 117
Binding local and special variables 126
&local Lambda-list Keyword 310
Local Network Problems 554
local-interned-symbols loop iteration path 225
sys: **local-network-error** flavor 554
local-declarations variable 316

local-declare special form 126, 315
Local variables 126, 331
locate-in-closure function 335
locate-in-instance function 440
Locate functions 147
Locating generalized variables 147
:location function spec type 297
location-boundp function 85
location-contents function 84
Location Errors 534
location-makunbound function 84
The Locative Data Type 5
:locative returned by **typep** 9
locativep function 9
Locative pointer 147
Locatives 67, 83
Cdr-coding and Locatives 83
Cells and Locatives 83
Functions That Operate on Locatives 84
locf macro 148
loci macro 83
Adding new symbols to locked packages 626
External-only Packages and Locking 626
logand function 113
Logarithm of nonpositive number error 534
Natural logarithms 106
%logdpb function 118
log function 106
Error logging code 494
Logical **and** function 178
Complement logical operation 114
Logical Operations on Numbers 113
Logical **or** function 179
Login errors 547
Login Problems 547
fs: login-problems flavor 547
fs: login-required flavor 548
logior function 113
%logldb function 117
lognot function 114
logxor function 113
cl: long-float format 23
File Lookup 548
SCL case checking in package-name lookup 640
File lookup errors 548
SCL and file lookup errors 645
Clm: the Top-level Loop 642
Data Types Recognized by **loop** 219
Introduction to **loop** 205
Expressions in loop constructs 206
fixnum loop data-type keyword 219
flonum loop data-type keyword 219
integer loop data-type keyword 219
notype loop data-type keyword 219
number loop data-type keyword 219
Loop exit code 205, 212
Loop initialization code 205, 212
Hash Tables and Loop iteration 75

Heaps and Loop Iteration 79
The **loop** Iteration Macro 205
loop Iteration Over Hash Tables or Heaps 224
array-element **loop** iteration path 225
array-elements **loop** iteration path 225, 257
hash-elements **loop** iteration path 224
heap-elements **loop** iteration path 224
interned-symbols **loop** iteration path 225
local-interned-symbols **loop** iteration path 225
above **loop** keyword 207, 225
always **loop** keyword 215
and **loop** keyword 207, 210, 216, 222
append **loop** keyword 212
appending **loop** keyword 212
as **loop** keyword 207
being **loop** keyword 207, 222
below **loop** keyword 207, 225
by **loop** keyword 207, 225
collect **loop** keyword 212
collecting **loop** keyword 212
count **loop** keyword 212
counting **loop** keyword 212
do **loop** keyword 212
doing **loop** keyword 212
downfrom **loop** keyword 207, 225
downto **loop** keyword 207, 225
each **loop** keyword 207, 222
else **loop** keyword 216
finally **loop** keyword 212
first **loop** keyword 207
for **loop** keyword 207
from **loop** keyword 207, 225
her **loop** keyword 222
his **loop** keyword 222
if **loop** keyword 216
in **loop** keyword 207, 222, 225
index **loop** keyword 222, 225
initially **loop** keyword 212
into **loop** keyword 212
it **loop** keyword 216
its **loop** keyword 207, 222
maximize **loop** keyword 212
minimize **loop** keyword 212
named **loop** keyword 218
nconc **loop** keyword 212
nconcing **loop** keyword 212
never **loop** keyword 215
nodeclare **loop** keyword 210
of **loop** keyword 222, 225
on **loop** keyword 207
repeat **loop** keyword 207
return **loop** keyword 216, 218
sum **loop** keyword 212
summing **loop** keyword 212
the **loop** keyword 207, 222
their **loop** keyword 222
then **loop** keyword 207
thereis **loop** keyword 215

to **loop** keyword 207, 225
unless **loop** keyword 216
until **loop** keyword 214
upfrom **loop** keyword 207
using **loop** keyword 222, 225
when **loop** keyword 216
while **loop** keyword 214
with **loop** keyword 210
with-key **loop** keyword 224
SCL **loop** macro 635
si: **loop-named-variable** function 229
si: **loop-use-system-destructuring?** variable 221
Loop epilogue 205, 212
loop-finish macro 215
loop iteration 221
loop macro 205
Loop prologue 205, 212
Bindings in loops 210, 220
Evaluation in loops 212
loop Synonyms 218
si: **loop-tassoc** function 228
si: **loop-tequal** function 228
Loop termination 212, 214
si: **loop-tmember** function 228
lower-case-p function 270
lsh function 114
Lsubrs 151

M M M

`m-sh-D` command 302
Zmacs Command: `m-.` 471
Maclisp 205, 305
Maclisp Array Compatibility 262
Readtable Functions for Maclisp Compatibility 36
dumparrays Maclisp function 262
loadarrays Maclisp function 262
***rearray** Maclisp function 262
Functions for Compatibility with Maclisp Lexprs 165
Maclisp property names 564
Maclisp system property names 564
expr Maclisp type 305
fexpr Maclisp type 305
Lexpr Maclisp type 305
macro Maclisp type 305
Maclisp-compatible String-manipulation Functions 292
Maclisp compatibility 503
Macro 144
#◊ Reader Macro 30
#- Reader Macro 30
#' Reader Macro 28
#+ Reader Macro 29
#, Reader Macro 28
#. Reader Macro 29
#(Reader Macro 30
#b Reader Macro 29
#m Reader Macro 29
#n Reader Macro 29

#o Reader Macro 29
#P Reader Macro 643
#q Reader Macro 29
#r Reader Macro 29
#x Reader Macro 29
or **#/** Reader Macro 27
#^ Reader Macro 28
#| Reader Macro 30
array macro 263
check-arg-type macro 506
check-arg macro 505
compile-flavor-methods macro 438, 485
dbg:with-erring-frame macro 494, 495
decf macro 149
defflavor macro 428
@define macro 308
define-loop-macro macro 218
define-loop-path macro 227
define-loop-sequence-path macro 225
defmacro macro 344
defmacro-displace macro 372
defmethod macro 429
defstruct macro 345, 383
defstruct-define-type macro 409
defunp macro 302, 305
defwhopper-subst macro 464
defwrapper macro 461
incf macro 149
locf macro 148
loci macro 83
loop macro 205
loop-finish macro 215
once-only macro 365
pkg-bind macro 598
pop macro 150
push macro 150
push-in-area macro 150
SCL and **#-** Reader Macro 644, 645
SCL and **#+** Reader Macro 644, 645
SCL and **defstruct** macro 642
SCL and **prog1** macro 639
SCL and **prog2** macro 639
SCL and **progv** macro 639
SCL and returned values from **cond** macro 639
SCL **loop** macro 635
setf macro 147
si:define-simple-method-combination macro 459
si:encapsulate macro 326
swapf macro 149
sys:defsubst-with-parent macro 319
sys:printing-random-object macro 18
The **loop** Iteration Macro 205
undefmethod macro 437, 468
unless macro 179
unwind-protect-case macro 199
when macro 179
macro and lexical scoping 138
:macro argument to **setsyntax** 36

Backquote (`) macro character 26
Comma (,) macro character 26
Quote (') macro character 26
Macro character 26
SCL and *non-terminating* *macro* character attribute 643
Clm: Macro Characters 643
Functions That Change Characters Into Macro Characters 35
How the Reader Recognizes Macro Characters 26
Clm: Macro Definition 639
SCL and equivalent macro definitions for special forms 638
Displaced macro expansions 371
Reader macro for infix expressions 30
SCL and **#S** macro form 642
macro Maclisp type 305
macro special form 305, 339, 355
si: **macro** syntax description 34
Hints to Macro Writers 357
Macro character 32
Semicolon (;) macro character 26
Sharp-sign (#) macro character 26
Clm: Standard Dispatching Macro Character Syntax 644
Selective evaluation in macro definitions 345
Macro-defining macros 343, 345
macroexpand-1 function 375
environment argument for macro-expander functions 138
Macro-expander functions and lexical scoping 138
Macro-expander functions 368
macroexpand function 375
si: ***macroexpand-hook*** variable 375
macrolet special form 144
flet, labels, and **macrolet** Special Forms 142
Macros 303, 305, 337
reader macros 27
Aid for Debugging Macros 369
Aids for Defining Macros 343
Alterant Macros 397
By-position Constructor Macros 396
Comments in macros 26
Constructor macros 379, 395
Displacing Macros 371
Functions to Expand Macros 375
Introduction to Macros 339
Introduction to Structure Macros 379
Lambda Macros 355
Lisp macros 337
Macro-defining macros 343, 345
Nesting Macros 366
Sharp-sign Reader Macros 27
Structure Macros 377
Symbol Macros 353
Using the Constructor and Alterant Macros 395
Macros expanded to Lisp functions 355
Macros Expanding Into Many Forms 360
Macros That Surround Code 362
Complex magnitude 101
BUG-LISPM mailing list 524
:displaced-conformally option for **make-array** 241
:displaced-index-offset option for **make-array** 240, 241

:displaced-to option for **make-array** 239, 241
:fill-pointer option for **make-array** 241
:initial-value option for **make-array** 241
:leader-length option for **make-array** 241
:leader-list option for **make-array** 241
:named-structure-symbol option for **make-array** 241
make-array-into-named-structure function 405
SCL and **make-concatenated-stream** function 643
SCL and **make-echo-stream** function 643
make-equal-hash-table function 72
make-hash-table function 72
SCL and **:rehash-size** keyword to **make-hash-table** function 641
SCL and **:rehash-threshold** keyword to **make-hash-table** function 641
SCL and **:test** keyword to **make-hash-table** function 641
:initial-value keyword for **make-list** 49
make-mouse-char function 274
:colon-mode option for **make-package** 585
:colon-mode Option for **defpackage** and **make-package** 601, 603
:external-only Option for **defpackage** and **make-package** 600, 603
:hash-inherited-symbols Option for **defpackage** and **make-package** 600, 603
:import-from option for **make-package** 573
:import-from Option for **defpackage** and **make-package** 600, 602
:new-symbol-function Option for **defpackage** and **make-package** 601, 603
:prefix-intern-function Option for **defpackage** and **make-package** 601, 603
:prefix-name Option for **defpackage** and **make-package** 599, 601
:relative-names option for **make-package** 582
:relative-names Option for **defpackage** and **make-package** 600, 602
:relative-names-for-me option for **make-package** 582
:relative-names-for-me Option for **defpackage** and **make-package** 600, 602
:shadowing-import option for **make-package** 573, 574
:shadowing-import Option for **defpackage** and **make-package** 600, 602
:default-value option for **make-plane** 261
:initial-dimensions option for **make-plane** 261
:initial-origins option for **make-plane** 261
:area option for **make-array** 241
:type option for **make-array** 241
:make-array option for **defstruct** 385, 389, 395
make-array function 235, 241
make-char function 270
make-condition function 505
make-heap function 77
make-instance function 430
:area keyword for **make-list** 49
make-list function 49, 56
:export option for **make-package** 573
:export Option for **defpackage** and **make-package** 599, 602
:import option for **make-package** 573
:import Option for **defpackage** and **make-package** 599, 602
:include Option for **defpackage** and **make-package** 600, 603
:invisible Option for **make-package** 601
:nicknames Option for **defpackage** and **make-package** 599, 601
:shadow option for **make-package** 574
:shadow Option for **defpackage** and **make-package** 599, 602
:size Option for **defpackage** and **make-package** 600, 602
:use option for **make-package** 571, 602
make-package function 601
:extension option for **make-plane** 261

:type option for **make-plane** 261
make-plane function 261
make-symbol function 566, 573
maknam function 293
makunbound-in-closure function 336
makunbound function 562
makunbound-globally function 562
How Programs Manipulate Definitions 316
Manipulating List Structure 41
Manipulating the readtable 33
Bit manipulation 114
Byte Manipulation Functions 115
Invoking Restart Handlers Manually 516
Macros Expanding Into Many Forms 360
Functions That Map Names to Symbols 605
mapatoms-all function 608, 625
mapatoms function 608
mapcan function 203
mapcar function 202
mapc function 202
mapcon function 203
map function 202
maphash-equal function 74
maphash function 74
:map-hash message 73
maplist function 202
Mapping 201
Mapping Between Names and Packages 607
Mapping from names to symbols 559
Mapping Names to Symbols 604
Break on exit from marked frame message 523
mask-field function 117
Reference Material: Default Handlers and Complex Modularity 510
Reference Material: Establishing Handlers 488
Reference Material: Proceeding 521
Reference Material: Restart Handlers 514
Reference Material: Signalling Conditions 503
Reference Material: Application: Handlers Examining the Stack 495
math:decompose function 259
math:determinant function 259
math:fill-2d-array function 260
math:invert-matrix function 259
math:list-2d-array function 260
math:multiply-matrices function 258
math:singular-matrix flavor 535
math:solve function 259
math:transpose-matrix function 259
Matrices and Systems of Linear Equations 258
Singular matrix operation error 535
Matrix operations 258
max function 100
maximize loop keyword 212
Maximum number of list elements to be printed 18
m-BREAK Debugger command 532
MDL programming environment 205
Encapsulation mechanism 523

Flavor inheritance mechanism 482
Signalling Mechanism 501, 509
memass function 65
member function 60
mem function 60
Memory allocation of conses 56
Memory cell as property list 67
memq function 59
Break on exit from marked frame message 523
:bug-report-description message 525
:bug-report-recipient-system message 524
:clear-hash message 73
:describe message 453
:directory-pathname message 549
:document-proceed-type message 517
:document-special-command message 526
:eval-inside-yourself message 454
:fasd-form message 465
:filled-elements message 73
:funcall-inside-yourself message 454
:get message 473
:get-handler-for message 454
:get-hash message 72
:getl message 473
:handle-condition-p message 511
:handle-condition message 511
:initialize-special-commands message 526
:map-hash message 73
:modify-hash message 73
:operation-handled-p message 453
:print-self message 453
:proceed message 520
:proceed-type-p message 519
:proceed-types message 519
:property-list message 474
:push-property message 473
:put-hash message 72
:putprop message 473
:rem-hash message 73
:remprop message 473
:report message 529
:send-if-handles message 454
Sending a message 421
:set-property-list message 474
:size message 73
:special-command message 525
:swap-hash message 73
:unclaimed-message message 454
:which-operations message 453
No method for message 542
:print-self message 16
Message Passing in the Flavor System 423
Message to an object of some flavor 461
Send message to self 159
:package message to **sys:external-symbol-not-found** 619
:string message to **sys:external-symbol-not-found** 619
:name message to **sys:package-not-found** 542, 619
:relative-to message to **sys:package-not-found** 542, 619

:package message to **sys:package-locked** 543, 619
:symbol message to **sys:package-locked** 543, 619
Message names 423
Message passing 421, 423
Message-receiving object 423
Flavor system messages 453
Functions for Passing Messages 430
Hash Table Messages 72
Proceed Type Messages 519
Property List Messages 473
Relationship between methods and messages 421
Sending messages 159
Messages and Init Options 529
Messages to Heaps 78
Method 421
Combined method 431
:proceed method 517, 520
:report method 485
:case method combination 517
:and method combination type 455
:append method combination type 455, 456
:case method combination type 455, 457
:daemon method combination type 455
:daemon-with-and method combination type 455, 456
:daemon-with-or method combination type 455, 456
:daemon-with-override method combination type 455, 457
:inverse-list method combination type 455, 456
:list method combination type 455
:nconc method combination type 455, 456
:or method combination type 455
:pass-on method combination type 455, 456
:progn method combination type 455
No method for message 542
:method function spec type 297
:method function specs 316
:document-proceed-type method of **condition** 529
:proceed-type-p method of **condition** 530
:proceed-types method of **condition** 530
:report method of **condition** 530
:report-string method of **condition** 531
:set-proceed-types method of **condition** 530
:special-command-p method of **condition** 530
:special-commands method of **condition** 530
:clear method of **si:heap** 78
:delete-by-item method of **si:heap** 78
:delete-by-key method of **si:heap** 78
:describe method of **si:heap** 78
:empty-p method of **si:heap** 78
:find-by-item method of **si:heap** 78
:find-by-key method of **si:heap** 78
:insert method of **si:heap** 79
:remove method of **si:heap** 79
:top method of **si:heap** 79
:after method type 455, 457
:and method type 455, 458
:before method type 455, 457
:combined method type 455, 458
:default method type 455, 457

:or method type 455, 458
:override method type 455, 457
:whopper method type 455, 458
:wrapper method type 455, 458
Combination Method Types 459
Method Combination 455
:method-combination Option for **defflavor** 441, 449
:method-order Option for **defflavor** 441, 449
Adding new methods 438
After-daemon methods 431
Before-daemon methods 431
Combined methods 438
Combining methods 461
Creating methods 429
Daemon methods 431
Defining methods 429
Modifying methods 468
Primary methods 431
Removing methods 437
Zmacs Command: Edit Methods 471
Zmacs Command: Edit Combined Methods 472
Zmacs Command: List Methods 471
Zmacs Command: List Combined Methods 471
Relationship between methods and messages 421
Ordering Flavors, Methods, and Wrappers 467
mexp function 369
min function 100
minimize loop keyword 212
minus function 101
minusp function 97
Printed Representation of Miscellaneous Data Types 18
Miscellaneous file operations failures 552
Miscellaneous Operations Failures 552
Miscellaneous Other Clauses 218
Miscellaneous System Errors Not Categorized by Base Flavor 536
SCL and **mismatch** function 641
Mixin flavor 451
Mixing Flavors 431
:mixture Option for **defflavor** 441, 445
:external colon mode 601, 603
:internal colon mode 601, 603
mod function 104
:modify-hash message 73
Clm: Modifying Sequences 640
Modifying flavors 438, 468
Modifying methods 468
Modifying wrappers 468
Default Handlers and Complex Modularity 509
Reference Material: Default Handlers and Complex Modularity 510
Modularity and Object-oriented Programming 418
Clm: Modules 640
Interface between two modules 569
tape: **mount-error** flavor 556
Mouse-1-1 32
Mouse-1-2 32
Mouse-2-1 32
Mouse-2-2 32

Mouse-3-1 32
Mouse-3-2 32
mouse-char-p function 273
Mouse-L-1 32
Mouse-L-2 32
Mouse-M-1 32
Mouse-M-2 32
Mouse-R-1 32
Mouse-R-2 32
Mouse Characters 273
Multics external arrays 262
Multidimensional array subscripts 245
Accessing Multidimensional Arrays as One-dimensional 245
Multidimensional Arrays 237
SCL and multidimensional arrays 641
Multilevel Qualified Package Names 586
Multipackage Programs 621
Multiple and Out-of-order Evaluation 363
Clm: Rules Governing the Passing of Multiple Values 639
Functions that return multiple values 167
Interaction of Some Common Special Forms with Multiple Values 169
Passing-back of Multiple Values 169
Primitive for Producing Multiple Values 167
Special Forms for Receiving Multiple Values 167
fs: **multiple-file-not-found** flavor 548
multiple-value-bind special form 168
multiple-value-call special form 168
multiple-value-list special form 168
multiple-value-prog1 special form 169
multiple-value special form 168
Multiple Values 167
Multiplication 102
math: **multiply-matrices** function 258
Hash table considerations while using multiprocessing 69
Where Is Symbol (m-X) Zmacs command 571

N N N

SCL and **ieee-floating-point** feature name 645
Package name conflicts 581
Symbol name conflicts 574
:name message to **sys:package-not-found** 542, 619
Print name of a symbol 3
The Print Name of a Symbol 565
name-char function 273
Checking for Package Name-conflict Errors 593
Introduction to Package Name-conflict Errors 593
Package Name-conflict Errors 593
Resolving Package Name-conflict Errors 595
sys: **name-conflict** flavor 619
Name Conflicts 357, 613
Special Form for Declaring a Named Constant 135
named loop keyword 218
:named option for **defstruct** 385, 389
SCL and **:named** option for **defstruct** 642
:named option for **defstruct-define-type** 410, 411
Named structure array 238, 241

Named structure symbol 403
Functions That Operate on Named Structures 405
Handler Functions for Named Structures 403
Introduction to Named Structures 403
Printed Representation of Arrays That Are Named Structures 16
Printed Representation of Arrays That Are Not Named Structures 17
:describe keyword for **named-structure-invoke** 403, 404
:print-self keyword for **named-structure-invoke** 403, 404
:which-operations keyword for **named-structure-invoke** 403, 404
named-structure-invoke function 405
named-structure-p function 405
:named-structure-symbol option for **make-array** 241
named-structure-symbol function 405
Named Structures 403
Names 559
Character Names 273
Data type names 576
Forms of qualified names 585
Introduction to Package Names 581
Introduction to Qualified Package Names 584
Maclisp property names 564
Maclisp system property names 564
Message names 423
Multilevel Qualified Package Names 586
Package Names 581
Proceed Type Names 520
Qualified names 589
Qualified Package Names 584
Relative Package Names 582, 610
SCL and printed character names 644
Shadowing package names 582
Special Character Names 32
Structure names 383
Mapping Between Names and Packages 607
Qualified Package Names as Interfaces 585
Names of functions 297
Qualified Names of Symbols 585
Functions That Map Names to Symbols 605
Mapping Names to Symbols 604
Mapping from names to symbols 559
Naming convention 7
Natural logarithms 106
nbutlast function 53, 252
nconc loop keyword 212
:nconc method combination type 455, 456
nconc function 52, 56
nconcing loop keyword 212
ncons-in-area function 45
ncons function 45, 56
Example of the Need for Packages 560
The Need for Packages 559
Testing for negative number 97
Square root of a negative number error 536
sys: **negative-sqrt** flavor 536
neq function 10
Printing nested lists 17
Nesting Macros 366

neti package 615
net package 615
Network connection problems 555
Local Network Problems 554
Remote Network Problems 554
sys: **network-resources-exhausted** flavor 554
sys: **network-stream-closed** flavor 556
Network character 32
sys: **network-error** flavor 554
Network Errors 554
network-internals package 615
fs: **network-lossage** flavor 547
network package 615
never loop keyword 215
Creating New Conditions 485
Adding new methods 438
Functions That Create New Readtables 33
Clm: Creating New Streams 643
Adding new symbols to locked packages 626
:new-symbol-function Option for **defpackage** and **make-package** 601, 603
:new-name proceed type 542, 619
Next frame 494
:nicknames Option for **defpackage** and **make-package** 599, 601
NIL 205, 220
:proceed Can Return **nil** 519
nil argument to **setsyntax** 36
nil symbol 561
nintersection function 62
nleft function 53
nlistp function 7
No method for message 542
sys: **no-action-mixin** flavor 532
condition-call and **:no-error** 493
fs: **no-file-system** flavor 546
fs: **no-more-room** flavor 552
:no-vanilla-flavor Option for **defflavor** 441, 445
:no-action proceed type 523, 542, 543, 619
nodeclare loop keyword 210
:non-complex-number argument to **typep** 9
sys: **non-positive-log** flavor 534
Nonlocal exit control structures 175
Nonlocal exit 197
Nonlocal Exits 197
SCL and non-null hosts 644
Logarithm of nonpositive number error 534
Character code for nonprinting characters 27
Support for Nonstandard Character Sets 275
SCL and *non-terminating macro* character attribute 643
***nopoint** variable 14
Miscellaneous System Errors Not Categorized by Base Flavor 536
Printed Representation of Arrays That Are Not Named Structures 17
fs: **not-enough-resources** flavor 547
fs: **not-logged-in** flavor 548
Exponential notation 14
fs: **not-available** flavor 553
Note on **record-source-file-name** 318

not function 12
SCL and **notinline** declaration 640
notype loop data-type keyword 219
nreconc function 53
nreverse function 51, 56
nsublis function 56
nsubst function 55
nsubstring function 240, 278
nsymbolp function 7
nthcdr function 48
nth function 47
:null argument to **typep** 9
null function 12
Testing for even number 97
Testing for negative number 97
Testing for odd number 97
Testing for positive number 97
Testing for sign of a number 97
:number argument to **typep** 9
Logarithm of nonpositive number error 534
Square root of a negative number error 536
Seed for random number generator 118
Read rational number in binary 29
number loop data-type keyword 219
Maximum number of list elements to be printed 18
Symbolics-Lisp Number Types 89
sys: **number-array-not-allowed** flavor 541
number-into-array function 281
numberp function 7
Numbers 87
32-bit Numbers 119
Addition of 32-bit numbers 119
Coercion Rules for Numbers 90
Complex numbers 89, 95
Conversion of numbers 90
Double-precision floating-point numbers 89
Fixed-point numbers 7
Floating-point numbers 4, 7, 94
How the Reader Recognizes Complex Numbers 24
How the Reader Recognizes Floating-point Numbers 22
Introduction to Numbers 89
Logical Operations on Numbers 113
Printed Representation of Numbers 90
Printed Representation of Complex Numbers 15
Printed Representation of Floating-point Numbers 14
Random Numbers 118
Rational Numbers 93
SCL and potential numbers 644
Single-precision floating-point numbers 89
Subtraction of 32-bit numbers 119
Types of Numbers 93
Types of numbers 4
Font numbers and character objects 267
Clm: Parsing of Numbers and Symbols 643
Numbers in the Compiler 90
Integer numerator 93
numerator function 108
Numeric argument descriptor 324

%%arg-desc-interpreted numeric argument descriptor field 324
%arg-desc-interpreted numeric argument descriptor field 324
%%arg-desc-max-args numeric argument descriptor field 324
%%arg-desc-min-args numeric argument descriptor field 324
%%arg-desc-quoted numeric argument descriptor field 324
%%arg-desc-rest-arg numeric argument descriptor field 324
Numeric Data Types 4
Numeric Type Conversions 107
Numeric arrays 5
Numeric Comparisons 98
Numeric Functions 97
Numeric Predicates 97
nunion function 62

O

O

Obarrays 262
Controlling the Printed Representation of an Object 19
Error object 529
Interactive handler object 511
Message-receiving object 423
Character Object Details 265
Message to an object of some flavor 461
Object comparisons 10
Object-oriented programming 417
Modularity and Object-oriented Programming 418
Basic Objects 1
Character Objects 265
Condition objects 479, 481, 519, 529
Creating condition objects 505
Font numbers and character objects 267
Functional objects 423
Generic Operations on Objects 421
Printed Representation of Common Lisp Character Objects 16
Lexical Environment Objects and Arguments 138
Objects and the Flavor System 417
Objects as hash table keys 69
Character objects code field 265
Character objects font field 265
Character objects style field 265
Reading octal character codes 25
Testing for odd number 97
oddp function 97
of loop keyword 222, 225
Index offset 240, 241
on loop keyword 207
once-only macro 365
Accessing Multidimensional Arrays as One-dimensional 245
Open coded functions 351
Open-coding 418
Open frame 494
Open subroutine 351
Functions That Operate on Locatives 84
Functions That Operate on Named Structures 405
Complement logical operation 114
Shadowing-import operation 573
Singular matrix operation error 535
Invalid file operation errors 550

O

operation-handled-p function 431
:operation-handled-p message 453
Basic List Operations 46
Basic String Operations 278
Boolean operations 113
Matrix operations 258
Truth table for the Boolean operations 114
SCL I/O operations and streams 634
Miscellaneous Operations Failures 552
Miscellaneous file operations failures 552
Performing arithmetic operations on characters in SCL 265
Logical Operations on Numbers 113
Generic Operations on Objects 421
Operations the User Can Perform on Functions 302
SCL and Optimization 635
SCL and **optimize** declaration 640
:proceed-types init option 519
:proceed-types init option for **condition** 530
:abstract-flavor Option for **defflavor** 441, 450
:accessor-prefix Option for **defflavor** 441, 448
:default-init-plist Option for **defflavor** 441, 442
:default-handler Option for **defflavor** 441, 447
:documentation Option for **defflavor** 441, 449
:export-instance-variables Option for **defflavor** 441, 448
:gettable-instance-variables Option for **defflavor** 441, 485
:included-flavors Option for **defflavor** 441, 444
:initable-instance-variables Option for **defflavor** 441, 485
:init-keywords Option for **defflavor** 441, 442
:method-combination Option for **defflavor** 441, 449
:method-order Option for **defflavor** 441, 449
:mixture Option for **defflavor** 441, 445
:no-vanilla-flavor Option for **defflavor** 441, 445
:ordered-instance-variables Option for **defflavor** 441, 447
:outside-accessible-instance-variables Option for **defflavor** 441, 447
:required-init-keywords Option for **defflavor** 441, 442
:required-instance-variables Option for **defflavor** 441, 442
:required-flavors Option for **defflavor** 441, 443
:required-methods Option for **defflavor** 441, 442
:settable-instance-variables Option for **defflavor** 441
:special-instance-variables Option for **defflavor** 441, 448
:colon-mode option for **defpackage** 585
:export option for **defpackage** 573
:import option for **defpackage** 573
:import-from option for **defpackage** 573
:relative-names-for-me option for **defpackage** 582
:relative-names option for **defpackage** 582
:shadow option for **defpackage** 574
:shadowing-import option for **defpackage** 573, 574
:use option for **defpackage** 571, 599
:colon-mode Option for **defpackage** and **make-package** 601, 603
:export Option for **defpackage** and **make-package** 599, 602
:external-only Option for **defpackage** and **make-package** 600, 603
:hash-inherited-symbols Option for **defpackage** and **make-package** 600, 603
:import Option for **defpackage** and **make-package** 599, 602
:import-from Option for **defpackage** and **make-package** 600, 602
:include Option for **defpackage** and **make-package** 600, 603
:new-symbol-function Option for **defpackage** and **make-package** 601, 603
:nicknames Option for **defpackage** and **make-package** 599, 601

:prefix-intern-function Option for **defpackage** and **make-package** 601, 603
:prefix-name Option for **defpackage** and **make-package** 599, 601
:relative-names Option for **defpackage** and **make-package** 600, 602
:relative-names-for-me Option for **defpackage** and **make-package** 600, 602
:shadow Option for **defpackage** and **make-package** 599, 602
:shadowing-import Option for **defpackage** and **make-package** 600, 602
:size Option for **defpackage** and **make-package** 600, 602
:alterant option for **defstruct** 385, 386
:but-first option for **defstruct** 385, 390
:callable-accessors option for **defstruct** 385, 391
:conc-name option for **defstruct** 385, 387
:constructor option for **defstruct** 385, 386, 396
:copier option for **defstruct** 385, 392
:default-pointer option for **defstruct** 385, 386
:eval-when option for **defstruct** 335, 391
:export option for **defstruct** 385, 386
:include option for **defstruct** 385, 387
:initial-offset option for **defstruct** 385, 390
:make-array option for **defstruct** 385, 389, 395
:named option for **defstruct** 385, 389
:predicate option for **defstruct** 385, 392
:print option for **defstruct** 385, 391
:property option for **defstruct** 385, 391
SCL and **:named** option for **defstruct** 642
SCL and **:type** option for **defstruct** 642
:size-macro option for **defstruct** 385, 390
:size-symbol option for **defstruct** 385, 390
:times option for **defstruct** 385, 390, 395
:type option for **defstruct** 385
:array option for **defstruct :type** 385
:array-leader option for **defstruct :type** 385
:fixnum option for **defstruct :type** 385, 386
:grouped-array option for **defstruct :type** 385, 386
:list option for **defstruct :type** 385
:named-array-leader option for **defstruct :type** 385
:named-array option for **defstruct :type** 385
:named-list option for **defstruct :type** 385
:tree option for **defstruct :type** 385, 386
:cons option for **defstruct-define-type** 410
:copier option for **defstruct-define-type** 410, 412
:defstruct option for **defstruct-define-type** 410, 412
:keywords option for **defstruct-define-type** 410, 412
:named option for **defstruct-define-type** 410, 411
:overhead option for **defstruct-define-type** 410, 411
:predicate option for **defstruct-define-type** 410, 412
:ref option for **defstruct-define-type** 410
:displaced-conformally option for **make-array** 241
:displaced-index-offset option for **make-array** 240, 241
:displaced-to option for **make-array** 239, 241
:fill-pointer option for **make-array** 241
:initial-value option for **make-array** 241
:leader-length option for **make-array** 241
:leader-list option for **make-array** 241
:named-structure-symbol option for **make-array** 241
:colon-mode option for **make-package** 585
:import option for **make-package** 573
:import-from option for **make-package** 573
:relative-names option for **make-package** 582

:relative-names-for-me option for **make-package** 582
:shadowing-import option for **make-package** 573, 574
:default-value option for **make-plane** 261
:area option for **make-array** 241
:invisible Option for **make-package** 601
:shadow option for **make-package** 574
:use option for **make-package** 571, 602
:extension option for **make-plane** 261
:initial-dimensions option for **make-plane** 261
:initial-origins option for **make-plane** 261
:type option for **make-array** 241
:export option for **make-package** 573
:type option for **make-plane** 261
:area init option for **si:eq-hash-table** 71
:growth-factor init option for **si:eq-hash-table** 71
:rehash-before-cold init option for **si:eq-hash-table** 71
:size init option for **si:eq-hash-table** 71
:rehash-threshold init option for **si:equal-hash-table** 72
:property-list init option for **si:property-list-mixin** 474
&optional keyword for **defmacro** 373
&optional Lambda-list Keyword 309
&optional keyword 151
:optional keyword 161
Optional parameters 154
Clm: **defstruct** Options 642
defflavor Options 441
Init options 425
Messages and Init Options 529
Options to **defstruct** 385
Options to **defstruct-define-type** 410
Inclusive **or** 179
Logical **or** function 179
:or method combination type 455
:or method type 455, 458
or special form 179
Row-major order 235
SCL and column-major order for arrays 641
:ordered-instance-variables Option for **defflavor** 441, 447
Column-major ordering 248
Ordering Flavors, Methods, and Wrappers 467
Miscellaneous Other Clauses 218
Clm: Other Environment Inquiries 645
Other Kinds of Functions 305
otherwise symbol 178
Elements out-of-bounds 541
Instance variable out-of-bounds 541
Out-of-bounds subscripts 541
Multiple and Out-of-order Evaluation 363
Displaying characters on output devices 268
Clm: Formatted Output to Character Streams 644
:outside-accessible-instance-variables Option for **defflavor** 441, 447
Exponent overflow 94
Exponent overflow error 535
Stack overflow error 537
:overhead option for **defstruct-define-type** 410, 411
Arrays Overlaid with Lists 250

:override method type 455, 457
Overstrike character 32
Overview and Definitions of Signalling and Handling 479

P

Change current package 597
chaos package 615
cl package 615, 633
Common Lisp Compatibility Package 615
common-lisp-global package 615
common-lisp-system package 633
common-lisp-user package 633
common-lisp package 633
compiler package 615
Condition system package 615
Current package 597
dbg package 615
debugger package 615
Defining a Package 598
Editor package 615
file-system package 615
fonts package 615
format package 615
fs package 615
global package 615
Home package 606
keyword package 615
Lisp language package 615
lmfs package 615
net package 615
neti package 615
network package 615
network-internals package 615
Pathnames package 615
Remove package 603
Removing symbol from package 606
scl package 633
SCL **gprint** package 633
SCL **language-tools** package 633
SCL **zl** package 633
SCL **zl-user** package 633
si package 615
symbolics-common-lisp package 633
sys package 615
system package 571, 615
system-internals package 615
The Current Package 597
The **global** Package 571
tv package 615
user package 579, 615
Window system package 615
Zmail package 615
zwei package 615
Package cell of a symbol 3
The Package Cell of a Symbol 566
Package Functions, Special Forms, and

P

P

Variables 597
:package message to **sys:external-symbol-not-found** 619
:package message to **sys:package-locked** 543, 619
Package name conflicts 581
Package Name-conflict Errors 593
Checking for Package Name-conflict Errors 593
Introduction to Package Name-conflict Errors 593
Resolving Package Name-conflict Errors 595
Introduction to Package Names 581
Introduction to Qualified Package Names 584
Multilevel Qualified Package Names 586
Qualified Package Names 584
Relative Package Names 582, 610
Shadowing package names 582
Qualified Package Names as Interfaces 585
Functions That Find the Home Package of a Symbol 607
Home Package of a Symbol 573
#: package qualifier 584
: package qualifier 585, 601, 603
:: package qualifier 601, 603
Compatibility with the Pre-release 5.0 Package System 625
How the Package System Allows Symbol Sharing 569
Package "Commands" 612
package-cell-location function 607
package-external-symbols function 611
:name message to **sys:** **package-not-found** 542, 619
:relative-to message to **sys:** **package-not-found** 542, 619
sys: **package-not-found** flavor 542, 619
package-shadowing-symbols function 612
package-use-list function 610
package-used-by-list function 610, 625
Package attribute 579
Package commands 612
package-declare special form 625
Interning Errors Based on **sys:** **package-error** 542
sys: **package-error** flavor 542, 619
Package Inheritance 571, 599, 602
Package Iteration 608
:package message to **sys:** **package-locked** 543, 619
:symbol message to **sys:** **package-locked** 543, 619
sys: **package-locked** flavor 543, 619
SCL case checking in package-name lookup 640
Package Names 581
Package-related Conditions 619
Packages 557
Adding new symbols to locked packages 626
Consistency Rules for Packages 591
Example of the Need for Packages 560
Examples of Symbol Sharing Among Packages 589
Invisible packages 581, 582
Mapping Between Names and Packages 607
SCL Packages 633
Sharing of Symbols Among Packages 569
Specifying Internal and External Symbols in Packages 585
System Packages 615
The Need for Packages 559
External-only Packages and Locking 626

Specifying Packages In Programs 579
Package system 566
package variable 579, 597
Page character 32
pairlis function 66
Specifying a Keyword Parameter's Symbol In Lambda Lists 156
Parameters 151
Actual parameters 151
Default forms of lambda-list parameters 151
Formal parameters 151
Keyword parameters 151
Optional parameters 154
Positional parameters 154
Required parameters 154
Rest parameters 154
SCL and **&rest** parameters for lambda-expressions 638
Binding Parameters to Arguments 153
SCL and **:junk-allowed** keyword for **parse-namestring** 644
fs: **parse-pathname-error** flavor 553
sys: **parse-error** flavor 543
parse-ferror flavor 543
parse-ferror function 504
SCL and **parse-namestring** function 644
parse-number function 281
SCL and **parse-pathname** function 644
Clm: Parsing of Numbers and Symbols 643
Message passing 421, 423
Message Passing In the Flavor System 423
Functions for Passing Messages 430
Clm: Rules Governing the Passing of Multiple Values 639
Passing-back of Multiple Values 169
:pass-on method combination type 455, 456
Drawing past edge of screen error 537
Read past the end-of-file error 536
array-element loop iteration path 225
array-elements loop iteration path 225, 257
hash-elements loop iteration path 224
heap-elements loop iteration path 224
interned-symbols loop iteration path 225
local-interned-symbols loop iteration path 225
The **interned-symbols** Path 225
An Example Path Definition 229
SCL and pathname components 644
Clm: Pathname Functions 644
Invalid pathname syntax errors 550
fs: **pathname-error** flavor 553
Pathname Errors 553
Pathnames 222
Clm: Pathnames 644
Printing pathnames 643
SCL pathnames and **:unspecific** 644
Pathnames package 615
Defining Iteration Paths 227
Iteration Paths 222
Predefined Iteration Paths 225
sys: **pdl-overflow** flavor 537
Operations the User Can Perform on Functions 302
Array Registers and Performance 255

Performing arithmetic operations on characters in SCL 265
phase function 107
pkg-add-relative-name function 582, 610
pkg-contained-in function 625
pkg-create-package function 626
pkg-debug-copy function 625
pkg-delete-relative-name function 582, 610
pkg-find-package function 607
pkg-global-package variable 617
pkg-keyword-package variable 617
pkg-refname-alist function 625
pkg-super-package function 625
pkg-system-package variable 617
pkg-bind macro 598
pkg-goto function 597
pkg-kill function 603
pkg-load function 625
pkg-name function 607
plane-aref function 261
plane-aset function 262
plane-default function 261
plane-extension function 261
plane-origin function 261
plane-ref function 261
Planes 260
plane-store function 262
Plist 67
plist function 565
Pluralizing words 280
plus function 100
Plus-Minus (+) character 32
plusp function 97
Trailing decimal point 14
Pointer 83
Fill pointer 238, 247
Frame pointer 494
Invisible pointer 56
Locative pointer 147
Conses represented as pointers 56
Restart points 479
pop macro 150
Extract position field of a byte specifier 116
Positional parameters 154
Testing for positive number 97
Post-step-endtest 221
SCL and potential numbers 644
Ppss 115
Pre-step-endtest 221
Predefined Iteration Paths 225
Predicate 7
:predicate option for **defstruct** 385, 392
:predicate option for **defstruct-define-type** 410, 412
Predicates 7
Character Predicates 270
Clm: Specific Data Type Predicates 638
Numeric Predicates 97
SCL sequence and list functions and two-argument predicates 641

Clm: Predicates on Characters 640
:prefix-intern-function Option for **defpackage** and **make-package** 601, 603
:prefix-name Option for **defpackage** and **make-package** 599, 601
Compatibility with the Pre-release 5.0 Package System 625
Previous frame 494
Primary methods 431
Hash Primitive 75
Primitive for Producing Multiple Values 167
princ function 585
prinlength variable 18
prinlevel variable 17
Clm: What the Print Function Produces 644
Print name of a symbol 3
The Print Name of a Symbol 565
:print option for **defstruct** 385, 391
dbg: **print-frame-locals** function 499
dbg: **print-function-and-args** function 499
sys: **print-not-readable** flavor 543
si: ***print-object-error-message*** variable 543
:print-self keyword for **named-structure-invoke** 403, 404
Maximum number of list elements to be printed 18
SCL and printed character names 644
Controlling the Printed Representation of an Object 19
Printed Representation of Arrays That Are Named Structures 16
Printed Representation of Arrays That Are Not Named Structures 17
Printed Representation of Common Lisp Character Objects 16
Printed Representation of Complex Numbers 15
Printed Representation of Conses 17
Printed Representation of Floating-point Numbers 14
Printed representation of functions 297
Printed Representation of Instances 16
Printed Representation of Integers 14
Printed Representation of Miscellaneous Data Types 18
Printed Representation of Numbers 90
Printed Representation of Ratios 14
Printed Representation of Strings 16
Printed Representation of Symbols 15
Errors Involving Lisp Printed Representations 543
Printed representation 3, 13, 559, 585
Printer 13, 585
Errors inside Lisp printer 543
What the Printer Produces 14
How the Printer Works 13
Effects of Slashification on Printing 13
Depth of recursion of printing lists 16
Printing nested lists 17
SCL and printing of structures 642
sys: **printing-random-object** macro 18
Printing pathnames 643
Print-Print consistency 591
si: **print-readably** variable 18

Print-Read consistency 591
:print-self message 16
:print-self message 453
Private symbol 571
Connection Problems 555
Local Network Problems 554
Login Problems 547
Network connection problems 555
Remote Network Problems 554
:proceed Can Return **nil** 519
:create-package proceed type 542, 619
:export proceed type 619
:new-name proceed type 542, 619
:no-action proceed type 523, 542, 543, 619
:shadow proceed type 619
:shadowing-import proceed type 619
:share proceed type 619
:skip proceed type 619
:unintern proceed type 619
:choose proceed type 619
Proceed Type Messages 519
Proceed Type Names 520
:proceed-type-p method of **condition** 530
:proceed-type-p message 519
dbg: ***proceed-type-special-keys*** variable 528
Proceedable condition functions 521
Proceedable conditions 517
Signalling proceedable conditions 521
dbg: **proceedable-ferror** flavor 532
Proceeding 517
Protocol for Proceeding 517
Reference Material: Proceeding 521
Proceeding with **condition-bind** Handlers 520
:proceed message 520
:proceed method 517, 520
Proceed type 517
:proceed-types init option 519
:proceed-types init option for **condition** 530
:proceed-types method of **condition** 530
:proceed-types message 519
Processes and SCL 635
Text processing 277
SCL and **proclaim** function 640
Clm: What the Print Function Produces 644
What the Printer Produces 14
Primitive for Producing Multiple Values 167
prog special form 167, 194, 359
prog* special form 196
SCL and **prog1** macro 639
prog1 special form 165
SCL and **prog2** macro 639
prog2 special form 165
prog-Context Conflicts 359
:progn method combination type 455
progn special form 164
Program control 175
Goto-less programming 194
Modularity and Object-oriented Programming 418

Object-oriented programming 417
MDL programming environment 205
Multipackage Programs 621
Specifying Packages in Programs 579
How Programs Examine Functions 322
How Programs Manipulate Definitions 316
How Applications Programs Treat Conditions 481
prog tags 189
SCL and **progv** macro 639
progv special form 131
progw special form 132
Loop prologue 205, 212
Prompt string 517, 528
si:defstruct-description property 407
File property errors 551
:property function spec type 297
Cons as property list 67
Disembodied property list 67
Memory cell as property list 67
Symbol associated with property list 67
Property list indicators 67
Property list keywords 67
Property List Messages 473
Property list of a symbol 3
The Property List of a Symbol 564
Property list values 67
Storing functions on property lists 297
Maclisp property names 564
Maclisp system property names 564
:property option for **defstruct** 385, 391
property-cell-location function 565
:property-list init option for **si:property-list-mixin** 474
:property-list init option for **si:** **property-list-mixin** 474
si: **property-list-mixin** flavor 473
Property list 67
:property-list message 474
Property Lists 67
Unwind protection 197
Protection-violation errors 549
Protocol for Proceeding 517
psetq special form 128
Pseudo-steps 221
Purpose of functions 339
SCL and **push** function 641
push-in-area macro 150
push macro 150
SCL and **pushnew** function 641
:push-property message 473
puthash-equal function 74
puthash function 74
:put-hash message 72
putprop function 68
:putprop message 473

Q

Qualified Names of Symbols 585
Qualified Package Names 584
Introduction to Qualified Package Names 584
Multilevel Qualified Package Names 586
Qualified Package Names as Interfaces 585
Qualified names 589
Forms of qualified names 585
#: package qualifier 584
: package qualifier 585, 601, 603
:: package qualifier 601, 603
Single quote (') 161
Quote (') macro character 26
&quote Lambda-list Keyword 310
quote special form 161
Quote character 32
quotient function 102
SCL and vertical bars for quoting 643

R

Radians in trigonometric functions 106
Integer radix 14, 20
Seed for random number generator 118
:random returned by **typep** 9
si: **random-create-array** function 119
Random-array 118
random function 118
si: **random-initialize** function 119
Random Numbers 118
rass function 66
rassoc function 66
rassq function 65
Read rational number in binary 29
:rational returned by **typep** 9
rational function 108
Rational Numbers 93
rationalp function 8
Ratios 89, 93
How the Reader Recognizes Ratios 22
Printed Representation of Ratios 14
Clm: What the Read Function Accepts 643
Syntax errors in read functions 543
Read past the end-of-file error 536
Read rational number in binary 29
cl: ***read-default-float-format*** variable 23
SCL and **read-delimited-list** function 644
sys: **read-end-of-file** flavor 544
si: ***read-extended-ibase-signed-number*** variable 21
SCL and ***read-extended-ibase-signed-numbers*** variable 643
si: ***read-extended-ibase-unsigned-number*** variable 21
SCL and ***read-extended-ibase-unsigned-numbers*** variable 643
SCL and **read-from-string** function 643
sys: **read-list-end-of-file** flavor 544

si: ***read-multi-dot-tokens-as-symbols**** variable 26
sys: **read-premature-end-of-symbol** flavor 544
sys: **read-string-end-of-file** flavor 544
SCL and ***read-base**** variable 643
SCL and **read-char** function 643
Reader 13
#- Reader Macro 30
#: Reader Macro 29
#◊ Reader Macro 30
#' Reader Macro 28
#+ Reader Macro 29
#, Reader Macro 28
#. Reader Macro 29
#(Reader Macro 30
#b Reader Macro 29
#m Reader Macro 29
#n Reader Macro 29
#o Reader Macro 29
#P Reader Macro 643
#q Reader Macro 29
#r Reader Macro 29
#x Reader Macro 29
or **#/** Reader Macro 27
#^ Reader Macro 28
#| Reader Macro 30
SCL and **#-** Reader Macro 644, 645
SCL and **#+** Reader Macro 644, 645
Reader macro for infix expressions 30
reader macros 27
Sharp-sign Reader Macros 27
What the Reader Recognizes 20
How the Reader Recognizes Complex Numbers 24
How the Reader Recognizes Conses 25
How the Reader Recognizes Floating-point Numbers 22
How the Reader Recognizes Integers 20
How the Reader Recognizes Macro Characters 26
How the Reader Recognizes Ratios 22
How the Reader Recognizes Strings 25
How the Reader Recognizes Symbols 24
How the Reader Works 19
sys: **read-error** flavor 544
read function 13, 56
Reading Integers in Bases Greater Than 10 21
Reading octal character codes 25
Read-only error 537
Read-Read consistency 591
si: **read-recursive** function 27
SCL and ***read-suppress**** variable 643
Readtable 33
Clm: the Readtable 644
Common Lisp readtable 27
Manipulating the readtable 33
The Readtable 33
Readtable Functions for Maclisp Compatibility 36
Readtables 262
Functions That Create New Readtables 33
readtable variable 33
Read-time conditionalization facility 29

realpart function 108
***rearray** Maclisp function 262
Special Forms for Receiving Multiple Values 167
Data Types Recognized by **loop** 219
What the Reader Recognizes 20
How the Reader Recognizes Complex Numbers 24
How the Reader Recognizes Conses 25
How the Reader Recognizes Floating-point Numbers 22
How the Reader Recognizes Integers 20
How the Reader Recognizes Macro Characters 26
How the Reader Recognizes Ratios 22
How the Reader Recognizes Strings 25
How the Reader Recognizes Symbols 24
recompile-flavor function 438
Note on **record-source-file-name** 318
record-source-file-name function 317, 319
Structured records 59
Recursion 175
Condition Bind Recursion 502
Depth of recursion of printing lists 16
Redefining functions 318
Illegal redefinition error 538
sys: **redefinition** flavor 538
:ref option for **defstruct-define-type** 410
Captured free reference 126
Clm: Reference 638
Condition Flavors Reference 529
Free reference 126
Reference Material: Default Handlers and Complex Modularity 510
Reference Material: Establishing Handlers 488
Reference Material: Proceeding 521
Reference Material: Restart Handlers 514
Reference Material: Signalling Conditions 503
Reference Material: Application: Handlers Examining the Stack 495
sys: **region-table-overflow** flavor 538
Array Register Restrictions 258
Array Registers 255
Hints for Using Array Registers 257
Array Registers and Performance 255
Rehash 75
:rehash-before-cold init option for **si:eq-hash-table** 71
SCL and **:rehash-size** keyword to **make-hash-table** function 641
:rehash-threshold init option for **si:equal-hash-table** 72
SCL and **:rehash-threshold** keyword to **make-hash-table** function 641
Interpackage Relations 610
Relationship between methods and messages 421
Relative Package Names 582, 610
:relative-names Option for **defpackage** and **make-package** 600, 602
:relative-names option for **make-package** 582
:relative-names-for-me option for **defpackage** 582
:relative-names-for-me Option for **defpackage** and

make-package 600, 602
:relative-names-for-me option for **make-package** 582
:relative-to message to **sys:package-not-found** 542, 619
:relative-names option for **defpackage** 582
rem-if-not function 63
remainder function 103
rem function 62
remhash-equal function 74
remhash function 74
:rem-hash message 73
rem-if function 64
remob function 573, 593, 606
Remote Network Problems 554
sys: **remote-network-error** flavor 554
:remove method of **si:heap** 79
remove function 62
Remove package 603
Removing a **defun-method** 437
Removing elements from list 150
Removing symbol from package 606
Removing flavor 437
Removing methods 437
remprop function 69
:remprop message 473
remq function 62
File rename errors 551
fs: **rename-across-directories** flavor 551
fs: **rename-across-hosts** flavor 551
fs: **rename-to-existing-file** flavor 551
si: **rename-within-new-definition-maybe** function 329
fs: **rename-failure** 551
fs: **rename-failure** flavor 551
Rename-within 329
Rename-within Encapsulations 329
Function renaming 329
repeat loop keyword 207
SCL and **:replace** keyword for **delete-duplicates** function 640
:report method of **condition** 530
:report message 529
:report method 485
Debugger Bug Reports 524
:report-string method of **condition** 531
D exponential representation 94
E exponential representation 94
IEEE Floating-point Representation 94
Printed representation 3, 13, 559, 585
E exponential representation 14
Controlling the Printed Representation of an Object 19
Printed Representation of Arrays That Are Named Structures 16
Printed Representation of Arrays That Are Not Named Structures 17
Printed Representation of Common Lisp Character Objects 16
Printed Representation of Complex Numbers 15
Printed Representation of Conses 17

Printed Representation of Floating-point Numbers 14
Printed representation of functions 297
Printed Representation of Instances 16
Printed Representation of Integers 14
String representation of integers 277
Printed Representation of Miscellaneous Data Types 18
Printed Representation of Numbers 90
Printed Representation of Ratios 14
Printed Representation of Strings 16
Printed Representation of Symbols 15
Array Representation Tools 237
Errors Involving Lisp Printed Representations 543
Conses represented as pointers 56
Request Failures Based on **fs:file-request-failure** 546
SCL and **require** function 640
:required-init-keywords Option for **defflavor** 441, 442
:required-instance-variables Option for **defflavor** 441, 442
:required-flavors Option for **defflavor** 441, 443
:required-methods Option for **defflavor** 441, 442
Required parameters 154
Resolving Package Name-conflict Errors 595
Safety of **&rest** Arguments 157
&rest keyword for **defmacro** 373
&rest Lambda-list Keyword 309
SCL and **&rest** parameters for lambda-expressions 638
rest1 function 47
rest2 function 47
rest3 function 47
rest4 function 47
Restart handler functions 514
Reference Material: Restart Handlers 514
Invoking Restart Handlers Manually 516
Restart handlers 501, 513
Restart points 479
&rest keyword 151
Rest parameters 154
SCL and **rplaca** restriction 638
SCL and **rplacd** restriction 638
SCL arrays and circular-structure labelling restriction 641
Restriction Due to Scope 503
Array Register Restrictions 258
Using the RESUME key with floating-point conditions 535
Resume character 32
RESUME key 526
Return from **typeof** function 642
SCL and **return** function 639
return loop keyword 216, 218
Functions that return multiple values 167
:proceed Can Return **nil** 519
return special form 167, 169, 185, 189
return-array function 250
Return character 32
:array returned by **typep** 9
:bignum returned by **typep** 9
:closure returned by **typep** 9

:compiled-function returned by **typep** 9
:complex returned by **typep** 9
:double-float returned by **typep** 9
:fixnum returned by **typep** 9
:list returned by **typep** 9
:locative returned by **typep** 9
:random returned by **typep** 9
:rational returned by **typep** 9
:select-method returned by **typep** 9
:single-float returned by **typep** 9
:stack-group returned by **typep** 9
:string returned by **typep** 9
:symbol returned by **typep** 9
SCL and returned values from **cond** macro 639
SCL and **return-from** function 639
return-from special form 167, 184, 218
Returning array elements 244
return-list function 185
reverse function 51
Roman-I character 32
Roman-II character 32
Roman-III character 32
Roman-IV character 32
Square root 106
Square root of a negative number error 536
Rotate bits 115
rot function 115
round function 111
Rounding 93
Row-major order 235
SCL and **rplaca** restriction 638
rplaca function 54, 56, 83, 236
SCL and **rplacd** restriction 638
rplacd function 54, 56, 83, 236
Rubout character 32
Search rule for invoking handlers 501, 509
Coercion rules 90
Coercion Rules for Numbers 90
Consistency Rules for Packages 591
Clm: Rules Governing the Passing of Multiple Values 639

S

S

S

Safety of **&rest** Arguments 157
Copying From and to the Same Array 253
samepnamep function 293, 565
sassoc function 66
sassq function 66
Flavors and SCL 635
Performing arithmetic operations on characters in SCL 265
Processes and SCL 635
Self-evaluating atoms and SCL 637
Using SCL 633
Zetalisp and SCL 631
SCL and * variable 642
SCL and **#-** Reader Macro 644, 645
SCL and **#+** Reader Macro 644, 645
SCL and **#S** macro form 642

SCL and ***default-pathname-defaults*** variable 644
SCL and ***features*** variable 645
SCL and ***read-extended-ibase-signed-numbers*** variable 643
SCL and *read-extended-ibase-unsigned-numbers* variable 643
SCL and ***read-base*** variable 643
SCL and ***read-suppress*** variable 643
SCL and adjustable arrays 641
SCL and **array-row-major-index** function 642
SCL and column-major order for arrays 641
SCL and Common Lisp Differences 637
SCL and Common Lisp Files 635
SCL and **compile-file** function 645
SCL and *constituent* character attribute 643
SCL and **declaration** declaration 640
SCL and **declare** special form 640
SCL and **decode-universal-time** function 645
SCL and **defsetf** function 639
SCL and **defstruct** macro 642
SCL and **defstruct** slot initializations 642
SCL and **describe** function 645
SCL and **directory** function 645
SCL and **documentation** as **setf** argument 639
SCL and **documentation** function 645
SCL and **dribble** function 645
SCL and **equal-typep** function 638
SCL and equivalent macro definitions for special forms 638
SCL and file lookup errors 645
SCL and **format** function directives 644
SCL and **ftype** declaration 640
SCL and **function** special form 638
SCL and **functionp** function 638
SCL and **get-macro-character** function 644
SCL and **go** special form 639
SCL and **ieee-floating-point** feature name 645
SCL and *illegal* character attributes 643
SCL and **inline** declaration 640
SCL and **:junk-allowed** keyword for **parse-namestring** 644
SCL and keywords in the ***features*** list 644
SCL and **make-concatenated-stream** function 643
SCL and **make-echo-stream** function 643
SCL and **mismatch** function 641
SCL and multidimensional arrays 641
SCL and **:named** option for **defstruct** 642
SCL and non-null hosts 644
SCL and *non-terminating macro* character attribute 643
SCL and **notinline** declaration 640
SCL and Optimization 635
SCL and **optimize** declaration 640
SCL and **parse-namestring** function 644
SCL and **parse-pathname** function 644
SCL and pathname components 644
SCL and potential numbers 644
SCL and printed character names 644

SCL and printing of structures 642
SCL and **proclaim** function 640
SCL and **prog1** macro 639
SCL and **prog2** macro 639
SCL and **progv** macro 639
SCL and **push** function 641
SCL and **pushnew** function 641
SCL and **read-delimited-list** function 644
SCL and **read-from-string** function 643
SCL and **read-char** function 643
SCL and **:rehash-size** keyword to **make-hash-table** function 641
SCL and **:rehash-threshold** keyword to make-hash-table function 641
SCL and **:replace** keyword for **delete-duplicates** function 640
SCL and **require** function 640
SCL and **&rest** parameters for lambda-expressions 638
SCL and **return** function 639
SCL and returned values from **cond** macro 639
SCL and **return-from** function 639
SCL and **rplaca** restriction 638
SCL and **rplacd** restriction 638
SCL and **set-macro-character** function 644
SCL and **set-syntax-from-char** function 643
SCL and slashification 644
SCL and Standard Character Syntax Types Table 643
SCL and Standard Constituent Character Attributes Table 643
SCL and **standard-char-p** function 640
SCL and Strings 634
SCL and **substitute** function 640
SCL and **substitute-if-not** function 640
SCL and **substitute-if** function 640
SCL and subtype variations 637
SCL and Symbolics Common Lisp Extensions 635
SCL and **:test** keyword to **make-hash-table** function 641
SCL and **the** special form 640
SCL and type common 637
SCL and **:type** option for **defstruct** 642
SCL and type string 637
SCL and type string-char 637
SCL and *value-type* argument for **the** special form 640
SCL and vertical bars for quoting 643
SCL and **zl:si:*flag-wrong-type-strings*** 637, 642
SCL and | 643
SCL and ~E directive 644
SCL and ~G directive 644
SCL and ~T directive 644
SCL arrays and circular-structure labelling restriction 641
SCL case checking in package-name lookup 640
Zetalisp and SCL character incompatibilities 265
SCL **coerce** function 637

SCL **gprint** package 633
SCL I/O operations and streams 634
SCL **language-tools** package 633
SCL **loop** macro 635
SCL pathnames and **:unspecific** 644
SCL sequence and list functions and two-argument predicates 641
Zetalisp and SCL string incompatibilities 265
SCL **zl** package 633
SCL **zl-user** package 633
SCL Documentation 636
scl package 633
SCL Packages 633
Dynamic scope 126
Lexical scope 126
Restriction Due to Scope 503
Lexically scoped compiler 137
Lexically scoped interpreter 137
defmacro and lexical scoping 138
Lexical Scoping 137, 142, 144
macro and lexical scoping 138
Macro-expander functions and lexical scoping 138
Drawing past edge of screen error 537
Search rule for invoking handlers 501, 509
Table searches 59
String Searching 286
String Searching Affected by Case, Style, and Bits 286
Handler-list searching functions 510
String Searching Ignoring Case, Style, and Bits 287
Clm: Searching Sequences for Items 641
second function 46
Seed for random number generator 118
select special form 180
Selective evaluation in macro definitions 345
Select-method 305
:select-method returned by **typep** 9
Select-method functions 305
selector special form 181
selectq special form 179
selectq-every special form 182
Send message to self 159
Self-evaluating atoms and SCL 637
self variable 437
Semicolon (;) macro character 26
Send message to self 159
send-if-handles function 431
:send-if-handles message 454
send function 160, 430
Sending a message 421
Sending messages 159
Case sensitivity of interning 604
SCL sequence and list functions and two-argument predicates 641
Sequence iteration 225
Clm: Modifying Sequences 640
Clm: Searching Sequences for Items 641
Clm: Sequences, Lists 641
Clm: Simple Sequencing 639

Special Forms for Sequencing 164
Set 59
Expanded character set 277
Creating a Set of Condition Flavors 486
set-char-bit function 269
set-character-translation function 34
set-in-closure function 335
set-in-instance function 439
SCL and **set-macro-character** function 644
:set-proceed-types method of **condition** 530
:set-property-list message 474
set-syntax-#-macro-char function 27, 36
set-syntax-from-char function 34
SCL and **set-syntax-from-char** function 643
set-syntax-from-description function 34
set-syntax-macro-char function 26, 35
setarg function 166
SCL and **documentation** as **setf** argument 639
setf special form 147
setf macro 147
set function 561
set-globally function 561
setplist function 565
setq special form 125, 128
Character sets 267
Support for Nonstandard Character Sets 275
Character Sets and Character Styles 267
:macro argument to **setsyntax** 36
nil argument to **setsyntax** 36
:single argument to **setsyntax** 36
:splicing argument to **setsyntax** 36
setsyntax-sharp-macro function 36
setsyntax function 36
:settable-instance-variables Option for **defflavor** 441
Setting variables 125, 126
Special Forms for Setting Variables 128
seventh function 47
:shadow option for **defpackage** 574
:shadow Option for **defpackage** and **make-package** 599, 602
:shadow option for **make-package** 574
:shadow proceed type 619
Functions That Import, Export, and Shadow Symbols 611
shadow function 574, 593, 612
Shadowing 571
Shadowing package names 582
:shadowing-import Option for **defpackage** and **make-package** 600, 602
:shadowing-import option for **make-package** 573, 574
:shadowing-import option for **defpackage** 573, 574
:shadowing-import proceed type 619
shadowing-import function 573, 574, 593, 611
Shadowing-import operation 573
Shadowing Symbols 574, 593, 611
:share proceed type 619
How the Package System Allows Symbol Sharing 569

Examples of Symbol
Sharing Among Packages 589
Sharing of Symbols Among Packages 569
Sharp-sign (#) macro character 26
Sharp-sign Reader Macros 27
Drawing on unprepared
sheet error 537
Shift bits 114
cl:
short-float format 23
si:*flavor-compile-trace* variable 440
si:*flavor-compilations* variable 440
si:*macroexpand-hook* variable 375
si:*print-object-error-message* variable 543
si:*read-extended-ibase-signed-number*
variable 21
si:*read-extended-ibase-unsigned-number*
variable 21
si:*read-multi-dot-tokens-as-symbols* variable 26
SCL and **zl:**
si:*flag-wrong-type-strings* 637, 642
:area init option for
si:eq-hash-table 71
:growth-factor init option for
si:eq-hash-table 71
:rehash-before-cold init option for
si:eq-hash-table 71
:size init option for
si:eq-hash-table 71
:rehash-threshold init option for
si:equal-hash-table 72
:clear method of
si:heap 78
:delete-by-item method of
si:heap 78
:delete-by-key method of
si:heap 78
:describe method of
si:heap 78
:empty-p method of
si:heap 78
:find-by-item method of
si:heap 78
:find-by-key method of
si:heap 78
:insert method of
si:heap 79
:remove method of
si:heap 79
:top method of
si:heap 79
:property-list init option for
si:property-list-mixin 474
si:alphabetic syntax description 34
si:break syntax description 34
si:circlecross syntax description 34
Side Effects 212
si:define-simple-method-combination macro 459
si:defstruct-description property 407
si:digested-lambda functions 297, 300, 304, 305
si:doublequote syntax description 34
si:encapsulated-definition debugging info alist
element 325
si:encapsulated-function variable 325
si:encapsulate macro 326
si:encapsulation-standard-order variable 327
si:eq-hash-table flavor 71
si:equal-hash-table flavor 72
si:equal-hash function 75
si:flavor-allowed-init-keywords function 439
si:flavor-default-init-get function 440
si:flavor-default-init-putprop function 440
si:flavor-default-init-remprop function 440
si:function-spec-get function 322
si:function-spec-putprop function 322
Testing for
sign of a number 97
signal-proceed-case special form 517, 521
signal function 482, 503, 517

Signallers 520
Signalling 482
Signalling a condition 479
Overview and Definitions of Signalling and Handling 479
Introduction to Signalling and Handling Conditions 479
Signalling proceedable conditions 521
Signalling Simple Conditions 502
Signalling Conditions 501
Reference Material: Signalling Conditions 503
Signalling Errors 502
Signalling functions 482, 503
Signalling Mechanism 501, 509
Significant bits 115
signp special form 97
signum function 106
si:initial-readtable variable 33
si:loop-named-variable function 229
si:loop-use-system-destructuring? variable 221
si:loop-tassoc function 228
si:loop-tequal function 228
si:loop-tmember function 228
si:macro syntax description 34
Signalling Simple Conditions 502
Simple Function Definitions 300
Examples of Simple Lambda Lists 154
Clm: Simple Sequencing 639
Simple Use of Flavors 425
Simple conditions 482
Simultaneous linear equations 258
sind function 106
Sine 106
sin function 106
:single argument to **setsyntax** 36
Single quote (') 161
si: **single** syntax description 34
sys: **single-float-p** function 8
Single-precision floating-point numbers 89
Single-character symbol 34
Single-float 22
:single-float returned by **typep** 9
cl: **single-float** format 23
Single-floats 4, 89
Singular matrix operation error 535
math: **singular-matrix** flavor 535
sinh function 107
si package 615
si:print-readably variable 18
si:property-list-mixin flavor 473
si:random-create-array function 119
si:random-initialize function 119
si:read-recursive function 27
si:rename-within-new-definition-maybe function 329
si:single syntax description 34
si:slash syntax description 34
si:unencapsulate-function-spec function 328
si:vanilla-flavor flavor 453
si:verticalbar syntax description 34

si:whitespace syntax description 34
sixth function 47
Extract size field of a byte specifier 116
:size init option for **si:eq-hash-table** 71
Changing the Size of an Array 249
Bit size of array elements 236
:size Option for **defpackage** and **make-package** 600, 602
:size-macro option for **defstruct** 385, 390
:size message 73
:size-symbol option for **defstruct** 385, 390
:skip proceed type 619
si: **slash** syntax description 34
SCL and slashification 644
Effects of Slashification on Printing 13
Slashifying 529
Slot 383
SCL and **defstruct** slot initializations 642
Altering slot values of structures 395
Slot description 407
math: **solve** function 259
Interaction of Some Common Special Forms with Multiple Values 169
Message to an object of some flavor 461
Some Functions and Special Forms 159
some function 64
sort-grouped-array-group-key function 81
sort-grouped-array function 81
sortcar function 81
sort function 56, 79
Sorting Arrays and Lists 79
Sorting compact lists 79
Sorting arrays 79
Sorting lists 79
Space character 32
SP character 32
Basic definition of the function spec 297, 325
:handler function spec type 297
:internal function spec type 297
:location function spec type 297
:method function spec type 297
:property function spec type 297
Symbol function spec type 297
:within function spec type 297
Special Character Names 32
Debugger Special Commands 525
#' special form 162
advise special form 302
and special form 178
argument-typecase special form 506
block special form 183
caseq special form 182
***catch** special form 169, 200
catch special form 169, 197
catch-error-restart special form 501, 502, 513, 515
catch-error-restart-if special form 513, 515
check-arg-type special form 576
comment special form 164

compiler-let special form 129
cond special form 177
cond-every special form 178
condition-bind-default special form 489, 501, 509
condition-bind-default-if special form 489
condition-bind-if special form 489
condition-call-if special form 493
condition-case-if special form 491
condition-bind special form 488, 501, 517
condition-call special form 492, 501
condition-case special form 490, 501
declare special form 126, 311
def special form 307
defconst special form 126, 135
defconstant special form 135
deff special form 307
deffunction special form 356
define-symbol-macro special form 353
deflambda-macro-displace special form 356
deflambda-macro special form 356
defmacro special form 305
defmethod special form 297
defpackage special form 598, 599
defprop special form 69
defselect special form 297, 305, 308
defselect-method special form 437
defsubst special form 305, 351
defun special form 297, 300, 305
defun-method special form 436, 437
defvar special form 126, 134
defwhopper special form 463
desetq special form 133
destructuring-bind special form 132
dispatch special form 182
dlet special form 133
dlet* special form 133
do special form 167, 189, 359
do* special form 191
do-all-symbols special form 609
do-external-symbols special form 609
do-local-symbols special form 609
dolist special form 193
do*-named special form 192
do-named special form 192
do-symbols special form 609
dotimes special form 192
error-restart-loop special form 501, 513, 514
error-restart special form 501, 513, 514
flet special form 142
function special form 162
go special form 187, 189
if special form 177
ignore-errors special form 493
keyword-extract special form 193
labels special form 144
lambda special form 163
lambda-macro special form 355
let special form 128, 142

let* special form 129
let-globally-if special form 131
let-closed special form 305, 335
letf special form 130
letf* special form 130
let-globally special form 131
let-if special form 130
local-declare special form 126, 315
macro special form 305, 339, 355
macrolet special form 144
multiple-value-bind special form 168
multiple-value-call special form 168
multiple-value-list special form 168
multiple-value-prog1 special form 169
multiple-value special form 168
or special form 179
package-declare special form 625
prog special form 167, 194, 359
prog* special form 196
prog1 special form 165
prog2 special form 165
progn special form 164
progv special form 131
progw special form 132
psetq special form 128
quote special form 161
return special form 167, 169, 185, 189
return-from special form 167, 184, 218
SCL and **declare** special form 640
SCL and **function** special form 638
SCL and **go** special form 639
SCL and **the** special form 640
SCL and *value-type* argument for **the** special form 640
select special form 180
selector special form 181
selectq special form 179
selectq-every special form 182
setf special form 147
setq special form 125, 128
signal-proceed-case special form 517, 521
signp special form 97
tagbody special form 187
throw special form 198
trace special form 302
typecase special form 181, 576
undefun-method special form 437
unwind-protect special form 169, 198
variable-boundp special form 562
variable-location special form 563
variable-makunbound special form 562
with-input-from-string special form 290
with-output-to-string special form 291
without-floating-underflow-traps special form 94
Special Form for Declaring a Named Constant 135
Clm: Special Forms 638
Defining special forms 362
flet, labels, and **macrolet** Special Forms 142
Function-defining Special Forms 305

SCL and equivalent macro definitions for special forms 638
Some Functions and Special Forms 159
Special Forms for Binding Variables 128
Functions and Special Forms for Constant Values 161
Special Forms for Defining Special Variables 134
Special Forms for Receiving Multiple Values 167
Special Forms for Sequencing 164
Special Forms for Setting Variables 128
Interaction of Some Common Special Forms with Multiple Values 169
Package Functions, Special Forms, and Variables 597
Evaluation of special functions 303
&special Lambda-list Keyword 310
Binding local and special variables 126
dbg: ***special-command-special-keys*** variable 528
:special-instance-variables Option for **defflavor** 441, 448
Special characters 32
:special-command message 525
:special-commands method of **condition** 530
special declaration 312
Special forms 303
Special functions 303
Special Keys 526
Special variables 126, 331
Defining special variables 134
Special Forms for Defining Special Variables 134
:special-command-p method of **condition** 530
dbg: **special-commands-mixin** flavor 525
Clm: Specific Data Type Predicates 638
Create a byte specifier 116
Extract position field of a byte specifier 116
Extract size field of a byte specifier 116
Byte specifiers 115, 399
Clm: Declaration Specifiers 640
%%ch- byte specifiers and file characters 268
%%kbd- byte specifiers and keyboard characters 268
Specifying a Keyword Parameter's Symbol in Lambda Lists 156
Specifying Aux-variables in Lambda Lists 157
Specifying Default Forms in Lambda Lists 155
Specifying Internal and External Symbols in Packages 585
Specifying Packages in Programs 579
Defining function specs 297
Function Specs 297
:method function specs 316
:splicing argument to **setsyntax** 36
:spread keyword 161
sqrt function 106
Square root of a negative number error 536
Square root 106
stable-sortcar function 81
stable-sort function 81
Application: Handlers Examining the Stack 494
Reference Material: Application: Handlers Examining the Stack 495
Unwinding a stack 197
Stack group state error 537

Stack groups used as functions 305
Stack overflow error 537
Stack-allocated closures 139
Stack frame 494
:stack-group returned by **typep** 9
Stack-groups 303
SCL and Standard Character Syntax Types Table 643
SCL and Standard Constituent Character Attributes Table 643
Clm: Standard Dispatching Macro Character Syntax 644
SCL and **standard-char-p** function 640
Standard Conditions 531
Stack group state error 537
Status character 32
Stepping 221
Stepping variables 221
Steps 221
Stop-Output character 32
Storage allocation error 538
Storage of arrays 235
Storage allocation 250
store-array-leader function 245
Cdr storing functions 297
Flavor system storing functions 297
Storing functions on property lists 297
Storing into array elements 244
Stream 290, 291
Tokens in the input stream 19
sys: **stream-closed** flavor 536
Clm: Creating New Streams 643
Clm: Formatted Output to Character Streams 644
Clm: Input From Character Streams 643
SCL I/O operations and streams 634
Prompt string 517, 528
SCL and type string 637
string≠ function 282
string≤ function 282
string≥ function 283
Clm: String Characters 637
String Comparisons Affected by Case, Style, and Bits 282
String Comparisons Ignoring Case, Style, and Bits 283
Documentation string functions 302, 322
:string message to **sys:external-symbol-not-found** 619
Basic String Operations 278
String representation of integers 277
:string returned by **typep** 9
String Searching Affected by Case, Style, and Bits 286
String Searching Ignoring Case, Style, and Bits 287
string-capitalize-words function 286
string-exact-compare function 283
sys: **%string-exact-compare** function 283
string-left-trim function 280
Maclisp-compatible String-manipulation Functions 292
string-nconc-portion function 279
string-not-equal function 284

string-not-greaterp function 284
string-not-lessp function 284
string-reverse-search-char function 287
string-reverse-search-exact-char function 286
string-reverse-search-exact function 287
string-reverse-search-not-char function 288
string-reverse-search-not-exact-char function 287
string-reverse-search-not-set function 290
string-reverse-search-set function 289
string-reverse-search function 288
string-right-trim function 280
string-search-char function 287
%string-search-char function 289
%string-search-exact-char function 287
string-search-exact-char function 286
string-search-exact function 287
string-search-not-char function 287
string-search-not-exact-char function 286
string-search-not-set function 289
string-search-set function 289
string-to-ascii function 290
string< function 282
string= function 282
%string= function 283
string> function 282
string-append function 279
SCL and type string-char 637
string-compare function 285
sys: **%string-compare** function 285
String comparisons 11, 282
String concatenation 279
String Conversions 285
string-downcase function 285
%string-equal function 284
string-equal function 283
string-flipcase function 286
string function 278
string-greaterp function 284
String incompatibilities 634
Zetalisp and SCL string incompatibilities 265
string-length function 278
string-lessp function 284
string-nconc function 279
string-nreverse function 280
stringp function 8
string-pluralize function 280
string-reverse function 280
Strings 5, 236, 277
Arrays, Characters, and Strings 233
ASCII Strings 290
Character strings 277
Clm: Strings 637, 642
How the Reader Recognizes Strings 25
I/O to Strings 290
Printed Representation of Strings 16
SCL and Strings 634
Character strings as array elements 236
Clm: Translating Strings to Symbols 640

string-search function 288
String Searching 286
string-trim function 280
string-upcase function 285
Alteration of List Structure 54
Manipulating List Structure 41
Named structure array 238, 241
Named structure symbol 403
Structured records 59
Structure Macros 377
Introduction to Structure Macros 379
Structure names 383
Altering slot values of structures 395
Clm: Structures 642
Creating instances of structures 395
defstruct Internal Structures 407
Exit control structures 175
Functions That Operate on Named Structures 405
Handler Functions for Named Structures 403
Initialization in structures 395
Introduction to Named Structures 403
Named Structures 403
Nonlocal exit control structures 175
Printed Representation of Arrays That Are Named Structures 16
Printed Representation of Arrays That Are Not Named Structures 17
SCL and printing of structures 642
Character objects style field 265
Character Comparisons Affected by Case, Style, and Bits 270
Character Comparisons Ignoring Case, Style, and Bits 271
String Comparisons Affected by Case, Style, and Bits 282
String Comparisons Ignoring Case, Style, and Bits 283
String Searching Affected by Case, Style, and Bits 286
String Searching Ignoring Case, Style, and Bits 287
Character Sets and Character Styles 267
sub1 function 104
Subform 169
The Device-font and Subindex Derived Fields 268
sublis function 55
Subprimitives 494
Subprograms 621
Closed subroutine 351
Open subroutine 351
subrp function 8
sys: **subscript-out-of-bounds** flavor 541
Array subscripts 235, 248
Multidimensional array subscripts 245
Out-of-bounds subscripts 541
subset function 63
subset-not function 63
subst function 55
subst functions 351
Substitutable Functions 351
SCL and **substitute** function 640
SCL and **substitute-if-not** function 640
SCL and **substitute-if** function 640
Substitution 54
substring function 278

Subtraction 101, 104, 105
Subtraction of 32-bit numbers 119
SCL and subtype variations 637
sum loop keyword 212
summing loop keyword 212
SUPER key 526
Supplied-p variable 155
Support for Nonstandard Character Sets 275
Macros That Surround Code 362
swapf macro 149
swaphash-equal function 74
swaphash function 74
:swap-hash message 73
sxhash function 59, 76
Atomic symbol 3
Binding of a symbol 3
:case-documentation symbol 520
Functions That Find the Home Package of a Symbol 607
Home Package of a Symbol 573
Interned symbol 566
Lambda symbol 151
lambda-list-keywords symbol 151
Named structure symbol 403
nil symbol 561
otherwise symbol 178
Package cell of a symbol 3
Print name of a symbol 3
Private symbol 571
Property list of a symbol 3
Single-character symbol 34
t symbol 561
The Function Cell of a Symbol 563
The Package Cell of a Symbol 566
The Print Name of a Symbol 565
The Property List of a Symbol 564
The Value Cell of a Symbol 561
Unbound symbol 561
Uninterned symbol 357, 566
:which-operations symbol 520
Symbol associated with property list 67
The Symbol Data Type 3
Removing symbol from package 606
Symbol function spec type 297
Specifying a Keyword Parameter's Symbol in Lambda Lists 156
:symbol message to **sys:package-locked** 543, 619
Where Is Symbol (m-X) Zmacs command 571
Symbol name conflicts 574
:symbol returned by **typep** 9
How the Package System Allows Symbol Sharing 569
Examples of Symbol Sharing Among Packages 589
Symbol definition 3, 563
Symbolics Common Lisp 629
Introduction to Symbolics Common Lisp 631
SCL and Symbolics Common Lisp Extensions 635
symbolics-common-lisp package 633
Symbolics-Lisp Number Types 89
Symbol Macros 353
symbol-package function 573, 607

symbolp function 7
Symbols 3, 559, 561
Clm: Parsing of Numbers and Symbols 643
Clm: Translating Strings to Symbols 640
Creating Symbols 566
Dot (.) in symbols 357
Exporting symbols 573, 589, 593, 611, 613
External Symbols 570, 571, 573
Functions That Import, Export, and Shadow Symbols 611
Functions That Map Names to Symbols 605
Global symbols 571
How the Reader Recognizes Symbols 24
Importing symbols 573, 589, 593, 611
Importing and Exporting Symbols 573
Internal symbols 570
Keyword symbols 151
Mapping from names to symbols 559
Mapping Names to Symbols 604
Printed Representation of Symbols 15
Qualified Names of Symbols 585
Shadowing Symbols 574, 593, 611
Uninterned symbols 29, 573
Sharing of Symbols Among Packages 569
Specifying Internal and External Symbols In Packages 585
Adding new symbols to locked packages 626
Symbols used as functions 297
symeval-in-closure function 335
symeval-in-instance function 439
symeval function 3, 561
symeval-globally function 562
loop Synonyms 218
#" syntax 637, 642
Circumflex (^) in integer syntax 20
Clm: Declaration Syntax 640
Clm: Standard Dispatching Macro Character Syntax 644
Functions That Change Character Syntax 34
Underscore (_) in integer syntax 20
si:alphabetic syntax description 34
si:break syntax description 34
si:circlecross syntax description 34
si:doublequote syntax description 34
si:macro syntax description 34
si:single syntax description 34
si:slash syntax description 34
si:verticalbar syntax description 34
si:whitespace syntax description 34
Invalid pathname syntax errors 550
Syntax errors in read functions 543
SCL and Standard Character Syntax Types Table 643
sys:%1d-aloc function 246
sys:%1d-aref function 246
sys:%1d-aset function 246
Base Flavor: **sys:arithmetic-error** 534
Base Flavor: **sys:cell-contents-error** 533
The **sys:downward-function** and **sys:downward-funarg** Declarations 140
The **sys:downward-function** and **sys:downward-funarg** Declarations 140
:package message to **sys:external-symbol-not-found** 619

:string message to **sys:external-symbol-not-found** 619
Base Flavor: **sys:floating-point-exception** 535
:name message to **sys:package-not-found** 542, 619
:relative-to message to **sys:package-not-found** 542, 619
:package message to **sys:package-locked** 543, 619
:symbol message to **sys:package-locked** 543, 619
sys:abort flavor 502, 526, 532
sys:area-overflow flavor 537
sys:arithmetic-error flavor 534
sys:array-has-no-leader flavor 540
sys:array-register-1d 245
sys:array-register-1d declaration 312
sys:array-wrong-number-of-dimensions flavor 540
sys:array-wrong-number-of-subscripts flavor 541
sys:array-register 255
sys:array-register declaration 312
sys:bad-array-type flavor 540
sys:bad-connection-state flavor 555
sys:bad-data-type-in-memory flavor 534
sys:bitblt-array-fractional-word-width flavor 537
sys:bitblt-destination-too-small flavor 537
sys:call-trap flavor 523
sys:cell-contents-error flavor 533
sys:connection-closed-locally flavor 555
sys:connection-no-more-data flavor 556
sys:connection-closed flavor 555
sys:connection-error flavor 555
sys:connection-lost flavor 556
sys:connection-refused flavor 555
sys:cons-in-fixed-area flavor 538
Sysdcl file 598
sys:defsubst-with-parent macro 319
sys:disk-error flavor 538
sys:divide-by-zero flavor 479, 534
sys:double-float-p function 8
sys:downward-funarg declaration 139, 141, 313
sys:downward-function declaration 139, 140, 312
sys:draw-off-end-of-screen flavor 537
sys:draw-on-unprepared-sheet flavor 537
sys:end-of-file flavor 536
sys:external-symbol-not-found flavor 619
sys:fceiling function 112
sys:fdefine-file-pathname variable 319
sys:fdefinition-location function 322
sys:ffloor function 111
sys:fill-pointer-not-fixnum flavor 540
sys:float-divide-by-zero flavor 535
sys:float-divide-zero-by-zero flavor 536
sys:float-inexact-result flavor 536
sys:float-invalid-compare-operation flavor 536
sys:float-invalid-operation flavor 536
sys:floating-exponent-overflow flavor 535
sys:floating-exponent-underflow flavor 536
sys:floating-point-exception flavor 535
sys:fround function 113
sys:ftruncate function 112
sys:function-parent declaration 312
Using the **sys:function-parent** Declaration 319

sys:function-parent function 319
sys:host-not-responding-during-connection flavor 555
sys:host-not-responding flavor 555
sys:host-stopped-responding flavor 555
sys:instance-variable-pointer-out-of-range flavor 538
sys:instance-variable-zero-referenced flavor 538
sys:invalid-function flavor 542
sys:local-network-error flavor 554
sys:name-conflict flavor 619
sys:negative-sqrt flavor 536
sys:network-resources-exhausted flavor 554
sys:network-stream-closed flavor 556
sys:network-error flavor 554
sys:no-action-mixin flavor 532
sys:non-positive-log flavor 534
sys:number-array-not-allowed flavor 541
sys package 615
sys:package-not-found flavor 542, 619
Interning Errors Based on **sys:package-error** 542
sys:package-error flavor 542, 619
sys:package-locked flavor 543, 619
sys:parse-error flavor 543
sys:pdl-overflow flavor 537
sys:print-not-readable flavor 543
sys:printing-random-object macro 18
sys:read-end-of-file flavor 544
sys:read-list-end-of-file flavor 544
sys:read-premature-end-of-symbol flavor 544
sys:read-string-end-of-file flavor 544
sys:read-error flavor 544
sys:redefinition flavor 538
sys:region-table-overflow flavor 538
sys:remote-network-error flavor 554
sys:single-float-p function 8
sys:stream-closed flavor 536
sys:%string-exact-compare function 283
sys:%string-compare function 285
sys:subscript-out-of-bounds flavor 541
Compatibility with the Pre-release 5.0 Package System 625
Introduction to the Flavor System 417
Message Passing in the Flavor System 423
Objects and the Flavor System 417
Package system 566
Using the Flavor System 425
How the Package System Allows Symbol Sharing 569
System declaration file 598, 621
Miscellaneous System Errors Not Categorized by Base Flavor 536
Flavor system messages 453
Condition system package 615
Window system package 615
Maclisp system property names 564
Flavor system storing functions 297
System character 32
system-internals package 615
system package 571, 615
System Packages 615

Matrices and Systems of Linear Equations 258
sys:throw-tag-not-seen flavor 538
sys:too-few-arguments flavor 539
sys:too-many-arguments flavor 539
sys:unbound-closure-variable flavor 533
sys:unbound-instance-variable flavor 533
sys:unbound-symbol flavor 533
sys:unbound-variable flavor 533
sys:unclaimed-message flavor 542
sys:undefined-keyword-argument flavor 542
sys:undefined-function flavor 534
sys:unknown-host-name flavor 554
sys:unknown-locf-reference flavor 534
sys:unknown-setf-reference flavor 534
sys:unknown-address flavor 554
sys:virtual-memory-overflow flavor 537
sys:write-in-read-only flavor 537
sys:wrong-stack-group-state flavor 537
sys:wrong-type-argument condition 505
sys:wrong-type-argument flavor 539
sys:zero-args-to-select-method flavor 539

T T T

Tab character 32
Hash table 75
SCL and Standard Character Syntax Types Table 643
SCL and Standard Constituent Character Attributes Table 643
Hash table considerations while using multiprocessing 69
Hash table facilities 59
Truth table for the Boolean operations 114
Hash Table Functions 73
Hash table keys 69
Objects as hash table keys 69
Trees as hash table keys 69
Hash Table Messages 72
Tables 59
Clm: Hash Tables 641
Creating Hash Tables 71
Hash Tables 69
Lists as Tables 59
Hash Tables and Loop Iteration 75
Hash Tables and the Garbage Collector 75
loop Iteration Over Hash Tables or Heaps 224
Dumping Hash Tables to Files 75
Table searches 59
Throw tag error 538
tagbody special form 187
prog tags 189
tailp function 61
tand function 107
tan function 106
Tangent 106
tanh function 107
tape: **tape-device-error** flavor 556
tape:end-of-tape flavor 556
tape: **tape-error** flavor 556
Tape Errors 556

tape:mount-error flavor 556
tape:tape-device-error flavor 556
tape:tape-error flavor 556
Lists as templates 345
Terminal character 32
Loop termination 212, 214
SCL and **:test** keyword to **make-hash-table** function 641
Testing for even number 97
Testing for negative number 97
Testing for odd number 97
Testing for positive number 97
Testing for sign of a number 97
Testing for zero 97
Aggregated Boolean Tests 215
Text processing 277
Reading Integers in Bases Greater Than 10 21
Printed Representation of Arrays That Are Named Structures 16
Printed Representation of Arrays That Are Not Named Structures 17
Functions That Change Character Syntax 34
Functions That Change Characters Into Macro Characters 35
Functions That Create New Readtables 33
Functions That Find the Home Package of a Symbol 607
Functions That Import, Export, and Shadow Symbols 611
Functions That Map Names to Symbols 605
Functions That Operate on Locatives 84
Functions That Operate on Named Structures 405
Functions that return multiple values 167
Macros That Surround Code 362
the loop keyword 207, 222
SCL and **the** special form 640
SCL and *value-type* argument for **the** special form 640
their loop keyword 222
then loop keyword 207
thereis loop keyword 215
third function 46
Throw 197
***throw** 482
throw special form 198
Throw tag error 538
sys: **throw-tag-not-seen** flavor 538
***throw** function 200
Clm: Time Functions 645
:times option for **defstruct** 385, 390, 395
times function 102
to loop keyword 207, 225
Tokens in the input stream 19
sys: **too-few-arguments** flavor 539
sys: **too-many-arguments** flavor 539
Array Representation Tools 237
Clm: Debugging Tools 645
:top method of **si:heap** 79
Clm: the Top-level Loop 642
trace special form 302
trace-conditions variable 523
Tracing Conditions 523
Trailing decimal point 14
Transcendental Functions 106
Transfer of Control 187

Clm: Translating Strings to Symbols 640
math: **transpose-matrix** function 259
Inexact-result trap 535
Trap on exit bit 523
Enabling and disabling of floating-point traps 535
How Applications Programs Treat Conditions 481
Tree 41
Tree of flavors 431
Trees as hash table keys 69
Trigonometric functions 106
Degrees in trigonometric functions 106
Radians in trigonometric functions 106
true function 164
truncate function 110
Truth table for the Boolean operations 114
t symbol 561
tv package 615
Two Kinds of Characters 268
Interface between two modules 569
SCL sequence and list functions and two-argument predicates 641
:after method type 455, 457
:and method type 455, 458
:and method combination type 455
:append method combination type 455, 456
art-fat-string Array Type 236
art-q-list Array Type 236
art-16b array type 236
art-1b array type 236
art-2b array type 236
art-4b array type 236
art-8b array type 236
art-boolean Array Type 237
art-*N*b Array Type 236
art-q Array Type 236
art-string Array Type 236
:before method type 455, 457
:case method combination type 455, 457
:combined method type 455, 458
:create-package proceed type 542, 619
:daemon method combination type 455
:daemon-with-and method combination type 455, 456
:daemon-with-or method combination type 455, 456
:daemon-with-override method combination type 455, 457
:default method type 455, 457
dtp-instance-header data type 467
dtp-select-method data type 467
dtp-instance data type 467
:export proceed type 619
expr Maclisp type 305
fexpr Maclisp type 305
:handler function spec type 297
:internal function spec type 297
:inverse-list method combination type 455, 456
Lexpr Maclisp type 305
:list method combination type 455
:location function spec type 297
macro Maclisp type 305
:method function spec type 297

:nconc method combination type 455, 456
:new-name proceed type 542, 619
:no-action proceed type 523, 542, 543, 619
:or method type 455, 458
:or method combination type 455
:override method type 455, 457
:pass-on method combination type 455, 456
Proceed type 517
:progn method combination type 455
:property function spec type 297
:shadow proceed type 619
:shadowing-import proceed type 619
:share proceed type 619
:skip proceed type 619
Symbol function spec type 297
The Array Data Type 5
The Compiled Function Data Type 4
The Cons Data Type 4
The List Data Type 5
The Locative Data Type 5
The Symbol Data Type 3
:unintern proceed type 619
:whopper method type 455, 458
:within function spec type 297
:wrapper method type 455, 458
:choose proceed type 619
Invalid type code error 534
SCL and type common 637
Clm: Type Conversion Function 637
Numeric Type Conversions 107
Clm: Type Declaration for Forms 640
Proceed Type Messages 519
Data type names 576
Proceed Type Names 520
:type option for **defstruct** 385
SCL and **:type** option for **defstruct** 642
:type option for **make-array** 241
:type option for **make-plane** 261
Clm: Specific Data Type Predicates 638
SCL and type string 637
SCL and type string-char 637
typecase special form 181, 576
Return from **typeof** function 642
:array returned by **typep** 9
:atom argument to **typep** 9
:bignum returned by **typep** 9
:closure returned by **typep** 9
:compiled-function returned by **typep** 9
:complex returned by **typep** 9
:double-float returned by **typep** 9
:fix argument to **typep** 9
:fixnum returned by **typep** 9
:float argument to **typep** 9
:instance argument to **typep** 9
:list returned by **typep** 9
:list-or-nil argument to **typep** 9
:locative returned by **typep** 9
:non-complex-number argument to **typep** 9

:null argument to **typep** 9
:number argument to **typep** 9
:random returned by **typep** 9
:rational returned by **typep** 9
:select-method returned by **typep** 9
:single-float returned by **typep** 9
:stack-group returned by **typep** 9
:string returned by **typep** 9
:symbol returned by **typep** 9
typep function 9, 335, 403, 576, 597
Abstract types 418, 425
Array Types 235, 241
Clm: Data Types 637
Combination Method Types 459
Combining abstract types 431
Creating data types 379
Data Types 3
Definition types 319
Numeric Data Types 4
Printed Representation of Miscellaneous Data Types 18
Symbolics-Lisp Number Types 89
Types of Numbers 93
Types of numbers 4
Data Types Recognized by **loop** 219
SCL and Standard Character Syntax Types Table 643

U

Un-garbage-collected arrays 262
Unbindings 125
Unbound closure variable error 533
Unbound instance variable error 533
Unbound variable errors 533
sys: **unbound-closure-variable** flavor 533
sys: **unbound-instance-variable** flavor 533
Unbound symbol 561
sys: **unbound-symbol** flavor 533
sys: **unbound-variable** flavor 533
unbreakon function 524
sys: **unclaimed-message** flavor 542
:unclaimed-message message 454
uncompile function 316
undefflavor function 437
Undefined function error 534
sys: **undefined-keyword-argument** flavor 542
fs: **undefined-logical-pathname-translation** flavor 554
sys: **undefined-function** flavor 534
undefmethod macro 437, 468
undefun function 322
undefun-method special form 437
Exponent underflow 94
Exponent underflow error 536
Underscore (_) in integer syntax 20
si: **unencapsulate-function-spec** function 328
unexport function 593, 611
fs: **unimplemented-option** flavor 552
:unintern proceed type 619
Uninterned symbol 357, 566

Uninterned symbols 29, 573
union function 62
sys: **unknown-host-name** flavor 554
sys: **unknown-locf-reference** flavor 534
fs: **unknown-pathname-host** flavor 553
sys: **unknown-setf-reference** flavor 534
sys: **unknown-address** flavor 554
fs: **unknown-operation** flavor 547
fs: **unknown-property** flavor 551
fs: **unknown-user** flavor 547
unless loop keyword 216
unless macro 179
Drawing on unprepared sheet error 537
unspecial declaration 312
SCL pathnames and **:unspecific** 644
until loop keyword 214
unuse-package function 593, 611
unwind-protect-case macro 199
Unwinding a stack 197
unwind-protect special form 169, 198
Unwind protection 197
Up-Arrow (↑) character 32
Update functions 147
Updating generalized variables 147
upfrom loop keyword 207
upper-case-p function 270
Upward funargs 139
:use option for **defpackage** 571, 599
:use option for **make-package** 571, 602
Arrays used as functions 305
Stack groups used as functions 305
Symbols used as functions 297
Functions Used During Expansion 368
use-package function 571, 593, 610
Operations the User Can Perform on Functions 302
user package 579, 615
using loop keyword 222, 225

V V V

Checking for valid arguments 505
Absolute value 101
Value of instance variables 439
value-cell-location function 563
Value cell 3, 83, 331, 561, 563
External value cell 331
Internal value cell 331
The Value Cell of a Symbol 561
Changing the Value of a Variable 125
Values 212
Clm: Rules Governing the Passing of Multiple Values 639
Functions and Special Forms for Constant Values 161
Functions that return multiple values 167
Interaction of Some Common Special Forms with Multiple Values 169
Multiple Values 167
Passing-back of Multiple Values 169
Primitive for Producing Multiple Values 167

Property list values 67
Special Forms for Receiving Multiple Values 167
Default values for instance variables 431
SCL and returned values from **cond** macro 639
Altering slot values of structures 395
values declaration 312
values function 167
values-list function 167
SCL and *value-type* argument for **the** special form 640
Vanilla Flavor 453
si:vanilla-flavor flavor 453
all-flavor-names variable 429
alphabetic-case-affects-string-comparison variable 273
arglist variable 325
array-bits-per-element variable 238
array-elements-per-q variable 238
Array-register variable 255
array-types variable 237
base variable 14
Changing the Value of a Variable 125
cl:*read-default-float-format* variable 23
dbg:*interactive-handlers* variable 511
dbg:*proceed-type-special-keys* variable 528
dbg:*special-command-special-keys* variable 528
ibase variable 20
inhibit-fdefine-warnings variable 317
lambda-list-keywords variable 309
local-declarations variable 316
***nopoint** variable 14
package variable 579, 597
pkg-global-package variable 617
pkg-keyword-package variable 617
pkg-system-package variable 617
prinlength variable 18
prinlevel variable 17
readtable variable 33
SCL and * variable 642
SCL and ***default-pathname-defaults*** variable 644
SCL and ***features*** variable 645
SCL and ***read-extended-ibase-signed-numbers*** variable 643
SCL and ***read-extended-ibase-unsigned-numbers*** variable 643
SCL and ***read-base*** variable 643
SCL and ***read-suppress*** variable 643
self variable 437
si:*flavor-compile-trace* variable 440
si:*flavor-compilations* variable 440
si:*macroexpand-hook* variable 375
si:*print-object-error-message* variable 543
si:*read-extended-ibase-signed-number* variable 21
si:*read-extended-ibase-unsigned-number* variable 21
si:*read-multi-dot-tokens-as-symbols* variable 26
si:encapsulated-function variable 325
si:encapsulation-standard-order variable 327
si:initial-readtable variable 33
si:loop-use-system-destructuring? variable 221
si:print-readably variable 18
Supplied-p variable 155

sys:fdefine-file-pathname variable 319
trace-conditions variable 523
Unbound closure variable error 533
Unbound instance variable error 533
Instance variable errors 538
Unbound variable errors 533
Variable of iteration 207
Instance variable out-of-bounds 541
variable-boundp special form 562
variable-location special form 563
variable-makunbound special form 562
Variables 125
Binding Variables 125, 331
Binding local and special variables 126
Clm: Generalized Variables 639
Decrementing generalized variables 149
Default values for instance variables 431
Defining special variables 134
Effect of compiler on variables 126
Generalized Variables 147
Global variables 126
Incrementing generalized variables 149
Initializing instance variables 425, 435, 455
Instance variables 126, 418, 485
Iteration variables 221
Kinds of Variables 126
Local variables 126, 331
Locating generalized variables 147
Package Functions, Special Forms, and Variables 597
Setting variables 125, 126
Special variables 126, 331
Special Forms for Binding Variables 128
Special Forms for Defining Special Variables 134
Special Forms for Setting Variables 128
Stepping variables 221
Updating generalized variables 147
Value of instance variables 439
Variables in lambda lists 151
Instance variables of combined flavors 431
SCL and subtype variations 637
eq versus **equal** 10
eq versus **equal** 11
SCL and vertical bars for quoting 643
si: **verticalbar** syntax description 34
sys: **virtual-memory-overflow** flavor 537

W W W

What is a Dynamic Closure? 331
What is a Function? 297
What is a Handler? 487
Clm: What the Print Function Produces 644
What the Printer Produces 14
Clm: What the Read Function Accepts 643
What the Reader Recognizes 20
when loop keyword 216
when macro 179
Where Is Symbol (m-X) Zmacs command 571

where-is function 571, 612, 625
:which-operations keyword for **named-structure-invoke** 403, 404
:which-operations message 453
:which-operations symbol 520
while loop keyword 214
Hash table considerations while using multiprocessing 69
si: **whitespace** syntax description 34
&whole Lambda-list Keyword 311, 374
:whopper method type 455, 458
Whoppers and Wrappers 461
fs: **wildcard-not-allowed** flavor 550
Window system package 615
with loop keyword 210
dbg: **with-erring-frame** macro 494, 495
with-input-from-string special form 290
with-output-to-string special form 291
:within function spec type 297
with-key loop keyword 224
without-floating-underflow-traps special form 94
Pluralizing words 280
How the Printer Works 13
How the Reader Works 19
:wrapper method type 455, 458
Modifying wrappers 468
Ordering Flavors, Methods, and Wrappers 467
Whoppers and Wrappers 461
sys: **write-in-read-only** flavor 537
Hints to Macro Writers 357
fs: **wrong-kind-of-file** 550
fs: **wrong-kind-of-file** flavor 550
sys: **wrong-stack-group-state** flavor 537
sys: **wrong-type-argument** condition 505
sys: **wrong-type-argument** flavor 539

X

X X

xcons-in-area function 45
xcons function 45, 56

Z

Z Z

Testing for zero 97
Division by zero error 534
sys: **zero-args-to-select-method** flavor 539
zerop function 97
Zetalisp and SCL 631
Zetalisp and SCL character incompatibilities 265
Zetalisp and SCL string incompatibilities 265
SCL **zl** package 633
SCL and **zl:si:*flag-wrong-type-strings*** 637, 642
SCL **zl-user** package 633
Where Is Symbol (m-X) Zmacs command 571
Zmacs Command: Describe Flavor 471
Zmacs Command: Edit Combined Methods 472
Zmacs Command: Edit Methods 471
Zmacs Command: List Combined Methods 471

Zmacs Command: List Methods 471
Zmacs Command: m-. 471
Zmacs Commands for Flavors 471
Zmail package 615
zwei package 615

\ \ \
\ function 103
\\ function 105

^ ^ ^
Circumflex (^) in integer syntax 20
^$ function 105
^ function 105

_ _ _
Underscore (_) in integer syntax 20

` ` `
Backquote character (`) 345
Backquote (`) macro character 26

| | |
SCL and | 643

~ ~ ~
SCL and ~E directive 644
SCL and ~G directive 644
SCL and ~T directive 644

∫ ∫ ∫
Integral (∫) character 32