

Data General COMMON LISP Reference Manual





62 T.W. Alexander Drive
Research Triangle Park
North Carolina 27709
Telephone: 919-549-8421

September 17, 1985

Dr. Guy Steele
Thinking Machines Incorporated
245 First Street
Cambridge MA 02142

Dear Dr. Steele:

As promised, here is a courtesy copy of our derivative Common LISP Manual. I would be very happy to receive your comments or suggestions on any aspect of this book. Dan Oldman tells me that he senses a strong interest in Common LISP among the user community.

Thanks for your help in making this excellent manuscript available to us for publication.

Yours truly,

A handwritten signature in cursive script that reads 'Carl M. Lewis'.

Carl M. Lewis

CML:jp

Enclosure

Introduction

1.1

1.2

1.3

1.4

1.5

1.6

1.7

1.8

1.9

1.10

1.11

1.12

1.13

1.14

1.15

1.16

1.17

1.18

1.19

1.20

1.21

1.22

1.23

1.24

1.25

1.26

1.27

1.28

1.29

1.30

1.31

1.32

1.33

1.34

1.35

1.36

1.37

1.38

1.39

1.40

1.41

1.42

1.43

1.44

1.45

1.46

1.47

1.48

1.49

1.50

1.51

1.52

1.53

1.54

1.55

1.56

1.57

1.58

1.59

1.60

1.61

1.62

1.63

1.64

1.65

1.66

1.67

1.68

1.69

1.70

1.71

1.72

1.73

1.74

1.75

1.76

1.77

1.78

1.79

1.80

1.81

1.82

1.83

1.84

1.85

1.86

1.87

1.88

1.89

1.90

1.91

1.92

1.93

1.94

1.95

1.96

1.97

1.98

1.99

1.100

1.101

1.102

1.103

1.104

1.105

1.106

1.107

1.108

1.109

1.110

1.111

1.112

1.113

1.114

1.115

1.116

1.117

1.118

1.119

1.120

1.121

1.122

1.123

1.124

1.125

1.126

1.127

1.128

1.129

1.130

1.131

1.132

1.133

1.134

1.135

1.136

1.137

1.138

1.139

1.140

1.141

1.142

1.143

1.144

1.145

1.146

1.147

1.148

1.149

1.150

1.151

1.152

1.153

1.154

1.155

1.156

1.157

1.158

1.159

1.160

1.161

1.162

1.163

1.164

1.165

1.166

1.167

1.168

1.169

1.170

1.171

1.172

1.173

1.174

1.175

1.176

1.177

1.178

1.179

1.180

1.181

1.182

1.183

1.184

1.185

1.186

1.187

1.188

1.189

1.190

1.191

1.192

1.193

1.194

1.195

1.196

1.197

1.198

1.199

1.200

1.201

1.202

1.203

1.204

1.205

1.206

1.207

1.208

1.209

1.210

1.211

1.212

1.213

1.214

1.215

1.216

1.217

1.218

1.219

1.220

1.221

1.222

1.223

1.224

1.225

1.226

1.227

1.228

1.229

1.230

1.231

1.232

1.233

1.234

1.235

1.236

1.237

1.238

1.239

1.240

1.241

1.242

1.243

1.244

1.245

1.246

1.247

1.248

1.249

1.250

1.251

1.252

1.253

1.254

1.255

1.256

1.257

1.258

1.259

1.260

1.261

1.262

1.263

1.264

1.265

1.266

1.267

1.268

1.269

1.270

1.271

1.272

1.273

1.274

1.275

1.276

1.277

1.278

1.279

1.280

1.281

1.282

1.283

1.284

1.285

1.286

1.287

1.288

1.289

1.290

1.291

1.292

1.293

1.294

1.295

1.296

1.297

1.298

1.299

1.300

1.301

1.302

1.303

1.304

1.305

1.306

1.307

1.308

1.309

1.310

1.311

1.312

1.313

1.314

1.315

1.316

1.317

1.318

1.319

1.320

1.321

1.322

1.323

1.324

1.325

1.326

1.327

Data General COMMON LISP Reference Manual

093-701017-00

This documentation is based on *COMMON LISP: The Language*, written by Guy L. Steele, Jr., and published by Digital Press (copyright 1984 by Digital Equipment Corporation), 30 North Avenue, Burlington, MA 01803. The original work constitutes the sole specification for *COMMON LISP* and any departures from that standard are the sole responsibility of Data General Corporation.

Ordering No. 093-701017

© Data General Corporation, 1985

All Rights Reserved

Licensed Material—Property of Data General Corporation

Printed in the United States of America

Revision 00, March 1985

NOTICE

Data General Corporation (DGC) has prepared this manual for use by DGC personnel, licensees, and customers. The information contained herein is the property of DGC and shall not be reproduced in whole or in part without DGC prior written approval.

DGC reserves the right to make changes in specifications and other information contained in this document without prior notice, and the reader should in all cases consult DGC to determine whether any such changes have been made.

THE TERMS AND CONDITIONS GOVERNING THE SALE OF DGC HARDWARE PRODUCTS AND THE LICENSING OF DGC SOFTWARE CONSIST SOLELY OF THOSE SET FORTH IN THE WRITTEN CONTRACT BETWEEN DGC AND ITS CUSTOMERS. NO REPRESENTATION OR OTHER AFFIRMATION OF FACT CONTAINED IN THIS DOCUMENT INCLUDING BUT NOT LIMITED TO STATEMENTS REGARDING CAPACITY, RESPONSE-TIME PERFORMANCE, SUITABILITY FOR USE OR PERFORMANCE OF PRODUCTS DESCRIBED HEREIN SHALL BE DEEMED TO BE A WARRANTY BY DGC FOR ANY PURPOSE, OR GIVE RISE TO ANY LIABILITY OF DGC WHATSOEVER.

This software is made available solely pursuant to the terms of a DGC license agreement which governs its use.

Data General COMMON LISP Reference Manual
Revision 00
March 1985

Copyright © Data General Corporation, 1985
All rights reserved, printed in USA

CEO, DASHER, DATAPREP, ECLIPSE, ECLIPSE MV/4000, ECLIPSE MV/6000, ECLIPSE MV/8000, ENTERPRISE, GENAP, INFOS, MANAP, microNOVA, NOVA, PROXI, PRESENT, SWAT, SUPERNOVA, and TRENDVIEW are U.S. registered trademarks of Data General Corporation.

AZ-TEXT, BusiGEN, BusiPEN, BusiTEXT, DEFINE, DG/L, DG/UX, DG/XAP, ECLIPSE MV/10000, GDC/1000, GW/4000, microECLIPSE, MV/UX, REV-UP, SLATE and XODIAC are U.S. trademarks of Data General Corporation.

Preface

This manual is the primary reference for the Data General COMMON LISP system that runs under the Advanced Operating System/Virtual Storage (AOS/VS) and its interface to the DG MV/UX[®] operating system. DG COMMON LISP comprises both a language system and an environment. The language system includes an interpreter, a compiler, and runtime libraries for COMMON LISP. The environment is a set of program development tools, including a debugger and the EMACS text editor.

This manual completely incorporates *COMMON LISP: the Language* by Guy L. Steele, Jr., with contributions by Scott E. Fahlman, Richard P. Gabriel, David A. Moon, and Daniel L. Weinreb; copyright held by Digital Equipment Corporation and published by Digital Press, Burlington, MA. The Steele book is the *de facto* standard for COMMON LISP; the Data General Corporation (DGC) implementation, DG COMMON LISP, supports most of this standard and provides some additional facilities. Within this book, descriptions of extensions and comments that are not part of the standard work are written in colored ink. COMMON LISP features that are not supported in this implementation or comments that are not relevant are shaded.

Who Should Read This Manual?

This is a reference manual for experienced LISP programmers. It assumes that the reader has enough experience with the AOS/VS Command Line Interpreter (CLI) or the MV/UX shell to manipulate files and execute programs. Knowledge of the EMACS text editor for COMMON LISP is not assumed; EMACS users are referred to the appropriate manual below.

Manual Organization

This manual is designed to give quick access to needed information. It is not intended to be read sequentially, except that later chapters do not redefine terms defined earlier (you can find the definitions through the index). Each section is self-contained, with references to other relevant sections.

The COMMON LISP notational conventions are described in Chapter 1. AOS/VS and MV/UX commands use their standard notations.

Prerequisite Manuals

- *Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS)* (093-000122) describes the interactive interface to AOS and AOS/VS.
- *MV/UX System User's Manual* (093-701001) explains the MV/UX system and how to use it.

Other Related Manuals

- *Data General EMACS Text Editor User's Manual* (093-701011) describes the text editor used with the Data General COMMON LISP system.
- *AOS/VS Programmer's Manual* Vol. I (093-000335) and Vol. II (093-000241) introduces system concepts and the AOS/VS functions for coding a program in assembly language. The system calls are in Volume II.
- *AOS/VS Link and Library File Editor (LFE) User's Manual* (093-000245) describes two fundamental AOS/VS utilities. Link consolidates object modules and library files into executable program files. LFE creates, edits, and analyzes library files.

Contacting Data General

- To order any Data General manual, notify your sales representative and supply the manual title and order number.
- If you have hardware problems, please notify your Support Center.
- If you have software problems, please notify your local Support Center or submit a Software Trouble Report (STR) to the Software Trouble Report Processing Center.

End of Preface

CONTENTS

Preface	p-1
Acknowledgements	a-1
Chapter 1—Introduction	1-1
Purpose	1-1
Notational Conventions	1-3
Decimal Numbers	1-3
Nil, False, and the Empty List	1-3
Evaluation, Expansion, and Equivalence	1-4
Errors	1-4
Descriptions of Functions and Other Entities	1-6
The Lisp Reader	1-7
Overview of Syntax	1-7
Chapter 2—Data Types	2-1
Numbers	2-3
Integers	2-3
Ratios	2-4
Floating-point Numbers	2-5
Complex Numbers	2-7
Characters	2-8
Standard Characters	2-8
Line Divisions	2-9
Non-standard Characters	2-10
Character Attributes	2-11
String Characters	2-11
Symbols	2-11
Lists and Conses	2-13
Arrays	2-14
Vectors	2-15
Strings	2-16
Bit-Vectors	2-17
Hash Tables	2-17
Readtables	2-17
Packages	2-17
Pathnames	2-17
Streams	2-18
Random-States	2-18
Structures	2-18
Functions	2-18
Unreadable Data Objects	2-19
Overlap, Inclusion, and Disjointness of Types	2-19
Chapter 3—Scope and Extent	3-1

Chapter 4—Type Specifiers	4-1
Type Specifier Symbols	4-1
Type Specifier Lists	4-1
Predicating Type Specifiers	4-2
Type Specifiers that Combine.....	4-2
Type Specifiers that Specialize.....	4-4
Type Specifiers that Abbreviate.....	4-6
Defining New Type Specifiers	4-8
Type Conversion Function	4-9
Determining the Type of an Object.....	4-10
Chapter 5—Program Structure	5-1
Forms	5-1
Self-Evaluating Forms.....	5-2
Variables	5-2
Special Forms	5-3
Macros	5-4
Function Calls.....	5-4
Functions	5-5
Named Functions	5-5
Lambda-Expressions	5-5
Top-Level Forms	5-11
Defining Named Functions.....	5-11
Declaring Global Variables and Named Constants.....	5-12
Control of Time of Evaluation	5-13
Chapter 6—Predicates	6-1
Logical Values	6-1
Data Type Predicates	6-2
General Type Predicates.....	6-2
Specific Data Type Predicates.....	6-2
Equality Predicates	6-6
Logical Operators.....	6-10
Chapter 7—Control Structure	7-1
Constants and Variables.....	7-1
Reference	7-2
Assignment	7-6
Generalized Variables	7-7
Function Invocation.....	7-19
Simple Sequencing.....	7-20
Establishing New Variable Bindings	7-21
Conditionals	7-25
Blocks and Exits	7-29
Iteration	7-30
Indefinite Iteration	7-30
General Iteration	7-31
Simple Iteration Constructs	7-34
Mapping	7-36

The “Program Feature”	7-38
Multiple Values	7-41
Constructs for Handling Multiple Values	7-41
Rules Governing the Passing of Multiple Values	7-44
Dynamic Non-local Exits	7-46
Chapter 8—Macros	8-1
Macro Definition	8-1
Macro Expansion	8-7
Chapter 9—Declarations	9-1
Declaration Syntax	9-1
Declaration Specifiers	9-4
Type Declaration for Forms	9-8
Chapter 10—Symbols	10-1
The Property List	10-1
The Print Name	10-5
Creating Symbols	10-5
Chapter 11—Packages	11-1
Consistency Rules	11-2
Package Names	11-3
Translating Strings to Symbols	11-3
Exporting and Importing Symbols	11-5
Name Conflicts	11-6
Built-in Packages	11-9
Package System Functions and Variables	11-9
Modules	11-15
An Example	11-16
Chapter 12—Numbers	12-1
Precision, Contagion, and Coercion	12-1
Predicates on Numbers	12-3
Comparisons on Numbers	12-4
Arithmetic Operations	12-6
Irrational and Transcendental Functions	12-9
Exponential and Logarithmic Functions	12-10
Trigonometric and Related Functions	12-11
Functions in the Complex Plane	12-15
Type Conversions and Component Extractions on Numbers	12-18
Logical Operations on Numbers	12-24
Byte Manipulation Functions	12-28
Random Numbers	12-30
Implementation Parameters	12-33
Chapter 13—Characters	13-1
Character Attributes	13-1

Predicates on Characters	13-2
Character Construction and Selection	13-6
Character Conversions	13-7
Character Control-Bit Functions	13-9
Chapter 14—Sequences	14-1
Simple Sequence Functions	14-3
Concatenating, Mapping, and Reducing Sequences	14-4
Modifying Sequences	14-7
Searching Sequences for Items	14-11
Sorting and Merging	14-12
Chapter 15—Lists	15-1
Conses	15-1
Lists	15-3
Alteration of List Structure	15-10
Substitution of Expressions	15-11
Using Lists as Sets	15-12
Association Lists	15-16
Chapter 16—Hash Tables	16-1
Hash Table Functions	16-2
Primitive Hash Function	16-4
Chapter 17—Arrays	17-1
Array Creation	17-1
Array Access	17-5
Array Information	17-6
Functions on Arrays of Bits	17-8
Fill Pointers	17-9
Changing the Dimensions of an Array	17-11
Chapter 18—Strings	18-1
String Access	18-1
String Comparison	18-2
String Construction and Manipulation	18-3
Chapter 19—Structures	19-1
Introduction to Structures	19-1
How to Use Defstruct	19-3
Using the Automatically Defined Constructor Function	19-4
Defstruct Slot-Options	19-5
Defstruct Options	19-5
By-position Constructor Functions	19-9
Explicitly Specified Representational Type Structures	19-10
Unnamed Structures	19-10
Named Structures	19-11
Other Aspects of Explicitly Specified Structures	19-12

Chapter 20—The Evaluator	20-1
Run-Time Evaluation of Forms	20-1
The Top-Level Loop	20-4
Chapter 21—Streams	21-1
Standard Streams	21-1
Creating New Streams	21-2
Operations on Streams	21-5
Chapter 22—Input/Output	22-1
Printed Representation of LISP Objects	22-1
What the Read Function Accepts	22-2
Parsing of Numbers and Symbols	22-7
Macro Characters	22-13
Standard Dispatching Macro Character Syntax	22-18
The Readtable	22-26
What the Print Function Produces	22-30
Input Functions	22-38
Input from Character Streams	22-38
Input from Binary Streams	22-44
Output Functions	22-45
Output to Character Streams	22-45
Output to Binary Streams	22-47
Formatted Output to Character Streams	22-47
Querying the User	22-67
Chapter 23—File System Interface	23-1
File Names	23-1
Pathnames	23-2
Pathname Functions	23-4
Opening and Closing Files	23-8
Renaming, Deleting, and Other File Operations	23-13
Loading Files	23-15
Accessing Directories	23-16
Chapter 24—Errors	24-1
General Error-Signalling Functions	24-1
Conditions	24-5
Defining and Signalling Conditions	24-6
Examples of Error Signalling	24-7
Predefined Conditions	24-10
Specialized Error-Signalling Forms and Macros	24-10
Special Forms for Exhaustive Case Analysis	24-12
Chapter 25—Miscellaneous Features	25-1
The Compiler	25-1
Documentation	25-3
Debugging Tools	25-4

Environment Inquiries	25-10
Time Functions	25-10
Storage Management	25-14
Other Environment Inquiries	25-15
Identity Function	25-16
Chapter 26—Foreign Calling	26-1
Representation Types	26-2
Predefined Representation Types	26-2
Representation Type Attributes	26-5
Default Attributes of Representation Types	26-7
Defining New Representation Types	26-8
Foreign Structures	26-9
Foreign Structure Options	26-10
Foreign Entries	26-11
Building a Foreign Code Region	26-13
Errors and Signals in Foreign Code	26-13
Sharing Process Resources with LISP	26-13
Sharing Address Space and Memory	26-14
Chapter 27—Operating System Interface	27-1
Logging on LISP	27-1
Switches	27-1
Heap Files	27-2
Interrupt Handling	27-2
Interacting with the Operating System	27-3
Chapter 28—References	28-1

Acknowledgements

COMMON LISP was designed by a diverse group of people affiliated with many institutions. Contributors to the design and implementation of COMMON LISP and to the polishing of this manual are hereby gratefully acknowledged:

Paul Anagnostopoulos	Digital Equipment Corporation
Dan Aronson	Carnegie-Mellon University
Alan Bawden	Massachusetts Institute of Technology
Eric Benson	University of Utah, Stanford University, and Symbolics, Incorporated
Jon Bentley	Carnegie-Mellon University and Bell Laboratories
Jerry Boetje	Digital Equipment Corporation
Gary Brooks	Texas Instruments
Rodney A. Brooks	Stanford University
Gary L. Brown	Digital Equipment Corporation
Richard L. Bryan	Symbolics, Incorporated
Glenn S. Burke	Massachusetts Institute of Technology
Howard I. Cannon	Symbolics, Incorporated
George J. Carrette	Massachusetts Institute of Technology
Robert Cassels	Symbolics, Incorporated
Monica Cellio	Carnegie-Mellon University
David Dill	Carnegie-Mellon University
Scott E. Fahlman	Carnegie-Mellon University
Richard J. Fateman	University of California, Berkeley
Neal Feinberg	Carnegie-Mellon University
Ron Fischer	Rutgers University
John Foderaro	University of California, Berkeley
Steve Ford	Texas Instruments
Richard Gabriel	Stanford University and Lawrence Livermore National Library
Joseph Ginder	Carnegie-Mellon University and Perq Systems Corp.
Bernard S. Greenberg	Symbolics, Incorporated
Richard Greenblatt	Lisp Machines Incorporated (LMI)
Martin L. Griss	University of Utah and Hewlett-Packard, Incorporated
Steven Handerson	Carnegie-Mellon University
Charles L. Hedrick	Rutgers University
Gail Kaiser	Carnegie-Mellon University
Earle A. Killian	Lawrence Livermore National Laboratory
Steve Krueger	Texas Instruments
John L. Kulp	Symbolics, Incorporated
Jim Large	Carnegie-Mellon University
Rob Maclachlan	Carnegie-Mellon University
William Maddox	Carnegie-Mellon University
Larry M. Masinter	Xerox Corporation
John McCarthy	Stanford University

Michael E. McMahon	Symbolics, Incorporated
Brian Milnes	Carnegie-Mellon University
David A. Moon	Symbolics, Incorporated
Beryl Morrison	Digital Equipment Corporation
Don Morrison	University of Utah
Dan Pierson	Digital Equipment Corporation
Kent M. Pitman	Massachusetts Institute of Technology
Jonathan Rees	Yale University
Walter van Roggen	Digital Equipment Corporation
Susan Rosenbaum	Texas Instruments
William L. Scherlis	Carnegie-Mellon University
Lee Schumacher	Carnegie-Mellon University
Richard M. Stallman	Massachusetts Institute of Technology
Barbara K. Steele	Carnegie-Mellon University
Guy L. Steele Jr.	Carnegie-Mellon University and Tartan Laboratories Incorporated
Peter Szolovits	Massachusetts Institute of Technology
William vanMelle	Xerox Corporation, Palo Alto Research Center
Ellen Waldrum	Texas Instruments
Allan C. Wechsler	Symbolics, Incorporated
Daniel L. Weinreb	Symbolics, Incorporated
Jon L. White	Xerox Corporation, Palo Alto Research Center
Skef Wholey	Carnegie-Mellon University
Richard Zippel	Massachusetts Institute of Technology
Leonard Zubkoff	Carnegie-Mellon University and Tartan Laboratories Incorporated

Some contributions were relatively small; others involved enormous expenditures of effort and great dedication. A few of the contributors served more as worthy adversaries than as benefactors (and do not necessarily endorse the final design reported here), but their pointed criticisms were just as important to the polishing of COMMON LISP as all the positively phrased suggestions. All of the people named above were helpful in one way or another, and I am grateful for the interest and spirit of cooperation that allowed most decisions to be made by consensus after due discussion.

Considerable encouragement and moral support were also provided by:

Norma Abel	Digital Equipment Corporation
Roger Bate	Texas Instruments
Harvey Cragon	Texas Instruments
Dennis Duncan	Digital Equipment Corporation
Sam Fuller	Digital Equipment Corporation
A. Nico Habermann	Carnegie-Mellon University
Berthold K. P. Horn	Massachusetts Institute of Technology
Gene Kromer	Texas Instruments
Gene Matthews	Texas Instruments
Allan Newell	Carnegie-Mellon University
Dana Scott	Carnegie-Mellon University
Harry Tennant	Texas Instruments
Patrick H. Winston	Massachusetts Institute of Technology
Lowell Wood	Lawrence Livermore National Laboratory
William A. Wulf	Carnegie-Mellon University and Tartan Laboratories Incorporated

I am very grateful to each of them.

Jan Zubkoff of Carnegie-Mellon University provided a great deal of organization, secretarial support, and unfailing good cheer in the face of adversity.

The development of COMMON LISP would most probably not have been possible without the electronic message system provided by the ARPANET. Design decisions were made on several hundred distinct points, for the most part by consensus, and by simple majority vote when necessary. Except for two one-day face-to-face meetings, all of the language design and discussion was done through the ARPANET message system, which permitted effortless dissemination of messages to dozens of people, and several interchanges per day. The message system also provided automatic archiving of the entire discussion, which has proved invaluable in the preparation of this reference manual. Over the course of thirty months, approximately 3000 messages were sent (an average of three per day), ranging in length from one line to twenty pages. Assuming 5000 characters per printed page of text, the entire discussion totaled about 1100 pages. It would have been substantially more difficult to have conducted this discussion by any other means, and would have required much more time.

The ideas in COMMON LISP have come from many sources and been polished by much discussion. I am responsible for the form of this manual, and for any errors or inconsistencies that may remain; but the credit for the design and support of COMMON LISP lies with the individuals named above, each of whom has made significant contributions.

The organization and content of this manual were inspired in large part by the *MacLISP Reference Manual* by David A. Moon and others [12], and by the *LISP Machine Manual* (fourth edition) by Daniel Weinreb and David Moon [21], which in turn acknowledges the efforts of Richard Stallman, Mike McMahon, Alan Bawden, Glenn Burke, and "many people too numerous to list."

I thank Phyllis Keenan, Chase Duffy, Virginia Anderson, John Osborn, and Jonathan Baker of Digital Press for their help in preparing this book for publication. Jane Blake did an admirable job of copy-editing. James Gibson and Katherine Downs of Waldman Graphics were most cooperative in typesetting this book from my on-line manuscript files.

I am grateful to Carnegie-Mellon University and to Tartan Laboratories Incorporated for supporting me in the writing of this manual over the last three years.

Part of the work on this book was done in conjunction with the Carnegie-Mellon University Spice Project, an effort to construct an advanced scientific software development for personal computers. The Spice Project is supported by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-78-C-1551. The views and conclusions contained in this book are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U. S. Government.

Most of the writing of this book took place between midnight and 5 A.M. I am grateful to Barbara, Julia, and Peter for putting up with it, and for their love.

Guy L. Steele Jr.
Pittsburgh, Pennsylvania
March 1984

Data General Corporation wishes to thank Carnegie-Mellon University and the Spice Project for providing the public domain software that was used as a basis for the DG COMMON LISP project.

End of Acknowledgements

Chapter 1

Introduction

COMMON LISP is a new dialect of LISP, a successor to MACLISP [12,15], influenced strongly by ZETALISP [21, 13] and also to some extent by SCHEME [18] and INTERLISP [20].

Purpose

COMMON LISP is intended to meet these goals:

Commonality

COMMON LISP originated in an attempt to focus the work of several implementation groups, each of which was constructing successor implementations of MACLISP for different computers. These implementations had begun to diverge because the differences in the implementation environments; microcoded personal computers (ZETALISP, SPICE LISP) commercial timeshared computers (NIL), and supercomputers (S-1 LISP). While the differences among the several implementation environments of necessity will continue to force certain incompatibilities among the implementations, COMMON LISP serves as a common dialect to which each implementation makes any necessary extensions.

Portability

COMMON LISP intentionally excludes features that cannot be implemented easily on a broad class of machines. On the one hand, features that are difficult or expensive to implement on hardware without special microcode are avoided or provided in a more abstract and efficiently implementable form. (Examples of this are the invisible forwarding pointers and locatives of ZETALISP. Some of the problems that they solve are addressed in different ways in COMMON LISP.) On the other hand, features that are useful only on certain "ordinary" or "commercial" processors are avoided or made optional. (An example of this is the type declaration facility, which is useful in some implementations and completely ignored in others. Type declarations are completely optional and for correct programs affect only efficiency, not semantics.) COMMON LISP is designed to make it easy to write programs that depend as little as possible on machine-specific characteristics, such as word length, while allowing some variety of implementation techniques.

Consistency

Most LISP implementations are internally inconsistent in that by default the interpreter and compiler may assign different semantics to correct programs. This semantic difference stems primarily from the fact that the interpreter assumes all variable to be dynamically scoped, whereas the compiler assumes all variables to be local unless explicitly directed otherwise. This difference has been the usual practice in LISP for the sake of convenience and efficiency, but can lead to very subtle bugs. The definition of COMMON LISP avoids such anomalies by explicitly requiring the interpreter and compiler to impose identical semantics on correct programs so far as possible.

Expressiveness

COMMON LISP culls what experience has shown to be the most useful and understandable constructs from not only MACLISP, but also INTERLISP, other LISP dialects, and other programming languages. Constructs judged to be awkward or less useful have been excluded. (An example is the store construct of MACLISP.)

Compatibility

Unless there is a good reason to the contrary, COMMON LISP strives to be compatible with ZETALISP, MACLISP, and INTERLISP, roughly in that order.

Efficiency

COMMON LISP has a number of features designed to facilitate the production of high-quality compiled code in those implementations whose developers care to invest effort in an optimizing compiler. One implementation of COMMON LISP, namely S-1 LISP, already has a compiler that produces code for numerical computations that is competitive in execution speed with that produced by a FORTRAN compiler [3]. The S-1 LISP compiler extends the work done in MACLISP to produce extremely efficient numerical code [7].

Power

COMMON LISP is a descendant of MACLISP, which has traditionally placed emphasis on providing system-building tools. Such tools may in turn be used to build the user-level packages such as INTERLISP provides; these packages are not, however, part of the COMMON LISP core specification. It is expected such packages will be built on top of the COMMON LISP core.

Stability

It is intended that COMMON LISP will change only slowly and with due deliberation. The various dialects that are supersets of COMMON LISP may serve as laboratories within which to test language extensions, but such extensions will be added to COMMON LISP only after careful examination and experimentation.

The goals of COMMON LISP are thus very close to those of STANDARD LISP [11] and PORTABLE STANDARD LISP [16]. COMMON LISP differs from STANDARD LISP primarily in incorporating more features, including a richer and more complicated set of data types and more complex control structures.

This book is intended to be a language specification rather than an implementation specification (although implementation notes are scattered throughout the text). It defines a set of standard language concepts and constructs that may be used for communication of data structures and algorithms in the COMMON LISP dialect. This set of concepts and constructs is sometimes referred to as the “core COMMON LISP language” because it contains conceptually necessary or important features. It is not necessarily implementationally minimal. While many features could be defined in terms of others by writing LISP code, and indeed may be implemented that way, it was felt that these features should be conceptually primitive so that there might be agreement among all users as to their usage. (For example, bignums and rational numbers could be implemented as LISP code given operations on fixnums. However, it is important to the conceptual integrity of the language that they be regarded by the user as primitive, and theory are useful enough to warrant a standard definition.)

For the most part, this book defines a programming language, not a programming environment. A few interfaces are defined for invoking such standard programming tools as a compiler, an editor, a program trace facility, and a debugger, but very little is said about their nature or operation. It is expected that one or more extensive programming environments will be built using COMMON LISP as a foundation, and will be documented separately.

Notational Conventions

A number of special notational conventions are used through this book.

Decimal Numbers

All numbers in this book are in decimal notation unless there is an explicit indication to the contrary. (Decimal notation is normally taken for granted, of course. Unfortunately, for certain other dialects of LISP, MACLISP in particular, the default notation for numbers is octal (base 8) rather than decimal, and so the use of decimal notation for describing COMMON LISP is, taken in its historical context, a bit unusual!)

Nil, False, and the Empty List

In COMMON LISP, as in most LISP dialects, the symbol `nil` is used to represent both the empty list and the “false” value for Boolean tests. An empty list may, of course, also be written `()`; this normally denotes the same object as `nil`. (It is possible, by extremely perverse manipulation of the package system, to cause the sequence of letters `nil` to be recognized not as the symbol that represents the empty list but as another symbol with the same name. This obscure possibility will be ignored in this manual.) These two notations may be used interchangeably as far as the LISP system is concerned. However, as a matter of style, this manual uses the notation `()` when it is desirable to emphasize the use of an empty list, and uses the notation `nil` when it is desirable to emphasize the use of the Boolean “false.” The notation `'nil` (note the explicit quotation mark) is used to emphasize the use of a symbol. For example:

```
(defun three () 3)      ;Emphasize empty parameter list.
(append '() '()) ⇒ () ;Emphasize use of empty lists
(not nil) ⇒ t         ;Emphasize use as Boolean “false”
(get 'nil 'color)     ;Emphasize use as a symbol
```

Any data object other than `nil` is construed to be Boolean “not false,” that is, “true.” The symbol `t` is conventionally used to mean “true” when no other value is more appropriate. When a function is said to “return *false*” or to “be *false*” in some circumstance, this means that it returns `nil`. However, when a function is said to “return *true*” or to “be *true*” in some circumstance, this means that it returns some value other than `nil`, but not necessarily `t`.

Evaluation, Expansion, and Equivalence

Execution of code in LISP is called *evaluation* because executing a piece of code normally results in a data object called the *value* produced by the code. The symbol \Rightarrow is used in examples to indicate evaluation. For example,

`(+ 4 5)` \Rightarrow 9

means “the result of evaluating the code `(+ 4 5)` is (or would be, or would have been) 9.”

The symbol \rightarrow is used in examples to indicate macro expansion. For example,

`(push x v)` \rightarrow `(seff v (cons x v))`

means “the result of expanding the macro-call form `(push x v)` is `seff v (cons x v)`.” This implies that the two pieces of code do the same thing; the second piece of code is the definition of what the first does.

The symbol \equiv is used in examples to indicate code equivalence. For example,

`(gcd x (gcd y z))` \equiv `(gcd (gcd x y) z)`

means “the value and effects of evaluating the form `(gcd x (gcd y z))` are always the same as the value and effects of `(gcd x y) z` for any values of the variables `x`, `y`, and `z`.” This implies that the two pieces of code do the same thing; however, neither directly defines the other in the way macro expansion does.

Errors

When this manual specifies that it “is an error” for some situation to occur, this means that:

- No valid COMMON LISP program should cause this situation to occur.
- If this situation occurs, the effects and results are completely undefined as far as adherence to the COMMON LISP specification is concerned.
- No COMMON LISP implementation is required to detect such an error. Of course, implementors are encouraged to provide for detection of such errors wherever reasonable.

This is not to say that some particular implementation might not define the effects and results for such a situation; the point is that no program conforming to the COMMON LISP specification may correctly depend on such effects or results.

On the other hand, if it is specified in this manual that in some situation “an error is signalled” this means that:

- If this situation occurs, an error will be signalled (see `error` and `error`).
- Valid COMMON LISP programs may rely on the fact that an error will be signalled.
- Every COMMON LISP implementation is required to detect such an error.

Table 1-1: Sample Function Description

`sample-function arg1 arg2 &optional arg3 arg4` [Function]

The function `sample-function` adds together `arg1` and `arg2`, and then multiplies the result by `arg3`. If `arg3` is not provided or is `nil`, the multiplication isn't done. `sample-function` then returns a list whose first element is this result and whose second element is `arg4` (which defaults to the symbol `foo`). For example:

```
(sample-function 3 4) ⇒ (7 foo)
(sample-function 1 2 2 'bar) ⇒ (6 bar)
```

In general, `(sample-function x y) ≡ (list (+ x y) 'foo)`.

Table 1-2: Sample Variable Description

`*sample-variable*` [Variable]

The variable `*sample-variable*` specifies how many times the special form `sample-special-form` should iterate. The value should always be a non-negative integer or `nil` (which means iterate indefinitely many times). The initial value is 0.

Table 1-3: Sample Constant Description

`sample-constant` [Constant]

The named constant `sample-constant` has as its value the height of the terminal screen in furlongs times the base-2 logarithm of the implementation's total disk capacity in bytes, as a floating-point number.

In places where it is stated that so-and-so “must” or “must not” or “may not” be the case, then it “is an error” if the stated requirement is not met. For example, if an argument “must be a symbol,” then it “is an error” if the argument is not a symbol. In all cases where an error is to be *signalled*, the word “signalled” is always used explicitly in this manual.

Descriptions of Functions and Other Entities

Functions, variables, named constants, special forms, and macros are described using a distinctive typographical format. Definition 1-1 illustrates the manner in which COMMON LISP functions are documented. The first line specifies the name of the function, the manner in which it accepts arguments, and the fact that it is a function. If the function takes many arguments, then the names of the arguments may spill across two or three lines. The paragraphs following this standard header explain the definition and uses of the function and often present examples or related functions.

Table 1-4: Sample Special Form Description

sample-special-form [*name*] (*{var}**) (*form*)⁺ [*Special form*]

This evaluates each form in sequence as an implicit *progn*, and does this as many times as specified by the global variable **sample-variable**. Each variable *var* is bound and initialized to 43 before the first iteration, and unbound after the last iteration. The name *name*, if supplied, may be used in a return-from form to exit from the loop prematurely. If the loop ends normally, *sample-special-form* returns *nil*. For example:

```
(setq *sample-variable* 3)
(sample-special-form () form1 form2)
```

This evaluates *form1, form2, form1, form2, form1, form2* in that order.

Table 1-5: Sample Macro Description

sample-macro *var* {*tag* | *statement*}* [*Macro*]

This evaluates the statements as a *prog* body, with the variable *var* bound to 43.

```
(sample-macro x (return (+ x x))) ⇒ 86
(sample-macro var . body) → (prog (( var 43)) . body)
```

Sometimes two or more related functions are explained in a single combined description. In this situation the headers for all the functions appear together, followed by the combined description.

In general, actual code (including actual names of functions) appears in this typeface: (cons a b). Names that stand for pieces of code (metavariables) are written in *italics*. In a function description, the names of the parameters appear in italics for expository purposes. The word &optional in the list of parameters indicates that all arguments past that point are optional; the default values for the parameters are described in the text. Parameter lists may also contain &rest, indicating that an indefinite number of arguments may appear, or &key, indicating that keyword arguments are accepted. (The &optional/&rest/&key syntax is actually used in COMMON LISP function definitions for these purposes.)

Definition 1-2 illustrates the manner in which a global variable is documented. The first line specifies the name of the variable and the fact that it is a variable. Purely as a matter of convention, all global variables used by COMMON LISP have names beginning and ending with an asterisk.

Definition 1-3 illustrates the manner in which a named constant is documented. The first line specifies the name of the constant and the fact that it is a constant. (A constant is just like a global variable, except that it is an error ever to alter its value or to bind it to a new value.)

Definitions 1-4 and 1-5 illustrate the documentation of special forms and macros, which are closely related in purpose. These are very different from functions. Functions are called according to a single, specific, consistent syntax; the *&optional/&rest/&key* syntax specifies how the function uses its arguments internally, but does not affect the syntax of a call. In contrast, each special form or macro can have its own idiosyncratic syntax. It is by special forms and macros that the syntax of COMMON LISP is defined and extended.

In the description of a special form or macro, an italicized word names a corresponding part of the form that invokes the special form or macro. Parentheses stand for themselves, and should be written as such when invoking the special form or macro. Brackets, braces, stars, plus signs, and vertical bars are metasyntactic marks. Brackets, [and], indicate that what they enclose is optional (may appear zero times or one time in that place); the square brackets should not be written in code. Braces, { and }, simply parenthesize what they enclose, but may be followed by a star, *, or a plus sign, +; a star indicates that what the braces enclose may appear any number of times (including zero, that is, not at all), whereas a plus sign indicates that what the braces enclose may appear any non-zero number of times (that is, must appear at least once). Within braces or brackets, a vertical bar, |, separates mutually exclusive choices. In summary, the notation $\{x\}^*$ means zero or more occurrences of x , the notation $\{x\}^+$ means one or more occurrences of x , and the notation $[x]$ means zero or one occurrence of x . These notations are also used for syntactic descriptions expressed as BNF-like productions, as in Table 22-2.

In the last example in Definition 1-5, notice the use of dot notation. The dot appearing in the expression (*sample-macro var . body*) means that the name *body* stands for a list of forms, not just a single form, at the end of a list. This notation is often used in examples.

The Lisp Reader

The term “Lisp reader” refers not to you, the reader of this manual, nor to some person reading LISP code, but specifically to a LISP procedure, namely the function *read*, that reads characters from an input stream and interprets them by parsing as representations of LISP objects.

Overview of Syntax

Certain characters are used in special ways in the syntax of COMMON LISP. The complete syntax is explained in detail in Chapter 22, but a quick summary here may be useful:

- { A left parenthesis begins a list of items. The list may contain any number of items, including zero. Lists may be nested. For example, `(cons (car x) (cdr y))` is a list of three things, of which the last two are themselves lists.
- }
-) A right parenthesis ends a list of items.
- ' An acute accent (also called single quote or apostrophe) followed by an expression *form* is an abbreviation for `(quote form)`. Thus `'foo` means `(quote foo)` and `'(cons 'a 'b)` means `(quote (cons (quote a) (quote b)))`.

Chapter 2

Data Types

COMMON LISP provides a variety of types of data objects. It is important to note that in LISP it is data objects that are typed, not variables. Any variable can have any LISP object as its value. (It is possible to make an explicit declaration that a variable will in fact take on one of only a limited set of values. However, such a declaration may always be omitted, and the program will still run correctly. Such a declaration merely constitutes advice from the user that may be useful in gaining efficiency. See `declare`.)

In COMMON LISP, a data type is a set (possibly infinite) of LISP objects. Many LISP objects belong to more than one such set, and so it doesn't always make sense to ask what *the* type of an object is; instead, one usually asks only whether an object belongs to a given type. The predicate `typep` may be used to ask whether an object belongs to a given type, and the function `type-of` returns *a* type to which a given object belongs.

The data types defined in COMMON LISP are arranged into a hierarchy (actually a partial order) defined by the subset relationship. Certain sets of objects, such as the set of numbers or the set of strings, are interesting enough to deserve labels. Symbols are used for most such labels (here, and throughout this book, the word "symbol" refers to atomic symbols, one kind of LISP object, elsewhere known as literal atoms). See Chapter 4 for a complete description of type specifiers.

The set of all objects is specified by the symbol `t`. The empty data type, which contains no objects, is denoted by `nil`. A type called `common` encompasses all the data objects required by the COMMON LISP language. A COMMON LISP implementation is free to provide other data types that are not subtypes of `common`.

The following categories of COMMON LISP objects are of particular interest: numbers, characters, symbols, lists, arrays, structures, and functions. There are others as well. Some of these categories have many subdivisions. There are also standard types defined to be the union of two or more of these categories. The categories listed above, while they are data types, are neither more nor less "real" than other data types; they simply constitute a particularly useful slice across the type hierarchy for expository purposes.

Here are brief descriptions of various COMMON LISP data types. The remaining sections of this chapter go into more detail, and also describe notations for objects of each type. Descriptions of LISP functions that operate on data objects of each type appear in later chapters.

- *Numbers* are provided in various forms and representations. COMMON LISP provides a true integer data type: any integer, positive or negative, has in principle a representation as a COMMON LISP data object, subject only to total memory limitations (rather than machine word width). A true rational data type is provided: the quotient of two integers, if not an integer, is a ratio. Floating-point numbers of various ranges and precisions are also provided, as well as Cartesian complex numbers.

- *Characters* represent printed glyphs such as letters or text formatting operations. Strings are one-dimensional arrays of characters. COMMON LISP provides for a rich character set, including ways to represent characters of various type styles.
- *Symbols* (sometimes called *atomic symbols* for emphasis or clarity) are named data objects. LISP provides machinery for locating a symbol object, given its name (in the form of a string). Symbols have *property lists*, which in effect allow symbols to be treated as record structures with an extensible set of named components, each of which may be any LISP object. Symbols also serve to name functions and variables within programs.
- *Lists* are sequences represented in the form of linked cells called *conses*. There is a special object (the symbol `nil`) that is an empty list. All other lists are built recursively by adding a new element to the front of an existing list. This is done by creating a new *cons*, which is an object having two components called the *car* and the *cdr*. The *car* may hold anything, and the *cdr* is made to point to the previously existing list. (Conses may actually be used completely generally as two-element record structures, but their most important use is to represent lists.)
- *Arrays* are dimensioned collections of objects. An array can have any non-negative number of dimensions and is indexed by a sequence of integers. A general array can have any LISP object as a component; other types of arrays are specialized for efficiency and can hold only certain types of LISP objects. It is possible for two arrays, possibly with differing dimension information, to share the same set of elements (such that modifying one array modifies the other also) by causing one to be *displaced* to the other. One-dimensional arrays of any kind are called *vectors*. One-dimensional arrays of characters are called *strings*. One-dimensional arrays of bits (that is, of integers whose values are 0 or 1) are called *bit-vectors*.
- *Hash tables* provide an efficient way of mapping any LISP object (a *key*) to an associated object.
- *Readtables* are used to control the built-in expression parser `read`.
- *Packages* are collections of symbols that serve as name spaces. The parser recognizes symbols by looking up character sequences in the current package.
- *Pathnames* represent names of files in a fairly implementation-independent manner. They are used to interface to the external file system.
- *Streams* represent sources or sinks of data, typically characters or bytes. They are used to perform I/O, as well as for internal purposes such as parsing strings.
- *Random-states* are data structures used to encapsulate the state of the built-in random-number generator.
- *Structures* are user-defined record structures, objects that have named components. The `defstruct` facility is used to define new structure types. Some COMMON LISP implementations may choose to implement certain system-supplied data types, such as *bignums*, *readtables*, *streams*, *hash tables*, and *pathnames*, as structures, but this fact will be invisible to the user.
- *Functions* are objects that can be invoked as procedures; these may take arguments and return values. (All LISP procedures can be construed to return values and therefore every procedure is a function.) Such objects include *compiled-functions* (compiled code objects). Some functions are represented as a list whose *car* is a particular symbol such as `lambda`. Symbols may also be used as functions.

These categories are not always mutually exclusive. The required relationships among the various data types are explained in more detail in "Overlap, Inclusion, and Disjointness of Types."

Numbers

Several kinds of numbers are defined in COMMON LISP. They are divided into *integers*, *ratios*, *floating-point numbers*, with names provided for up to four different floating-point representations; and *complex numbers*.

Integers

The *integer* data type is intended to represent mathematical integers. Unlike most programming languages, COMMON LISP in principle imposes no limit on the magnitude of an integer; storage is automatically allocated as necessary to represent large integers.

In every COMMON LISP implementation there is a range of integers that are represented more efficiently than others; each such integer is called a *fixnum*, and an integer that is not a fixnum is called a *bignum*. COMMON LISP is designed to hide this distinction as much as possible; the distinction between fixnums and bignums is visible to the user in only a few places where the efficiency of representation is important. Exactly which integers are fixnums is implementation-dependent; typically they will be those integers in the range -2^n to $2^n - 1$, inclusive, for some n not less than 15. See *most-positive-fixnum* and *most-negative-fixnum*.

Integers are ordinarily written in decimal notation, as a sequence of decimal digits, optionally preceded by a sign and optionally followed by a decimal point. For example:

0	;Zero
-0	;This <i>always</i> means the same as 0
+6	;The first perfect number
28	;The second perfect number
1024.	;Two to the tenth power
-1	;e ^{π} i
15511210043330985984000000.	;25 factorial (25!), probably a bignum

Compatibility note: MACLISP and ZETALISP normally assume that integers are written in octal (radix-8) notation unless a decimal point is present. INTERLISP assumes integers are written in decimal notation and uses a trailing Q to indicate octal radix; however, a decimal point, even in trailing position, *always* indicates a floating-point number. This is of course consistent with FORTRAN. ADA does not permit trailing decimal points, but instead requires them to be embedded. In COMMON LISP, integers written as described above are always construed to be in decimal notation, whether or not the decimal point is present; allowing the decimal point to be present permits compatibility with MACLISP.

Integers may be notated in radices other than ten. The notation

`#nnrdddd` or `#nnRdddd`

means the integer in radix-*nn* notation denoted by the digits *dddd*. More precisely, one may write #, a non-empty sequence of decimal digits representing an unsigned decimal integer *n*, *r* (or *R*), an optional sign, and a sequence of radix-*n* digits, to indicate an integer written in radix *n* (which must be between 2 and 36, inclusive). Only legal digits for the specified radix may be used; for example, an octal number may contain only the digits 0 through 7. For digits above 9, letters of the alphabet of either case may be used in order. Binary, octal, and hexadecimal radices are useful enough to warrant the special abbreviations #b for #2r, #o for #8r, and #x for #16r. For example:

```
#2r11010101 ;Another way of writing 213 decimal
#b11010101   ;Ditto
#b+11010101  ;Ditto
#o325        ;Ditto, in octal radix
#xD5         ;Ditto, in hexadecimal radix
#16r+D5      ;Ditto
#o-300       ;Decimal -192, written in base 8
#3r-21010    ;Same thing in base 3
#25R-7H      ;Same thing in base 25
#xACCDEDED   ;181202413, in hexadecimal radix
```

Ratios

A ratio is a number representing the mathematical ratio of two integers. Integers and ratios collectively constitute the type *rational*. The canonical representation of a rational number is as an integer if its value is integral, and otherwise as the ratio of two integers, the *numerator* and *denominator*, whose greatest common divisor is one, and of which the denominator is positive (and in fact greater than 1, or else the value would be integral). A ratio is notated with / as a separator, thus: 3/5. It is possible to notate ratios in non-canonical (unreduced) forms, such as 4/6, but the LISP function `print` always prints the canonical form for a ratio.

If any computation produces a result that is a ratio of two integers such that the denominator evenly divides the numerator, then the result is immediately converted to the equivalent integer. This is called the rule of *rational canonicalization*.

Rational numbers may be written as the possibly signed quotient of decimal numerals: an optional sign followed by two non-empty sequences of digits separated by a /. This syntax may be described as follows:

ratio ::= [*sign*] {*digit*}+ / {*digit*}+

The second sequence may not consist entirely of zeros. For example:

```
2/3           ;This is in canonical form
4/6           ;A non-canonical form for the same number
-17/23        ;A not very interesting ratio
-30517578125/32768 ;This is (-5/2)15
10/5          ;The canonical form for this is 2
```

To notate rational numbers in radices other than ten, one uses the same radix specifiers (one of #nnR, #O, #B, or #X) as for integers. For example:

```
#o-101/75           ;Octal notation for -65/61
#3r120/21          ;Ternary notation for 15/7
#Xbc/ad            ;Hexadecimal notation for 188/173
#xFADED/FACADE    ;Hexadecimal notation for 1027565/16435934
```

Floating-point Numbers

COMMON LISP allows an implementation to provide one or more kinds of floating-point number, which collectively make up the type `float`. A floating-point number is a (mathematical) rational number of the form $s \cdot f \cdot b^{e-p}$, where s is $+1$ or -1 , the *sign*; b is an integer greater than 1, the *base* or *radix* of the representation; p is a positive integer, the *precision* (in base- b digits) of the floating-point number; f is a positive integer between b^{p-1} and $b^p - 1$ (inclusive), the *significand*; and e is an integer, the *exponent*. The value of p and the range of e depend on the implementation and on the type of floating-point number within that implementation. In addition, there is a floating-point zero; depending on the implementation, there may also be a “minus zero.” If there is no minus zero, then 0.0 and -0.0 are both interpreted as simply a floating-point zero.

Implementation note: The form of the above description should not be construed to require the internal representation to be in sign-magnitude form. Two’s-complement and other representations are also acceptable. Note that the radix of the internal representation may be other than 2, as on the IBM 360 and 370, which use radix 16; see `float-radix`.

Floating-point numbers may be provided in a variety of precisions and sizes, depending on the implementation. High-quality floating-point software tends to depend critically on the precise nature of the floating-point arithmetic, and so may not always be completely portable. To aid in writing programs that are moderately portable, however, certain definitions are made here:

- A *short* floating-point number (type `short-float`) is of the representation of smallest fixed precision provided by an implementation.
- A *long* floating-point number (type `long-float`) is of the representation of the largest fixed precision provided by an implementation.
- Intermediate between short and long formats are two others, arbitrarily called *single* and *double* (types `single-float` and `double-float`).

Table 2-1: Recommended Minimum Floating-Point Precision and Exponent Size

<i>Format</i>	<i>Minimum Precision</i>	<i>Minimum Exponent Size</i>
Short	13 bits	5 bits
Single	24 bits	8 bits
Double	50 bits	8 bits
Long	50 bits	8 bits

The precise definition of these categories is implementation-dependent. However, the rough intent is that short floating-point numbers be precise to at least four decimal places or so (but also have a space-efficient representation); single floating-point numbers, to at least seven decimal places; and double floating-point numbers, to at least fourteen decimal places. It is suggested that the precision (measured in "bits," computed as $p \log_2 b$) and the exponent size (also measured in "bits," computed as the base-2 logarithm of one plus the maximum exponent value) be at least as great as the values in Table 2-1.

Floating-point numbers are written in either decimal fraction or computerized scientific notation: an optional sign, then a non-empty sequence of digits with an embedded decimal point, then an optional decimal exponent specification. If there is no exponent specifier, then the decimal point is required, and there must be digits after it. The exponent specifier consists of an exponent marker, an optional sign, and a non-empty sequence of digits. For preciseness, here is a modified-BNF description of floating-point notation.

```
floating-point-number ::= [sign] {digit}* decimal-point {digit}+ [exponent]
                        | [sign] {digit}+ [decimal-point{digit}*] exponent
sign ::= + | -
decimal-point ::= .
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
exponent ::= exponent-marker [sign] {digit}+
exponent-marker ::= e | s | f | d | l | E | S | F | D | L
```

If no exponent specifier is present, or if the exponent marker *e* (or *E*) is used, then the precise format to be used is not specified. When such a representation is read and converted to an internal floating-point data object, the format specified by the variable `*read-default-float-format*` is used; the initial value of this variable is `single-float`.

The letters *s*, *f*, *d*, and *l* (or their respective uppercase equivalents) explicitly specify the use of *short*, *single*, *double*, and *long* format, respectively.

Examples of floating-point numbers:

0.0	;Floating-point zero in default format
0E0	;Also floating-point zero in default format
-.0	;This may be a zero or a minus zero, depending ; on the implementation
0.	;The <i>integer</i> zero, not a floating-point zero!
0.0s0	;A floating-point zero in <i>short</i> format
0s0	;Also a floating-point zero in <i>short</i> format
3.1415926535897932384d0	;A <i>double</i> -format approximation to π
6.02E + 23	;Avogadro's number, in default format
602E + 21	;Also Avogadro's number, in default format
3.1010299957f-1	;log ₁₀ 2, in <i>single</i> format
-0.000000001s9	;e ^{πi} in <i>short</i> format, the hard way

The internal format used for an external representation depends only on the exponent marker, and not on the number of decimal digits in the external representation.

While COMMON LISP provides terminology and notation sufficient to accommodate four distinct floating-point formats, not all implementations will have the means to support that many distinct formats. An implementation is therefore permitted to provide fewer than four distinct internal floating-point formats, in which case at least one of them will be "shared" by more than one of the external format names *short*, *single*, *double*, and *long* according to the following rules:

- If one internal format is provided, then it is considered to be *single*, but serves also as *short*, *double*, and *long*. The data types short-float, single-float, double-float, and long-float are considered to be identical. An expression such as (eql 1.0s0 1.0d0) will be true in such an implementation because the two numbers 1.0s0 and 1.0d0 will be converted into the same internal format and therefore be considered to have the same data type, despite the differing external syntax. Similarly, (typep 1.0L0 'short-float) will be true in such an implementation. For output purposes all floating-point numbers are assumed to be of *single* format, and so will print using the exponent letter E or F.
- If two internal formats are provided, then either of two correspondences may be used, depending on which is the more appropriate:
 - One format is *short*; the other is *single* and serves also as *double* and *long*. The data types single-float, double-float, and long-float are considered to be identical, but short-float is distinct. An expression such as (eql 1.0s0 1.0d0) will be false, but (eql 1.0f0 1.0d0) will be true. Similarly, (typep 1.0L0 'short-float) will be false, but (typep 1.0L0 'single-float) will be true. For output purposes all floating-point numbers are assumed to be of *short* or *single* format.
 - One format is *single* and serves also as *short*; the other is *double* and serves also as *long*. The data types short-float and single-float are considered to be identical, and the data types double-float and long-float are considered to be identical. An expression such as (eql 1.0s0 1.0d0) will be false, as will (eql 1.0f0 1.0d0); but (eql 1.0d0 1.0L0) will be true. Similarly, (typep 1.0L0 'short-float) will be false, but (typep 1.0L0 'double-float) will be true. For output purposes all floating-point numbers are assumed to be of *single* or *double* format.
- If three internal formats are provided, then either of two correspondences may be used, depending on which is the more appropriate:
 - One format is *short*; another format is *single*; and the third format is *double* and serves also as *long*. Similar constraints apply.
 - One format is *single* and serves also as *short*; another is *double*; and the third format is *long*.

Implementation note: It is recommended that an implementation provide as many distinct floating-point formats as feasible, given Table 2-1 as a guideline. Ideally, short-format floating-point numbers should have an "immediate" representation that does not require heap allocation; single-format floating-point numbers should approximate IEEE proposed standard single-format floating-point numbers; and double-format floating-point numbers should approximate IEEE proposed standard double-format floating-point numbers [9, 5, 6].

Complex Numbers

Complex numbers (type `complex`) are represented in Cartesian form, with a real part and an imaginary part each of which is a non-complex number (integer, ratio, or floating-point number). It should be emphasized that the parts of a complex number are not necessarily floating-point numbers; in this, COMMON LISP is like PL/I and differs from FORTRAN. However, both parts must be of the same type: either both are rational, or both are of the same floating-point format.

Complex numbers may be notated by writing the characters `#C` followed by a list of the real and imaginary parts. If the two parts as notated are not of the same type, then they are converted according to the rules of floating-point contagion as described in Chapter 12. (Indeed, `#C(a b)` is equivalent to `#(complex a b)`; see the description of the function `complex`.) For example:

```
#C{3.0s1 2.0s-1)
#C{5 -3)           ;A Gaussian integer
#C{5:3 7.0)       ;Will be converted internally to ; #C{1.66666 7.0)
#C{0 1)           ;The imaginary unit, that is, i
```

The type of a specific complex number is indicated by a list of the word `complex` and the type of the components; for example, a specialized representation for complex numbers with short floating-point parts would be of type `(complex short-float)`. The type `complex` encompasses all complex representations.

A complex number of type `(complex rational)`, that is, one whose components are rational, can never have a zero imaginary part. If the result of any computation would be a complex rational with a zero imaginary part, the result is immediately converted to a non-complex rational number by taking the real part. This is called the rule of *complex canonicalization*. This rule does not apply to complex numbers whose parts are floating-point numbers: `#C{5.0 0.0)` and `5.0` are different.

Characters

Characters are represented as data objects of type `character`. There are two subtypes of interest, called `standard-char` and `string-char`.

A character object can be notated by writing `#\` followed by the character itself. For example, `#\g` means the character object for a lowercase `g`. This works well enough for printing characters. Non-printing characters have names, and can be notated by writing `#\` and then the name; for example, `#\Space` (or `#\SPACE` or `#\space` or `#\SPaCE`) means the space character. The syntax for character names after `#\` is the same as that for symbols. However, only character names that are known to the particular implementation may be used.

Standard Characters

COMMON LISP defines a “standard character set” (subtype `standard-char`) for two purposes. COMMON LISP programs that are *written in* the standard character set can be read by any COMMON LISP implementation; and COMMON LISP programs that *use* only standard characters as data objects are most likely to be portable. The COMMON LISP character set consists of a space character `#\Space`, a newline character `#\Newline`, and the following ninety-four non-blank printing characters or their equivalents:

```
! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _
` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~
```

The COMMON LISP standard character set is apparently equivalent to the ninety-five standard ASCII printing characters plus a newline character. Nevertheless, COMMON LISP is designed to be relatively independent of the ASCII character encoding. For example, the collating sequence is not specified except to say that digits must be properly ordered, the uppercase letters must be properly ordered, and the lowercase letters must be properly ordered (see `char<` for a precise specification). Other character encodings, particularly EBCDIC, should be easily accommodated (with a suitable mapping of printing characters).

Of the ninety-four non-blank printing characters, the following are used in only limited ways in the syntax of COMMON LISP programs:

[] { } ? ! ^ _ ` \$ %

All of these characters except `!` and `_` are used within `format` strings as formatting directives. Except for this, `[`, `]`, `{`, `}`, `?`, and `!` are not used in COMMON LISP and are reserved to the user for syntactic extensions; `^` and `_` are not yet used in COMMON LISP, but are part of the syntax of reserved tokens, and are reserved to implementors; ``` is not yet used in COMMON LISP, and is reserved to implementors; and `$` and `%` are normally regarded as alphabetic characters, but are not used in the names of any standard COMMON LISP functions, variables, or other entities.

The following characters are called *semi-standard*:

`#\Backspace` `#\Tab` `#\Linefeed` `#\Page` `#\Return` `#\Rubout`

Not all implementations of COMMON LISP need to support them; but those implementations that use the standard ASCII character set should support them, treating them as corresponding respectively to the ASCII characters BS (octal code 010), HT (011), LF (012), FF (014), CR (015), and DEL (177). These characters are not members of the subtype `standard-char` unless synonymous with one of the standard characters specified above. For example, in a given implementation it might be sensible for the implementor to define `#\Linefeed` or `#\Return` to be synonymous with `#\Newline`, or `#\Tab` to be synonymous with `#\Space`.

Line Divisions

The treatment of line divisions is one of the most difficult issues in designing portable software, simply because there is so little agreement among operating systems. Some use a single character to delimit lines; the recommended ASCII character for this purpose is the line feed character LF (also called the new line character, NL) but some systems use the carriage return character CR. Much more common is the two-character sequence CR followed by LF. Frequently line divisions have no representation as a character but are implicit in the structuring of a file into records, each record containing a line of text. A deck of punched cards has this structure, for example.

COMMON LISP provides an abstract interface by requiring that there be a single character, `#\Newline`, that within the language serves as a line delimiter. (The language C has a similar requirement.) An implementation of COMMON LISP must translate between this internal single-character representation and whatever external representation(s) may be used.

Implementation note: How the character called `#\Newline` is represented internally is not specified here, but it is strongly suggested that the ASCII LF character be used in COMMON LISP implementations that use the ASCII character encoding. The ASCII CR character is a workable, but in most cases inferior, alternative.

The requirement that a line division be represented as a single character has certain consequences. A character string written in the middle of a program in such a way as to span more than one line must contain exactly one character to represent each line division. Consider this code fragment:

```
(setq a-string "This string
contains
forty-two characters.")
```

Between `g` and `c` there must be exactly one character, `#\Newline`; a two-character sequence, such as `#\Return` and then `#\Newline`, is not acceptable, nor is the absence of a character. The same is true between `s` and `f`.

When the character `#\Newline` is written to an output file, the COMMON LISP implementation must take the appropriate action to produce a line division. This might involve writing out a record or translating `#\Newline` to a CR/LF sequence.

Implementation note: If an implementation uses the ASCII character encoding, uses the CR/LF sequence externally to delimit lines, uses LF to represent `#\Newline` internally, and supports `#\Return` as a data object corresponding to the ASCII character CR, the question arises as to what action to take when the program writes out `#\Return` followed by `#\Newline`. It should first be noted that `#\Return` is not a standard COMMON LISP character, and the action to be taken when `#\Return` is written out is therefore not defined by the COMMON LISP language. A plausible approach is to buffer the `#\Return` character, and suppress it if and only if the next character is `#\Newline` (the net effect is to generate a CR/LF sequence). Another plausible approach is simply to ignore the difficulty and declare that writing `#\Return` and then `#\Newline` results in the sequence CR/CR/LF in the output.

Non-standard Characters

Any implementation may provide additional characters, whether printing characters or named characters. Some plausible examples:

```
#\pi #\alpha #\Break #\Home-Up #\Escape
```

The use of such characters may render COMMON LISP programs non-portable.

Character Attributes

Every object of type `character` has three attributes: *code*, *bits*, and *font*. The *code* attribute is intended to distinguish among the printed glyphs and formatting functions for characters; it is a numerical encoding of the character proper. The *bits* attribute allows extra flags to be associated with a character. The *font* attribute permits a specification of the style of the glyphs (such as italics). Each of these attributes may be understood to be a non-negative integer.

The *font* attribute may be notated in unsigned decimal notation between the `#` and the `\`. For example, `#3\σ` means the letter `σ` in font 3. This might mean the same thing as `#\α` if font 3 were used to represent Greek letters. Note that not all COMMON LISP implementations provide for non-zero font attributes; see `char-font-limit`.

The *bits* attribute may be notated by preceding the name of the character by the names or initials of the bits, separated by hyphens. The character itself may be written instead of the name, preceded if necessary by `\`. For example:

<code>#\Control-Meta-Return</code>	<code>#\Meta-Control-Q</code>
<code>#\Hyper-Space</code>	<code>#\Meta-\α</code>
<code>#\Control-A</code>	<code>#\Meta-Hyper-\:</code>
<code>#\C-M-Return</code>	<code>#\Hyper-\π</code>

Note that not all COMMON LISP implementations provide for non-zero bits attributes; see `char-bits-limit`.

String Characters

Any character whose bits and font attributes are zero may be contained in strings. All such characters together constitute a subtype of the characters; this subtype is called `string-char`.

Symbols

Symbols are LISP data objects that serve purposes and have several interesting characteristics. Every object of type `symbol` has a name, called its *print name*. Given a symbol, one can obtain its name in the form of a string. Conversely, given the name of a symbol as a string one can obtain the symbol itself. (More precisely, symbols are organized into *packages*, and all the symbols in a package are uniquely identified by name. See Chapter 11.)

Symbols have a component called the *property list*, or *plist*. By convention this is always a list whose even-numbered components (calling the first component zero) are symbols, here functioning as property names, and whose odd-numbered components are associated property values. Functions are provided for manipulating this property list; in effect, these allow a symbol to be treated as an extensible record structure.

Symbols are also used to represent certain kinds of variables in LISP programs, and there are functions for dealing with the values associated with symbols in this role.

A symbol can be notated simply by writing its name. If its name is not empty, and if the name consists only of uppercase alphabetic, numeric, or certain "pseudo-alphabetic" special characters (but not delimiter characters such as parentheses or space), and if the name of the symbol cannot be mistaken for a number, then the symbol can be notated by the sequence of characters in its name. Any uppercase letters that appear in the (internal) name may be written in either case in the external notation (more on this below). For example:

```
FROBBOZ           ;The symbol whose name is FROBBOZ
frobboz          ;Another way to notate the same symbol
fRObBoz         ;Yet another way to notate it
unwind-protect  ;A symbol with a - in its name
+$              ;The symbol named +$
1+              ;The symbol named 1+
+1              ;This is the integer 1, not a symbol.
pascal_style    ;This symbol has an underscore in its name.
b^2-4*a*c      ;This is a single symbol!
                ; It has several special characters in its name.
file.rel.43     ;This symbol has periods in its name.
/usr/games/zork ;This symbol has slashes in its name.
```

In addition to letters and numbers, the following characters are normally considered to be "alphabetic" for the purposes of notating symbols:

```
+ - * / @ $ % ^ & _ \ < > `
```

Some of these characters have conventional purposes for naming things; for example, symbols that name special variables generally have names beginning and ending with *. The last character listed above, the period, is considered alphabetic *provided* that a token does not consist entirely of periods. A single period standing by itself is used in the notation of conses and dotted lists; a token consisting of two or more periods is syntactically illegal. (The period also serves as the decimal point in the notation of numbers.)

The following characters are also alphabetic by default, but are explicitly reserved to the user for definition as reader macro characters (see "Macro Characters," Chapter 22) or any other desired purpose, and therefore should not be used routinely in names of symbols:

```
? ! [ ] { }
```

A symbol may have uppercase letters, lowercase letters, or both in its print name. However, the LISP reader normally converts lowercase letters to the corresponding uppercase letters when reading symbols. The net effect is that most of the time case makes no difference when *notating* symbols. Case *does* make a difference internally and when printing a symbol. Internally the symbols that name all standard COMMON LISP functions, variables, and keywords have uppercase names; their names appear in lower case in this manual for readability. Typing such names with lowercase letters works because the function `read` will convert lowercase letters to the equivalent uppercase letters.

If a symbol cannot be simply notated by the characters of its name because the (internal) name contains special characters or lowercase letters, then there are two "escape" conventions for notating them. Writing a \ character before any character causes the character to be treated itself as an ordinary character for use in a symbol name; in particular, it suppresses internal conversion of lowercase letters to their uppercase equivalents. If any character in a notation is preceded by \, then that notation can never be interpreted as a number. For example:

<code>\{</code>	;The symbol whose name is {
<code>\+1</code>	;The symbol whose name is +1
<code>+ \1</code>	;Also the symbol whose name is +1
<code>\frobboz</code>	;The symbol whose name is fROBBOZ
<code>3.14159265\s0</code>	;The symbol whose name is 3.14159265s0
<code>3.14159265\s0</code>	;A different symbol, whose name is 3.14159265S0
<code>3.14159265s0</code>	;A short-format floating-point approximation to π
<code>APL\360</code>	;The symbol whose name is APL\360
<code>apl\360</code>	;Also the symbol whose name is APL\360
<code>\(b^2)\ -\ 4*a+c</code>	;The name is (B^2) - 4*A+C.
	; It has parentheses and two spaces in it.
<code>\(\b^2)\ -\ 4*\a*\c.</code>	;The name is (b^2) - 4*a*c.
	; The letters are explicitly lowercase.

It may be tedious to insert a `\` before *every* delimiter character in the name of a symbol if there are many of them. An alternative convention is to surround the name of a symbol with vertical bars; these cause every character between them to be taken as part of the symbol's name, as if `\` had been written before each one, excepting only `|` itself and `\`, which must nevertheless be preceded by `\`. For example:

<code> "</code>	;The same as writing <code>\"</code>
<code> {(b^2) - 4*a+c </code>	;The name is (b^2) - 4*a*c
<code> frobboz </code>	;The name is frobboz, not FROBBOZ
<code> APL\360 </code>	;The name is APL360, because
	; the <code>\</code> quotes the 3
<code> APL\360 </code>	;The name is APL\360
<code> apl\360 </code>	;The name is apl\360
<code> \ \ </code>	;Same as <code>\ \ </code> ; the name is <code> </code>
<code> {(B^2) - 4*A*C </code>	;The name is (B^2) - 4*A*C.
	; It has parentheses and two spaces in it.
<code> {(b^2) - 4*a*c </code>	;The name is (b^2) - 4*a*c.

Lists and Conses

A cons is a record structure containing two components called the *car* and the *cdr*. Conses are used primarily to represent lists.

A *list* is recursively defined to be either the empty list or a cons whose *cdr* component is a list. A list is therefore a chain of conses linked by their *cdr* components and terminated by `nil`, the empty list. The *car* components of the conses are called the *elements* of the list. For each element of the list there is a cons. The empty list has no elements at all.

A list is notated by writing the elements of the list in order, separated by blank space (space, tab, or return characters), and surrounded by parentheses. For example:

<code>(a b c)</code>	;A list of three symbols
<code>(2.0s0 (a 1) #*)</code>	;A list of three things: a short floating-point
	; number, another list, and a character object

The empty list `nil` therefore can be written as `()`, because it is a list with no elements.

A *dotted list* is one whose last cons does not have `nil` for its *cdr*, rather some other data object (which is also not a cons, or the first-mentioned cons would not be the last cons of the list). Such a list is called "dotted" because of the special notation used for it: the elements of the list are written between parentheses as before, but after the last element and before the right parenthesis are written a dot (surrounded by blank space) and then the *cdr* of the last cons. As a special case, a single cons is notated by writing the *car* and the *cdr* between parentheses and separated by a space-surrounded dot. For example:

```
(a . 4)           ;A cons whose car is a symbol
                  ; and whose cdr is an integer
(a b c . d)      ;A dotted list with three elements whose last cons
                  ; has the symbol d in its cdr
```

Compatibility note: In `MACLISP`, the dot in dotted-list notation need not be surrounded by white space or other delimiters. The dot is required to be delimited in `COMMON LISP`, as in `ZETALISP`.

It is legitimate to write something like `(a b . (c d))`; this means the same as `(a b c d)`. The standard `LISP` output routines will never print a list in the first form, however; they will avoid dot notation wherever possible.

Often the term *list* is used to refer either to true lists or to dotted lists. When the distinction is important, the term "true list" will be used to refer to a list terminated by `nil`. Most functions advertised to operate on lists expect to be given true lists. Throughout this manual, unless otherwise specified, it is an error to pass a dotted list to a function that is specified to require a list as an argument.

Implementation note: Implementors are encouraged to use the equivalent of the predicate `endp` wherever it is necessary to test for the end of a list. Whenever feasible, this test should explicitly signal an error if a list is found to be terminated by a non-`nil` atom. However, such an explicit error signal is not required, because some such tests occur in important loops where efficiency is important. In such cases, the predicate `atom` may be used to test for the end of the list, quietly treating any non-`nil` list-terminating atom as if it were `nil`.

Sometimes the term *tree* is used to refer to some cons and all the other conses transitively accessible to it through *car* and *cdr* links until non-conses are reached; these non-conses are called the *leaves* of the tree.

Lists, dotted lists, and trees are not mutually exclusive data types; they are simply useful points of view about structures of conses. There are yet other terms, such as *association list*. None of these are true `LISP` data types. Conses are a data type, and `nil` is the sole object of type null. The `LISP` data type `list` is taken to mean the union of the cons and null data types, and therefore encompasses both true lists and dotted lists.

Arrays

An array is an object with components arranged according to a Cartesian coordinate system. In general, these components may be any `LISP` data objects.

The number of dimensions of an array is called its *rank* (this terminology is borrowed from APL); the rank is a non-negative integer. Likewise, each dimension is itself a non-negative integer. The total number of elements in the array is the product of all the dimensions.

An implementation of COMMON LISP may impose a limit on the rank of an array, but this limit may not be smaller than 7. Therefore, any COMMON LISP program may assume the use of arrays of rank 7 or less. (A program may determine the actual limit on array ranks for a given implementation by examining the constant `array-rank-limit`.)

It is permissible for a dimension to be zero. In this case, the array has no elements, and any attempt to access an element is in error. However, other properties of the array, such as the dimensions themselves, may be used. If the rank is zero, then there are no dimensions, and the product of the dimensions is then by definition 1. A zero-rank array therefore has a single element.

An array element is specified by a sequence of indices. The length of the sequence must equal the rank of the array. Each index must be a non-negative integer strictly less than the corresponding array dimension. Array indexing is therefore zero-origin, not one-origin as in (the default case of) FORTRAN.

As an example, suppose that the variable `foo` names a 3-by-5 array. Then the first index may be 0, 1, or 2, and the second index may be 0, 1, 2, 3, or 4. One may refer to array elements using the function `aref`; for example, `(aref foo 2 1)` refers to element (2, 1) of the array. Note that `aref` takes a variable number of arguments: an array, and as many indices as the array has dimensions. A zero-rank array has no dimensions, and therefore `aref` would take such an array and no indices, and return the sole element of the array.

In general, arrays can be multidimensional, can share their contents with other array objects, and can have their size altered dynamically (either enlarging or shrinking) after creation. A one-dimensional array may also have a *fill pointer*.

Multidimensional arrays store their components in row-major order; that is, internally a multidimensional array is stored as a one-dimensional array, with the multidimensional index sets ordered lexicographically, last index varying fastest. This is important in two situations: (1) when arrays with different dimensions share their contents, and (2) when accessing very large arrays in a virtual-memory implementation. (The first situation is a matter of semantics; the second, a matter of efficiency.)

An array that is not displaced to another array, has no fill pointer, and is not to have its size adjusted dynamically after creation is called a *simple* array. The user may provide declarations that certain arrays will be simple. Some implementations can handle simple arrays in an especially efficient manner; for example, simple arrays may have a more compact representation than non-simple arrays.

Vectors

One-dimensional arrays are called *vectors* in COMMON LISP and constitute the type `vector` (which is therefore a subtype of `array`). Vectors and lists are collectively considered to be *sequences*. They differ in that any component of a one-dimensional array can be accessed in constant time, whereas the average component access time for a list is linear in the length of the list; on the other hand, adding a new element to the front of a list takes constant time, whereas the same operation on an array takes time linear in the length of the array.

A general vector (a one-dimensional array that can have any data object as an element, but has no additional paraphernalia) can be notated by notating the components in order, separated by whitespace and surrounded by # (and). For example:

```
#(a b c)                ;A vector of length 3
#(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47)
                        ;A vector containing the primes below 50
#()                    ;An empty vector
```

Note that when the function `read` parses this syntax, it always constructs a *simple* general vector.

Rationale: Many people have suggested that brackets be used to notate vectors, as `[a b c]` instead of `#(a b c)`. This notation would be shorter, perhaps more readable, and certainly in accord with cultural conventions in other parts of computer science and mathematics. However, to preserve the usefulness of the user-definable macro-character feature of the function `read`, it is necessary to leave some characters to the user for this purpose. Experience in `MACLISP` has shown that users, implementors of languages for use in artificial intelligence research, often want to define special kinds of brackets. Therefore `COMMON LISP` avoids using brackets and braces for any syntactic purpose.

Implementations may provide certain specialized representations of arrays for efficiency in the case where all the components are of the same specialized (typically numeric) type. All implementations provide specialized arrays for the cases when the components are characters (or rather, a special subset of the characters); the one-dimensional instances of this specialization are called *strings*. All implementations are also required to provide specialized arrays of bits, that is, arrays of type `(array bit)`; the one-dimensional instances of this specialization are called *bit-vectors*.

Strings

A string is simply a vector of characters. More precisely, a string is a specialized vector whose elements are of type `string-char`. The type `string` is therefore a subtype of the type `vector`. A string can be written as the sequence of characters contained in the string, preceded and followed by a " (double quote) character. Any " or \ character in the sequence must additionally have a \ character before it. For example:

```
"Foo"                ;A string with three characters in it
""                   ;An empty string
"\\"APL\\360?\" he cried." ;A string with twenty characters
"|x| = |-x|"         ;A ten-character string
```

Notice that any vertical bar `|` in a string need not be preceded by a `\`. Similarly, any double quote in the name of a symbol written using vertical-bar notation need not be preceded by a `\`. The double-quote and vertical-bar notations are similar but distinct: double quotes indicate a character string containing the sequence of characters, whereas vertical bars indicate a symbol whose name is the contained sequence of characters.

The characters contained by the double quotes, taken from left to right, occupy locations within the string with increasing indices. The leftmost character is string element number 0, the next one is element number 1, and so on.

Note that the function `print` will print any character vector (not just a simple one) using this syntax, but the function `read` will always construct a simple string when it reads this syntax.

Bit-Vectors

A bit-vector can be written as the sequence of bits contained in the string, preceded by `#*`; any delimiter character, such as whitespace, will terminate the bit-vector syntax. For example:

```
#*10110           ;A five-bit bit-vector; bit 0 is a 1
#*                ;An empty bit-vector
```

The bits notated following the `#*`, taken from left to right, occupy locations within the bit-vector with increasing indices. The leftmost notated bit is bit-vector element number 0, the next one is element number 1, and so on.

The function `print` will print any bit-vector (not just a simple one) using this syntax, but the function `read` will always construct a simple bit-vector when it reads this syntax.

Hash Tables

Hash tables provide an efficient way of mapping any LISP object (a *key*) to an associated object. They are provided as primitives of COMMON LISP because some implementations may need to use internal storage management strategies that would make it very difficult for the user to implement hash tables himself in a portable fashion. Hash tables are described in Chapter 16.

Readtables

A readtable is a data structure that maps characters into syntax types for the LISP expression parser. In particular, a readtable indicates for each character with syntax *macro character* what its macro definition is. This is a mechanism by which the user may reprogram the parser to a limited but useful extent. See "The Read Table," Chapter 22.

Packages

Packages are collections of symbols that serve as name spaces. The parser recognizes symbols by looking up character sequences in the current package. Packages can be used to hide names internal to a module from other code. Mechanisms are provided for exporting symbols from a given package to the primary "user" package. See Chapter 11.

Pathnames

Pathnames are the means by which a COMMON LISP program can interface to an external file system in a reasonably implementation-independent manner. See "Pathnames," Chapter 23.

Streams

A stream is a source or sink of data, typically characters or bytes. Nearly all functions that perform I/O do so with respect to a specified stream. The function `open` takes a pathname and returns a stream connected to the file specified by the pathname. There are a number of standard streams that are used by default for various purposes. See Chapter 21.

Random-States

An object of type `random-state` is used to encapsulate state information used by the pseudo-random number generator. For more information about `random-state` objects, see "Random Numbers," Chapter 12.

Structures

Structures are instances of user-defined data types that have a fixed number of named components. They are analogous to records in PASCAL. Structures are declared using the `defstruct` construct; `defstruct` automatically defines access and constructor functions for the new data type.

Different structures may print out in different ways; the definition of a structure type may specify a print procedure to use for objects of that type (see the `:print-function` option to `defstruct`). The default notation for structures is:

```
#S(structure-name
    slot-name-1 slot-value-1
    slot-name-2 slot-value-2
    . . . )
```

where `#S` indicates structure syntax, *structure-name* is the name (a symbol) of the structure type, each *slot-name* is the name (also a symbol) of a component, and each corresponding *slot-value* is the representation of the LISP object in that slot.

Functions

A *function* is anything that may be correctly given to the `funcall` or `apply` function, and is to be executed as code when arguments are supplied

A *compiled-function* is a compiled code object.

A lambda-expression (a list whose *car* is the symbol `lambda`) may serve as a function. Depending on the implementation, it may be possible for other lists to serve as functions. For example, an implementation might choose to represent a "lexical closure" as a list whose *car* contains some special marker.

A symbol may serve as a function; an attempt to invoke a symbol as a function causes the contents of the symbol's function cell to be used. See `symbol-function` and `defun`.

The result of evaluating a function special form will always be a function.

Unreadable Data Objects

Some objects may print in implementation-dependent ways. Such objects cannot necessarily be reliably reconstructed from a printed representation, and so they are usually printed in a format informative to the user but not acceptable to the `read` function:

```
#<useful information>
```

The LISP reader will signal an error on encountering `#<`.

As a hypothetical example, an implementation might print

```
#<stack-pointer si:rename-within-new-definition-maybe #o311037552>
```

for an implementation-specific “internal stack pointer” data type whose printed representation includes the name of the type, some information about the stack slot pointed to, and the machine address (in octal) of the stack slot.

Overlap, Inclusion, and Disjointness of Types

The COMMON LISP data type hierarchy is tangled and purposely left somewhat open-ended so that implementors may experiment with new data types as extensions to the language. This section explicitly states all the defined relationships between types, including subtype/supertype relationships, disjointness, and exhaustive partitioning. The user of COMMON LISP should not depend on any relationships not explicitly stated here. For example, it is not valid to assume that because a number is not complex and not rational that it must be a float, because implementations are permitted to provide yet other kinds of numbers.

First we need some terminology. If x is a supertype of y , then any object of type y is also of type x , and y is said to be a subtype of x . If types x and y are disjoint, then no object (in any implementation) may be both of type x and of type y . Types a_1 through a_n are an *exhaustive union* of type x if each a_j is a subtype of x , and any object of type x is necessarily of at least one of the types a_j ; a_1 through a_n are furthermore an *exhaustive partition* if they are also pairwise disjoint.

- The type `t` is a supertype of every type whatsoever. Every object belongs to type `t`.
- The type `nil` is a subtype of every type whatsoever. No object belongs to type `nil`.
- The types `cons`, `symbol`, `array`, `number`, and `character` are pairwise disjoint.
- The types `rational`, `float`, and `complex` are pairwise disjoint subtypes of `number`.
- The types `integer` and `ratio` are disjoint subtypes of `rational`.

Rationale: It might be thought that `integer` and `ratio` should form an exhaustive partition of the type `rational`. This is purposely avoided here in order to permit compatible experimentation with extensions to the COMMON LISP rational number system.

- The types `fixnum` and `bignum` are disjoint subtypes of `integer`.

Rationale: It might be thought that `fixnum` and `bignum` should form an exhaustive partition of the type `integer`. This is purposely avoided here in order to permit compatible experimentation with extensions to the COMMON LISP integer number system, such as the idea of adding explicit representations of infinity or of positive and negative infinity.

- The types `short-float`, `single-float`, `double-float`, and `long-float` are subtypes of `float`. Any two of them must be either disjoint or identical; if identical, then any other types between them in the above ordering must also be identical to them (for example, if `single-float` and `long-float` are identical types, then `double-float` must be identical to them also).
- The type `null` is a subtype of `symbol`; the only object of type `null` is `nil`.
- The types `cons` and `null` form an exhaustive partition of the type `list`.
- The type `standard-char` is a subtype of `string-char`; `string-char` is a subtype of `character`.
- The type `string` is a subtype of `vector`, for `string` means `(vector string-char)`.
- The type `bit-vector` is a subtype of `vector`, for `bit-vector` means `(vector bit)`.
- The types `(vector t)`, `string`, and `bit-vector` are disjoint.
- The type `vector` is a subtype of `array`; for all types `x`, the type `(vector x)` is the same as the type `(array x (*))`.
- The type `simple-array` is a subtype of `array`.
- The types `simple-vector`, `simple-string`, and `simple-bit-vector` are disjoint subtypes of `simple-array`, for they respectively mean `(simple-array t (*))`, `(simple-array string-char (*))`, and `(simple-array bit (*))`.
- The type `simple-vector` is a subtype of `vector`, and indeed is a subtype of `(vector t)`.
- The type `simple-string` is a subtype of `string`. (Note that although `string` is a subtype of `vector`, `simple-string` is not a subtype of `simple-vector`.)

Rationale: The type `simple-vector` might better have been designated `simple-general-vector`, but in this instance euphony and user convenience were deemed more important to the design of COMMON LISP than a rigid symmetry.

- The type `simple-bit-vector` is a subtype of `bit-vector`. (Note that although `bit-vector` is a subtype of `vector`, `simple-bit-vector` is not a subtype of `simple-vector`).
- The types `vector` and `list` are disjoint subtypes of `sequence`.
- The types `hash-table`, `readtable`, `package`, `pathname`, `stream`, and `random-state` are pairwise disjoint.

- Any two types created by `defstruct` are disjoint unless one is a supertype of the other by virtue of the `:include` option.
- An exhaustive union for the type `common` is formed by the types `cons`, `symbol`, `(array x)` where `x` is either `t` or a subtype of `common`, `string`, `fixnum`, `bignum`, `ratio`, `short-float`, `single-float`, `double-float`, `long-float`, `(complex x)` where `x` is a subtype of `common`, `standard-char`, `hash-table`, `readtable`, `package`, `pathname`, `stream`, `random-state`, and all types created by the user via `defstruct`. An implementation may not unilaterally add subtypes to `common`; however, future revisions to the COMMON LISP standard may extend the definition of the `common` data type.

Note that a type such as `number` or `array` may or may not be a subtype of `common`, depending on whether or not the given implementation has extended the set of objects of that type.

End of Chapter

Chapter 3

Scope and Extent

In describing various features of the COMMON LISP language, the notions of *scope* and *extent* are frequently useful. These notions arise when some object or construct must be referred to from some distant part of a program. *Scope* refers to the spatial or textual region of the program within which references may occur. *Extent* refers to the interval of time during which references may occur.

As a simple example, consider this program:

```
(defun copy-cell (x) (cons (car x) (cdr x)))
```

The scope of the parameter named *x* is the body of the `defun` form. There is no way to refer to this parameter from any other place but within the body of the `defun`. Similarly, the extent of the parameter *x* (for any particular call to `copy-cell`) is the interval from the time the function is invoked to the time it is exited. (In the general case, the extent of a parameter may last beyond the time of function exit, but that cannot occur in this simple case.)

Within COMMON LISP, a referenceable entity is *established* by the execution of some language construct, and the scope and extent of the entity are described relative to the construct and the time (during execution of the construct) at which the entity is established. For the purposes of this discussion, the term “entity” refers not only to COMMON LISP data objects, such as symbols and conses, but also to variable bindings (both ordinary and special), catchers, and `go` targets. It is important to distinguish between an entity and a name for the entity. In a function definition such as

```
(defun foo (x y) (* x (+ y 1)))
```

there is a single name, *x*, used to refer to the first parameter of the procedure whenever it is invoked; however, a new binding is established on every invocation. A *binding* is a particular parameter instance. The value of a reference to the name *x* depends not only on the scope within which it occurs (the one in the body of `foo` in the example occurs in the scope of the function definition’s parameters) but also on the particular binding or instance involved. (In this case, it depends on the invocation during which the reference is made.) More complicated examples appear at the end of this chapter.

There are a few kinds of scope and extent that are particularly useful in describing COMMON LISP:

- *Lexical scope*. Here references to the established entity can occur only within certain program portions that are lexically (that is, textually) contained within the establishing construct. Typically the construct will have a part designated the *body*, and the scope of all entities established will be (or include) the body.

Example: the names of parameters to a function normally are lexically scoped.

- *Indefinite scope.* References may occur anywhere, in any program.
- *Dynamic extent.* References may occur at any time in the interval between establishment of the entity and the explicit disestablishment of the entity. As a rule, the entity is disestablished when execution of the establishing construct completes or is otherwise terminated. Therefore entities with dynamic extent obey a stack-like discipline, paralleling the nested executions of their establishing constructs.

Example: the `with-open-file` construct opens a connection to a file and creates a stream object to represent the connection. The stream object has indefinite extent, but the connection to the open file has dynamic extent: when control exits the `with-open-file` construct, either normally or abnormally, the stream is automatically closed.

Example: the binding of a “special” variable has dynamic extent.

- *Indefinite extent.* The entity continues to exist so long as the possibility of reference remains. (An implementation is free to destroy the entity if it can prove that reference to it is no longer possible. Garbage collection strategies implicitly employ such proofs.)

Example: most COMMON LISP data objects have indefinite extent.

Example: the bindings of lexically scoped parameters of a function have indefinite extent. (By contrast, in ALGOL the bindings of lexically scoped parameters of a procedure have dynamic extent.) The function definition

```
(defun compose (f g)
  #'(lambda (x) (funcall f (funcall g x))))
```

when given two arguments, immediately returns a function as its value. The parameter bindings for `f` and `g` do not disappear because the returned function, when called, could still refer to those bindings. Therefore

```
(funcall (compose #'sqrt #'abs) -9. 0)
```

produces the value 3.0. (An analogous procedure would not necessarily work correctly in typical ALGOL implementations, or, for that matter, in most LISP dialects.)

In addition to the above terms, it is convenient to define *dynamic scope* to mean *indefinite scope and dynamic extent*. Thus we speak of “special” variables as having dynamic scope, or being dynamically scoped, because they have indefinite scope and dynamic extent: a special variable can be referred to anywhere as long as its binding is currently in effect.

The above definitions do not take into account the possibility of *shadowing*. Remote reference of entities is accomplished by using *names* of one kind or another. If two entities have the same name, then the second may shadow the first, in which case an occurrence of the name will refer to the second and cannot refer to the first.

In the case of lexical scope, if two constructs that establish entities with the same name are textually nested, then references within the inner construct refer to the entity established by the inner one; the inner one shadows the outer one. Outside the inner construct but inside the outer one, references refer to the entity established by the outer construct. For example:

```
(defun test (x z)
  (let ((z (* x 2))) (print z))
  z)
```

The binding of the variable `z` by the `let` construct shadows the parameter binding for the function `test`. The reference to the variable `z` in the `print` form refers to the `let` binding. The reference to `z` at the end of the function refers to the parameter named `z`.

In the case of dynamic extent, if the time intervals of two entities overlap, then one interval will necessarily be nested within the other one. This is a property of the design of COMMON LISP.

Implementation note: Behind the assertion that dynamic extents nest properly is the assumption that there is only a single program or process. COMMON LISP does not address the problems of multiprogramming (timesharing) or multiprocessing (more than one active processor) within a single LISP environment. The documentation for implementations that extend COMMON LISP for multiprogramming or multiprocessing should be very clear on what modifications are induced by such extensions to the rules of extent and scope. Implementors should note that COMMON LISP has been carefully designed to allow special variables to be implemented using either the "deep binding" technique or the "shallow binding" technique, but the two techniques have different semantic and performance implications for multiprogramming and multiprocessing.

A reference by name to an entity with dynamic extent will always refer to the entity of that name that has been most recently established that has not yet been disestablished. For example:

```
(defun fun1 (x)
  (catch 'trap (+ 3 (fun2 x))))

(defun fun2 (y)
  (catch 'trap (* 5 (fun3 y))))

(defun fun3 (z)
  (throw 'trap z))
```

Consider the call `(fun1 7)`. The result will be 10. At the time the `throw` is executed, there are two outstanding catchers with the name `trap`: one established within procedure `fun1`, and the other within procedure `fun2`. The latter is the more recent, and so the value 7 is returned from the catch form in `fun2`. Viewed from within `fun3`, the catch in `fun2` shadows the one in `fun1`. Had `fun2` been defined as

```
(defun fun2 (y)
  (catch 'snare (* 5 (fun3 y))))
```

then the two catchers would have different names, and therefore the one in `fun1` would not be shadowed. The result would then have been 7.

As a rule this manual simply speaks of the scope or extent of an entity; the possibility of shadowing is left implicit.

The important scope and extent rules in COMMON LISP follow:

- Variable bindings normally have lexical scope and indefinite extent.
- Variable bindings that are declared to be special have dynamic scope (indefinite scope and dynamic extent).
- A catcher established by a catch or unwind-protect special form has dynamic scope.
- An exit point established by a block construct has lexical scope and dynamic extent. (Such exit points are also established by do, prog, and other iteration constructs.)
- The go targets established by a tagbody, named by the tags in the tagbody, and referred to by go have lexical scope and dynamic extent. (Such go targets may also appear as tags in the bodies of do, prog, and other iteration constructs.)
- Named constants such as nil and pi have indefinite scope and indefinite extent.

The rules of lexical scoping imply that lambda-expressions appearing in the function construct will, in general, result in "closures" over those non-special variables visible to the lambda-expression. That is, the function represented by a lambda-expression may refer to any lexically apparent non-special variable and get the correct value, even if the construct that established the binding has been exited in the course of execution. The compose example shown earlier in this chapter provides one illustration of this. The rules also imply that special variable bindings are not "closed over" as they may be in certain other dialects of LISP.

Constructs that use lexical scope effectively generate a new name for each established entity on each execution. Therefore dynamic shadowing cannot occur (though lexical shadowing may). This is of particular importance when dynamic extent is involved. For example:

```
(defun contorted-example (f g x)
  (if (= x 0)
      (funcall f)
      (block here
        (+ 5 (contorted-example g
                                #'(lambda ()
                                   (return-from here 4))
                                (- x 1))))))
```

Consider the call (contorted-example nil nil 2). This produces the result 4. During the course of execution, there are three calls on contorted-example, interleaved with two establishments of blocks:

```
(contorted-example nil nil 2)

(block here, . . . )

(contorted-example nil
  #'(lambda () (return-from here, 4))
  1)

(block here2 . . . )
```

```
(contorted-example #'(lambda () (return-from here, 4))
                  #'(lambda () (return-from here2 4))
                  0)
(funcall f)
  where f ⇒ #'(lambda () (return-from here, 4))

(return-from here, 4)
```

At the time the `funcall` is executed there are two block exit points outstanding, each apparently named `here`. In the trace above, these exit points are distinguished for expository purposes by subscripts. The `return-from` form executed as a result of the `funcall` operation refers to the *outer* outstanding exit point (`here1`), not the inner one (`here2`). This is a consequence of the rules of lexical scoping: it refers to that exit point textually visible at the point of execution of the function construct (`here` abbreviated by the `#'` syntax) that resulted in creation of the function object actually invoked by the `funcall`.

If, in this example, one were to change the form `(funcall f)` to `(funcall g)`, then the value of the call `(contorted-example nil nil 2)` would be 9. The value would change because the `funcall` would cause the execution of `(return-from here2 4)`, thereby causing a return from the inner exit point (`here2`). When that occurs, the value 4 is returned from the middle invocation of `contorted-example`, 5 is added to that to get 9, and that value is returned from the outer block and the outermost call to `contorted-example`. The point is that the choice of exit point returned from has nothing to do with its being innermost or outermost; rather, it depends on the lexical scoping information that is effectively packaged up with a lambda-expression when the function construct is executed.

This function `contorted-example` works only because the function named by `f` is invoked during the extent of the exit point. Block exit points are like non-special variable bindings in having lexical scope, but differ in having dynamic extent rather than indefinite extent. Once the flow of execution has left the block construct, the exit point is disestablished. For example:

```
(defun illegal-example ()
  (let ((y (block here #'(lambda (z) (return-from here z)))))
    (if (numberp y) y (funcall y 5))))
```

One might expect the call `(illegal-example)` to produce 5 by the following incorrect reasoning: the `let` statement binds the variable `y` to the value of the block construct; this value is a function resulting from the lambda-expression. Because `y` is not a number, it is invoked on the value 5. The `return-from` should then return this value from the exit point named `here`, thereby exiting from the block *again* and giving `y` the value 5 which, being a number, is then returned as the value of the call to `illegal-example`.

The argument fails only because exit points are defined in COMMON LISP to have dynamic extent. The argument is correct up to the execution of the `return-from`. The execution of the `return-from` is an error, however, *not* because it cannot refer to the exit point, but because it does correctly refer to an exit point *and* that exit point has been disestablished.

End of Chapter

Chapter 4

Type Specifiers

In COMMON LISP, types are named by LISP objects, specifically symbols and lists, called *type specifiers*. Symbols name predefined classes of objects, whereas lists usually indicate combinations or specializations of simpler types. Symbols or lists may also be abbreviations for types that could be specified in other ways.

Type Specifier Symbols

The type symbols defined by the system include those shown in Table 4-1. In addition, when a structure type is defined using `defstruct`, the name of the structure type becomes a valid type symbol.

Table 4-1: Standard Type Specifier Symbols

array	fixnum	package	simple-string
atom	float	pathname	simple-vector
bignum	function	random-state	single-float
bit	hash-table	ratio	standard-char
bit-vector	integer	rational	stream
character	keyword	readtable	string
common	list	sequence	string-char
compiled-function	long-float	short-float	symbol
complex	nil	signed-byte	t
cons	null	simple-array	unsigned-byte
double-float	number	simple-bit-vector	vector

Type Specifier Lists

If a type specifier is a list, the *car* of the list is a symbol, and the rest of the list is subsidiary type information. In many cases a subsidiary item may be *unspecified*. The unspecified subsidiary item is indicated by writing `*`. For example, to completely specify a vector type, one must mention the type of the elements and the length of the vector, as for example:

```
(vector double-float 100)
```

To leave the length unspecified, one would write

```
(vector double-float *)
```

To leave the element type unspecified, one would write

```
(vector * 100)
```

Suppose that two type specifiers are the same except that the first has a * where the second has a more explicit specification. Then the second denotes a subtype of the type denoted by the first.

As a convenience, if a list has one or more unspecified items at the end, such items may simply be dropped rather than writing an explicit * for each one. If dropping all occurrences of * results in a singleton list, then the parentheses may be dropped as well (the list may be replaced by the symbol in its *car*). For example, (vector double-float *) may be abbreviated to (vector double-float), and (vector * *) may be abbreviated to (vector) and then to simply vector.

Predicating Type Specifiers

A type specifier list (*satisfies predicate-name*) denotes the set of all objects that satisfy the predicate named by *predicate-name*, which must be a symbol whose global function definition is a one-argument predicate. (A name is required; lambda-expressions are disallowed in order to avoid scoping problems.) For example, the type (satisfies numberp) is the same as the type number. The call (typep x '(satisfies p)) results in applying p to x and returning t if the result is true and nil if the result is false.

As an example, the type string-char could be defined as

```
(deftype string-char ()
  '(and character (satisfies string-char-p)))
```

See deftype.

It is not a good idea for a predicate appearing in a satisfies type specifier to cause any side effects when invoked.

Type Specifiers that Combine

The following type specifier lists define a data type in terms of other types or objects.

```
(member object1 object2 . . .)
```

This denotes the set containing precisely those objects named. An object is of this type if and only if it is eql to one of the specified objects.

Compatibility note: This is approximately equivalent to what the INTERLISP DECL package calls `memq`.

(not *type*)

This denotes the set of all those objects that are *not* of the specified type.

(and *type1 type2 . . .*)

This denotes the intersection of the specified types.

Compatibility note: This is roughly equivalent to what the INTERLISP DECL package calls `allof`.

When `typep` processes an `and` type specifier, it always tests each of the component types in order from left to right and stops processing as soon as one component of the intersection has been found to which the object in question does not belong. In this respect an `and` type specifier is similar to an executable `and` form. The purpose of this similarity is to allow a `satisfies` type specifier to depend on filtering by previous type specifiers. For example, suppose there were a function `primep` that takes an integer and says whether it is prime. Suppose also that it is an error to give any object other than an integer to `primep`. Then the type specifier

(and integer (satisfies primep))

is guaranteed never to result in an error because the function `primep` will not be invoked unless the object in question has already been determined to be an integer.

(or *type1 type2 . . .*)

This denotes the union of the specified types. For example, the type `list` by definition is the same as `(or null cons)`. Also, the value returned by the function `position` is always of type `(or null (integer 0 *))` (either `nil` or a non-negative integer).

Compatibility note: This is roughly equivalent to what the INTERLISP DECL package calls `oneof`.

As for `and`, when `typep` processes an `or` type specifier, it always tests each of the component types in order from left to right and stops processing as soon as one component of the union has been found to which the object in question belongs.

Type Specifiers that Specialize

Some type specifier lists denote *specializations* of data types named by symbols. These specializations may be reflected by more efficient representations in the underlying implementation. As an example, consider the type (array short-float). Implementation A may choose to provide a specialized representation for arrays of short floating-point numbers, and implementation B may choose not to.

If you should want to create an array for the express purpose of holding only short-float objects, you may optionally specify to make-array the element type short-float. This does not *require* make-array to create an object of type (array short-float); it merely *permits* it. The request is construed to mean, "Produce the most specialized array representation capable of holding short-floats that the implementation can provide." Implementation A will then produce a specialized array of type (array short-float), and implementation B will produce an ordinary array of type (array t).

If one were then to ask whether the array were actually of type (array short-float), implementation A would say "yes," but implementation B would say "no." This is a property of make-array and similar functions: what you ask for is not necessarily what you get.

Types can therefore be used for two different purposes: *declaration* and *discrimination*. Declaring to make-array that elements will always be of type short-float permits optimization. Similarly, declaring that a variable takes on values of type (array short-float) amounts to saying that the variable will take on values that might be produced by specifying element type short-float to make-array. On the other hand, if the predicate typep is used to test whether an object is of type (array short-float), only objects actually of that specialized type can satisfy the test; in implementation B no object can pass that test.

The valid list-format names for data types are as follows:

(array *element-type dimensions*)

This denotes the set of specialized arrays whose elements are all members of the type *element-type* and whose dimensions match *dimensions*. For declaration purposes, this type encompasses those arrays that can result by specifying *element-type* as the element type to the function make-array; this may be different from what the type means for discrimination purposes. *element-type* must be a valid type specifier or unspecified. *dimensions* may be a non-negative integer, which is the number of dimensions, or it may be a list of non-negative integers representing the length of each dimension (any dimension may be unspecified instead), or it may be unspecified. For example:

(array integer 3)	;Three-dimensional arrays of integers
(array integer (* * *))	;Three-dimensional arrays of integers
(array * (4 5 6))	;4-by-5-by-6 arrays
(array character (3 *))	;Two-dimensional arrays of characters
	; that have exactly three rows
(array short-float ())	;Zero-rank arrays of short-format
	; floating-point numbers

Note that `(array t)` is a proper subset of `(array *)`. The reason is that `(array t)` is the set of arrays that can hold any COMMON LISP object (the elements are of type `t`, which includes all objects). On the other hand, `(array *)` is the set of all arrays whatsoever, including for example arrays that can hold only characters. Now `(array character)` is not a subset of `(array t)`; the two sets are in fact disjoint because `(array character)` is not the set of all arrays that can hold characters, but rather the set of arrays that are specialized to hold precisely characters and no other objects. To test whether an array `foo` can hold a character, one should not use

```
(typep foo '(array character))
```

but rather

```
(subtypep 'character (array-element-type foo))
```

See `array-element-type`.

```
(simple-array element-type dimensions)
```

This is equivalent to `(array element-type dimensions)` except that it additionally specifies that objects of the type are *simple* arrays. (See "Arrays," Chapter 2.)

```
(vector element-type size)
```

This denotes the set of specialized one-dimensional arrays whose elements are all of type *element-type* and whose lengths match *size*. This is entirely equivalent to `(array element-type (size))`. For example:

```
(vector double-float)           ;Vectors of double-format
                                ; floating-point numbers
(vector * 5)                    ;Vectors of length 5
(vector t 5)                    ;General vectors of length 5
(vector (mod 32) *)             ;Vectors of integers between 0 and 31
```

The specialized types `(vector string-char)` and `(vector bit)` are so useful that they have the special names `string` and `bit-vector`. Every implementation of COMMON LISP must provide distinct representations for these as distinct specialized data types.

```
(simple-vector size)
```

This is the same as `(vector t size)` except that it additionally specifies that its elements are *simple* general vectors.

(complex *type*)

Every element of this type is a complex number whose real part and imaginary part are each of type *type*. For declaration purposes, this type encompasses those complex numbers that can result by giving numbers of the specified type to the function `complex`; this may be different from what the type means for discrimination purposes. As an example, Gaussian integers might be described as (complex integer), even in implementations where giving two integers to the function `complex` results in an object of type (complex rational). Note that complex values and operations are not supported in this implementation. See Chapter 12.

(function (*arg1-type arg2-type . . .*) *value-type*)

This type may be used only for declaration and not for discrimination; `typep` will signal an error if it encounters a specifier of this form. Every element of this type is a function that accepts arguments at *least* of the types specified by the *argj-type* forms and returns a value that is a member of the types specified by the *value-type* form. The `&optional`, `&rest`, and `&key` markers may appear in the list of argument types. The *value-type* may be a values type specifier in order to indicate the types of multiple values.

As an example, the function `cons` is of type (function (t t) cons), because it can accept any two arguments and always returns a cons. The function `cons` is also of type (function (float string) list), because it can certainly accept a floating-point number and a string (among other things), and its result is always of type list (in fact a cons is never null, but that does not matter for this type declaration). The function `truncate` is of type (function (number number) (values number number)), as well as of type (function (integer (mod 8)) integer).

(values *value1-type value2-type . . .*)

This type specifier is extremely restricted: it may be used *only* as the *value-type* in a function type specifier or in a the special form. It is used to specify individual types when multiple values are involved. The `&optional`, `&rest`, and `&key` markers may appear in the *value-type* list; they thereby indicate the parameter list of a function that, when given to `multiple-value-call` along with the values, would be suitable for receiving those values.

Type Specifiers that Abbreviate

The following type specifiers are, for the most part, abbreviations for other type specifiers that would be far too verbose to write out explicitly (using, for example, `member`).

(integer *low high*)

Denotes the integers between *low* and *high*. The limits *low* and *high* must each be an integer, a list of an integer, or unspecified. An integer is an inclusive limit, a list of an integer is an exclusive limit, and `*` means that a limit does not exist and so effectively denotes minus or plus infinity, respectively. The type `fixnum` is simply a name for (integer *smallest largest*) for implementation-dependent values of *smallest* and *largest* (see `most-negative-fixnum` and `most-positive-fixnum`). The type (integer 0 1) is so useful that it has the special name `bit`.

(mod n)

Denotes the set of non-negative integers less than n . This is equivalent to (integer 0 $n-1$) or to (integer 0 (n)).

(signed-byte s)

Denotes the set of integers that can be represented in two's-complement form in a byte of s bits. This is equivalent to (integer -2^{s-1} $2^{s-1}-1$). Simply signed-byte or (signed-byte $*$) is the same as integer.

(unsigned-byte s)

Denotes the set of non-negative integers that can be represented in a byte of s bits. This is equivalent to (mod 2^s), that is, (integer 0 2^s-1). Simply unsigned-byte or (unsigned-byte $*$) is the same as (integer 0 $*$), the set of non-negative integers.

(rational low $high$)

Denotes the rationals between low and $high$. The limits low and $high$ must each be a rational, a list of a rational, or unspecified. A rational is an inclusive limit, a list of a rational is an exclusive limit, and $*$ means that a limit does not exist and so effectively denotes minus or plus infinity, respectively.

(float low $high$)

Denotes the set of floating-point numbers between low and $high$. The limits low and $high$ must each be a floating-point number, a list of a floating-point number, or unspecified; a floating-point number is an inclusive limit, a list of a floating-point number is an exclusive limit, and $*$ means that a limit does not exist and so effectively denotes minus or plus infinity, respectively.

In a similar manner, one may use:

(short-float low $high$)

(single-float low $high$)

(double-float low $high$)

(long-float low $high$)

In this case, if a limit is a floating-point number (or a list of one), it must be one of the appropriate format.

(string $size$)

Means the same as (array string-char ($size$)): the set of strings of the indicated size.

(simple-string *size*)

Means the same as (simple-array string-char (*size*)): the set of simple strings of the indicated size.

(bit-vector *size*)

Means the same as (array bit (*size*)): the set of bit-vectors of the indicated size.

(simple-bit-vector *size*)

This means the same as (simple-array bit (*size*)): the set of bit-vectors of the indicated size.

Defining New Type Specifiers

New type specifiers can come into existence in two ways. First, defining a new structure type with `defstruct` automatically causes the name of the structure to be a new type specifier symbol. Second, the `deftype` special form can be used to define new type-specifier abbreviations.

```
deftype name lambda-list {declaration | doc-string}* {form}* [Macro]
```

This is very similar to `defmacro` macro: *name* is the symbol that identifies the type specifier being defined, *lambda-list* is a lambda-list (and may contain `&optional` and `&rest` markers), and the *forms* constitute the body of the expander function. If we view a type specifier list as a list containing the type specifier name and some argument forms, the argument forms (unevaluated) are bound to the corresponding parameters in *lambda-list*. Then the body forms are evaluated as an implicit `progn`, and the value of the last form is interpreted as a new type specifier for which the original specifier was an abbreviation. The *name* is returned as the value of the `deftype` form.

`deftype` differs from `defmacro` in that if no *initform* is specified for an `&optional` parameter, the default value is `*`, not `nil`.

If the optional documentation string *doc-string* is present, then it is attached to the *name* as a documentation string of type type; see documentation.

Here are some examples of the use of `deftype`:

```
(deftype mod (n) `(integer 0 (.n)))
```

```
(deftype list () `(or null cons))
```

```
(deftype square-matrix (&optional type size)
  "SQUARE-MATRIX includes all square two-dimensional arrays."
  `(array .type (.size .size)))
```

```
(square-matrix short-float 7) means (array short-float (7 7))
```

```
(square-matrix bit) means (array bit (* *))
```

If the type name defined by `deftype` is used simply as a type specifier symbol, it is interpreted as a type specifier list with no argument forms. Thus, in the example above, `square-matrix` would mean `(array *(+ *))`, the set of two-dimensional arrays. This would unfortunately fail to convey the constraint that the two dimensions be the same; `(square-matrix bit)` has the same problem. A better definition is:

```
(defun equidimensional (a)
  (or (< (array-rank a) 2)
      (apply #'= (array-dimensions a))))

(deftype square-matrix (&optional type size)
  `(and (array .type (.size .size))
        (satisfies equidimensional)))
```

Type Conversion Function

The following function may be used to convert an object to an equivalent object of another type.

`coerce object result-type` [Function]

The *result-type* must be a type specifier; the *object* is converted to an “equivalent” object of the specified type. If the coercion cannot be performed, then an error is signalled. In particular, `(coerce x 'nil)` always signals an error. If *object* is already of the specified type, as determined by `typep`, then it is simply returned. It is not generally possible to convert any object to be of any type whatsoever; only certain conversions are permitted:

- Any sequence type may be converted to any other sequence type, provided the new sequence can contain all actual elements of the old sequence (it is an error if it cannot). If the *result-type* is specified as simply `array`, for example, then `(array t)` is assumed. A specialized type such as `string` or `(vector (rational -1-2 1-2))` `(vector (complex short-float))` may be specified; of course, the result may be of either that type or some more general type, as determined by the implementation. Elements of the new sequence will be `eql` to corresponding elements of the old sequence. If the *sequence* is already of the specified type, it may be returned without copying it; in this, `(coerce sequence type)` differs from `(concatenate type sequence)` for the latter is required to copy the argument *sequence*. In particular, if one specifies `sequence`, then the argument may simply be returned if it already is a sequence.

```
(coerce '(a b c) 'vector) ⇒ #(a b c)
```

- Some strings, symbols, and integers may be converted to characters. If *object* is a string of length 1, then the sole element of the string is returned. If *object* is a symbol whose print name is of length 1, then the sole element of the print name is returned. If *object* is an integer *n*, then `(int-char n)` is returned. See `character`.

```
(coerce "a" 'character) ⇒ #\a
```

- Any non-complex number can be converted to a short-float, single-float, double-float, or long-float. If simply `float` is specified, and *object* is not already a float of some kind, then the object is converted to a single-float.

4-10

(coerce 0 'short-float) ⇒ 0.0S0

(coerce 3.5L0 'float) ⇒ 3.5L0

(coerce 7/2 'float) ⇒ 3.5

- * Any number can be converted to a complex number. If the number is not already complex, then a zero imaginary part is provided by coercing the integer zero to the type of the given real part. (If the given real part is rational, however, then the rule of canonical representation for complex rationals will result in the immediate re-conversion of the result from type complex back to type rational.)

(coerce 4.5s0 'complex) ⇒ #C(4.5S0 0.0S0)

(coerce 7.2 'complex) ⇒ 7.2

(coerce #C(7.2 0) '(complex double-float)) ⇒ #C(3.5D0 0.0D0)

- Any object may be coerced to type *t*.

(coerce x 't) ≡ (identity x) ≡ x

Coercions from floating-point numbers to rationals and from ratios to integers are purposely *not* provided because of rounding problems. The functions `rational`, `rationalize`, `floor`, `ceiling`, `truncate`, and `round` may be used for such purposes. Similarly, coercions from characters to integers are purposely not provided; `char-code` or `char-int` may be used explicitly to perform such conversions.

Determining the Type of an Object

The following function may be used to obtain a type specifier describing the type of a given object.

`type-of object`

[Function]

(`type-of object`) returns an implementation-dependent result: some *type* of which the *object* is a member. Implementors are encouraged to arrange for `type-of` to return the most specific type that can be conveniently computed and is likely to be useful to the user. If the argument is a user-defined named structure created by `defstruct`, then `type-of` will return the type name of that structure. Because the result is implementation-dependent, it is usually better to use `type-of` primarily for debugging purposes; however, in a few situations portable code requires the use of `type-of`, such as when the result is to be given to the `coerce` or `map` function. On the other hand, often the `typep` function or the `typecase` construct is more appropriate than `type-of`.

Compatibility note: In MACLISP the function `type-of` is called `typep`, and anomalously so, for it is not a predicate.

End of Chapter

Chapter 5

Program Structure

In Chapter 2 the syntax was sketched for notating data objects in COMMON LISP. The same syntax is used for notating programs because all COMMON LISP programs have a representation as COMMON LISP data objects.

LISP programs are organized as forms and functions. Forms are *evaluated* (relative to some context) to produce values and side effects. Functions are invoked by *applying* them to arguments. The most important kind of form performs a function call; conversely, a function performs computation by evaluating forms.

In this chapter forms are discussed first, and then functions. Finally, certain “top level” special forms are discussed; the most important of these is `defun`, whose purpose is to define a named function.

Forms

The standard unit of interaction with a COMMON LISP implementation is the *form*, which is simply a data object meant to be *evaluated* as a program to produce one or more *values* (which are also data objects). One may request evaluation of *any* data object, but only certain ones are meaningful. For instance, symbols and lists are meaningful forms, while arrays normally are not. Examples of meaningful forms are 3, whose value is 3, and (+ 3 4), whose value is 7. We write $3 \Rightarrow 3$ and $(+ 3 4) \Rightarrow 7$ to indicate these facts. (\Rightarrow means “evaluates to.”)

Meaningful forms may be divided into three categories: self-evaluating forms, such as numbers; symbols, which stand for variables; and lists. The lists in turn may be divided into three categories: special forms, macro calls, and function calls.

Any COMMON LISP data object not explicitly defined here to be a valid form is not a valid form. It is an error to evaluate anything but a valid form.

Implementation note: An implementation is free to make implementation-dependent extensions to the evaluator, but is strongly encouraged to signal an error on any attempt to evaluate anything but a valid form or an object for which a meaningful evaluation extension has been purposely defined.

Self-Evaluating Forms

All numbers, characters, strings, and bit-vectors are *self-evaluating* forms. When such an object is evaluated, that object (or possibly a copy in the case of numbers or characters) is returned as the value of the form. The empty list `()`, which is also the false value `nil`, is also a self-evaluating form: the value of `nil` is `nil`. Keywords (symbols written with a leading colon) also evaluate to themselves: the value of `:start` is `:start`.

Variables

Symbols are used as names of variables in COMMON LISP programs. When a symbol is evaluated as a form, the value of the variable it names is produced. For example, after doing `(setq items 3)`, which assigns the value 3 to the variable named `items`, then `items` \Rightarrow 3. Variables can be *assigned* to, as by `setq`, or *bound*, as by `let`. Any program construct that binds a variable effectively saves the old value of the variable and causes it to have a new value, and on exit from the construct the old value is reinstated.

There are actually two kinds of variables in COMMON LISP, called *lexical* (or *static*) variables and *special* (or *dynamic*) variables. At any given time either or both kinds of variable with the same name may have a current value. Which of the two kinds of variable is referred to when a symbol is evaluated depends on the context of the evaluation. The general rule is that if the symbol occurs textually within a program construct that creates a *binding* for a variable of the same name, then the reference is to the variable specified by the binding; if no such program construct textually contains the reference, then it is taken to refer to the special variable of that name.

The distinction between the two kinds of variable is one of scope and extent. A lexically bound variable can be referred to *only* by forms occurring at any *place* textually within the program construct that binds the variable. A dynamically bound (special) variable can be referred to at any *time* from the time the binding is made until the time evaluation of the construct that binds the variable terminates. Therefore lexical binding of variables imposes a spatial limitation on occurrences of references (but no temporal limitation, for the binding continues to exist as long as the possibility of reference remains). Conversely, dynamic binding of variables imposes a temporal limitation on occurrences of references (but no spatial limitation). For more information on scope and extent, see Chapter 3.

The value a special variable has when there are currently no bindings of that variable is called the *global* value of the (special) variable. A global value can be given to a variable only by assignment, because a value given by binding is by definition not global.

It is possible for a special variable to have no value at all, in which case it is said to be *unbound*. By default, every global variable is unbound unless and until explicitly assigned a value, except for those global variables defined in this manual or by the implementation already to have values when the LISP system is first started. It is also possible to establish a binding of a special variable and then cause that binding to be valueless by using the function `makunbound`. In this situation the variable is also said to be “unbound,” although this is a misnomer; precisely speaking, it is bound but valueless. It is an error to refer to a variable that is unbound.

Certain global variables are reserved as “named constants.” They have a global value and may not be bound or assigned to. For example, the symbols `t` and `nil` are reserved. One may not assign a value to `t` or `nil`, and one may not bind `t` or `nil`. The global value of `t` is always `t`, and the global value of `nil` is always `nil`. Constant symbols defined by `defconstant` also become

reserved and may not be further assigned to or bound (although they may be redefined, if necessary, by using `defconstant` again). Keyword symbols, which are notated with a leading colon, are reserved and may never be assigned to or bound; a keyword always evaluates to itself.

Special Forms

If a list is to be evaluated as a form, the first step is to examine the first element of the list. If the first element is one of the symbols appearing in Table 5-1, then the list is called a *special form*. (This use of the word “special” is unrelated to its use in the phrase “special variable.”)

Special forms are generally environment and control constructs. Every special form has its own idiosyncratic syntax. An example is the `if` special form: `(if p (+ x 4) 5)` in COMMON LISP means what “if `p` then `x+4` else 5” would mean in ALGOL.

The evaluation of a special form normally produces a value or values, but the evaluation may instead call for a non-local exit; see `return-from`, `go`, and `throw`.

Table 5-1: Names of All COMMON LISP Special Forms

<code>block</code>	<code>if</code>	<code>progv</code>
<code>catch</code>	<code>labels</code>	<code>quote</code>
<code>compiler-let</code>	<code>let</code>	<code>return-from</code>
<code>declare</code>	<code>let*</code>	<code>setq</code>
<code>eval-when</code>	<code>macrolet</code>	<code>tagbody</code>
<code>flet</code>	<code>multiple-value-call</code>	<code>the</code>
<code>function</code>	<code>multiple-value-prog1</code>	<code>throw</code>
<code>go</code>	<code>progn</code>	<code>unwind-protect</code>

The set of special forms is fixed in COMMON LISP; no way is provided for the user to define more. The user can create new syntactic constructs, however, by defining macros.

The set of special forms in COMMON LISP is purposely kept very small because any program-analyzing program must have special knowledge about every type of special form. Such a program needs no special knowledge about macros because it is simple to expand the macro and operate on the resulting expansion. (This is not to say that many such programs, particularly compilers, will not have such special knowledge. A compiler may be able to produce much better code if it recognizes such constructs as `typecase` and `multiple-value-bind` and gives them customized treatment.)

An implementation is free to implement as a macro any construct described herein as a special form. Conversely, an implementation is free to implement as a special form any construct described herein as a macro if an equivalent macro definition is also provided. The practical consequence is that the predicates `macro-function` and `special-form-p` may both be true of the same symbol. It is recommended that a program-analyzing program process a form that is a list whose *car* is a symbol as follows:

1. If the program has particular knowledge about the symbol, process the form using special-purpose code. All of the symbols listed in Table 5-1 should fall into this category.

2. Otherwise, if `macro-function` is true of the symbol, apply either `macroexpand` or `macroexpand-1`, as appropriate, to the entire form and then start over.
3. Otherwise, assume it is a function call.

Macros

If a form is a list and the first element is not the name of a special form, it may be the name of a *macro*; if so, the form is said to be a *macro call*. A macro is essentially a function from forms to forms that will, given a call to that macro, compute a new form to be evaluated in place of the macro call. (This computation is sometimes referred to as *macro expansion*.) For example, the macro named `return` will take a form such as `(return x)` and from that form compute a new form `(return-from nil x)`. We say that the old form *expands* into the new form. The new form is then evaluated in place of the original form; the value of the new form is returned as the value of the original form.

There are a number of standard macros in `COMMON LISP`, and the user can define more by using `defmacro`.

Macros provided by a `COMMON LISP` implementation as described herein may expand into code that is not portable among differing implementations. That is, a macro call may be implementation-independent because the macro is defined in this manual, but the expansion need not be.

Implementation note: Implementors are encouraged to implement the macros defined in this manual, as far as is possible, in such a way that the expansion will not contain any implementation-dependent special forms, nor contain as forms data objects that are not considered to be forms in `COMMON LISP`. The purpose of this restriction is to ensure that the expansion can be processed by a program-analyzing program in an implementation-independent manner. There is no problem with a macro expansion containing calls to implementation-dependent functions. This restriction is not a requirement of `COMMON LISP`; it is recognized that certain complex macros may be able to expand into significantly more efficient code in certain implementations by using implementation-dependent special forms in the macro expansion.

Function Calls

If a list is to be evaluated as a form and the first element is not a symbol that names a special form or macro, then the list is assumed to be a *function call*. The first element of the list is taken to name a function. Any and all remaining elements of the list are forms to be evaluated; one value is obtained from each form, and these values become the *arguments* to the function. The function is then *applied* to the arguments. The functional computation normally produces a value, but it may instead call for a non-local exit; see `throw`. A function that does return may produce no value or several values; see `values`. If and when the function returns, whatever values it returns become the values of the function-call form.

For example, consider the evaluation of the form `(+ 3 (* 4 5))`. The symbol `+` names the addition function, not a special form or macro. Therefore the two forms `3` and `(* 4 5)` are evaluated

to produce arguments. The form 3 evaluates to 3, and the form (* 4 5) is a function call (to the multiplication function). Therefore the forms 4 and 5 are evaluated, producing arguments 4 and 5 for the multiplication. The multiplication function calculates the number 20 and returns it. The values 3 and 20 are then given as arguments to the addition function, which calculates and returns the number 23. Therefore we say (+ 3 (* 4 5)) \Rightarrow 23.

Functions

There are two ways to indicate a function to be used in a function call form. One is to use a symbol that names the function. This use of symbols to name functions is completely independent of their use in naming special and lexical variables. The other way is to use a *lambda-expression*, which is a list whose first element is the symbol `lambda`. A lambda-expression is *not* a form; it cannot be meaningfully evaluated. Lambda-expressions and symbols, when used in programs as names of functions, can appear only as the first element of a function-call form, or as the second element of the function special form. Note that symbols and lambda-expressions are treated as *names* of functions in these two contexts. This should be distinguished from the treatment of symbols and lambda-expressions as *function objects*, that is, objects that satisfy the predicate `functionp`, as when giving such an object to `apply` or `funcall` to be invoked.

Named Functions

A name can be given to a function in one of two ways. A *global name* can be given to a function by using the `defun` construct. A *local name* can be given to a function by using the `flet` or `labels` special form. When a function is named, a lambda-expression is effectively associated with that name along with information about the entities that are lexically apparent at that point.

If a symbol appears as the first element of a function-call form, then it refers to the definition established by the innermost `flet` or `labels` construct that textually contains the reference, or to the global definition (if any) if there is no such containing construct.

Lambda-Expressions

A *lambda-expression* is a list with the following syntax:

```
(lambda lambda-list . body)
```

The first element must be the symbol `lambda`. The second element must be a list. It is called the *lambda-list*, and specifies names for the *parameters* of the function. When the function denoted by the lambda-expression is applied to arguments, the arguments are matched with the parameters specified by the lambda-list. The *body* may then refer to the arguments by using the parameter names. The *body* consists of any number of forms (possibly zero). These forms are evaluated in sequence, and the results of the *last* form only are returned as the results of the application (the value `nil` is returned if there are zero forms in the body). The complete syntax of a lambda-expression is:

```
(lambda ({var}*
  [&optional {var | (var [initform [svar]])}*]
  [&rest var]
  [&key {var} | ({var | (keyword var)} [initform [svar]])*]
  [&allow-other-keys]
  [&aux {var | (var [initform])}*])
{declaration | documentation-string}*
{form}*)
```

Each element of a lambda-list is either a *parameter specifier* or a *lambda-list keyword*; lambda-list keywords begin with &. (Note that lambda-list keywords are not keywords in the usual sense; they do not belong to the keyword package. They are ordinary symbols each of whose names begins with an ampersand. This terminology is unfortunately confusing but is retained for historical reasons.)

In all cases a *var* or *svar* must be a symbol, the name of a variable; each *keyword* must be a keyword symbol, such as `:start`. An *initform* may be any form.

A lambda-list has five parts, any or all of which may be empty:

- Specifiers for the *required* parameters. These are all the parameter specifiers up to the first lambda-list keyword; if there is no such lambda-list keyword, then all the specifiers are for required parameters.
- Specifiers for *optional* parameters. If the lambda-list keyword `&optional` is present, the *optional* parameter specifiers are those following the lambda-list keyword `&optional` up to the next lambda-list keyword or the end of the list.
- A specifier for a *rest* parameter. The lambda-list keyword `&rest`, if present, must be followed by a single *rest* parameter specifier, which in turn must be followed by another lambda-list keyword or the end of the lambda-list.
- Specifiers for *keyword* parameters. If the lambda-list keyword `&key` is present, all specifiers up to the next lambda-list keyword or the end of the list are *keyword* parameter specifiers. The keyword parameter specifiers may optionally be followed by the lambda-list keyword `&allow-other-keys`.
- Specifiers for *aux* variables. These are not really parameters. If the lambda-list keyword `&aux` is present, all specifiers after it are *auxiliary variable* specifiers.

When the function represented by the lambda-expression is applied to arguments, the arguments and parameters are processed in order from left to right. In the simplest case, only required parameters are present in the lambda-list; each is specified simply by a name *var* for the parameter variable. When the function is applied, there must be exactly as many arguments as there are parameters, and each parameter is bound to one argument. Here, and in general, the parameter is bound as a lexical variable unless a declaration has been made that it should be a special binding; see `defvar`, `proclaim`, and `declare`.

In the more general case, if there are n required parameters (n may be zero), there must be at least n arguments, and the required parameters are bound to the first n arguments. The other parameters are then processed using any remaining arguments.

If *optional* parameters are specified, then each one is processed as follows. If any unprocessed arguments remain, then the parameter variable *var* is bound to the next remaining argument, just as for a required parameter. If no arguments remain, however, then the *initform* part of the parameter specifier is evaluated, and the parameter variable is bound to the resulting value (or to *nil* if no *initform* appears in the parameter specifier). If another variable name *svar* appears in the specifier, it is bound to *true* if an argument was available, and to *false* if no argument remained (and therefore *initform* had to be evaluated). The variable *svar* is called a *supplied-p* parameter; it is bound not to an argument but to a value indicating whether or not an argument had been supplied for another parameter.

After all *optional* parameter specifiers have been processed, then there may or may not be a *rest* parameter. If there is a *rest* parameter, it is bound to a list of all as-yet-unprocessed arguments. (If no unprocessed arguments remain, the *rest* parameter is bound to the empty list.) If there is no *rest* parameter and there are no *keyword* parameters, then there should be no unprocessed arguments (it is an error if there are).

Next, any *keyword* parameters are processed. For this purpose the same arguments are processed that would be made into a list for a *rest* parameter. (Indeed, it is permitted to specify both *&rest* and *&key*. In this case the remaining arguments are used for both purposes; that is, all remaining arguments are made into a list for the *&rest* parameter, and are also processed for the *&key* parameters. This is the only situation in which an argument is used in the processing of more than one parameter specifier.) If *&key* is specified, there must remain an even number of arguments; these are considered as pairs, the first argument in each pair being interpreted as a keyword name and the second as the corresponding value. It is an error for the first object of each pair to be anything but a keyword.

Rationale: This last restriction is imposed so that a compiler may issue warnings about certain malformed calls to functions that take keyword arguments. It must be remembered that the arguments in a function call that evaluate to keywords are just like any other arguments, and may be any evaluable forms. A compiler could not, without additional context, issue a warning about the call

```
(fill seq item x y)
```

because in principle the variable *x* might have as its value a keyword such as *:start*. However, a compiler would be justified in issuing a warning about the call

```
(fill seq item 0 10)
```

because the constant *0* is definitely not a keyword. Similarly, if in the first case the variable *x* had been declared to be of type *integer* then type analysis could enable the compiler to justify a warning.

In each keyword parameter specifier must be a name *var* for the variable. If an explicit *keyword* is specified, then that is the keyword name for the parameter. Otherwise the name *var* serves to indicate the keyword name, in that a keyword with the same name (in the keyword package) is used as the keyword. Thus

```
(defun foo (&key radix (type 'integer)) . . . )
```

means exactly the same as

```
(defun foo (&key ([:radix radix]) ([:type type] 'integer)) . . . )
```

The keyword parameter specifiers are, like all parameter specifiers, effectively processed from left to right. For each keyword parameter specifier, if there is an argument pair whose keyword name matches that specifier's keyword name (that is, the names are *eq*), then the parameter variable for that specifier is bound to the second item (the value) of that argument pair. If more than one such argument pair matches, it is not an error; the leftmost argument pair is used. If no such argument pair exists, then the *initform* for that specifier is evaluated and the parameter variable is bound to that value (or to *nil* if no *initform* was specified). The variable *svar* is treated as for ordinary *optional* parameters: it is bound to *true* if there was a matching argument pair, and to *false* otherwise.

It is an error if an argument pair has a keyword name not matched by any parameter specifier, unless at least one of the following two conditions is met:

- `&allow-other-keys` was specified in the lambda-list.
- Among the keyword argument pairs is a pair whose keyword is `:allow-other-keys` and whose value is not *nil*.

If either condition obtains, then it is not an error for an argument pair to match no parameter specified, and the argument pair is simply ignored (but such an argument pair is accessible through the `&rest` parameter if one was specified). The purpose of these mechanisms is to allow sharing of argument lists among several functions and to allow either the caller or the called function to specify that such sharing may be taking place.

After all parameter specifiers have been processed, the auxiliary variable specifiers (those following the lambda-list keyword `&aux`) are processed from left to right. For each one, the *initform* is evaluated and the variable *var* bound to that value (or to *nil* if no *initform* was specified). Nothing can be done with `&aux` variables that cannot be done with the special form `let*`:

```
(lambda (x y &aux (a (car x)) (b 2) c) . . . )
≡ (lambda (x y) (let* ((a (car x)) (b 2) c) . . . ))
```

Which to use is purely a matter of style.

Whenever any *initform* is evaluated for any parameter specifier, that form may refer to any parameter variable to the left of the specifier in which the *initform* appears, including any supplied-*p* variables, and may rely on the fact that no other parameter variable has yet been bound (including its own parameter variable).

Once the lambda-list has been processed, the forms in the body of the lambda-expression are executed. These forms may refer to the arguments to the function by using the names of the parameters. On exit from the function, either by a normal return of the function's value(s) or by a non-local exit, the parameter bindings, whether lexical or special, are no longer in effect. (The bindings are not necessarily permanently discarded, for a lexical binding can later be reinstated if a "closure" over that binding was created, perhaps by using `function`, and saved before the exit occurred.)

Examples of &optional and &rest parameters:

```

((lambda (a b) (+ a (* b 3))) 4 5) ⇒ 19
((lambda (a &optional (b 2)) (+ a (* b 3))) 4 5) ⇒ 19
((lambda (a &optional (b 2)) (+ a (* b 3))) 4) ⇒ 10
((lambda (&optional (a 2 b) (c 3 d) &rest x) (list a b c d x)))
  ⇒ (2 nil 3 nil nil)
((lambda (&optional (a 2 b) (c 3 d) &rest x) (list a b c d x))
 6)
  ⇒ (6 t 3 nil nil)
((lambda (&optional (a 2 b) (c 3 d) &rest x) (list a b c d x))
 6 3)
  ⇒ (6 t 3 t nil)
((lambda (&optional (a 2 b) (c 3 d) &rest x) (list a b c d x))
 6 3 8)
  ⇒ (6 t 3 t (8))
((lambda (&optional (a 2 b) (c 3 d) &rest x) (list a b c d x))
 6 3 8 9 10 11)
  ⇒ (6 t 3 t (8 9 10 11))

```

Examples of &key parameters:

```

((lambda (a b &key c d) (list a b c d)) 1 2)
  ⇒ (1 2 nil nil)
((lambda (a b &key c d) (list a b c d)) 1 2 :c 6)
  ⇒ (1 2 6 nil)
((lambda (a b &key c d) (list a b c d)) 1 2 :d 8)
  ⇒ (1 2 nil 8)
((lambda (a b &key c d) (list a b c d)) 1 2 :c 6 :d 8)
  ⇒ (1 2 6 8)
((lambda (a b &key c d) (list a b c d)) 1 2 :d 8 :c 6)
  ⇒ (1 2 6 8)
((lambda (a b &key c d) (list a b c d)) :a 1 :d 8 :c 6)
  ⇒ (:a 1 6 8)
((lambda (a b &key c d) (list a b c d)) :a :b :c :d)
  ⇒ (:a :b :d nil)

```

Examples of mixtures:

```

((lambda (a &optional (b 3) &rest x &key c (d a))
  (list a b c d x))
 1) ⇒ (1 3 nil 1 ())

((lambda (a &optional (b 3) &rest x &key c (d a))
  (list a b c d x))
 1 2) ⇒ (1 2 nil 1 ())

((lambda (a &optional (b 3) &rest x &key c (d a))
  (list a b c d x))
 :c 7) ⇒ (:c 7 nil :c ())

```

```
((lambda (a &optional (b 3) &rest x &key c (d a))
  (list a b c d x))
1 6 :c 7) ⇒ (1 6 7 1 (:c 7))
```

```
((lambda (a &optional (b 3) &rest x &key c (d a))
  (list a b c d x))
1 6 :d 8) ⇒ (1 6 nil 8 (:d 8))
```

```
((lambda (a &optional (b 3) &rest x &key c (d a))
  (list a b c d x))
1 6 :d 8 :c 9 :d 10) ⇒ (1 6 9 8 (:d 8 :c 9 :d 10))
```

All lambda-list keywords are permitted, but not terribly useful, in lambda-expressions appearing explicitly as the first element of a function-call form. They are extremely useful, however, in functions given global names by `defun`.

All symbols whose names begin with `&` are conventionally reserved for use as lambda-list keywords and should not be used as variable names. Implementations of COMMON LISP are free to provide additional lambda-list keywords.

`lambda-list-keywords`

[*Constant*]

The value of `lambda-list-keywords` is a list of all the lambda-list keywords used in the implementation, including the additional ones used only by `defmacro`. This list must contain at least the symbols `&optional`, `&rest`, `&key`, `&allow-other-keys`, `&aux`, `&body`, `&whole`, and `&environment`.

As an example of the use of `&allow-other-keys` and `:allow-other-keys`, consider a function that takes two keyword arguments of its own and also accepts additional keyword arguments to be passed to `make-array`:

```
(defun array-of-strings (str dims &rest keyword-pairs
  &key (start 0) end &allow-other-keys)
  (apply #'make-array dims
    :initial-element (subseq str start end)
    :allow-other-keys t
    keyword-pairs))
```

This function takes a string and dimensioning information and returns an array of the specified dimensions, each of whose elements is the specified string. However, `:start` and `:end` keyword arguments may be used in the usual manner (see Chapter 14) to specify that a substring of the given string should be used. In addition, the presence of `&allow-other-keys` in the lambda-list indicates that the caller may specify additional keyword arguments; the `&rest` argument provides access to them. These additional keyword arguments are fed to `make-array`. Now `make-array` normally does not allow the keywords `:start` and `:end` to be used, and it would be an error to specify such keyword arguments to `make-array`. However, the presence in the call to `make-array` of the keyword argument `:allow-other-keys` with a non-`nil` value causes any extraneous keyword arguments, including `:start` and `:end`, to be acceptable and ignored.

lambda-parameters-limit

[Constant]

The value of lambda-parameters-limit is a positive integer that is the upper exclusive bound on the number of distinct parameter names that may appear in a single lambda-list. This bound depends on the implementation but will not be smaller than 50. Implementors are encouraged to make this limit as large as practicable without sacrificing performance. See call-arguments-limit.

Top-Level Forms

The standard way for the user to interact with a COMMON LISP implementation is via a *read-eval-print loop*: the system repeatedly reads a form from some input source (such as a keyboard or a disk file), evaluates it, and then prints the value(s) to some output sink (such as a display screen or another disk file). Any form (evaluable data object) is acceptable; however, certain special forms are specifically designed to be convenient for use as *top-level* forms, rather than as forms embedded within other forms in the way that `(+ 3 4)` is embedded within `(if p (+ 3 4) 6)`. These top-level special forms may be used to define globally-named functions, to define macros, to make declarations, and to define global values for special variables.

It is not illegal to use these forms at other than top level, but whether it is meaningful to do so depends on context. Compilers, for example, may not recognize these forms properly in other than top-level contexts. (As a special case, however, if a `progn` form appears at top level, then all forms within that `progn` are considered by the compiler to be top-level forms.)

Compatibility note: In MACLISP, a top-level `progn` is considered to contain top-level forms only if the first form is `(quote compile)`. This odd marker is unnecessary in COMMON LISP.

Macros are usually defined by using the special form `defmacro`. This facility is fairly complicated, and is described in Chapter 8.

Defining Named Functions

The `defun` special form is the usual means of defining named functions.

```
defun name lambda-list [declaration | doc-string]* form* [Macro]
```

Evaluating a `defun` form causes the symbol *name* to be a global name for the function specified by the lambda-expression

```
(lambda lambda-list [declaration | doc-string]* form*)
```

defined in the lexical environment in which the `defun` form was executed. Because `defun` forms normally appear at top level, this is normally the null lexical environment.

If the optional documentation string *doc-string* is present, then it is attached to the *name* as a documentation string of type function; see documentation. If *doc-string* is not followed by a declaration, it may be present only if at least one *form* is also specified, as it is otherwise taken to be a *form*. It is an error if more than one *doc-string* is present.

The *forms* constitute the body of the defined function; they are executed as an implicit progn.

The body of the defined function is implicitly enclosed in a block construct whose name is the same as the *name* of the function. Therefore return-from may be used to exit from the function.

Other implementation-dependent bookkeeping actions may be taken as well by defun. The *name* is returned as the value of the defun form. For example:

```
(defun discriminant (a b c)
  (declare (number a b c))
  "Compute the discriminant for a quadratic equation.
  Given a, b, and c, the value b^2-4*a*c is calculated.
  The quadratic equation a*x^2+b*x+c=0 has real, multiple, or complex roots depending on
  whether this calculated value is positive, zero, or negative, respectively."
  (- (* b b) (* 4 a c)))
⇒ discriminant
and now (discriminant 1 2/3 -2) ⇒ 76/9
```

It is permissible to use defun to redefine a function, to install a corrected version of an incorrect definition, for example! It is permissible to redefine a macro as a function. It is an error to attempt to redefine the name of a special form (see Table 5-1) as a function.

Declaring Global Variables and Named Constants

The defvar and defparameter special forms are the usual means of specifying globally-defined variables. The defconstant special form is used for defining named constants.

```
defvar name [initial-value [documentation]] [Macro]
defparameter name initial-value [documentation] [Macro]
defconstant name initial-value [documentation] [Macro]
```

defvar is the recommended way to declare the use of a special variable in a program.

```
(defvar variable)
```

proclaims *variable* to be special (see proclaim), and may perform other system-dependent book-keeping actions. If a second "argument" is supplied,

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

then *variable* is initialized to the result of evaluating the form *initial-value* unless it already has a value. The *initial-value* form is not evaluated unless it is used; this fact is useful if evaluation of the *initial-value* form does something expensive like creating a large data structure. The

initialization is performed by assignment, and so assigns a global value to the variable unless there are currently special bindings of that variable. Normally there should not be any such special bindings.

`defvar` also provides a good place to put a comment describing the meaning of the variable, whereas an ordinary special proclamation offers the temptation to declare several variables at once and not have room to describe them all.

```
(defvar *visible-windows* 0
  "Number of windows at least partially visible on the screen")
```

`defparameter` is similar to `defvar`, but `defparameter` requires an *initial-value* form, always evaluates the form, and assigns the result to the variable. The semantic distinction is that `defvar` is intended to declare a variable changed by the program, whereas `defparameter` is intended to declare a variable that is normally constant but can be changed (possibly at run time), where such a change is considered a change *to* the program. `defparameter` therefore does not indicate that the quantity *never* changes; in particular, it does not license the compiler to build assumptions about the value into programs being compiled.

`defconstant` is like `defparameter` but *does* assert that the value of the variable *name* is fixed and does license the compiler to build assumptions about the value into programs being compiled. (However, if the compiler chooses to replace references to the name of the constant by the value of the constant in code to be compiled, perhaps in order to allow further optimization, the compiler must take care that such "copies" appear to be eq to the object that is the actual value of the constant. For example, the compiler may freely make copies of numbers but must exercise care when the value is a list.)

It is an error if there are any special bindings of the variable at the time the `defconstant` form is executed (but implementations may or may not check for this).

Once a name has been declared by `defconstant` to be constant, any further assignment to or binding of that special variable is an error. This is the case for such system-supplied constants as `t` and `most-positive-fixnum`. A compiler may also choose to issue warnings about bindings of the lexical variable of the same name.

For any of these constructs, the documentation should be a string. The string is attached to the name of the variable, parameter, or constant under the variable documentation type; see the documentation function.

These constructs are normally used only as top-level forms. The value returned by each of these constructs is the *name* declared.

Control of Time of Evaluation

The `eval-when` special form allows pieces of code to be executed only at compile time, only at load time, or when interpreted but not compiled. Its uses are relatively esoteric.

```
eval-when ({situation}*) {form}* [Special Form]
```

The body of an `eval-when` form is processed as an implicit `progn`, but only in the situations listed. Each *situation* must be a symbol, either `compile`, `load`, or `eval`.

`eval` specifies that the interpreter should process the body. `compile` specifies that the compiler should evaluate the body at compile time in the compilation context. `load` specifies that the compiler should arrange to evaluate the forms in the body when the compiled file containing the `eval-when` form is loaded.

The `eval-when` construct may be more precisely understood in terms of a model of how the compiler processes forms in a file to be compiled. Successive forms are read from the file using the function `read`. These top-level forms are normally processed in what we shall call *not-compile-time* mode. There is another mode called *compile-time-too* mode. The `eval-when` special form controls which of these two modes to use.

Every form is processed as follows:

- If the form is an `eval-when` form:
 - If the situation `load` is specified:
 - If the situation `compile` is also specified, *or* if the current processing mode is *compile-time-too* and the situation `eval` is also specified, then process each of the forms in the body in *compile-time-too* mode.
 - Otherwise, process each of the forms in the body in *not-compile-time* mode.
 - If the situation `load` is not specified:
 - If the situation `compile` is specified, *or* if the current processing mode is *compile-time-too* and the situation `eval` is specified, then evaluate each of the forms in the body in the compiler's executing environment.
 - Otherwise, ignore the `eval-when` form entirely.
- If the form is not an `eval-when` form, then do two things. First, if the current processing mode is *compile-time-too* mode, then evaluate the form in the compiler's executing environment. Second, perform normal compiler processing of the form (compiling functions defined by `defun` forms, and so on).

One example of the use of `eval-when` is that if the compiler is to be able to properly read a file that uses user-defined reader macro characters, it is necessary to write

```
(eval-when (compile load eval)
  (set-macro-character #\$ #'(lambda (stream char)
    (declare (ignore char))
    (list 'dollar (read stream))))))
```

This causes the call to `set-macro-character` to be executed in the compiler's execution environment, thereby modifying its reader syntax table.

End of Chapter

Chapter 6

Predicates

A *predicate* is a function that tests for some condition involving its arguments and returns `nil` if the condition is false, or some non-`nil` value if the condition is true. One may think of a predicate as producing a Boolean value, where `nil` stands for *false* and anything else stands for *true*. Conditional control structures such as `cond`, `if`, `when`, and `unless` test such Boolean values. We say that a predicate *is true* when it returns a non-`nil` value, and *is false* when it returns `nil`; that is, it is true or false according to whether the condition being tested is true or false.

By convention, the names of predicates usually end in the letter `p` (which stands for “predicate”). `COMMON LISP` uses a uniform convention in hyphenating names of predicates. If the name of the predicate is formed by adding a `p` to an existing name, such as the name of a data type, a hyphen is placed before the final `p` if and only if there is a hyphen in the existing name. For example, `number` begets `numberp` but `standard-char` begets `standard-char-p`. On the other hand, if the name of a predicate is formed by adding a prefixing qualifier to the front of an existing predicate name, the two names are joined with a hyphen and the presence or absence of a hyphen before the final `p` is not changed. For example, the predicate `string-lessp` has no hyphen before the `p` because it is the string version of `lessp` (a `MACLISP` function that has been renamed `<` in `COMMON LISP`). The name `string-less-p` would incorrectly imply that it is a predicate that tests for a kind of object called a `string-less`, and the name `stringlessp` would connote a predicate that tests whether something has no strings (is “stringless”)!

The control structures that test Boolean values only test for whether or not the value is `nil`, which is considered to be false. Any other value is considered to be true. Often a predicate will return `nil` if it “fails” and some *useful* value if it “succeeds”; such a function can be used not only as a test but also for the useful value provided in case of success. An example is `member`.

If no better non-`nil` value is available for the purpose of indicating success, by convention the symbol `t` is used as the “standard” true value.

Logical Values

The names `nil` and `t` are constants in `COMMON LISP`. Although they are symbols like any other symbols, and appear to be treated as variables when evaluated, it is not permitted to modify their values. See `defconstant`.

`nil`

[*Constant*]

The value of `nil` is always `nil`. This object represents the logical *false* value and also the empty list. It can also be written `()`.

t

[Constant]

The value of t is always t.

Data Type Predicates

Perhaps the most important predicates in LISP are those that deal with data types; that is, given a data object one can determine whether or not it belongs to a given type, or one can compare two type specifiers.

General Type Predicates

If a data type is viewed as the set of all objects belonging to the type, then the `typep` function is a set membership test, while `subtypep` is a subset test.

`typep object type`

[Function]

`typep` is a predicate that is true if *object* is of type *type*, and is false otherwise. Note that an object can be "of" more than one type, since one type can include another. The *type* may be any of the type specifiers mentioned in Chapter 4 *except* that it may not be or contain a type specifier list whose first element is *function* or *values*. A specifier of the form *(satisfies fn)* is handled simply by applying the function *fn* to *object* (see `funcall`); the *object* is considered to be of the specified type if the result is not `nil`.

`subtypep type1 type2`

[Function]

The arguments must be type specifiers that are acceptable to `typep`. The two type specifiers are compared; this predicate is true if *type1* is definitely a (not necessarily proper) subtype of *type2*. If the result is `nil`, however, then *type1* may or may not be a subtype of *type2* (sometimes it is impossible to tell, especially when *satisfies* type specifiers are involved). A second returned value indicates the certainty of the result; if it is true, then the first value is an accurate indication of the subtype relationship. Thus there are three possible result combinations:

t t	<i>type1</i> is definitely a subtype of <i>type2</i>
nil t	<i>type1</i> is definitely not a subtype of <i>type2</i>
nil nil	<code>subtypep</code> could not determine the relationship

Specific Data Type Predicates

The following predicates test for individual data types.

`null object`

[Function]

`null` is true if its argument is `()`, and otherwise is false. This is the same operation performed by the function `not`; however, `not` is normally used to invert a Boolean value, whereas `null` is normally used to test for an empty list. The programmer can therefore express *intent* by the choice of function name.

`(null x) ≡ (typep x 'null) ≡ (eq x '())`

symbolp *object*

[Function]

symbolp is true if its argument is a symbol, and otherwise is false.

(symbolp x) ≡ (typep x 'symbol)

Compatibility note: The INTERLISP equivalent of symbolp is called litatom.

atom *object*

[Function]

The predicate atom is true if its argument is not a cons, and otherwise is false. Note that (atom '()) is true, because () ≡ nil.

(atom x) ≡ (typep x 'atom) ≡ (not (typep x 'cons))

Compatibility note: In some LISP dialects, notably INTERLISP, only symbols and numbers are considered to be atoms; arrays and strings are considered to be neither atoms nor lists (conses).

consp *object*

[Function]

The predicate consp is true if its argument is a cons, and otherwise is false. Note that the empty list is not a cons, so (consp '()) ≡ (consp 'nil) ⇒ nil.

(consp x) ≡ (typep x 'cons) ≡ (not (typep x 'atom))

Compatibility note: Some LISP implementations call this function pairp or listp. The name pairp was rejected for COMMON LISP because it emphasizes too strongly the dotted-pair notion rather than the usual usage of conses in lists. On the other hand, listp too strongly implies that the cons is in fact part of a list, which after all it might not be; moreover, () is a list, though not a cons. The name consp seems to be the appropriate compromise.

listp *object*

[Function]

listp is true if its argument is a cons or the empty list (), and otherwise is false. It does not check for whether the list is a "true list" (one terminated by nil) or a "dotted list" (one terminated by a non-null atom).

(listp x) ≡ (typep x 'list) ≡ (typep x '(or cons null))

`integerp` *object* [Function]

`integerp` is true if its argument is any kind of number, and otherwise is false.

`(integerp x)` \equiv `(typep x 'number)`

`integerp` *object* [Function]

`integerp` is true if its argument is an integer, and otherwise is false.

`(integerp x)` \equiv `(typep x 'integer)`

Compatibility note: In MACLISP this is called `fixp`. Users have been confused as to whether this meant `integerp` or `fixnum`, and so the name `integerp` has been adopted here.

`rationalp` *object* [Function]

`rationalp` is true if its argument is a rational number (a ratio or an integer), and otherwise is false.

`(rationalp x)` \equiv `(typep x 'rational)`

`floatp` *object* [Function]

`floatp` is true if its argument is a floating-point number, and otherwise is false.

`(floatp x)` \equiv `(typep x 'float)`

`complexp` *object* [Function]

`complexp` is true if its argument is a complex number, and otherwise is false.

`(complexp x)` \equiv `(typep x 'complex)`

`characterp` *object* [Function]

`characterp` is true if its argument is a character, and otherwise is false.

`(characterp x)` \equiv `(typep x 'character)`

`stringp` *object* [Function]

`stringp` is true if its argument is a string, and otherwise is false.

`(stringp x)` \equiv `(typep x 'string)`

`bit-vector-p` *object* [Function]

`bit-vector-p` is true if its argument is a bit-vector, and otherwise is false.

`(bit-vector-p x) ≡ (typep x 'bit-vector)`

`vectorp` *object* [Function]

`vectorp` is true if its argument is a vector, and otherwise is false.

`(vectorp x) ≡ (typep x 'vector)`

`simple-vector-p` *object* [Function]

`simple-vector-p` is true if its argument is a simple general vector, and otherwise is false.

`(simple-vector-p x) ≡ (typep x 'simple-vector)`

`simple-string-p` *object* [Function]

`simple-string-p` is true if its argument is a simple string, and otherwise is false.

`(simple-string-p x) ≡ (typep x 'simple-string)`

`simple-bit-vector-p` *object* [Function]

`simple-bit-vector-p` is true if its argument is a simple-bit-vector, and otherwise is false.

`(simple-bit-vector-p x) ≡ (typep x 'simple-bit-vector)`

`arrayp` *object* [Function]

`arrayp` is true if its argument is an array, and otherwise is false.

`(arrayp x) ≡ (typep x 'array)`

`packagep` *object* [Function]

`packagep` is true if its argument is a package, and otherwise is false.

`(packagep x) ≡ (typep x 'package)`

`functionp` *object* [Function]

`functionp` is true if its argument is suitable for applying to arguments, using for example the `funcall` or `apply` function. Otherwise `functionp` is false.

functionp is always true of symbols, lists whose *car* is the symbol *lambda*, any value returned by the function special form, and any values returned by the function *compile* when the first argument is *nil*.

compiled-function-p *object* [Function]

compiled-function-p is true if its argument is any compiled code object, and otherwise is false.

(compiled-function-p x) ≡ (typep x 'compiled-function)

commonp *object* [Function]

commonp is true if its argument is any standard COMMON LISP data type, and otherwise is false.

(commonp x) ≡ (typep x 'common)

See also *standard-char-p*, *string-char-p*, *stream-p*, *random-state-p*, *readtable-p*, *hash-table-p*, and *pathname-p*.

Equality Predicates

COMMON LISP provides a spectrum of predicates for testing for equality of two objects: *eq* (the most specific), *eqi*, *equal*, and *equalp* (the most general). *eq* and *equal* have the meanings traditional in LISP. *eqi* was added because it is frequently needed, and *equalp* was added primarily in order to have a version of *equal* that would ignore type differences when comparing numbers and case differences when comparing characters. If two objects satisfy any one of these equality predicates, then they also satisfy all those that are more general.

eq *x* *y* [Function]

(*eq* *x* *y*) is true if and only if *x* and *y* are the same identical object. (Implementationally, *x* and *y* are usually *eq* if and only if they address the same identical memory location.)

It should be noted that things that print the same are not necessarily *eq* to each other. Symbols with the same print name usually are *eq* to each other because of the use of the *intern* function. However, numbers with the same value need not be *eq*, and two similar lists are usually not *eq*. For example:

(*eq* 'a 'b) is false.
 (*eq* 'a 'a) is true.
 (*eq* 3 3) might be true or false, depending on the implementation.
 (*eq* 3 3.0) is false.
 (*eq* 3.0 3.0) might be true or false, depending on the implementation.
 (*eq* #c(3 -4) #c(3 -4)) might be true or false, depending on the implementation.
 (*eq* #c(3 -4.0) #c(3 -4)) is false.
 (*eq* (cons 'a 'b) (cons 'a 'c)) is false.
 (*eq* (cons 'a 'b) (cons 'a 'b)) is false.
 (*eq* '(a . b) '(a . b)) might be true or false.
 (progn (setq x (cons 'a 'b)) (*eq* x x)) is true.

(progn (setq x '(a . b)) (eq x x)) is true.
 (eq #\A #\A) might be true or false, depending on the implementation.
 (eq "Foo" "Foo") might be true or false.
 (eq "Foo" (copy-seq "Foo")) is false.
 (eq "FOO" "foo") is false.

In COMMON LISP, unlike some other LISP dialects, the implementation is permitted to make "copies" of characters and numbers at any time. (This permission is granted because it allows tremendous performance improvements in many common situations.) The net effect is that COMMON LISP makes no guarantee that eq will be true even when both its arguments are "the same thing" if that thing is a character or number. For example:

(let ((x 5)) (eq x x)) might be true or false.

The predicate eql is the same as eq, except that if the arguments are characters or numbers of the same type then their values are compared. Thus eql tells whether two objects are *conceptually* the same, whereas eq tells whether two objects are *implementationally* identical. It is for this reason that eql, not eq, is the default comparison predicate for the sequence functions defined in Chapter 14.

Implementation note: eq simply compares the two given pointers, so any kind of object that is represented in an "immediate" fashion will indeed have like-valued instances satisfy eq. In some implementations, for example, fixnums and characters happen to "work." However, no program should depend on this, as other implementations of COMMON LISP might not use an immediate representation for these data types.

An additional problem with eq is that the implementation is permitted to "collapse" constants (or portions thereof) appearing in code to be compiled if they are equal. An object is considered to be a constant in code to be compiled if it is self-evaluating form or is contained in a quote form. This is why (eq "Foo" "Foo") might be true or false; in interpreted code it would normally be false, because reading in the form (eq "Foo" "Foo") would construct distinct strings for the two arguments to eq, but the compiler might choose to use the same identical string or two distinct copies as the two arguments in the call to eq. Similarly, (eq '(a . b) '(a . b)) might be true or false, depending on whether the constant conses appearing in the quote forms were collapsed by the compiler. However, (eq (cons 'a 'b) (cons 'a 'b)) is always false, because every distinct call to the cons function necessarily produces a new and distinct cons.

eql x y

[Function]

The eql predicate is true if its arguments are eq, or if they are numbers of the same type with the same value, or if they are character objects that represent the same character. An example follows.

(eql 'a 'b) is false.
 (eql 'a 'a) is true.
 (eql 3 3) is true.
 (eql 3 3.0) is false.
 (eql 3.0 3.0) is true.

```

(eql #c(3 -4) #c(3 -4)) is true.
(eql #c(3 -4.0) #c(3 -4)) is false.
(eql (cons 'a 'b) (cons 'a 'c)) is false.
(eql (cons 'a 'b) (cons 'a 'b)) is false.
(eql '(a . b) '(a . b)) might be true or false.
(progn (setq x (cons 'a 'b)) (eql x x)) is true.
(progn (setq x '(a . b)) (eql x x)) is true.
(eql #\A #\A) is true.
(eql "Foo" "Foo") might be true or false.
(eql "Foo" (copy-seq "Foo")) is false.
(eql "FOO" "foo") is false.

```

Normally (eql 1.0s0 1.0d0) would be false, under the assumption that 1.0s0 and 1.0d0 are of distinct data types. However, implementations that do not provide four distinct floating-point formats are permitted to “collapse” the four formats into some smaller number of them; in such an implementation (eql 1.0s0 1.0d0) might be true. The predicate = will compare the values of two numbers even if the numbers are of different types.

If an implementation supports positive and negative zeros as distinct values (as in the IEEE proposed standard floating-point format), then (eql 0.0 -0.0) will be false. Otherwise, when the syntax -0.0 is read it will be interpreted as the value 0.0, and so (eql 0.0 -0.0) will be true. The predicate = differs from eql in that (= 0.0 -0.0) will always be true, because = compares the mathematical values of its operands, whereas eql compares the representational values, so to speak.

Two complex numbers are considered to be eql if their real parts are eql and their imaginary parts are eql. For example, (eql #C(4 5) #C(4 5)) is true and (eql #C(4 5) #C(4.0 5.0)) is false. Note that while (eql #C(5.0 0.0) 5.0) is false, (eql #C(5 0) 5) is true. In the case of (eql #C(5.0 0.0) 5.0) the two arguments are of different types, and so cannot satisfy eql; that’s all there is to it. In the case of (eql #C(5 0) 5), however, #C(5 0) is not a complex number, but is always automatically reduced by the rule of complex canonicalization to the integer 5, just as the apparent ration 20/4 is always simplified to 5.

The case of (eql "Foo" "Foo") is discussed above in the description of eq. While eql compares the values of numbers and characters, it does not compare the contents of strings. To compare the characters of two strings, one should use equal, equalp, string=, or string-equal.

Compatibility note: The COMMON LISP function eql is similar to the INTERLISP function eqp. However, eql considers 3 and 3.0 to be different, whereas eqp considers them to be the same; eqp behaves like the COMMON LISP = function, not like eql, when both arguments are numbers.

equal x y

[Function]

The equal predicate is true if its arguments are structurally similar (isomorphic) objects. A rough rule of thumb is that two objects are equal if and only if their printed representations are the same.

Numbers and characters are compared as for `eq`. Symbols are compared as for `eq`. This method of comparing symbols can violate the rule of thumb for `equal` and printed representations, but only in the infrequently occurring case of two distinct symbols with the same print name.

Certain objects that have components are `equal` if they are of the same type and corresponding components are `equal`. This test is implemented in a recursive manner and may fail to terminate for circular structures.

For conses, `equal` is defined recursively as the two `car`'s being `equal` and the two `cdr`'s being `equal`.

Two arrays are `equal` only if they are `eq`, with one exception: strings and bit-vectors are compared element-by-element. If either argument has a fill pointer, the fill pointer limits the number of elements examined by `equal`. Uppercase and lowercase letters in strings are considered by `equal` to be distinct. (In contrast, `equalp` ignores case distinctions in strings.)

Compatibility note: In ZETALISP, `equal` ignores the difference between uppercase and lowercase letters in strings. This violates the rule of thumb about printed representations, however, which is very useful, especially to novices. It is also inconsistent with the treatment of single characters, which in ZETALISP are represented as `fixnums`.

Two pathname objects are `equal` if and only if all the corresponding components (host, device, and so on) are equivalent. (Whether or not uppercase and lowercase letters are considered equivalent in strings appearing in components depends on the file name conventions of the file system.) Pathnames that are `equal` should be functionally equivalent.

```
(equal 'a 'b) is false.
(equal 'a 'a) is true.
(equal 3 3) is true.
(equal 3 3.0) is false.
(equal 3.0 3.0) is true.
(equal #c(3 -4) #c(3 -4)) is true.
(equal #c(3 -4.0) #c(3 -4)) is false.
(equal (cons 'a 'b) (cons 'a 'c)) is false.
(equal (cons 'a 'b) (cons 'a 'b)) is true.
(equal '(a . b) '(a . b)) is true.
(progn (setq x (cons 'a 'b)) (equal x x)) is true.
(progn (setq x '(a . b)) (equal x x)) is true.
(equal #\A #\A) is true.
(equal "Foo" "Foo") is true.
(equal "Foo" (copy-seq "Foo")) is true.
(equal "FOO" "foo") is false.
```

To compare a tree of conses, using `eq` (or any other desired predicate) on the leaves, use `tree-equal`.

`equalp x y`

[Function]

Two objects are `equalp` if they are `equal`; if they are characters and satisfy `char-equal`, which ignores alphabetic case and certain other attributes of characters; if they are numbers and have the same numerical value, even if they are different types; or if they have components that are all `equalp`.

Objects that have components are `equalp` if they are of the same type and corresponding components are `equalp`. This test is implemented in a recursive manner and may fail to terminate for circular structures. For conses, `equalp` is defined recursively as the two *car*'s being `equalp` and the two *cdr*'s being `equalp`.

Two arrays are `equalp` if and only if they have the same number of dimensions, the dimensions match, and the corresponding components are `equalp`. The specializations need not match; for example, a string and a general array that happens to contain the same characters will be `equalp` (though definitely not `equal`). If either argument has a fill pointer, the fill pointer limits the number of elements examined by `equalp`. Because `equalp` performs element-by-element comparisons of strings and ignores the alphabetic case of characters, case distinctions are therefore also ignored when `equalp` compares strings.

Two symbols can be `equalp` only if they are `eq`, that is, the same identical object.

```
(equalp 'a 'b) is false.
(equalp 'a 'a) is true.
(equalp 3 3) is true.
(equalp 3 3.0) is true.
(equalp 3.0 3.0) is true.
(equalp #c(3 -4) #c(3 -4)) is true.
(equalp #c(3 -4.0) #c(3 -4)) is true.
(equalp (cons 'a 'b) (cons 'a 'c)) is false.
(equalp (cons 'a 'b) (cons 'a 'b)) is true.
(equalp '(a . b) '(a . b)) is true.
(progn (setq x (cons 'a 'b)) (equalp x x)) is true.
(progn (setq x '(a . b)) (equalp x x)) is true.
(equalp #\A #\A) is true.
(equalp "Foo" "Foo") is true.
(equalp "Foo" (copy-seq "Foo")) is true.
(equalp "FOO" "foo") is true.
```

Logical Operators

COMMON LISP provides three operators on Boolean values: `and`, `or`, and `not`. Of these, `and` and `or` are also control structures because their arguments are evaluated conditionally. The function `not` necessarily examines its single argument, and so is a simple function.

`not x`

[Function]

`not` returns `t` if `x` is `nil`, and otherwise returns `nil`. It therefore inverts its argument considered as a Boolean value.

nil is the same as not; both functions are included for the sake of clarity. As a matter of style, it is customary to use nil to check whether something is the empty list and to use not to invert the sense of a logical value.

and *{form}** [Macro]

(and *form1 form2 . . .*) evaluates each *form*, one at a time, from left to right. If any *form* evaluates to nil the value nil is immediately returned without evaluating the remaining *forms*. If every *form* but the last evaluates to a non-nil value, and returns whatever the last *form* returns. Therefore in general and can be used both for logical operations, where nil stands for *false* and non-nil values stand for *true*, and as a conditional expression. An example follows.

```
(if (and (>= n 0)
        (< n (length a-simple-vector))
        (eq (elt a-simple-vector n) 'foo))
    (princ "Foo!"))
```

The above expression prints Foo! if element *n* of *a-simple-vector* is the symbol *foo*, provided also that *n* is indeed a valid index for *a-simple-vector*. Because and guarantees left-to-right testing of its parts, elt is not called if *n* is out of range.

To put it another way, the and macro does *short-circuit* Boolean evaluation, like the **and** then operator in ADA and what in some PASCAL-like languages is called **cand** (for "conditional and"); the LISP and macro is unlike the PASCAL or ADA **and** operator, which always evaluates both arguments.

In the previous example writing

```
(and (>= n 0)
      (< n (length a-simple-vector))
      (eq (elt a-simple-vector n) 'foo))
      (princ "Foo!"))
```

would accomplish the same thing. The difference is purely stylistic. Some programmers never use expressions containing side effects within and, preferring to use if or when for that purpose.

From the general definition, one can deduce that (and *x*) ≡ *x*. Also, (and) evaluates to t, which is an identity for this operation.

One can define and in terms of cond in this way:

```
(and x y z . . . w) ≡ (cond ((not x) nil)
                          ((not y) nil)
                          ((not z) nil)
                          . . .
                          (t w))
```

See if and when, which are sometimes stylistically more appropriate than and for conditional purposes. If it is necessary to test whether a predicate is true of all elements of a list or vector (element 0 *and* element 1 *and* element 2 *and* . . .), then the function every may be useful.

or *{form}**

[Macro]

(or *form1 form2 . . .*) evaluates each *form*, one at a time, from left to right. If any *form* other than the last evaluates to something other than nil, or immediately returns that non-nil value without evaluating the remaining *forms*. If every *form* but the last evaluates to nil, or returns whatever evaluation of the last of the *forms* returns. Therefore in general or can be used both for logical operations, where nil stands for *false* and non-nil values stand for *true*, and as a conditional expression.

To put it another way, the or macro does *short-circuit* Boolean evaluation, like the **or else** operator in ADA and what in some PASCAL-like languages is called **cor** (for "conditional or"); the LISP or macro is unlike the PASCAL or ADA **or** operator, which always evaluates both arguments.

From the general definition, one can deduce that (or *x*) = *x*. Also, (or) evaluates to nil, which is the identity for this operation.

One can define or in terms of cond in this way:

(or *x y z . . . w*) ≡ (cond (*x*) (*y*) (*z*) . . . (t *w*))

See if and unless, which are sometimes stylistically more appropriate than or for conditional purposes. If it is necessary to test whether a predicate is true of one or more elements of a list or vector (element 0 or element 1 or element 2 or . . .), then the function some may be useful.

End of Chapter

Chapter 7

Control Structure

COMMON LISP provides a variety of special structures for organizing programs. Some have to do with flow of control (control structures), while others control access to variables (environment structures). Some of these features are implemented as special forms; other are implemented as macros, which typically expand into complex program fragments expressed in terms of special forms or other macros.

Function application is the primary method for construction of LISP 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. LISP functions may call upon themselves recursively, either directly or indirectly.

While the LISP language is more applicative in style than statement-oriented, it nevertheless provides many operations that produce side effects, and consequently requires constructs for controlling the sequencing of side effects. The construct `progn`, which is roughly equivalent to an ALGOL **begin-end** block with all its semicolons, executes a number of forms sequentially, discarding the values of all but the last. Many LISP control constructs include sequencing implicitly, in which case they are said to provide an “implicit `progn`.” Other sequencing constructs include `prog1` and `prog2`.

For looping, COMMON LISP provides the general iteration facility `do` as well as a variety of special-purpose iteration facilities for iterating or mapping over various data structures.

COMMON LISP provides the simple one-way conditionals `when` and `unless`, the simple two-way conditional `if`, and the more general multi-way conditionals such as `cond` and `case`. The choice of which form to use in any particular situation is a matter of taste and style.

Constructs for performing non-local exits with various scoping disciplines are provided: `block`, `return`, `return-from`, `catch`, and `throw`.

The multiple-value constructs provide an efficient way for a function to return more than one value; see `values`.

Constants and Variables

Because some LISP data objects are used to represent programs, one cannot always notate a constant data object in a program simply by writing the notation for the object unadorned; it would be ambiguous whether a constant object or a program fragment was intended. The quote special form resolves this ambiguity.

There are two spaces of variables in COMMON LISP, in effect: ordinary variables and function names. There are some similarities between the two kinds, and in a few cases there are similar functions for dealing with them, for example `boundp` and `fboundp`. However, for the most part the two kinds of variables are used for very different purposes: one to name defined functions, macros, and special forms, and the other to name data objects.

Reference

The value of an ordinary variable may be obtained simply by writing the name of the variable as a form to be executed. Whether this is treated as the name of a special variable or a lexical variable is determined by the presence or absence of an applicable special declaration; see Chapter 9.

The following functions and special forms allow reference to the values of constants and variables in other ways.

`quote` *object* [Special form]

`(quote x)` simply returns *x*. The *object* is not evaluated and may be any LISP object whatsoever. This construct allows any LISP object to be written as a constant value in a program. For example:

```
(setq a 43)
(list a (cons a 3)) ⇒ (43 (43 . 3))
(list (quote a) (quote (cons a 3))) ⇒ (a (cons a 3))
```

Since `quote` forms are so frequently useful but somewhat cumbersome to type, a standard abbreviation is defined for them: any form *f* preceded by a single quote (`'`) character is assumed to have `(quote)` wrapped around it to make `(quote f)`. For example:

```
(setq x '(the magic quote hack))
```

is normally interpreted by `read` to mean

```
(setq x (quote (the magic quote hack)))
```

See “Macro Characters,” Chapter 22.

`function` *fn* [Special form]

The value of `function` is always the functional interpretation of *fn*; *fn* is interpreted as if it had appeared in the functional position of a function invocation. In particular, if *fn* is a symbol, the functional definition associated with that symbol is returned; see `symbol-function`. If *fn* is a lambda-expression, then a “lexical closure” is returned, that is, a function that when invoked will execute the body of the lambda-expression in such a way as to observe the rules of lexical scoping properly. For example:

```
(defun adder (x) (function (lambda (y) (+ x y))))
```

The result of `(adder 3)` is a function that will add 3 to its argument:

```
(setq add3 (adder 3))
(funcall add3 5) ⇒ 8
```

This works because `function` creates a closure of the inner lambda-expression that is able to refer to the value 3 of the variable `x` even after control has returned from the function `adder`.

More generally, a lexical closure in effect retains the ability to refer to lexically visible *bindings*, not just values. Consider this code:

```
(defun two-funs (x)
  (list (function (lambda () x))
        (function (lambda (y) (setq x y))))))
(setq funs (two-funs 6))
(funcall (car funs)) ⇒ 6
(funcall (cadr funs) 43) ⇒ 43
(funcall (car funs)) ⇒ 43
```

The function `two-funs` returns a list of two functions, each of which refers to *the binding* of the variable `x` created on entry to the function `two-funs` when it was called with argument 6. This binding has the value 6 initially, but `setq` can alter a binding. The lexical closure created for the first lambda-expression does not “snapshot” the value 6 for `x` when the closure is created. The second function can be used to alter the binding (to 43, in the example), and this altered value then becomes accessible to the first function.

In situations where a closure of a lambda-expression over the same set of bindings may be produced more than once, the various resulting closures may or may not be `eq`, at the discretion of the implementation. For example:

```
(let ((x 5) (funs '()))
  (dotimes (j 10)
    (push #'(lambda (z)
              (if (null z) (setq x 0) (+ x z)))
          funs))
  funs)
```

The result of the above expression is a list of ten closures. Each logically requires only the binding of `x`. It is the same binding in each case, so the ten closures may or may not be the same identical (`eq`) object. On the other hand, the result of the expression

```
(let ((funs '()))
  (dotimes (j 10)
    (let ((x 5))
      (push (function (lambda (z)
                        (if (null z) (setq x 0) (+ x z))))
            funs)))
  funs)
```

is also a list of ten closures. However, in this case no two of the closures may be `eq`, because each closure is over a distinct binding of `x`, and these bindings can be behaviorally distinguished because of the use of `setq`.

The question of distinguishable behavior is important; the result of the simpler expression

```
(let ((funs '()))
  (dotimes (j 10)
    (let ((x 5))
      (push (function (lambda (z) (+ x z)))
            funs)))
  funs)
```

is a list of ten closures that *may* be pairwise `eq`. Although one might think that a different binding of `x` is involved for each closure (which is indeed the case), the bindings cannot be distinguished because their values are identical and immutable, there being no occurrence of `setq` on `x`. A compiler would therefore be justified in transforming the expression to

```
(let ((funs '()))
  (dotimes (j 10)
    (push (function (lambda (z) (+ 5 z)))
          funs))
  funs)
```

where clearly the closures may be the same after all. The general rule, then, is that the implementation is free to have two distinct evaluations of the same function form produce identical (`eq`) closures if it can prove that the two conceptually distinct resulting closures must in fact be behaviorally identical with respect to invocation. This is merely a permitted optimization; a perfectly valid implementation might simply cause every distinct evaluation of a function form to produce a new closure object not `eq` to any other.

Frequently a compiler can deduce that a closure in fact does not need to close over any variable bindings. For example, in the code fragment

```
(mapcar (function (lambda (x) (+ x 2))) y)
```

the function `(lambda (x) (+ x 2))` contains no references to any outside entity. In this important special case, the same “closure” may be used as the value for all evaluations of the function special form. Indeed, this value need not be a closure object at all; it may be a simple compiled function containing no environment information. This example is simply a special case of the foregoing discussion and is included as a hint to implementors familiar with previous methods of implementing LISP. The distinction between closures and other kinds of functions is somewhat pointless, actually, as COMMON LISP defines no particular representation for closures and no way to distinguish between closure and non-closure functions. All that matters is that the rules of lexical scoping be obeyed.

Since function forms are so frequently useful but somewhat cumbersome to type, a standard abbreviation is defined for them: any form `f` preceded by `#'` (`#` followed by an apostrophe) is assumed to have function wrapped around it to make function `f`. For example,

```
(remove-if #'numberp '(1 a b 3))
```

is normally interpreted by read to mean

```
(remove-if (function numberp) '(1 a b 3))
```

See “Standard Dispatching Macro Character Syntax,” Chapter 22.

symbol-value *symbol*

[Function]

symbol-value returns the current value of the dynamic (special) variable named by *symbol*. An error occurs if the symbol has no value; see `boundp` and `makunbound`. Note that constant symbols are really variables that cannot be changed, and so symbol-value may be used to get the value of a named constant. In particular, symbol-value of a keyword will return that keyword.

symbol-value cannot access the value of a lexical variable.

This function is particularly useful for implementing interpreters for languages embedded in LISP. The corresponding assignment primitive is `set`; alternatively, symbol-value may be used with `self`.

symbol-function *symbol*

[Function]

symbol-function returns the current global function definition named by *symbol*. An error is signalled if the symbol has no function definition; see `fboundp`. Note that the definition may be a function or may be an object representing a special form or macro. In the latter case, however, it is an error to attempt to invoke the object as a function. If it is desired to process macros, special forms, and functions equally well, as when writing an interpreter, it is best first to test the symbol with `macro-function` and `special-form-p` and then to invoke the functional value only if these two tests both yield false.

This function is particularly useful for implementing interpreters for languages embedded in LISP.

symbol-function cannot access the value of a lexical function name produced by `filet` or `labels`; it can access only the global function value.

The global function definition of a symbol may be altered by using `self` with symbol-function. Performing this operation causes the symbol to have *only* the specified definition as its global function definition; any previous definition, whether as a macro or as a function, is lost. It is an error to attempt to redefine the name of a special form (see Table 5-1).

boundp *symbol*

[Function]

boundp is true if the dynamic (special) variable named by *symbol* has a value; otherwise, it returns nil.

See also `set` and `makunbound`.

fboundp *symbol*

[Function]

fboundp is true if the symbol has a global function definition. Note that fboundp is true when the symbol names a special form or macro. `macro-function` and `special-form-p` may be used to test for these cases.

See also symbol-function and `fmakunbound`.

special-form-p *symbol*

[Function]

The function `special-form-p` takes a symbol. If the symbol globally names a special form, then a non-`nil` value is returned; otherwise `nil` is returned. A returned non-`nil` value is typically a function of implementation-dependent nature that can be used to interpret (evaluate) the special form.

It is possible for *both* `special-form-p` and `macro-function` to be true of a symbol. This is possible because an implementation is permitted to implement any macro also as a special form for speed. On the other hand, the macro definition must be available for use by programs that understand only the standard special forms listed in Table 5-1.

Assignment

The following facilities allow the value of a variable (more specifically, the value associated with the current binding of the variable) to be altered. Such alteration is different from establishing a new binding. Constructs for establishing new bindings of variables are described below in “Establishing New Variable Bindings.”

`setq {var form}*`

[Special form]

The special form `(setq var1 form1 var2 form2 . . .)` is the “simple variable assignment statement” of LISP. First *form1* is evaluated and the result is stored in the variable *var1*, then *form2* is evaluated and the result stored in *var2*, and so forth. The variables are represented as symbols, of course, and are interpreted as referring to static or dynamic instances according to the usual rules. Therefore `setq` may be used for assignment of both lexical and special variables.

`setq` returns the last value assigned, that is, the result of the evaluation of its last argument. As a boundary case, the form `(setq)` is legal and returns `nil`. There must be an even number of argument forms. For example, in

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

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

See also the description of `setf`, the COMMON LISP “general assignment statement” that is capable of assigning to variables, array elements, and other locations.

`psetq {var form}*`

[Macro]

A `psetq` form is just like a `setq` form, except that the assignments happen in parallel. First all of the forms are evaluated, and then the variables are set to the resulting values. The value of the `psetq` form is `nil`. For example:

```
(setq a 1)
(setq b 2)
(psetq a b b a)
a ⇒ 2
b ⇒ 1
```

In this example, the values of *a* and *b* are exchanged by using parallel assignment. (If several variables are to be assigned in parallel in the context of a loop, the `do` construct may be appropriate.)

See also the description of `psetf`, the COMMON LISP “general parallel assignment statement” that is capable of assigning to variables, array elements, and other locations.

`set` *symbol value* [Function]

`set` allows alteration of the value of a dynamic (special) variable. `set` causes the dynamic variable named by *symbol* to take on *value* as its value. Only the value of the current dynamic binding is altered; if there are no bindings in effect, the most global value is altered. For example:

```
(set (if (eq a b) 'c 'd) 'foo)
```

will either set *c* to `foo` or set *d* to `foo`, depending on the outcome of the test `(eq a b)`.

`set` returns *value* as its result.

`set` cannot alter the value of a local (lexically bound) variable. The special form `setq` is usually used for altering the values of variables (lexical or dynamic) in programs. `set` is particularly useful for implementing interpreters for languages embedded in LISP. See also `progv`, a construct that performs binding rather than assignment of dynamic variables.

`makunbound` *symbol* [Function]
`fmakunbound` *symbol* [Function]

`makunbound` causes the dynamic (special) variable named by *symbol* to become unbound (have no value). `fmakunbound` does the analogous thing for the global function definition named by *symbol*. For example:

```
(setq a 1)
a ⇒ 1
(makunbound 'a)
a ⇒ causes an error
```

```
(defun foo (x) (+ x 1))
(foo 4) ⇒ 5
(fmakunbound 'foo)
(foo 4) ⇒ causes an error
```

Both functions return *symbol* as the result value.

Generalized Variables

In LISP, a variable can remember one piece of data, that is, one LISP object. The main operations on a variable are to recover that object, and to alter the variable to remember a new object; these operations are often called *access* and *update* operations. 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. Examples of such storage locations are the *car* and *cdr* of a cons, elements of an array, and components of a structure.

For each kind of generalized variable, typically there are two functions that implement the conceptual *access* and *update* operations. For a variable, merely mentioning the name of the variable accesses it, while the `setq` special form can be used to update it. The function `car` accesses the *car* of a cons, and the function `rplaca` updates it. The function `symbol-value` accesses the dynamic value of a variable named by a given symbol, and the function `set` updates it.

Rather than thinking about two distinct functions that respectively access and update a storage location somehow deduced from their arguments, we can instead simply think of a call to the access function with given arguments as a *name* for the storage location. Thus, just as `x` may be considered a name for a storage location (a variable), so `(car x)` is a name for the *car* of some cons (which is in turn named by `x`). Now, rather than having to remember two functions for each kind of generalized variable (having to remember, for example, that `rplaca` corresponds to `car`), we adopt a uniform syntax for updating storage locations named in this way, using the `setf` macro. 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. The uniformity of this approach is illustrated in the following table.

Access function	Update function	Update using <code>setf</code>
<code>x</code>	<code>(setq x datum)</code>	<code>(setf x datum)</code>
<code>(car x)</code>	<code>(rplaca x datum)</code>	<code>(setf (car x) datum)</code>
<code>(symbol-value-x)</code>	<code>(set x datum)</code>	<code>(setf (symbol-value x) datum)</code>

`setf` is actually a macro that examines an access form and produces a call to the corresponding update function.

Given the existence of `setf` in COMMON LISP, it is not necessary to have `setq`, `rplaca`, and `set`; they are redundant. They are retained in COMMON LISP because of their historical importance in LISP. However, most other update functions (such as `putprop`, the update function for `get`) have been eliminated from COMMON LISP in the expectation that `setf` will be uniformly used in their place.

`setf` *{place newvalue}** [Macro]

`(setf place newvalue)` takes a form *place* that when evaluated *accesses* a data object in some location and “inverts” it to produce a corresponding form to *update* the location. A call to the `setf` macro therefore expands into an update form that stores the result of evaluating the form *newvalue* into the place referred to by the access-form.

If more than one *place-newvalue* pair is specified, the pairs are processed sequentially; that is,

```
(setf place1 newvalue1
      place2 newvalue2)
      . . .
      placen newvaluen)
```

is precisely equivalent to

```
(progn (setf place1 newvalue1)
       (setf place2 newvalue2)
       . . .
       (setf placen newvaluen))
```

For consistency, it is legal to write (setf), which simply returns nil. The form *place* may be any one of the following:

- The name of a variable (either lexical or dynamic).
- A function call form whose first element is the name of any one of the following functions:

aref	car	svref	symbol-value
nth	cdr	get	symbol-function
elt	caar	getf	symbol-plist
rest	cadr	gethash	macro-function
first	cdar	documentation	cdaaar
second	cddr	fill-pointer	cdaadr
third	caaar	caaar	cdadar
fourth	caadr	caadr	cdaddr
fifth	cadar	caadar	cddaar
sixth	caddr	caaddr	cddadr
seventh	cdaar	cadaar	cdddar
eighth	cdadr	cadadr	cdddr
ninth	cddar	caddar	
tenth	cdddr	caddr	

- A function call form whose first element is the name of a selector function constructed by destruct.
- A function call form whose first element is the name of any one of the following functions, provided that the new value is of the specified type so that it can be used to replace the specified “location” (which is in each of these cases not truly a generalized variable):

Function name	Required type
char	string-char
schar	string-char
bit	bit
sbit	bit
subseq	sequence

In the case of *subseq*, the replacement value must be a sequence whose elements may be contained by the sequence argument to *subseq*. (Note that this is not so stringent as to require that the replacement value be a sequence of the same type as the sequence of which the subsequence is specified.) If the length of the replacement value does not equal the length of the subsequence to be replaced, then the shorter length determines the number of elements to be stored, as for the function *replace*.

- A function call form whose first element is the name of any one of the following functions, provided that the specific argument to that function is in turn a *place* form; in this case the new *place* has stored back into it the result of applying the specified “update” function (which is in each of these cases not a true update function):

Function name	Argument that is a <i>place</i>	Update function used
char-bit	first	set-char-bit
ldb	second	dpb
mask-field	second	deposit-field

- A the type declaration form, in which case the declaration is transferred to the *newvalue* form, and the resulting *self* form is analyzed. For example:

```
(self (the integer (cadr x)) (+ y 3))
```

is processed as if it were

```
self (cadr x) (the integer (+ y 3)))
```

- A call to *apply* where the first argument form is of the form *#'name*, that is, (function *name*) where *name* is the name of a function, calls to which are recognized as places by *self*. Suppose that the use of *self* with *apply* looks like this:

```
(self (apply #'name x1 x2 . . . xn rest) x0)
```

The *self* method for the function *name* must be such that

```
(self (name z1 z2 . . . zm) z0)
```

expands into a store form

```
(storefn zi1 zi2 . . . zik zm)
```

That is, it must expand into a function call such that all arguments but the last may be any permutation or subset of the new value *z0* and the arguments of the access form, but the *last* argument of the storing call must be the same as the last argument of the access call. See *define-self-method* for more details on accessing and storing forms.

Given this, the *self*-of-*apply* form shown above expands into

```
(apply #'storefn xi1 xi2 . . . xik rest)
```

As an example, suppose that the variable *indexes* contains a list of subscripts for a multidimensional array *foo* whose rank is not known until run time. One may access the indicated element of the array by writing

```
(apply #'aref foo indexes)
```

and one may alter the value of the indicated element to that of *newvalue* by writing

```
(self (apply #'aref foo indexes) newvalue)
```

- A macro call, in which case *self* expands the macro call and then analyzes the resulting form.
- Any form for which a *defself* or *define-self-method* declaration has been made.

self carefully arranges to preserve the usual left-to-right order in which the various subforms are evaluated. On the other hand, the exact expansion for any particular form is not guaranteed and may even be implementation-dependent; all that is guaranteed is that the expansion of a *self* form will be an update form that works for that particular implementation, and that the left-to-right evaluation of subforms is preserved.

The ultimate result of evaluating a `setf` form is the value of *newvalue*. Therefore `(setf (car x) y)` does not expand into precisely `(rplaca x y)`, but into something more like

```
(let ((G1 x) (G2 y)) (rplaca G1 G2) G2)
```

the precise expansion being implementation-dependent.

The user can define new `setf` expansions by using `defsetf`.

`psetf` *{place newvalue}** [Macro]

`psetf` is like `setf` except that if more than one *place-newvalue* pair is specified then the assignments of new values to places are done in parallel. More precisely, all subforms that are to be evaluated are evaluated from left to right; after all evaluations have been performed, all of the assignments are performed in an unpredictable order. (The unpredictability matters only if more than one *place* form refers to the same place.) `psetf` always returns `nil`.

`shifff` *{place}⁺ newvalue* [Macro]

Each *place* form may be any form acceptable as a generalized variable to `setf`. In the form `(shifff place1 place2 . . . placen newvalue)` the values in *place1* through *placen* are accessed and saved, and *newvalue* is evaluated, for a total of $n + 1$ values in all. Values 2 through $n + 1$ are then stored into *place1* through *placen*, and value 1 (the original value of *place1*) is returned. It is as if all the places form a shift register; the *newvalue* is shifted in from the right, all values shift over to the left one place, and the value shifted out of *place1* is returned. For example:

```
(setq x (list 'a 'b 'c)) ⇒ (a b c)
```

```
(shifff (cadr x) 'z) ⇒ b
  and now x ⇒ (a z c)
```

```
(shifff (cadr x) (caddr x) 'a) ⇒ z
  and now x ⇒ (a (c) . a)
```

The effect of `(shifff place1 place2 . . . placen newvalue)` is equivalent to

```
(let ((var1 place1)
      (var2 place2)
      . . .
      (varn placen))
  (setf place1 var2)
  (setf place2 var3)
  . . .
  (setf placen newvalue)
  var1)
```

except that the latter would evaluate any subforms of each *place* twice, whereas `shifff` takes care to evaluate them only once. For example:

```
(setq n 0)
(setq x '(a b c d))
(shifff (nth (setq n (+ n 1)) x) 'z) ⇒ b
  and now x ⇒ (a z c d)
```

but

```
(setq n 0)
(setq x '(a b c d))
(prog1 (nth (setq n (+ n 1)) x)
  (setf (nth (setq n (+ n 1)) x) 'z)) ⇒ b
  and now x ⇒ (a b z d)
```

Moreover, for certain *place* forms *shifff* may be significantly more efficient than the *prog1* version.

Rationale: *shifff* and *rotatet* have been included in COMMON LISP as generalizations of two-argument versions formerly called *swapf* and *exchf*. The two-argument versions have been found to be very useful, but the names were easily confused. The generalization to many argument forms and the change of names were both inspired by the work of Suzuki [19], which indicates that use of these primitives can make certain complex pointer-manipulation programs clearer and easier to prove correct.

rotatet {*place*}*

[Macro]

Each *place* form may be any form acceptable as a generalized variable to *setf*. In the form (*rotatet place1 place2 . . . placen*) the values in *place1* through *placen* are accessed and saved. Values 2 through *n* and value 1 are then stored into *place1* through *placen*. It is as if all the places form an end-around shift register that is rotated one place to the left, with the value of *place1* being shifted around the end to *placen*. Note that (*rotatet place1 place2*) exchanges the contents of *place1* and *place2*.

The effect of (*rotatet place1 place2 . . . placen newvalue*) is roughly equivalent to

```
(psetf place1 place2
  place2 place3
  . . .
  placen place1)
```

except that the latter would evaluate any subforms of each *place* twice, whereas *rotatet* takes care to evaluate them only once. Moreover, for certain *place* forms *rotatet* may be significantly more efficient.

rotatet always returns *nil*.

Other macros that manipulate generalized variables include *getf*, *remf*, *incf*, *decf*, *push*, *pop*, *assert*, *ctypecase*, and *ccase*.

Macros that manipulate generalized variables must guarantee the “obvious” semantics: subforms of generalized-variable references are evaluated exactly as many times as they appear in the source program, and they are evaluated in exactly the same order as they appear in the source program.

In generalized-variable references such as `shfff`, `incf`, `push`, and `setf` of `ldb`, the generalized variables are both read and written in the same reference. Preserving the source program order of evaluation and the number of evaluations is particularly important.

As an example of these semantic rules, in the generalized-variable reference (*setf reference value*) the *value* form must be evaluated *after* all the subforms of the reference because the *value* form appears to the right of them.

The expansion of these macros must consist of code that follows these rules or has the same effect as such code. This is accomplished by introducing temporary variables bound to the subforms of the reference. As an optimization in the implementation, temporary variables may be eliminated whenever it can be proven that removing them has no effect on the semantics of the program. For example, a constant need never be saved in a temporary variable. A variable, or any form that does not have side effects, need not be saved in a temporary variable if it can be proven that its value will not change within the scope of the generalized-variable reference.

COMMON LISP provides built-in facilities to take care of these semantic complications and optimizations. Since the required semantics can be guaranteed by these facilities, the user does not have to worry about writing correct code for them, especially in complex cases. Even experts can become confused and make mistakes while writing this sort of code.

Another reason for building in these functions is that the appropriate optimizations will differ from implementation to implementation. In some implementations most of the optimization is performed by the compiler, while in others a simpler compiler is used and most of the optimization is performed in the macros. The cost of binding a temporary variable relative to the cost of other LISP operations may differ greatly between one implementation and another, and some implementations may find it best never to remove temporary variables except in the simplest cases.

A good example of the issues involved can be seen in the following generalized-variable reference:

```
(incf (ldb byte-field variable))
```

This ought to expand into something like

```
(setq variable
      (dpb (1+ (ldb byte-field variable))
           byte-field
           variable))
```

In this expansion example we have ignored the further complexity of returning the correct value, which is the incremented byte, not the new value of `variable`. Note that the variable `byte-field` is evaluated twice, and the variable `variable` is referred to three times: once as the location in which to store a value, and twice during the computation of that value.

Now consider this expression:

```
(incf (ldb (aref byte-fields (incf i))
           (aref (determine-words-array) i)))
```

It ought to expand into something like this:

```
(let ((temp1 (aref byte-fields (incf i)))
      (temp2 (determine-words-array)))
  (setf (aref temp2 i)
        (dpb (1+ (ldb temp1 (aref temp2 i)))
              temp1
              (aref temp2 i))))
```

Again we have ignored the complexity of returning the correct value. What is important here is that the expressions `(incf i)` and `(determine-words-array)` must not be duplicated because each may have a side effect or be affected by side effects.

The COMMON LISP facilities provided to deal with these semantic issues include:

- Built-in macros such as `setf` and `push` that follow the semantic rules.
- The `define-modify-macro` macro, which allows new generalized-variable manipulating macros (of a certain restricted kind) to be defined easily. It takes care of the semantic rules automatically.
- The `defsetf` macro, which allows new types of generalized-variable references to be defined easily. It takes care of the semantic rules automatically.
- The `define-setf-method` macro and the `get-setf-method` function, which provide access to the internal mechanisms when it is necessary to define a complicated new type of generalized-variable reference or generalized-variable-manipulating macro.

`define-modify-macro` *name lambda-list function* [*doc-string*] [Macro]

This macro defines a read-modify-write macro named *name*. An example of such a macro is `incf`. The first subform of the macro will be a generalized-variable reference. The *function* is literally the function to apply to the old contents of the generalized-variable to get the new contents; it is not evaluated. *lambda-list* describes the remaining arguments for the *function*; these arguments come from the remaining subforms of the macro after the generalized-variable reference. *lambda-list* may contain `&optional` and `&rest` markers. (The `&key` marker is not permitted here; `&rest` suffices for the purposes of `define-modify-macro`.) *doc-string* is documentation for the macro *name* being defined.

The expansion of a `define-modify-macro` is equivalent to the following, except that it generates code that follows the semantic rule outlined above.

```
(defmacro name (reference . lambda-list)
  doc-string
  `(setf reference
        (function reference .arg1 .arg2 . . . )))
```

where *arg1*, *arg2*, . . . , are the parameters appearing in *lambda-list*; appropriate provision is made for an `&rest` parameter.

As an example, `incf` could have been defined by:

```
(define-modify-macro incf (&optional (delta 1)) +)
```

An example of a possibly useful macro not predefined in COMMON LISP is:

```
(define-modify-macro unionf (other-set &rest keywords) union)
```

```
defsetf access-fn {update-fn [doc-string] | lambda-list (store-variable)}          [Macro]
                {declaration | doc-string}* {form}*
```

This defines how to `setf` a generalized-variable reference of the form (*access-fn* . . .). The value of a generalized-variable reference can always be obtained simply by evaluating it, so *access-fn* should be the name of a function or a macro.

The user of `defsetf` provides a description of how to store into the generalized-variable reference and return the value that was stored (because `setf` is defined to return this value). The implementation of `defsetf` takes care of ensuring that subforms of the reference are evaluated exactly once and in the proper left-to-right order. In order to do this, `defsetf` requires that *access-fn* be a function or a macro that evaluates its arguments, behaving like a function. Furthermore, a `setf` of a call on *access-fn* will also evaluate all of *access-fn*'s arguments; it cannot treat any of them specially. This means that `defsetf` cannot be used to describe how to store into a generalized variable that is a byte, such as (ldb field reference). To handle situations that do not fit the restrictions imposed by `defsetf`, use `define-setf-method`, which gives the user additional control at the cost of increased complexity.

A `defsetf` declaration may take one of two forms. The simple form of `defsetf` is

```
(defsetf access-fn update-fn [doc-string])
```

The *update-fn* must name a function (or macro) that takes one more argument than *access-fn* takes. When `setf` is given a *place* that is a call on *access-fn*, it expands into a call on *update-fn* that is given all the arguments to *access-fn* and also, as its last argument, the new value (which must be returned by *update-fn* as its value). For example, the effect of

```
(defsetf symbol-value set)
```

is built into the COMMON LISP system. This causes the form (`setf` (symbol-value foo) fu) to expand into (`set` foo fu). Note that

```
(defsetf car rplaca)
```

would be incorrect because `rplaca` does not return its last argument.

The complex form of `defsetf` looks like

```
(defsetf access-fn lambda-list (store-variable) . body)
```

and resembles `defmacro`. The *body* must compute the expansion of a `setf` of a call on *access-fn*.

The *lambda-list* describes the arguments of *access-fn*. `&optional`, `&rest`, and `&key` markers are permitted in *lambda-list*. Optional arguments may have defaults and "supplied-p" flags. The *store-variable* describes the value to be stored into the generalized-variable reference.

Rationale: The *store-variable* is enclosed in parentheses to provide for an extension to multiple store variables that would receive multiple values from the second subform of `self`. The rules given below for coding `self` methods discuss the proper handling of multiple store variables to allow for the possibility that this extension may be incorporated into COMMON LISP in the future.

The *body* forms can be written as if the variables in the *lambda-list* were bound to subforms of the call on *access-fn* and the *store-variable* were bound to the second subform of `self`. However, this is not actually the case. During the evaluation of the *body* forms, these variables are bound to names of temporary variables, generated as if by `gensym` or `gentemp`, that will be bound by the expansion of `self` to the values of those subforms. This binding permits the *body* forms to be written without regard for order-of-evaluation issues. `defself` arranges for the temporary variables to be optimized out of the final result in cases where that is possible. In other words, an attempt is made by `defself` to generate the best code possible in a particular implementation.

Note that the code generated by the *body* forms must include provision for returning the correct value (the value of *store-variable*). This is handled by the *body* forms rather than by `defself` because in many cases this value can be returned at no extra cost, by calling a function that simultaneously stores into the generalized variable and returns the correct value.

An example of the use of the complex form of `defself`:

```
(defself subseq (sequence start &optional end) (new-sequence)
  (progn (replace ,sequence ,new-sequence
                :start1 ,start :end1 ,end)
        ,new-sequence))
```

The underlying theory by which `self` and related macros arrange to conform to the semantic rules given above is that from any generalized-variable reference one may derive its “`self` method,” which describes how to store into that reference and which subforms of it are evaluated.

Compatibility note: To avoid confusion, it should be noted that the use of the word “method” here in connection with `self` has nothing to do with its use in ZETALISP in connection with message-passing and the ZETALISP “flavor system.”

Given knowledge of the subforms of the reference, it is possible to avoid evaluating them multiple times or in the wrong order. A `self` method for a given access form can be expressed as five values:

- A list of *temporary variables*.
- A list of *value forms* (subforms of the given form) to whose values the temporary variables are to be bound.
- A second list of temporary variables, called *store variables*.
- A *storing form*.
- An *accessing form*.

The temporary variables will be bound to the values of the value forms as if by `let*`; that is, the value forms will be evaluated in the order given and may refer to the values of earlier value forms by using the corresponding variables.

The store variables are to be bound to the values of the *newvalue* form, that is, the values to be stored into the generalized variable. In almost all cases only a single value is to be stored, and there is only one store variable.

The storing form and the accessing form may contain references to the temporary variables (and also, in the case of the storing form, to the store variables). The accessing form returns the value of the generalized variable. The storing form modifies the value of the generalized variable and guarantees to return the values of the store variables as its values; these are the correct values for `self` to return. (Again, in most cases there is a single store variable and thus a single value to be returned.) The value returned by the accessing form is, of course, affected by execution of the storing form, but either of these forms may be evaluated any number of times, and therefore should be free of side effects (other than the storing action of the storing form).

The temporary variables and the store variables are generated names, as if by `gensym` or `gentemp`, so that there is never any problem of name clashes among them, or between them and other variables in the program. This is necessary to make the special forms that do more than one `self` in parallel work properly; these are `pself`, `shifff`, and `rotatof`. Computation of the `self` method must always create new variable names; it may not return the same ones every time.

Some examples of `self` methods for particular forms:

- For a variable `x`:

```
()
()
(g0001)
(setq x g0001)
x
```

- For `(car exp)`:

```
(g0002)
(exp)
(g0003)
(progn (rplaca g0002 g0003) g0003)
(car g0002)
```

- For `(subseq seq s e)`:

```
(g0004 g0005 g0006)
(seq s e)
(g0007)
(progn (replaca g0004 g0007 :start1 g0005 :end1 g0006)
       g0007)
(subseq g0004 g0005 g0006)
```

define-self-method *access-fn lambda-list {declaration | doc-string}* {form}* [Macro]*

This defines how to self a generalized-variable reference that is of the form (*access-fn* . . .). The value of a generalized-variable reference can always be obtained simply by evaluating it, so *access-fn* should be the name of a function or a macro.

The *lambda-list* describes the subforms of the generalized-variable reference, as with `defmacro`. The result of evaluating the *forms* in the body must be five values representing the self method, as described above. Note that `define-self-method` differs from the complex form of `defself` in that while the body is being executed the variables in *lambda-list* are bound to parts of the generalized-variable reference, not to temporary variables that will be bound to the values of such parts. In addition, `define-self-method` does not have `defself`'s restriction that *access-fn* must be a function or a function-like macro; an arbitrary `defmacro` destructuring pattern is permitted in *lambda-list*.

By definition there are no good small examples of `define-self-method` because the easy cases can all be handled by `defself`. A typical use is to define the self method for `ldb`:

```
;;; SETF method for the form (LDB bytespec int).
;;; Recall that the int form must itself be suitable for SETF.

(define-self-method ldb (bytespec int)
  (multiple-value-bind (temps vals stores
                       store-form access-form)
    (get-self-method int) ;Get SETF method for int.
    (let ((btemp (gensym)) ;Temp var for byte specifier.
          (store (gensym)) ;Temp var for byte to store.
          (stemp (first stores))) ;Temp var for int to store.
      ;; Return the SETF method for LDB as five values.
      (values (cons btemp temps) ;Temporary variables.
              (cons bytespec vals) ;Value forms.
              (list store) ;Store variables.
              `(let ((.stemp (dpb ,store ,btemp ,access-form)))
                  ,store-form
                  ,store) ;Storing form.
              `(ldb ,btemp ,access-form ;Accessing form.
                  )))
```

get-self-method *form* [Function]

`get-self-method` returns five values constituting the self method for *form*. The *form* must be a generalized-variable reference, `get-self-method` takes care of error-checking and macro expansion and guarantees to return exactly one store-variable.

As an example, an extremely simplified version of self, allowing no more and no fewer than two subforms, containing no optimization to remove unnecessary variables, and not allowing storing of multiple values, could be defined by:

```
(defmacro setf (reference value)
  (multiple-value-bind (vars vals stores store-form access-form)
    (get-setf-method reference)
    (declare (ignore access-form))
    `(let* ,(mapcar #'list
                    (append vars stores)
                    (append vals (list value)))
       ,store-form)))
```

get-setf-method-multiple-value *form*

[Function]

get-setf-method-multiple-value returns five values constituting the setf method for *form*. The *form* must be a generalized-variable reference. This is the same as get-setf-method except that it does not check the number of store-variables; use this in cases that allow storing multiple values into a generalized variable. There are no such cases in standard COMMON LISP, but this function is provided to allow for possible extensions.

Function Invocation

The most primitive form for function invocation in LISP of course has no name; any list that has no other interpretation as a macro call or special form is taken to be a function call. Other constructs are provided for less common but nevertheless frequently useful situations.

apply *function arg* &rest *more-args*

[Function]

This applies *function* to a list of arguments. *function* may be a compiled-code object, or a lambda-expression, or a symbol; in the latter case the global functional value of that symbol is used (but it is illegal for the symbol to be the name of a macro or special form). The arguments for the *function* consist of the last argument to apply appended to the end of a list of all the other arguments to apply but the *function* itself; it is as if all the arguments to apply except the *function* were given to list* to create the argument list. For example:

```
(setq f '+) (apply f '(1 2)) ⇒ 3
(setq f #-) (apply f '(1 2)) ⇒ -1
(apply #'max 3 5 '(2 7 3)) ⇒ 7
(apply 'cons '((+ 2 3) 4)) ⇒
  ((+ 2 3) . 4) not (5 . 4)
(apply #'+ '()) ⇒ 0
```

Note that if the function takes keyword arguments, the keywords as well as the corresponding values must appear in the argument list:

```
(apply #'(lambda (&key a b) (list a b)) '(:b 3)) ⇒ (nil 3)
```

This can be very useful in conjunction with the &allow-other-keys feature:

```
(defun foo (size &rest keys &key double &allow-other-keys)
  (let ((v (apply #'make-array size :allow-other-keys t keys)))
    (if double (concatenate (type-of v) v v) v)))

(foo 4 :initial-contents '(a b c d) :double t)
⇒ #(a b c d a b c d)
```

`funcall fn &rest arguments`

[*Function*]

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

```
(cons 1 2) ⇒ (1 . 2)
(setq cons (symbol-function '+))
(funcall cons 1 2) ⇒ 3
```

The difference between `funcall` and an ordinary function call is that the function is obtained by ordinary LISP evaluation rather than by the special interpretation of the function position that normally occurs.

Compatibility note: The COMMON LISP function `funcall` corresponds roughly to the INTERLISP primitive `apply*`

`call-arguments-limit`

[*Constant*]

The value of `call-arguments-limit` is a positive integer that is the upper exclusive bound on the number of arguments that may be passed to a function. This bound depends on the implementation, but will not be smaller than 50. (Implementors are encouraged to make this limit as large as practicable without sacrificing performance.) The value of `call-arguments-limit` must be at least as great as that of `lambda-parameters-limit`. See also `multiple-values-limit`.

Simple Sequencing

Each of the constructs in this section simply evaluates all the argument forms in order. They differ only in what results are returned.

`progn {form}*`

[*Special form*]

The `progn` construct takes a number of forms and evaluates them sequentially, in order, from left to right. The values of all the forms but the last are discarded; whatever the last form returns is returned by the `progn` form. One says that all the forms but the last are evaluated for *effect*, because their execution is useful only for the side effects caused, but the last form is executed for *value*.

`progn` is the primitive control structure construct for “compound statements,” such as **begin-end** blocks in ALGOL-like languages. Many LISP constructs are “implicit `progn`” forms, in that as part of their syntax each allows many forms to be written that are to be evaluated sequentially, discarding the results of all forms but the last and returning the results of the last form.

If the last form of the `progn` returns multiple values, then those multiple values are returned by the `progn` form. If there are no forms for the `progn`, then the result is `nil`. These rules generally hold for implicit `progn` forms as well.

`prog1 first {form}`* [Macro]

`prog1` is similar to `progn`, but it returns the value of its *first* form. All the argument forms are executed sequentially; the value the first form produces is saved while all the others are executed and is then returned.

`prog1` is most commonly used to evaluate an expression with side effects and return a value that must be computed *before* the side effects happen. For example:

```
(prog1 (car x) (rplaca x 'foo))
```

alters the *car* of *x* to be *foo* and returns the old *car* of *x*.

`prog1` always returns a single value, even if the first form tries to return multiple values. As a consequence of this, `(prog1 x)` may behave differently if *x* can produce multiple values. See `multiple-value-prog1`. A point of style: although `prog1` can be used to force exactly a single value to be returned, it is conventional to use the function `values` for this purpose.

`prog2 first second {form}`* [Macro]

`prog2` is similar to `prog1`, but it returns the value of its *second* form. All the argument forms are executed sequentially; the value of the second form is saved while all the other forms are executed and is then returned. `prog2` is provided mostly for historical compatibility.

```
(prog2 a b c . . . z) ≡ (progn a (prog1 b c . . . z))
```

Occasionally it is desirable to perform one side effect, then a value-producing operation, then another side effect. In such a peculiar case, `prog2` is fairly perspicuous. For example:

```
(prog2 (open-a-file) (process-the-file) (close-the-file))
;value is that of process-the file
```

`prog2`, like `prog1`, always returns a single value, even if the second form tries to return multiple values. As a consequence of this, `(prog2 x y)` and `(progn x y)` may behave differently if *y* can produce multiple values.

Establishing New Variable Bindings

During the invocation of a function represented by a lambda-expression (or a closure of a lambda-expression, as produced by `function`), new bindings are established for the variables that are the parameters of the lambda-expression. These bindings initially have values determined by the parameter-binding protocol discussed in “Lambda-Expressions,” Chapter 5.

The following constructs may also be used to establish bindings of variables, both ordinary and functional.

`let` `{(var | (var value))* }` `{declaration}* {form}`* [Special form]

A `let` form can be used to execute a series of forms with specified variables bound to specified values.

More precisely, the form

```
(let ((var1 value1)
      (var2 value2)
      . . .
      (varm valuem))
  declaration1
  declaration2
  . . .
  declarationp
  body1
  body2
  . . .
  bodyn)
```

first evaluates the expressions *value1*, *value2*, and so on, in that order, saving the resulting values. Then all the variables *varj* are bound to the corresponding values in parallel; each binding will be a lexical binding unless there is a special declaration to the contrary. The expressions *bodyk* are then evaluated in order; the values of all but the last are discarded (that is, the body of a `let` form is an implicit `progn`). The `let` form returns what evaluating *body_n* produces (if the body is empty, which is fairly useless, `let` returns `nil` as its value). The bindings of the variables have lexical scope and indefinite extent.

Instead of a list (*varj valuej*), one may write simply *varj*. In this case *varj* is initialized to `nil`. As a matter of style, it is recommended that *varj* be written only when that variable will be stored into (such as by `setq`) before its first use. If it is important that the initial value is `nil` rather than some undefined value, then it is clearer to write out (*varj nil*) if the initial value is intended to mean "false" or (*varj '()*) if the initial value is intended to be an empty list. Note that the code

```
(let (x)
  (declare (integer x))
  (setq x (gcd y z))
  . . . )
```

is incorrect; although *x* is indeed set before it is used, and is set to a value of the declared type integer, nevertheless *x* momentarily takes on the value `nil` in violation of the type declaration.

Declarations may appear at the beginning of the body of a `let`. See `declare`.

`let*` (*{var | (var value)}*)* *{declaration}** *{form}** [*Special form*]

`let*` is similar to `let`, but the bindings of variables are performed sequentially rather than in parallel. This allows the expression for the value of a variable to refer to variables previously bound in the `let*` form.

More precisely, the form

```
{let* ((var1 value1)
      (var2 value2)
      . . .
      (varm valuem))
  declaration1
  declaration2
  . . .
  declarationp
  body1
  body2
  . . .
  bodyn)
```

first evaluates the expression *value1*, then binds the variable *var1* to that value; then it evaluates *value2* and binds *var2*; and so on. The expressions *bodyj* are then evaluated in order; the values of all but the last are discarded (that is, the body of a `let*` form is an implicit `progn`). The `let*` form returns the results of evaluating *bodyn* (if the body is empty, which is fairly useless, `let*` returns `nil` as its value). The bindings of the variables have lexical scope and indefinite extent.

Instead of a list *(varj valuej)*, one may write simply *varj*. In this case *varj* is initialized to `nil`. As a matter of style, it is recommended that *varj* be written only when that variable will be stored into (such as by `setq`) before its first use. If it is important that the initial value is `nil` rather than some undefined value, then it is clearer to write out *(varj nil)* if the initial value is intended to mean “false” or *(varj ())* if the initial value is intended to be an empty list.

Declarations may appear at the beginning of the body of a `let*`. See `declare`.

```
compiler-let ({var | (var value)}*) {form}* [Special form]
```

When executed by the LISP interpreter, `compiler-let` behaves exactly like `let` with all the variable bindings implicitly declared `special`. When the compiler processes this form, however, no code is compiled for the bindings; instead, the processing of the body by the compiler (including, in particular, the expansion of any macro calls within the body) is done with the special variables bound to the indicated values *in the execution context of the compiler*. This is primarily useful for communication among complicated macros.

Declarations may *not* appear at the beginning of the body of a `compiler-let`.

Rationale: Because of the unorthodox handling by `compiler-let` of its variable bindings, it would be complicated and confusing to permit declarations that apparently referred to the variables bound by `compiler-let`. Disallowing declarations eliminates the problem.

macrolet is similar in form to flet but defines local macros, using the same format used by defmacro. The names established by macrolet as names for macros are lexically scoped.

Macros often must be expanded at “compile time” (more generally, at a time before the program itself is executed), and so the run-time values of variables are not available to macros defined by macrolet. The precise rule is that the macro-expansion functions defined by macrolet are defined in the *global* environment; lexically scoped entities that would ordinarily be lexically apparent are not visible within the expansion functions. However, lexically scoped entities *are* visible within the body of the macrolet form and *are* visible to the code that is the expansion of a macro call. The following example should make this clear:

```
(defun foo (x flag)
  (macrolet ((fudge (z)
              ;The parameters x and flag are not accessible
              ; at this point; a reference to flag would be to
              ; the global variable of that name.
              `(if flag (* ,z ,z) ,z)))
    ;The parameters x and flag are accessible here.
    (+ x
      (fudge x)
      (fudge (+ x 1)))))
```

The body of the macrolet becomes

```
(+ x
  (if flag (* x x) x)
  (if flag (* (+ x 1) (+ x 1)) (+ x 1)))
```

after macro expansion. The occurrences of *x* and *flag* legitimately refer to the parameters of the function *foo* because those parameters are visible at the site of the macro call which produced the expansion.

Conditionals

The traditional conditional construct in LISP is *cond*. However, *if* is much simpler and is directly comparable to conditional constructs in other programming languages, so it is considered to be primitive in COMMON LISP and is described first. COMMON LISP also provides the dispatching constructs *case* and *typecase*, which are often more convenient than *cond*.

if test then [else]

[*Special form*]

The *if* special form corresponds to the **if-then-else** construct found in most algebraic programming languages. First the form *test* is evaluated. If the result is not nil, then the form *then* is selected; otherwise the form *else* is selected. Whichever form is selected is then evaluated, and *if* returns whatever evaluation of the selected form returns.

$(\text{if } test \text{ then } else) \equiv (\text{cond } (test \text{ then}) (t \text{ else}))$

but *if* is considered more readable in some situations.

The *else* form may be omitted, in which case if the value of *test* is *nil* then nothing is done and the value of the *if* form is *nil*. If the value of the *if* form is important in this situation, then the *and* construct may be stylistically preferable, depending on the context. If the value is not important, but only the effect, then the *when* construct may be stylistically preferable.

*when test {form}**

[Macro]

(*when test form1 form2 . . .*) first evaluates *test*. If the result is *nil*, then no *form* is evaluated, and *nil* is returned. Otherwise the *forms* constitute an implicit *progn* and are evaluated sequentially from left to right, and the value of the last one is returned.

```
(when p a b c) ≡ (and p (progn a b c))
(when p a b c) ≡ (cond (p a b c))
(when p a b c) ≡ (if p (progn a b c) nil)
(when p a b c) ≡ (unless (not p) a b c)
```

As a matter of style, *when* is normally used to conditionally produce some side effects, and the value of the *when*-form is normally not used. If the value is relevant, then it may be stylistically more appropriate to use *and* or *if*.

*unless test {form}**

[Macro]

(*unless test form1 form2 . . .*) first evaluates *test*. If the result is *not nil*, then the *forms* are not evaluated, and *nil* is returned. Otherwise the *forms* constitute an implicit *progn* and are evaluated sequentially from left to right, and the value of the last one is returned.

```
(unless p a b c) ≡ (cond ((not p) a b c))
(unless p a b c) ≡ (if p nil (progn a b c))
(unless p a b c) ≡ (when (not p) a b c)
```

As a matter of style, *unless* is normally used to conditionally produce some side effects, and the value of the *unless*-form is normally not used. If the value is relevant, then it may be stylistically more appropriate to use *if*.

cond {(test {form})}**

[Macro]

A *cond* form has a number (possibly zero) of *clauses*, which are lists of forms. Each clause consists of a *test* followed by zero or more *consequents*. For example:

```
(cond (test-1 consequent-1-1 consequent-1-2 . . . )
      (test-2)
      (test-3 consequent-3-1 . . . )
      . . . )
```

The first clause whose *test* evaluates to non-*nil* is selected; all other clauses are ignored, and the *consequents* of the selected clause are evaluated in order (as an implicit *progn*).

More specifically, `cond` processes its clauses in order from left to right. For each clause, the *test* is evaluated. If the result is `nil`, `cond` advances to the next clause. Otherwise, the *cdr* of the clause is treated as a list of forms, or consequents; these forms are evaluated in order from left to right, as an implicit `progn`. After evaluating the consequents, `cond` returns without inspecting any remaining clauses. The `cond` special form returns the results of evaluating the last of a selected consequents; if there were no consequents in the selected clause, then the single (and necessarily non-null) value of the *test* is returned. If `cond` runs out of clauses (every test produced `nil`, and therefore no clause was selected), the value of the `cond` form is `nil`.

If it is desired to select the last clause unconditionally if all others fail, the standard convention is to use `†` for the *test*. As a matter of style, it is desirable to write a last clause `(† nil)` if the value of the `cond` form is to be used for something. Similarly, it is in questionable taste to let the last clause of a `cond` be a “singleton clause”; an explicit `†` should be provided. (Note moreover that `(cond . . . (x))` may behave differently from `(cond . . . († x))` if *x* might produce multiple values; the former always returns a single value, whereas the latter returns whatever values *x* returns. However, as a matter of style it is preferable to obtain this behavior by writing `(cond . . . († (values x)))`, using the `values` function explicitly to indicate the discarding of any excess values.) For example:

<code>(setq z (cond (a 'foo) (b 'bar)))</code>	;Possibly confusing
<code>(setq z (cond (a 'foo) (b 'bar) († nil)))</code>	;Better
<code>(cond (a b) (c d) (e))</code>	;Possibly confusing
<code>(cond (a b) (c d) († e))</code>	;Better
<code>(cond (a b) (c d) († (values e)))</code>	;Better (if one value ; needed)
<code>(cond (a b) (c))</code>	;Possibly confusing
<code>(cond (a b) († c))</code>	;Better
<code>(if a b c)</code>	;Also better

A LISP `cond` form may be compared to a continued **if-then-else** as found in many algebraic programming languages:

<code>(cond (p . . .)</code>		if <i>p</i> then . . .
<code>(q . . .)</code>	roughly	else if <i>q</i> then . . .
<code>(r . . .)</code>	corresponds	else if <i>r</i> then . . .
<code>. . .</code>	to	. . .
<code>(† . . .)</code>		else . . .

`case` *keyform* {{{({key}*) | key} {form}*)}* [Macro]

`case` is a conditional that chooses one of its clauses to execute by comparing a value to various constants, which are typically keyword symbols, integers, or characters (but may be any objects). Its form is as follows:

```
(case keyform
  (keylist-1 consequent-1-1 consequent-1-2 . . .)
  (keylist-2 consequent-2-1 . . .)
  (keylist-3 consequent-3-1 . . .)
  . . . )
```

Structurally `case` is much like `cond`, and it behaves like `cond` in selecting one clause and then executing all consequents of that clause. However, `case` differs in the mechanism of clause selection.

The first thing `case` does is to evaluate the form *keyform* to produce an object called the *key object*. Then `case` considers each of the clauses in turn. If *key* is in the *keylist* (that is, is `eq` to any item in the *keylist*) of a clause, the consequents of that clause are evaluated as an implicit `progn`; `case` returns what was returned by the last consequent (or `nil` if there are no consequents in that clause). If no clause is satisfied, `case` returns `nil`.

The keys in the keylists are *not* evaluated; literal key values must appear in the keylists. It is an error for the same key to appear in more than one clause; a consequence is that the order of the clauses does not affect the behavior of the `case` construct.

Instead of a *keylist*, one may write one of the symbols `t` and `otherwise`. A clause with such a symbol always succeeds and must be the last clause (this is an exception to the order-independence of clauses). See also `ecase` and `ccase`, each of which provides an implicit `otherwise` clause to signal an error if no clause is satisfied.

If there is only one key for a clause, then that key may be written in place of a list of that key, provided that no ambiguity results. Such a “singleton key” may not be `nil` (which is confusable with `()`, a list of no keys), `t`, `otherwise`, or a `cons`.

Compatibility note: The ZETALISP `caseq` construct uses `eq` for the comparison. In ZETALISP `caseq` therefore works for fixnums but not bignums. The MACLISP `caseq` construct simply prohibits the use of bignums; indeed, it permits only fixnums and symbols as clause keys. In the interest of hiding the fixnum-bignum distinction, and for general language consistency, `case` uses `eq` in COMMON LISP.

The INTERLISP `selectq` construct is similar to `case`.

`typecase` *keyform* {(*type* {*form*}*)}* [Macro]

`typecase` is a conditional that chooses one of its clauses by examining the type of an object. Its form is as follows:

```
(typecase keyform
  (type-1 consequent-1-1 consequent-1-2 . . . )
  (type-2 consequent-2-1 . . . )
  (type-3 consequent-3-1 . . . )
  . . . )
```

Structurally `typecase` is much like `cond` or `case`, and it behaves like them in selecting one clause and then executing all consequents of that clause. It differs in the mechanism of clause selection.

The first thing `typecase` does is to evaluate the form *keyform* to produce an object called the *key object*. Then `typecase` considers each of the clauses in turn. The *type* that appears in each clause is a type specifier; it is not evaluated, but is a literal type specifier. The first clause for which the key is of that clause’s specified *type* is selected, the consequents of this clause are evaluated as an implicit `progn`, and `typecase` returns what was returned by the last consequent (or `nil` if there are no consequents in that clause). If no clause is satisfied, `typecase` returns `nil`.

As for *case*, the symbol *t* or otherwise may be written for *type* to indicate that the clause should always be selected. See also *etypecase* and *ctypecase*, each of which provides an implicit otherwise clause to signal an error if no clause is satisfied.

It is permissible for more than one clause to specify a given type, particularly if one is a subtype of another; the earliest applicable clause is chosen. Thus for *typecase*, unlike *case*, the order of the clauses may affect the behavior of the construct. For example:

```
(typecase an-object
  (string . . . )           ;This clause handles strings.
  ((array t) . . . )       ;This clause handles general arrays.
  ((array bit) . . . )     ;This clause handles bit arrays.
  (array . . . )           ;This handles all other arrays.
  ((or list number) . . . ) ;This handles lists and numbers.
  (t . . . ))              ;This handles all other objects.
```

A COMMON LISP compiler may choose to issue a warning if a clause cannot be selected because it is completely shadowed by earlier clauses.

Blocks and Exits

The *block* and *return-from* constructs provide a structured lexical non-local exit facility. At any point lexically within a *block* construct, a *return-from* with the same name may be used to perform an immediate transfer of control that exits from the *block*. In the most common cases this mechanism is more efficient than the dynamic non-local exit facility provided by *catch* and *throw*, described below in “Dynamic Non-local Exits.”

*block name {form}** [*Special form*]

The *block* construct executes each *form* from left to right, returning whatever is returned by the last *form*. If, however, a *return* or *return-from* form that specifies the same *name* is executed during the execution of some *form*, then the results specified by the *return* or *return-from* are immediately returned as the value of the *block* construct, and execution proceeds as if the *block* had terminated normally. In this, *block* differs from *progn*; the *progn* construct has nothing to do with *return*.

The *name* is not evaluated; it must be a symbol. The scope of the *name* is lexical; only a *return* or *return-from* textually contained in some *form* can exit from the *block*. The extent of the name is dynamic. Therefore it is only possible to exit from a given run-time incarnation of a *block* once, either normally or by explicit *return*.

The *defun* form implicitly puts a *block* around the body of the function defined; the *block* has the same name as the function. Therefore one may use *return-from* to return prematurely from a function defined by *defun*. In the same manner, *defmacro*, *deftype*, *defsetf*, *define-setf-method*, and *macrolet* put a named *block* around the body of the function or macro defined.

The lexical scoping of the *block name* is fully general and has consequences that may be surprising to users and implementors of other LISP systems. For example, the *return-from* in the following example actually does “work” in COMMON LISP as one might expect:

```
(block loser
  (catch 'stuff
    (mapcar #'(lambda (x) (if (numberp x)
                            (hairyfun x)
                            (return-from loser nil)))
            items)))
```

Depending on the situation, a return in COMMON LISP may not be simple. A return can break up catchers if necessary to get to the block in question. It is possible for a "closure" created by function for a lambda-expression to refer to a block name as long as the name is lexically apparent.

```
return-from name [result] [Special form]
return [result] [Macro]
```

return-from is used to return from a block or from such constructs as do and prog that implicitly establish a block. The *name* is not evaluated and must be a symbol. A block construct with the same name must lexically enclose the occurrence of return-from; whatever the evaluation of *result* produces is immediately returned from the block. (If the *result* form is omitted, it defaults to nil. As a matter of style, this form ought to be used to indicate that the particular value returned doesn't matter.)

The return-from form itself never returns and cannot have a value; it causes results to be returned from a block construct. If the evaluation of *result* produces multiple values, those multiple values are returned by the construct exited.

(*return form*) is identical in meaning to (return-from nil *form*); it returns from a block named nil. Blocks established implicitly by iteration constructs such as do are named nil, so that return will exit properly from such a construct.

Iteration

COMMON LISP provides a number of iteration constructs. The loop construct provides a trivial iteration facility; it is little more than a progn with a branch from the bottom back to the top. The do and do* constructs provide a general iteration facility for controlling the variation of several variables on each cycle. For specialized iterations over the elements of a list or *n* consecutive integers, dolist and dotimes are provided. The tagbody construct is the most general, permitting arbitrary go statements within it. (The traditional prog construct is a synthesis of tagbody, block, and let.) Most of the iteration constructs permit statically defined non-local exits in the form of the return-from and return statements.

Indefinite Iteration

The loop construct is the simplest iteration facility. It controls no variables, and simply executes its body repeatedly.

```
loop {form}* [Macro]
```

Each *form* is evaluated in turn from left to right. When the last *form* has been evaluated, then the first *form* is evaluated again, and so on, in a never-ending cycle. The loop construct never returns a value. Its execution must be terminated explicitly, using return or throw, for example.

loop, like most iteration constructs, establishes an implicit block named nil. Thus return may be used to exit from a loop with specified results.

A loop construct has this meaning only if every *form* is non-atomic (a list). The case where some *form* (possibly more than one) is atomic is reserved for future extensions.

Implementation note: There have been several proposals for a powerful iteration mechanism to be called loop. One version is provided in ZETALISP. Implementors are encouraged to experiment with extensions to the loop syntax, but users should be advised that in all likelihood some specific set of extensions to loop will be adopted in a future revision of COMMON LISP.

General Iteration

In contrast to loop, do and do* provide a powerful and general mechanism for repetitively recalculating many variables.

```
do {{{(var [init [step]])*} (end-test {result}*)}          [Macro]
   {declaration}* {tag | statement}*
do* {{{(var [init [step]])*} (end-test {form}*)}         [Macro]
     {declaration}* {tag | statement}*
```

The do macro form provides a generalized iteration facility, with an arbitrary number of “index variables.” These variables are bound within the iteration and stepped in parallel in specified ways. They may be used both to generate successive values of interest (such as successive integers) and to accumulate results. When an end condition is met, the iteration terminates with a specified value.

In general, a do loop looks like this:

```
(do ((var1 init1 step1)
     (var2 init2 step2)
     . . .
     (varn initn stepn))
    (end-test . result)
  {declaration}*
  . tagbody)
```

A do* loop looks exactly the same except that the name do is replaced by do*.

The first item in the form is a list of zero or more index-variable specifiers. Each index-variable specifier is a list of the name of a variable *var*, an initial value *init*, and a stepping form *step*. If *init* is omitted, it defaults to nil. If *step* is omitted, the *var* is not changed by the do construct between repetitions (though code within the do is free to alter the value of the variable by using setq).

An index-variable specifier can also be just the name of a variable. In this case, the variable has an initial value of `nil` and is not changed between repetitions. As a matter of style, it is recommended that an unadorned variable name be written only when that variable will be stored into (such as by `setq`) before its first use. If it is important that the initial value is `nil` rather than some undefined value, then it is clearer to write out `(varj nil)` if the initial value is intended to mean “false” or `(varj ())` if the initial value is intended to be an empty list.

Before the first iteration, all the *init* forms are evaluated, and each *var* is bound to the value of its respective *init*. This is a binding, not an assignment; when the loop terminates, the old values of those variables will be restored. For `do`, all of the *init* forms are evaluated *before* any *var* is bound; hence all the *init* forms may refer to the old bindings of all the variables (that is, to the values visible before beginning execution of the `do` construct). For `do*`, the first *init* form is evaluated, then the first *var* is bound to that value, then the second *init* form is evaluated, then the second *var* is bound, and so on; in general, the *initj* form can refer to the *new* binding *varj* if $k < j$, and otherwise to the *old* binding of *varj*.

The second element of the loop is a list of an end-testing predicate form *end-test* and zero or more *result* forms. This resembles a `cond` clause. At the beginning of each iteration, after processing the variables, the *end-test* is evaluated. If the result is `nil`, execution proceeds with the body of the `do` (or `do*`) form. If the result is not `nil`, the *result* forms are evaluated in order as an implicit `progn`, and then `do` returns. `do` returns the results of evaluating the last *result* form. If there are no *result* forms, the value of `do` is `nil`. Note that this is not quite analogous to the treatment of clauses in a `cond` form, because a `cond` clause with no result forms returns the (non-`nil`) result of the test.

At the beginning of each iteration other than the first, the index variables are updated as follows. All the *step* forms are evaluated, from left to right, and the resulting values are assigned to the respective index variables. Any variable that has no associated *step* form is not assigned to. For `do`, all the *step* forms are evaluated before any variable is updated; the assignment of values to variables is done in parallel, as if by `psetq`. Because all of the *step* forms are evaluated before any of the variables are altered, a *step* form when evaluated always has access to the *old* values of all the index variables, even if other *step* forms precede it. For `do*`, the first *step* form is evaluated, then the value is assigned to the first *var*, then the second *step* form is evaluated, then the value is assigned to the second *var*, and so on; the assignment of values to variables is done sequentially, as if by `setq`. For either `do` or `do*`, after the variables have been updated, the end-test is evaluated as described above, and the iteration continues.

If the end-test of a `do` form is `nil`, the test will never succeed. Therefore this provides an idiom for “do forever”: the *body* of the `do` is executed repeatedly, stepping variables as usual. (The loop construct performs a “do forever” that steps no variables.) The infinite loop can be terminated by the use of `return`, `return-from`, `go` to an outer level, or `throw`. For example:

```
(do ((j 0 (+ j 1)))
    (nil) ;Do forever.
    (format t "~%Input ~D:" j)
    (let ((item (read)))
      (if (null item) (return) ;Process items until nil seen.
          (format t "~&Output ~D: ~S" j (process item))))))
```

The remainder of the `do` form constitutes an implicit *tagbody*. Tags may appear within the body of a `do` loop for use by `go` statements appearing in the body (but such `go` statements may not appear in the variable specifiers, the *end-test*, or the *result* forms). When the end of a `do` body is reached, the next iteration cycle (beginning with the evaluation of *step* forms) occurs.

An implicit block named `nil` surrounds the entire `do` form. A `return` statement may be used at any point to exit the loop immediately.

`declare` forms may appear at the beginning of a `do` body. They apply to code in the `do` body, to the bindings of the `do` variables, to the *init* forms, to the *step* forms, to the *end-test*, and to the *result* forms.

Compatibility note: “Old-style” `MACLISP` `do` loops, that is, those of the form `(do var init step end-test . body)`, are not supported in `COMMON LISP`. Such old-style loops are considered obsolete, and in any case are easily converted to a new-style `do` with the insertion of three pairs of parentheses. In practice the compiler can catch nearly all instances of old-style `do` loops because they will not have a legal format anyway.

Here are some examples of the use of `do`:

```
(do ((i 0 (+ i 1)) ;Sets every null element of a-vector to zero.
      (n (length a-vector)))
    ((= i n)
     (when (null (aref a-vector i))
          (setf (aref a-vector i) 0))))
```

The construction

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

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

Very often an iterative algorithm can be most clearly expressed entirely in the *step* forms of a `do`, and the *body* is empty. For example:

```
(do ((x foo (cdr x))
      (y bar (cdr y))
      (z '() (cons (f (car x) (car y)) z)))
    ((or (null x) (null y))
     (nreverse z)))
```

does the same thing as `(mapcar #'f foo bar)`. Note that the *step* computation for `z` exploits the fact that variables are stepped in parallel. Also, the body of the loop is empty. Finally, the use of `nreverse` to put an accumulated `do` loop result into the correct order is a standard idiom. Another example:

```
(defun list-reverse (list)
  (do ((x list (cdr x))
        (y '() (cons (car x) y)))
      ((endp x) y)))
```

Note the use of `endp` rather than `null` or `atom` to test for the end of a list; this may result in more robust code.

As an example of nested loops, suppose that `env` holds a list of conses. The `car` of each cons is a list of symbols, and the `cdr` of each cons is a list of equal length containing corresponding values. Such a data structure is similar to an association list, but is divided into “frames”; the overall structure resembles a rib-cage. A lookup function on such a data structure might be:

```
(defun ribcage-lookup (sym ribcage)
  (do ((r ribcage (cdr r)))
      ((null r) nil)
    (do ((s (caar r) (cdr s))
        (v (cdar r) (cdr v)))
        ((null s)
         (when (eq (car s) sym)
           (return-from ribcage-lookup (car v)))))))
```

(Notice the use of indentation in the above example to set off the bodies of the `do` loops.)

A `do` loop may be explained in terms of the more primitive constructs `block`, `return`, `let`, `loop`, `tagbody`, and `psetq` as follows:

```
(block nil
  (let ((var1 init1)
        (var2 init2)
        . . .
        (varn initn))
    {declaration}*
    (loop (when end-test (return (progn . result)))
          (tagbody . tagbody)
          (psetq var1 step1
                 var2 step2
                 . . .
                 varn stepn))))
```

`do*` is exactly like `do` except that the bindings and steppings of the variables are performed sequentially rather than in parallel. It is as if, in the above explanation, `let` were replaced by `let*` and `psetq` were replaced by `setq`.

Simple Iteration Constructs

The constructs `dolist` and `dotimes` execute a body of code once for each value taken by a single variable. They are expressible in terms of `do`, but capture very common patterns of use.

Both `dolist` and `dotimes` perform a body of statements repeatedly. On each iteration a specified variable is bound to an element of interest that the body may examine. `dolist` examines successive elements of a list, and `dotimes` examines integers from 0 to $n-1$ for some specified positive integer n .

The value of any of these constructs may be specified by an optional result form, which if omitted defaults to the value `nil`.

The `return` statement may be used to return immediately from a `dolist` or `dotimes` form, discarding any following iterations that might have been performed; in effect, a block named `nil` surrounds the construct. The body of the loop is implicitly a `tagbody` construct; it may contain tags to serve as the targets of `go` statements. Declarations may appear before the body of the loop.

`dolist` (*var listform* [*resultform*]) {*declaration*}* {*tag* | *statement*}* [Macro]

`dolist` provides straightforward iteration over the elements of a list. First `dolist` evaluates the form *listform*, which should produce a list. It then executes the body once for each element in the list, in order, with the variable *var* bound to the element. Then *resultform* (a single form, *not* an implicit `progn`) is evaluated, and the result is the value of the `dolist` form. (When the *resultform* is evaluated, the control variable *var* is still bound, and has the value `nil`.) If *resultform* is omitted, the result is `nil`.

```
(dolist (x '(a b c d)) (print x) (princ " ")) ⇒ nil
  after printing "a b c d "
```

An explicit `return` statement may be used to terminate the loop and return a specified value.

`dotimes` (*var countform* [*resultform*]) {*declaration*}* {*tag* | *statement*}* [Macro]

`dotimes` provides straightforward iteration over a sequence of integers. The expression `(dotimes (var countform resultform) . progbody)` evaluates the form *countform*, which should produce an integer. It then performs *progbody* once for each integer from zero (inclusive) to *count* (exclusive), in order, with the variable *var* bound to the integer; if the value of *countform* is zero or negative, then the *progbody* is performed zero times. Finally, *resultform* (a single form, *not* an implicit `progn`) is evaluated, and the result is the value of the `dotimes` form. (When the *resultform* is evaluated, the control variable *var* is still bound, and has as its value the number of times the body was executed.) If *resultform* is omitted, the result is `nil`.

An explicit `return` statement may be used to terminate the loop and return a specified value.

Here is an example of the use of `dotimes` in processing strings:

```
;;; True if the specified subsequence of the string is a
;;; palindrome (reads the same forwards and backwards).
(defun palindromep (string &optional
                  (start 0)
                  (end (length string)))
  (dotimes (k (floor (- end start) 2) t)
    (unless (char-equal (char string (+ start k))
                       (char string (- end k 1)))
      (return nil))))

(palindromep "Able was I ere I saw Elba") ⇒ t
(palindromep "A man, a plan, a canal--Panama!") ⇒ nil

(remove-if-not #'alpha-char-p "A man, a plan, a canal--Panama!")
;Remove punctuation.
⇒ "AmanaplanacanalPanama"
```

```
(palindromep
 (remove-if-not #'alpha-char-p
  "A man, a plan, a canal--Panamal")) ⇒ t
```

```
(palindromep
 (remove-if-not
  #'alpha-char-p
  "Unremarkable was I ere I saw Elba Kramer, nu?")) ⇒ t
```

```
(palindromep
 (remove-if-not
  #'alpha-char-p
  "A man, a plan, a cat, a ham, a yak,
   a yam, a hat, a canal--Panamal")) ⇒ t
```

Altering the value of *var* in the body of the loop (by using `setq`, for example) will have unpredictable, possibly implementation-dependent results. A COMMON LISP compiler may choose to issue a warning if such a variable appears in a `setq`.

Compatibility note: The `dotimes` construct is the closest thing in COMMON LISP to the INTERLISP `rptq` construct.

See also `do-symbols`, `do-external-symbols`, and `do-all-symbols`.

Mapping

Mapping is a type of iteration in which a function is successively applied to pieces of one or more sequences. The result of the iteration is a sequence containing the respective results of the function applications. 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.

The function `map` may be used to map over any kind of sequence. The following functions operate only on lists.

<code>mapcar</code> <i>function list &rest more-lists</i>	[Function]
<code>maplist</code> <i>function list &rest more-lists</i>	[Function]
<code>mapc</code> <i>function list &rest more-lists</i>	[Function]
<code>mapi</code> <i>function list &rest more-lists</i>	[Function]
<code>mapcan</code> <i>function list &rest more-lists</i>	[Function]
<code>mapcon</code> <i>function list &rest more-lists</i>	[Function]

For each of these mapping functions, the first argument is a function and the rest must be lists. The function must take as many arguments as there are lists.

`mapcar` operates on successive elements of the lists. First the function is applied to the *car* of each list, then to the *cadr* of each list, and so on. (Ideally all the lists are the same length; if not, the iteration terminates when the shortest list runs out, and excess elements in other lists

are ignored.) The value returned by `mapcar` is a list of the results of the successive calls to the function. For example:

```
(mapcar #'abs '(3 -4 2 -5 -6)) ⇒ (3 4 2 5 6)
(mapcar #'cons '(a b c) '(1 2 3)) ⇒ ((a . 1) (b . 2) (c . 3))
```

`maplist` is like `mapcar` except that the function is applied to the list and successive `cdr`'s of that list rather than to successive elements of the list. For example:

```
(maplist #'(lambda (x) (cons 'foo x))
 '(a b c d))
⇒ ((foo a b c d) (foo b c d) (foo c d) (foo d))
(maplist #'(lambda (x) (if (member (car x) (cdr x)) 0 1))
 '(a b a c d b c))
⇒ (0 0 1 0 1 1 1)
;An entry is 1 if the corresponding element of the input
; list was the last instance of that element in the input list.
```

`mapl` and `mapc` are like `maplist` and `mapcar` respectively, except that they do not accumulate the results of calling the function.

Compatibility note: In all LISP systems since LISP 1.5, `mapl` has been called `map`. In the chapter on sequences it is explained why this was a bad choice. Here the name `map` is used for the far more useful generic sequence mapper, in closer accordance to the computer science literature, especially the growing body of papers on functional programming.

These functions are used when the function is being called merely for its side effects, rather than its returned values. The value returned by `mapl` or `mapc` is the second argument, that is, the first sequence argument.

`mapcon` and `mapcon` are like `mapcar` and `maplist` respectively, except that they combine the results of the function using `nconc` instead of `list`. That is,

```
(mapcon f x1 . . . xn)
≡ (apply #'nconc (maplist f x1 . . . xn))
```

and similarly for the relationship between `mapcon` and `mapcar`. Conceptually, these functions allow the mapped function to return a variable number of items to be put into the output list. This is particularly useful for effectively returning zero or one item:

```
(mapcon #'(lambda (x) (and (numberp x) (list x)))
 '(a 1 b c 3 4 d 5))
⇒ (1 3 4 5)
```

In this case the function serves as a filter; this is a standard LISP idiom using `mapcon`. (The function `remove-if-not` might have been useful in this particular context, however.) Remember that `nconc` is a destructive operation, and therefore so are `mapcon` and `mapcon`; the lists returned by the *function* are altered in order to concatenate them.

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

The functional argument to a mapping function must be acceptable to `apply`; it cannot be a macro or the name of a special form. Of course, there is nothing wrong with using a function that has `&optional` and `&rest` parameters as the functional argument.

The “Program Feature”

LISP implementations since LISP 1.5 have had what was originally called “the program feature,” as if it were impossible to write programs without it! The `prog` construct allows one to write in an ALGOL-like or FORTRAN-like statement-oriented style, using `go` statements that can refer to tags in the body of the `prog`. Modern LISP programming style tends to use `prog` rather infrequently. The various iteration constructs, such as `do`, have bodies with the characteristics of a `prog`. (However, the ability to use `go` statements within iteration constructs is very seldom used in practice.)

Three distinct operations are performed by `prog`: it binds local variables, it permits use of the `return` statement, and it permits use of the `go` statement. In COMMON LISP, these three operations have been separated into three distinct constructs: `let`, `block`, and `tagbody`. These three constructs may be used independently as building blocks for other types of constructs.

`tagbody {tag | statement}*` [Special form]

The part of a `tagbody` after the variable list is called the *body*. An item in the body may be a symbol or an integer, in which case it is called a *tag*, or an item in the body may be a list, in which case it is called a *statement*.

Each element of the body is processed from left to right. A *tag* is ignored; a *statement* is evaluated, and its results are discarded. If the end of the body is reached, the `tagbody` returns `nil`.

If `(go tag)` is evaluated, control jumps to the part of the body labelled with the *tag*.

Compatibility note: The “computed `go`” feature of MACLISP is not supported. The syntax of a computed `go` is idiosyncratic, and the feature is not supported by ZETALISP, NIL, or INTERLISP. The computed `go` has been infrequently used in MACLISP anyway, and is easily simulated with no loss of efficiency by using a `case` statement each of whose clauses performs a (non-computed) `go`.

The scope of the tags established by a `tagbody` is lexical, and the extent is dynamic. Once a `tagbody` construct has been exited, it is no longer legal to `go` to a *tag* in its body. It is permissible for a `go` to jump to a `tagbody` that is not the innermost `tagbody` construct containing that `go`; the tags established by a `tagbody` will only shadow other tags of like name.

The lexical scoping of the `go` targets named by tags is fully general and has consequences that may be surprising to users and implementors of other LISP systems. For example, the `go` in the following example actually does “work” in COMMON LISP as one might expect:

```
(tagbody
  (catch 'stuff
    (mapcar #'(lambda (x) (if (numberp x)
                            (hairyfun x)
                            (go lose)))
            items))
  (return)
lose
  (error "I lost big!"))
```

Depending on the situation, a `go` in COMMON LISP does not necessarily correspond to a simple machine “jump” instruction! A `go` can break up catchers if necessary to get to the target. It is possible for a “closure” created by function for a lambda-expression to refer to a `go` target as long as the tag is lexically apparent. See Chapter 3 for an elaborate example of this.

```
prog ({var | (var [init])}*) {declaration}* {tag | statement}*           [Macro]
prog* ({var | (var [init])}*) {declaration}* {tag | statement}*       [Macro]
```

The `prog` construct is a synthesis of `let`, `block`, and `tagbody`, allowing bound variables and the use of `return` and `go` within a single construct. A typical `prog` construct looks like this:

```
(prog (var1 var2 (var3 init3) var4 (var5 init5))
  {declaration}*
  statement1
tag1
  statement2
  statement3
  statement4
tag2
  statement5
  . . .
)
```

The list after the keyword `prog` is a set of specifications for binding `var1`, `var2`, etc., which are temporary variables bound locally to the `prog`. This list is processed exactly as the list in a `let` statement: first all the `init` forms are evaluated from left to right (where `nil` is used for any omitted `init` form), and then the variables are all bound in parallel to the respective results. Any `declaration` appearing in the `prog` is used as if appearing at the top of the `let` body.

The body of the `prog` is executed as if it were a `tagbody` construct; the `go` statement may be used to transfer control to a `tag`.

A `prog` implicitly establishes a block named `nil` around the entire `prog` construct, so that `return` may be used at any time to exit from the `prog` construct.

Here is a fine example of what can be done with prog:

```
(defun king-of-confusion (w)
  "Take a cons of two lists and make a list of conses.
  Think of this function as being like a zipper."
  (prog (x y z)           ;Initialize x, y, z to nil
        (setq y (car w) z (cdr w))
    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
      (cerror "Will self-pair extraneous items"
              "Mismatch - gleep! S" y)
      (setq z y)
      (go rejoin)))
```

which is accomplished somewhat more perspicuously by:

```
(defun prince-of-clarity (w)
  "Take a cons of two lists and make a list of conses.
  Think of this function as being like a zipper."
  (do ((y (car w) (cdr y))
       (z (cdr w) (cdr z))
       (x '() (cons (cons (car y) (car z)) x)))
      ((null y) x)
    (when (null z)
      (cerror "Will self-pair extraneous items"
              "Mismatch - gleep! S" y)
      (setq z y))))
```

The prog construct may be explained in terms of the simpler constructs block, let, and tagbody as follows:

```
(prog variable-list {declaration}* . body)
≡ (block nil (let variable-list {declaration}* (tagbody . body)))
```

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 that the *init* form for each one can use the values of previous ones. Therefore prog* is to prog as let* is to let. For example,

```
(prog* ((y z) (x (car y)))
  (return x))
```

returns the *car* of the value of z.

`go tag`

[*Special form*]

The (`go tag`) special form is used to do a “go to” within a `tagbody` construct. The *tag* must be a symbol or an integer; the *tag* is not evaluated. `go` transfers control to the point in the body labelled by a `tag eq` to the one given. If there is no such tag in the body, the bodies of lexically containing `tagbody` constructs (if any) are examined as well. It is an error if there is no matching tag lexically visible to the point of the `go`.

The `go` form does not ever return a value.

As a matter of style, it is recommended that the user think twice before using a `go`. Most purposes of `go` can be accomplished with one of the iteration primitives, nested conditional forms, or `return-from`. If the use of `go` seems to be unavoidable, perhaps the control structure implemented by `go` should be packaged as a macro definition.

Multiple Values

Ordinarily the result of calling a LISP function is a single LISP object. Sometimes, however, it is convenient for a function to compute several objects and return them. COMMON LISP provides a mechanism for handling multiple values directly. This mechanism is cleaner and more efficient than the usual tricks involving returning a list of results or stashing results in global variables.

Constructs for Handling Multiple Values

Normally multiple values are not used. Special forms are required both to *produce* multiple values and to *receive* them. If the caller of a function does not request multiple values, but the called function produces multiple values, then the first value is given to the caller and all others are discarded; if the called function produces zero values, then the caller gets `nil` as a value.

The primary primitive for producing multiple values is `values`, which takes any number of arguments and returns that many values. If the last form in the body of a function is a `values` with three arguments, then a call to that function will return three values. Other special forms also produce multiple values, but they can be described in terms of `values`. Some built-in COMMON LISP functions, such as `floor`, return multiple values; those that do are so documented.

The special forms and macros for receiving multiple values are as follows:

`multiple-value-list`
`multiple-value-call`
`multiple-value-prog1`
`multiple-value-bind`
`multiple-value-setq`

These specify a form to evaluate and an indication of where to put the values returned by that form.

values &rest *args*

[*Function*]

All of the arguments are returned, in order, as values. For example:

```
(defun polar (x y)
  (values (sqrt (+ (* x x) (* y y))) (atan y x)))

(multiple-value-bind (r theta) (polar 3.0 4.0)
  (vector r theta))
⇒ #(5.0 0.9272952)
```

The expression (values) returns zero values. This is the standard idiom for returning no values from a function.

Sometimes it is desirable to indicate explicitly that a function will return exactly one value. For example, the function

```
(defun foo (x y)
  (floor (+ x y) y))
```

will return two values because floor returns two values. It may be that the second value makes no sense, or that for efficiency reasons it is desired not to compute the second value. The values function is the standard idiom for indicating that only one value is to be returned, as shown in the following example.

```
(defun foo (x y)
  (values (floor (+ x y) y)))
```

This works because values returns exactly *one* value for each of its argument forms; as for any function call, if any argument form to values produces more than one value, all but the first are discarded.

There is absolutely no way in COMMON LISP for a caller to distinguish between returning a single value in the ordinary manner and returning exactly one “multiple value.” For example, the values returned by the expressions (+ 1 2) and (values (+ 1 2)) are identical in every respect: the single value 3.

multiple-values-limit

[*Constant*]

The value of multiple-values-limit is a positive integer that is the upper exclusive bound on the number of values that may be returned from a function. This bound depends on the implementation, but will not be smaller than 20. (Implementors are encouraged to make this limit as large as practicable without sacrificing performance.) See lambda-parameters-limit and call-arguments-limit.

values-list *list*

[*Function*]

All of the elements of *list* are returned as multiple values. For example:

```
(values-list (list a b c)) ≡ (values a b c)
```

In general,

$(\text{values-list } list) \equiv (\text{apply } \#'\text{values } list)$

but `values-list` may be clearer or more efficient.

`multiple-value-list` *form*

[*Macro*]

`multiple-value-list` evaluates *form* and returns a list of the multiple values it returned. For example:

$(\text{multiple-value-list } (\text{floor } -3 \ 4)) \Rightarrow (-1 \ 1)$

`multiple-value-list` and `values-list` are therefore inverses of each other.

`multiple-value-call` *function* *{form}**

[*Special form*]

`multiple-value-call` first evaluates *function* to obtain a function and then evaluates all of the *forms*. All the values of the *forms* are gathered together (not just one value from each) and are all given as arguments to the function. The result of `multiple-value-call` is whatever is returned by the function. For example:

$(+ (\text{floor } 5 \ 3) (\text{floor } 19 \ 4))$
 $\equiv (+ \ 1 \ 4) \Rightarrow 5$

$(\text{multiple-value-call } \#'+ (\text{floor } 5 \ 3) (\text{floor } 19 \ 4))$
 $\equiv (+ \ 1 \ 2 \ 4 \ 3) \Rightarrow 10$

$(\text{multiple-value-list } form) \equiv (\text{multiple-value-call } \#'\text{list } form)$

`multiple-value-prog1` *form* *{form}**

[*Special form*]

`multiple-value-prog1` evaluates the first *form* and saves all the values produced by that form. It then evaluates the other *forms* from left to right, discarding their values. The values produced by the first *form* are returned by `multiple-value-prog1`. See `prog1`, which always returns a single value.

`multiple-value-bind` (*{var}**) *values-form* *{declaration}** *{form}**

[*Macro*]

The *values-form* is evaluated, and each of the variables *var* is bound to the respective value returned by that form. If there are more variables than values returned, extra values of `nil` are given to the remaining variables. If there are more values than variables, the excess values are simply discarded. The variables are bound to the values over the execution of the forms, which make up an implicit `progn`. For example:

$(\text{multiple-value-bind } (x) (\text{floor } 5 \ 3) (\text{list } x)) \Rightarrow (1)$

$(\text{multiple-value-bind } (x \ y) (\text{floor } 5 \ 3) (\text{list } x \ y)) \Rightarrow (1 \ 2)$

$(\text{multiple-value-bind } (x \ y \ z) (\text{floor } 5 \ 3) (\text{list } x \ y \ z)) \Rightarrow (1 \ 2 \ \text{nil})$

multiple-value-setq *variables form*

[Macro]

The *variables* must be a list of variables. The *form* is evaluated, and the variables are *set* (not bound) to the values returned by that form. If there are more variables than values returned, extra values of `nil` are assigned to the remaining variables. If there are more values than variables, the excess values are simply discarded.

Compatibility note: In ZETALISP this is called `multiple-value`. The added clarity of the name `multiple-value-setq` in COMMON LISP was deemed worth the incompatibility with ZETALISP.

`multiple-value-setq` always returns a single value, which is the first value returned by *form*, or `nil` if *form* produces zero values.

Rules Governing the Passing of Multiple Values

It is often the case that the value of a special form or macro call is defined to be the value of one of its subforms. 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, then the original form will also produce all of those values. This *passing back* of multiple values of course has no effect unless eventually one of the special forms for receiving multiple values is reached.

To be explicit, multiple values can result from a special form under precisely these circumstances:

Evaluation and Application

- `eval` returns multiple values if the form given it to evaluate produces multiple values.
- `apply`, `funcall`, and `multiple-value-call` pass back multiple values from the function applied or called.

Implicit progn contexts

- The special form `progn` passes back multiple values resulting from evaluation of the last subform. Other situations referred to as “implicit `progn`,” where several forms are evaluated and the results of all but the last form are discarded, also pass back multiple values from the last form. These situations include the body of a lambda-expression, in particular those constructed by `defun`, `defmacro`, and `deftype`. Also included are bodies of the constructs `eval-when`, `progv`, `let`, `let*`, `when`, `unless`, `block`, `multiple-value-bind`, and `catch`, as well as clauses in such conditional constructs as `case`, `typecase`, `ecase`, `etypecase`, `coase`, and `ctypecase`.

Conditional constructs

- `if` passes back multiple values from whichever subform is selected (the *then* form or the *else* form).
- `and` and `or` pass back multiple values from the last subform but not from subforms other than the last.

- `cond` passes back multiple values from the last subform of the implicit `progn` of the selected clause. If, however, the clause selected is a singleton clause, then only a single value (the non-`nil` predicate value) is returned. This is true even if the singleton clause is the last clause of the `cond`. It is *not* permitted to treat a final clause (`x`) as being the same as `(t x)` for this reason; the latter passes back multiple values from the form `x`.

Returning from a block

- The `block` construct passes back multiple values from its last subform when it exits normally. If `return-from` (or `return`) is used to terminate the block prematurely, then `return-from` passes back multiple values from its subform as the values of the terminated block. Other constructs that create implicit blocks, such as `do`, `dolist`, `dotimes`, `prog`, and `prog*`, also pass back multiple values specified by `return-from` (or `return`).
- `do` passes back multiple values from the last form of the exit clause, exactly as if the exit clause were a `cond` clause. Similarly, `dolist` and `dotimes` pass back multiple values from the *resultform* if that is executed. These situations are all examples of implicit uses of `return-from`.

Throwing out of a catch

- The `catch` construct returns multiple values if the result form in a `throw` exiting from such a `catch` produces multiple values.

Miscellaneous situations

- `multiple-value-prog1` passes back multiple values from its first subform. However, `prog1` always returns a single value.
- `unwind-protect` returns multiple values if the form it protects returns multiple values.
- `the` returns multiple values if the form it contains returns multiple values.

Among special forms that *never* pass back multiple values are `setq`, `multiple-value-setq`, `prog1`, and `prog2`. The conventional way to force only one value to be returned from a form `x` is to write `(values x)`.

The most important rule about multiple values is: No matter how many values a form produces, if the form is an argument form in a function call, then exactly *one* value (the first one) is used.

For example, if you write `(cons (floor x))`, then `cons` will always receive *exactly* one argument (which is of course an error), even though `floor` returns two values. To pass both values from `floor` to `cons`, one must write something like `(multiple-value-call #'cons (floor x))`. In an ordinary function call, each argument form produces exactly *one* argument; if such a form returns zero values, `nil` is used for the argument, and if more than one value, all but the first are discarded. Similarly, conditional constructs such as `if` that test the value of a form will use exactly one value, the first one, from that form and discard the rest; such constructs will use `nil` as the test value if zero values are returned.

Dynamic Non-local Exits

COMMON LISP provides a facility for exiting from a complex process in a non-local, dynamically scoped manner. There are two classes of special forms for this purpose, called *catch* forms and *throw* forms, or simply *catches* and *throws*. A *catch* form evaluates some subforms in such a way that, if a *throw* form is executed during such evaluation, the evaluation is aborted at that point and the *catch* form immediately returns a value specified by the *throw*. Unlike *block* and *return* ("Blocks and Exits"), which allow for exiting a *block* form from any point lexically within the body of the *block*, the *catch/throw* mechanism works even if the *throw* form is not textually within the body of the *catch* form. The *throw* need only occur within the extent (time span) of the evaluation of the body of the *catch*. This is analogous to the distinction between dynamically bound (special) variables and lexically bound (local) variables.

*catch tag {form}** [Special form]

The *catch* special form serves as a target for transfer of control by *throw*. The form *tag* is evaluated first to produce an object that names the *catch*; it may be any LISP object. A catcher is then established with the object as the *tag*. The *forms* are evaluated as an implicit *progn*, and the results of the last form are returned, except that if during the evaluation of the *forms* a *throw* should be executed such that the *tag* of the *throw* matches (is *eq* to) the *tag* of the *catch* and the catcher is the most recent outstanding catcher with that *tag*, then the evaluation of the *forms* is aborted and the results specified by the *throw* are immediately returned from the *catch* expression. The catcher established by the *catch* expression is disestablished just before the results are returned.

The *tag* is used to match *throws* with *catches*. (*catch 'foo form*) will catch a (*throw 'foo form*) but not a (*throw 'bar form*). It is an error if *throw* is done when there is no suitable *catch* ready to catch it.

Catch tags are compared using *eq*, not *eq!*; therefore numbers and characters should not be used as *catch tags*.

Compatibility note: The name *catch* comes from MACLISP, but the syntax of *catch* in COMMON LISP is different. The MACLISP syntax was (*catch form tag*), where the *tag* was not evaluated.

*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 (start-motor)
      (drill-hole)
      (stop-motor))
```

The non-local exit facility of COMMON LISP creates a situation in which the above code won't work, however: if *drill-hole* should do a *throw* to a *catch* that is outside of the *progn* form (perhaps because the drill bit broke), then *(stop-motor)* will never be evaluated (and the motor

will presumably be left running). This is particularly likely if `drill-hole` causes a LISP error and the user tells the error-handler to give up and abort the computation. (A possibly more practical example might be:

```
(prog2 (open-a-file)
      (process-file)
      (close-the-file))
```

where it is desired always to close the file when the computation is terminated for whatever reason. This case is so important that COMMON LISP provides the special form `with-open-file` for this purpose.)

In order to allow the example hole-drilling program to work, it can be rewritten using `unwind-protect` as follows:

```
(unwind-protect
 (progn (start-motor)
       (drill-hole))
 (stop-motor))
```

If `drill-hole` does a throw that attempts to quit out of the `unwind-protect`, then `(stop-motor)` will be executed.

This example assumes that it is correct to call `stop-motor` even if the motor has not yet been started. Remember that an error or interrupt may cause an exit even before any initialization forms have been executed. Any state restoration code should operate correctly no matter where in the protected code an exit occurred. For example, the following code is not correct:

```
(unwind-protect
 (progn (incf *access-count*)
       (perform-access))
 (decf *access-count*))
```

If an exit occurs before completion of the `incf` operation the `decf` operation will be executed anyway, resulting in an incorrect value for `*access-count*`. The correct way to code this is as follows:

```
(let ((old-count *access-count*))
 (unwind-protect
 (progn (incf *access-count*)
       (perform-access))
 (setq *access-count* old-count)))
```

As a general rule, `unwind-protect` guarantees to execute the *cleanup-forms* before exiting, whether it terminates normally or is aborted by a throw of some kind. (If, however, an exit occurs during execution of the *cleanup-forms*, no special action is taken. The *cleanup-forms* of an `unwind-protect` are not protected by that `unwind-protect`, though they may be protected if that `unwind-protect` occurs within the protected form of another `unwind-protect`.) `unwind-protect` returns whatever results from evaluation of the *protected-form* and discards all the results from the *cleanup-forms*.

It should be emphasized that `unwind-protect` protects against *all* attempts to exit from the protected form, including not only such “dynamic exit” facilities as `throw` but also such “lexical exit” facilities as `go` and `return-from`. Consider this situation:

```
(tagbody
  (let ((x 3))
    (unwind-protect
      (if (numberp x) (go out))
      (print x)))
  out
  . . .)
```

When the `go` is executed, the call to `print` is executed first, and then the transfer of control to the tag `out` is completed.

`throw tag result`

[*Special form*]

The `throw` special form transfers control to a matching `catch` construct. The *tag* is evaluated first to produce an object called the throw tag; then the *result* form is evaluated, and its results are saved (if the *result* form produces multiple values, then *all* the values are saved). The most recent outstanding `catch` whose tag matches the throw tag is exited; the saved results are returned as the value(s) of the `catch`. A `catch` matches only if the catch tag is `eq` to the throw tag.

In the process, dynamic variable bindings are undone back to the point of the `catch`, and any intervening `unwind-protect` cleanup code is executed. The *result* form is evaluated before the unwinding process commences, and whatever results it produces are returned from the `catch`.

If there is no outstanding catcher whose tag matches the throw tag, no unwinding of the stack is performed, and an error is signalled. When the error is signalled, the outstanding catchers and the dynamic variable bindings are those in force at the point of the throw.

Implementation note: These requirements imply that throwing should typically make two passes over the control stack. In the first pass it simply searches for a matching `catch`. In this search every `catch` must be considered, but every `unwind-protect` should be ignored. On the second pass the stack is actually unwound, one frame at a time, undoing dynamic bindings and outstanding `unwind-protect` constructs in reverse order of creation until the matching `catch` is reached.

Compatibility note: The name `throw` comes from `MACLISP`, but the syntax of `throw` in `COMMON LISP` is different. The `MACLISP` syntax was `(throw form tag)`, where the *tag* was not evaluated.

End of Chapter

Chapter 8

Macros

The COMMON LISP macro facility allows the user to define arbitrary functions that convert certain LISP forms into different forms before evaluating or compiling them. This is done at the expression level, not at the character-string level as in most other languages. Macros are important in the writing of good code: they make it possible to write code that is clear and elegant at the user level, but that is converted to a more complex or more efficient internal form for execution.

When `eval` is given a list whose *car* is a symbol, it looks for local definitions of that symbol (by `let`, `labels`, and `macrolet`); if that fails, it looks for a global definition. If the definition is a macro definition, then the original list is said to be a *macro call*. Associated with the definition will be a function of two arguments, called the *expansion function*. This function is called with the entire macro call as its first argument (the second argument is a lexical environment); it must return some new LISP form, called the *expansion* of the macro call. (Actually, a more general mechanism is involved; see `macroexpand`.) This expansion is then evaluated in place of the original form.

When a function is being compiled, any macros it contains are expanded at compilation time. This means that a macro definition must be seen by the compiler before the first use of the macro.

More generally, an implementation of COMMON LISP has great latitude in deciding exactly when to expand macro calls within a program. For example, it is acceptable for the `defun` special form to expand all macro calls within its body at the time the `defun` form is executed and record the fully expanded body as the body of the function being defined. (An implementation might even choose always to compile functions defined by `defun`, even when operating in an "interpretive" mode!)

Macros should be written in such a way as to depend as little as possible on the execution environment to produce a correct expansion. To ensure consistent behavior, it is best to ensure that all macro definitions are available, whether to the interpreter or compiler, before any code containing calls to those macros is introduced.

In COMMON LISP, macros are not functions. In particular, macros cannot be used as functional arguments to such functions as `apply`, `funcall`, or `map`; in such situations, the list representing the "original macro call" does not exist, and cannot exist, because in some sense the arguments have already been evaluated.

Macro Definition

The function `macro-function` determines whether a given symbol is the name of a macro. The `defmacro` construct provides a convenient way to define new macros.

macro-function *symbol*

[Function]

The argument must be a symbol. If the symbol has a global function definition that is a macro definition, then the expansion function (a function of two arguments, the macro-call form and an environment) is returned. If the symbol has no global function definition, or has a definition as an ordinary function or as a special form but not as a macro, then `nil` is returned. The function `macroexpand` is the best way to invoke the expansion function.

It is possible for *both* `macro-function` and `special-form-p` to be true of a symbol. This is possible because an implementation is permitted to implement any macro also as a special form for speed. On the other hand, the macro definition must be available for use by programs that understand only the standard special forms listed in Table 5-1.

`macro-function` cannot be used to determine whether a symbol names a locally defined macro established by `macrolet`; `macro-function` can examine only global definitions.

`self` may be used with `macro-function` to install a macro as a symbol's global function definition:

```
(self (macro-function symbol) fn)
```

The value installed must be a function that accepts two arguments, an entire macro call and an environment, and computes the expansion for that call. Performing this operation causes the symbol to have *only* that macro definition as its global function definition; any previous definition, whether as a macro or as a function, is lost. It is an error to attempt to redefine the name of a special form (see Table 5-1).

```
defmacro name lambda-list {declaration | doc-string}* {form}*
```

[Macro]

`defmacro` is a macro-defining macro that arranges to decompose the macro-call form in an elegant and useful way. `defmacro` has essentially the same syntax as `defun`: *name* is the symbol whose macro definition we are creating, *lambda-list* is similar in form to a lambda-list, and the *forms* constitute the body of the expander function. The `defmacro` construct arranges to install this expander function, as the global macro definition of *name*. The expander function is effectively defined in the *global* environment; lexically scoped entities established outside the `defmacro` form that would ordinarily be lexically apparent are not visible within the body of the expansion function. The *name* is returned as the value of the `defmacro` form.

If we view the macro call as a list containing a function name and some argument forms, in effect the expander function and the list of (unevaluated) argument forms is given to apply. The parameter specifiers are processed as for any lambda-expression, using the macro-call argument forms as the arguments. Then the body forms are evaluated as an implicit `progn`, and the value of the last form is returned as the expansion of the macro call.

If the optional documentation string *doc-string* is present (if not followed by a declaration, it may be present only if at least one *form* is also specified, as it is otherwise taken to be a *form*), then it is attached to the *name* as a documentation string of type function; see documentation.

Like the lambda-list in a `defun`, a `defmacro` *lambda-list* may contain the lambda-list keywords `&optional`, `&rest`, `&key`, `&allow-other-keys`, and `&aux`. For `&optional` and `&key` parameters, initialization forms and "supplied-p" parameters may be specified, just as for `defun`. Three additional markers are allowed in `defmacro` variable lists only:

- `&body` This is identical in function to `&rest`, but it informs certain output-formatting and editing functions that the remainder of the form is treated as a body, and should be indented accordingly. (Only one of `&body` or `&rest` may be used.)
- `&whole` This is followed by a single variable that is bound to the entire macro-call form; this is the value that the macro definition function receives as its single argument. `&whole` and the following variable should appear first in the lambda-list, before any other parameter or lambda-list keyword.
- `&environment` This is followed by a single variable that is bound to an environment representing the lexical environment in which the macro call is to be interpreted. This environment may not be the complete lexical environment; it should be used only with the function `macroexpand` for the sake of any local macro definitions that the `macrolet` construct may have established within that lexical environment. This is useful primarily in the rare cases where a macro definition must explicitly expand any macros in a subform of the macro call before computing its own expansion.

See `lambda-list-keywords`.

`defmacro`, unlike any other COMMON LISP construct that has a lambda-list as part of its syntax, provides an additional facility known as *destructuring*. Anywhere in the lambda-list where a parameter name may appear, and where ordinary lambda-list syntax (as described in “Lambda-Expressions,” Chapter 5) does not otherwise allow a list, a lambda-list may appear in place of the parameter name. When this is done, then the argument form that would match the parameter is treated as a (possibly dotted) list, to be used as an argument forms list for satisfying the parameters in the embedded lambda-list. As an example, one could write the macro definition for `dolist` in this manner:

```
(defmacro dolist ((var listform &optional resultform)
                 &rest body)
  . . . )
```

More examples of embedded lambda-lists in `defmacro` are shown below.

Another destructuring rule is that `defmacro` allows any lambda-list (whether top-level or embedded) to be dotted, ending in a parameter name. This situation is treated exactly as if the parameter name that ends the list had appeared preceded by `&rest`. For example, the definition skeleton for `dolist` shown above could instead have been written

```
(defmacro dolist ((var listform &optional resultform)
                  . body)
  . . . )
```

If the compiler encounters a `defmacro`, the new macro is added to the compilation environment, and a compiled form of the expansion function is also added to the output file so that the new macro will be operative at run time. If this is not the desired effect, the `defmacro` form can be wrapped in an `eval-when` construct.

It is permissible to use `defmacro` to redefine a macro (for example, to install a corrected version of an incorrect definition!), or to redefine a function as a macro. It is an error to attempt to redefine the name of a special form (see Table 5-1) as a macro.

See also `macrolet`, which establishes macro definitions over a restricted lexical scope.

Suppose, for the sake of example, that it were desirable to implement a conditional construct analogous to the FORTRAN arithmetic IF statement. (This of course requires a certain stretching of the imagination and suspension of disbelief.) The construct should accept four forms: a *test-value*, a *neg-form*, a *zero-form*, and a *pos-form*. One of the last three forms is chosen to be executed according to whether the value of the *test-form* is positive, negative, or zero. Using `defmacro`, a definition for such a construct might look like this:

```
(defmacro arithmetic-if (test neg-form zero-form pos-form)
  (let ((var (gensym)))
    `(let ((.var ,test))
      (cond ((< .var 0) ,neg-form)
            ((= .var 0) ,zero-form)
            (t ,pos-form))))))
```

Note the use of the backquote facility in this definition. See "Macro Characters," Chapter 22. Also note the use of `gensym` to generate a new variable name. This is necessary to avoid conflict with any variables that might be referred to in *neg-form*, *zero-form*, or *pos-form*.

If the form is executed by the interpreter, it will cause the function definition of the symbol `arithmetic-if` to be a macro associated with which is a two-argument expansion function roughly equivalent to:

```
(lambda (calling-form environment)
  (declare (ignore environment))
  (let ((var (gensym)))
    (list 'let
          (list (list 'var (cadr calling-form)))
          (list 'cond
                (list (list '< var '0) (caddr calling-form))
                (list (list '= var '0) (caddr calling-form))
                (list 't (fifth calling-form)))))))
```

The lambda-expression is produced by the `defmacro` declaration. The calls to `list` are the (hypothetical) result of the backquote (`'`) macro character and its associated commas. The precise macro expansion function may depend on the implementation, for example providing some degree of explicit error checking on the number of argument forms in the macro call.

Now, if `eval` encounters

```
(arithmetic-if (- x 4.0)
  (- x)
  (error "Strange zero")
  x)
```

this will be expanded into something like

```
(let ((g407 (- x 4.0)))
  (cond ((< g407 0) (- x))
        ((= g407 0) (error "Strange zero"))
        (t x)))
```

and `eval` tries again on this new form. (It should be clear now that the backquote facility is very useful in writing macros, since the form to be returned is normally a complex list structure, typically consisting of a mostly constant template with a few evaluated forms here and there. The backquote template provides a “picture” of the resulting code, with places to be filled in indicated by preceding commas.)

To expand on this example, stretching credibility to its limit, we might allow the *pos-form* and *zero-form* to be omitted, allowing their values to default to `nil`, in much the same way that the *else* form of a COMMON LISP `if` construct may be omitted:

```
(defmacro arithmetic-if (test neg-form
                        &optional zero-form pos-form)
  (let ((var (gensym)))
    `(let ((,var ,test))
      (cond ((< ,var 0) ,neg-form)
            ((= ,var 0) ,zero-form)
            (t ,pos-form))))))
```

Then one could write

```
(arithmetic-if (- x 4.0) (print x))
```

which would be expanded into something like

```
(let ((g408 (- x 4.0)))
  (cond ((< g408 0) (print x))
        ((= g408 0) nil)
        (t nil)))
```

The resulting code is correct but rather silly-looking. One might rewrite the macro definition to produce better code when *pos-form* and possibly *zero-form* are omitted, or one might simply rely on the COMMON LISP implementation to provide a compiler smart enough to improve the code itself.

Destructuring is a very powerful facility that allows the `defmacro` lambda-list to express the structure of a complicated macro-call syntax. If no lambda-list keywords appear, then the `defmacro` lambda-list is simply a list, nested to some extent, containing parameter names at the leaves. The macro-call form must have the same list structure. For example, consider this macro definition:

```
(defmacro halibut ((mouth eye1 eye2)
                  ((fin1 length1) (fin2 length2))
                  tail)
  ... )
```

Now consider this macro call:

```
(halibut (m (car eyes) (cdr eyes))
         ((f1 (count-scales f1)) (f2 (count-scales f2)))
         my-favorite-tail)
```

This would cause the expansion function to receive the following values for its parameters:

<u>Parameter</u>	<u>Value</u>
mouth	m
eye1	(car eyes)
eye2	(cdr eyes)
fin1	f1
length1	(count-scales f1)
fin2	f2
length2	(count-scales f2)
tail	my-favorite-tail

The following macro call would be in error because there would be no argument form to match the parameter length1:

```
(halibut (m (car eyes) (cdr eyes))
         ((f1) (f2 (count-scales f2)))
         my-favorite-tail)
```

The following macro call would be in error because a symbol appears in the call where the structure of the lambda-list requires a list.

```
(halibut my-favorite-head
         ((f1 (count-scales f1)) (f2 (count-scales f2)))
         my-favorite-tail)
```

The fact that the value of the variable my-favorite-head might happen to be a list is irrelevant here. It is the macro call itself whose structure must match that of the defmacro lambda-list.

The use of lambda-list keywords adds even greater flexibility. For example, suppose it is convenient within the expansion function for halibut to be able to refer to the list whose components are called mouth, eye1, and eye2 as head. One may write this:

```
(defmacro halibut ((&whole head mouth eye1 eye2)
                  ((fin1 length1) (fin2 length2))
                  tail)
```

Now consider the same valid macro call as before:

```
(halibut (m (car eyes) (cdr eyes))
         ((f1 (count-scales f1)) (f2 (count-scales f2)))
         my-favorite-tail)
```

This would cause the expansion function to receive the same values for its parameters and also a value for the parameter head:

<u>Parameter</u>	<u>Value</u>
head	(m (car eyes) (cdr eyes))

The stipulation, that an embedded lambda-list is permitted only where ordinary lambda-list syntax would permit a parameter name but not a list, is made to prevent ambiguity. For example, one may not write

```
(defmacro loser (x &optional (a b &rest c) &rest z)
  . . . )
```

because ordinary lambda-list syntax does permit a list following `&optional`; the list `(a b &rest c)` would be interpreted as describing an optional parameter named `a` whose default value is that of the form `b`, with a supplied-p parameter named `&rest` (not legal), and an extraneous symbol `c` in the list (also not legal). An almost correct way to express this is

```
(defmacro loser (x &optional ((a b &rest c)) &rest z)
  . . . )
```

The extra set of parentheses removes the ambiguity. However, the definition is now incorrect because a macro call such as `(loser (car pool))` would not provide any argument form for the lambda-list `(a b &rest c)`, and so the default value against which to match the lambda-list would be `nil` because no explicit default value was specified. This is an error because `nil` is an empty list; it does not have forms to satisfy the parameters `a` and `b`. The fully correct definition would be either

```
(defmacro loser (x &optional ((a b &rest c) '(nil nil)) &rest z)
  . . . )
```

or

```
(defmacro loser (x &optional ((&optional a b &rest c)) &rest z)
  . . . )
```

These differ slightly: the first requires that if the macro call specifies `a` explicitly then it must also specify `b` explicitly, whereas the second does not have this requirement. For example,

```
(loser (car pool) ((+ x 1)))
```

would be a valid call for the second definition but not for the first.

Macro Expansion

The `macroexpand` function is the conventional means for expanding a macro call. A hook is provided for a user function to gain control during the expansion process.

```
macroexpand form &optional env [Function]
macroexpand-1 form &optional env [Function]
```

If *form* is a macro call, then `macroexpand-1` will expand the macro call *once* and return two values: the expansion and `t`. If *form* is not a macro call, then the two values *form* and `nil` are returned.

A *form* is considered to be a macro call only if it is a cons whose *car* is a symbol that names a macro. The environment *env* is similar to that used within the evaluator (see *evalhook*); it defaults to a null environment. Any local macro definitions established within *env* by *macrolet* will be considered. If only *form* is given as an argument, then the environment is effectively null, and only global macro definitions (as established by *defmacro*) will be considered.

Macro expansion is carried out as follows. Once a *macroexpand-1* has determined that a symbol names a macro, it obtains the expansion function for that macro. The value of the variable **macroexpand-hook** is then called as a function of three arguments: the expansion function, the *form*, and the environment *env*. The value returned from this call is taken to be the expansion of the macro call. The initial value of **macroexpand-hook** is *funcall*, and the net effect is to invoke the expansion function, giving it *form* and *env* as its two arguments. (The purpose of **macroexpand-hook** is to facilitate various techniques for improving interpretation speed by catching macro expansions.)

The evaluator expands macro calls as if through the use of *macroexpand-1*; the point is that *eval* also uses **macroexpand-hook**.

macroexpand is similar to *macroexpand-1*, but repeatedly expands *form* until it is no longer a macro call. (In effect, *macroexpand* simply calls *macroexpand-1* repeatedly until the second value returned is *nil*.) A second value of *t* or *nil* is returned as for *macroexpand-1*, indicating whether the original *form* was a macro call.

macroexpand-hook

[*Variable*]

The value of **macroexpand-hook** is used as the expansion interface hook by *macroexpand-1*.

End of Chapter

Chapter 9

Declarations

Declarations allow you to specify extra information about your program to the LISP system. With one exception, declarations are completely optional and correct declarations do not affect the meaning of a correct program. The exception is that special declarations *do* affect the interpretation of variable bindings and references, and so *must* be specified where appropriate. All other declarations are of an advisory nature, and may be used by the LISP system to aid the programmer by performing extra error checking or producing more efficient compiled code. Declarations are also a good way to add documentation to a program.

Note that it is considered an error for a program to violate a declaration (such as a type declaration), but an implementation is not required to detect such errors (though such detection, where feasible, is to be encouraged).

Declaration Syntax

The `declare` construct is used for embedding declarations within executable code. Global declarations and declarations that are computed by a program are established by the `proclaim` construct.

`declare {decl-spec}*` [*Special form*]

A `declare` form is known as a *declaration*. Declarations may occur only at the beginning of the bodies of certain special forms; that is, a declaration may occur only as a statement of such a special form, and all statements preceding it (if any) must also be `declare` forms (or possibly documentation strings, in some cases). Declarations may occur in lambda-expressions and in the forms listed here.

<code>defmacro</code>	<code>dotimes</code>
<code>defself</code>	<code>flet</code>
<code>deftype</code>	<code>labels</code>
<code>defun</code>	<code>let</code>
<code>do*</code>	<code>let*</code>
<code>do-all-symbols</code>	<code>locally</code>
<code>do-external-symbols</code>	<code>macrolet</code>
<code>do-symbols</code>	<code>multiple-value-bind</code>
<code>do</code>	<code>prog</code>
<code>dolist</code>	<code>prog*</code>

It is an error to attempt to evaluate a declaration. Those special forms that permit declarations to appear perform explicit checks for their presence.

Compatibility note: In MACLISP, `declare` is a special form that does nothing but return the symbol `declare` as its result. The MACLISP interpreter knows nothing about declarations but just blindly evaluates them, effectively ignoring them. The MACLISP compiler recognizes declarations but processes them simply by evaluating the subforms of the declaration in the compilation context. In COMMON LISP it is important that both the interpreter and compiler recognize declarations (especially *special* declarations) and treat them consistently, and so the rules about the structure and use of declarations have been made considerably more stringent. The odd tricks played in MACLISP by writing arbitrary forms to be evaluated within a `declare` form are better done in both MACLISP and COMMON LISP by using `eval-when`.

It is permissible for a macro call to expand into a declaration and be recognized as such, provided that the macro call appears where a declaration may legitimately appear. (However, a macro call may not appear in place of a *decl-spec*.)

Each *decl-spec* is a list whose *car* is a symbol specifying the kind of declaration to be made. Declarations may be divided into two classes: those that concern the bindings of variables, and those that do not. (The *special* declaration is the sole exception: it effectively falls into both classes, as explained below.) Those that concern variable bindings apply only to the bindings made by the form at the head of whose body they appear. For example, in

```
(defun foo (x)
  (declare (type float x)) . . .
  (let ((x 'a)) . . . )
  . . . )
```

the *type* declaration applies only to the outer binding of `x`, and not to the binding made in the `let`.

Compatibility note: This represents a difference from MACLISP, in which *type* declarations are pervasive.

Declarations that do not concern themselves with variable bindings are pervasive, affecting all code in the body of the special form. As an example of a pervasive declaration,

```
(defun foo (x y) (declare (notinline floor)) . . . )
```

advises that everywhere within the body of `foo` the function `floor` should not be open-coded but called as an out-of-line subroutine.

Some special forms contain pieces of code that, properly speaking, are not part of the body of the special form. Examples of this are initialization forms that provide values for bound variables, and the result forms of iteration constructs. In all cases such additional code is within the scope of any pervasive declarations appearing before the body of the special form. Non-pervasive declarations have no effect on such code, except (of course) in those situations where the code is defined to be within the scope of the variables affected by such non-pervasive declarations. For example:

```
(defun few (x &optional (y *print-circle*))
  (declare (special *print-circle*))
  . . . )
```

The reference to `*print-circle*` in the first line of this example is special because of the declaration in the second line.

```
(defun nonsense (k x z)
  (foo z x) ;First call to foo
  (let ((j (foo k x)) ;Second call to foo
        (x (* k k)))
    (declare (inline foo) (special x z))
    (foo x j z))) ;Third call to foo
```

In this rather nonsensical example, the `inline` declaration applies to the second and third calls to `foo`, but not to the first one. The `special` declaration of `x` causes the `let` form to make a special binding for `x`, and causes the reference to `x` in the body of the `let` to be a special reference. The reference to `x` in the second call to `foo` is also a special reference. The reference to `x` in the first call to `foo` is a local reference, not a special one. The `special` declaration of `z` causes the reference to `z` in the call to `foo` to be a special reference; it will not refer to the parameter to `nonsense` named `z`, because that parameter binding has not been declared to be `special`. (The `special` declaration of `z` does not appear in the body of the `defun`, but in an inner construct, and therefore does not affect the binding of the parameter.)

`locally` *{declaration}* {form}** [Macro]

This special form may be used to make local pervasive declarations where desired. It does not bind any variables and therefore cannot be used meaningfully for declarations of variable bindings. (Note that the `special` declaration may be used with `locally` to pervasively affect references to, rather than bindings of, variables.) For example:

```
(locally (declare (inline floor) (notinline car cdr))
  (declare (optimize space))
  (floor (car x) (cdr y)))
```

`proclaim` *decl-spec* [Function]

The function `proclaim` takes a *decl-spec* as its argument and puts it into effect globally. (Such a global declaration is called a *proclamation*.) Because `proclaim` is a function, its argument is always evaluated. This allows a program to compute a declaration and then put it into effect by calling `proclaim`.

Any variable names mentioned are assumed to refer to the dynamic values of the variable. For example, the proclamation

```
(proclaim '(type float tolerance))
```

once executed, specifies that the dynamic value of `tolerance` should always be a floating-point number. Similarly, any function names mentioned are assumed to refer to the global function definition.

A proclamation constitutes a universal declaration, always in force unless locally shadowed. For example:

```
(proclaim '(inline floor))
```

advises that `floor` should normally be open-coded in-line by the compiler (but in the situation

```
(defun foo (x y) (declare (notinline floor)) . . . )
```

it will be compiled out-of-line anyway in the body of `foo`, because of the shadowing local declaration to that effect).

As a special case (so to speak), `proclaim` treats a special *declaration-form* as applying to all bindings as well as to all references of the mentioned variables. For example, after

```
(proclaim '(special x))
```

then in a function definition such as

```
(defun example (x) . . . )
```

the parameter `x` will be bound as a special (dynamic) variable rather than as a lexical (static) variable. This facility should be used with caution. The usual way to define a globally special variable is with `defvar` or `defparameter`.

Declaration Specifiers

Here is a list of valid declaration specifiers for use in `declare`. A construct is said to be "affected" by a declaration if it occurs within the scope of a declaration.

`special`

`(special var1 var2 . . .)` specifies that all of the variables named are to be considered *special*. This specifier affects variable bindings but also pervasively affects references. All variable bindings affected are made to be dynamic bindings, and affected variable references refer to the current dynamic binding rather than the current local binding. For example:

```
(defun hack (thing *mod*) ;The binding of the parameter
  (declare (special *mod*)) ; *mod* is visible to hack1,
  (hack1 (car thing))) ; but not that of thing.
```

```
defun hack1 (arg) ;Declare references to *mod*
  (declare (special *mod*)) ; within hack1 to be special.
```

```
(if (atom arg) *mod*
    (cons (hack1 (car arg)) (hack1 (cdr arg))))
```

Note that it is conventional, though not required, to give special variables names that begin and end with an asterisk.

A special declaration does *not* affect bindings pervasively. Inner bindings of a variable implicitly shadow a special declaration and must be explicitly re-declared to be special. (However, a special proclamation *does* pervasively affect bindings; this exception is made for reasons of convenience and compatibility with MACLISP.) For example:

```
(proclaim '(special x)                ;x is always special.

(defun example (x y)
  (declare (special y))
  (let ((y 3) (x (* x 2)))
    (print (+ y (locally (declare (special y)) y)))
    (let ((y 4)) (declare (special y)) (foo x))))
```

In the contorted code above, the outermost and innermost bindings of *y* are special and therefore dynamically scoped, but the middle binding is lexically scoped. The two arguments to *+* are different, one being the value, which is 3, of the lexically bound variable *y*, and the other being the value of the special variable named *y* (a binding of which happens, coincidentally, to lexically surround it at an outer level). All the bindings of *x* and references to *x* are special, however, because of the proclamation that *x* is always special.

As a matter of style, use of special proclamations should be avoided. The *defvar* and *defparameter* macros are the conventional means for proclaiming special variables in a program.

type

(*type type var1 var2 . . .*) affects only variable bindings and specifies that the variables mentioned will take on values only of the specified type. In particular, values assigned to the variables by *setq*, as well as the initial values of the variables, must be of the specified type.

type

(*type var1 var2 . . .*) is an abbreviation for (*type type var1 var2 . . .*) provided that *type* is one of the symbols appearing in Table 4-1.

ftype

(*ftype type function-name-1 function-name-2 . . .*) specifies that the named functions will be of the functional type *type*, an example of which follows.

```
(declare (ftype (function (integer list) t) nth)
         (ftype (function (number) float) sin cos))
```

Note that rules of lexical scoping are observed; if one of the functions mentioned has a lexically apparent local definition (as made by *flet* or *labels*), then the declaration applies to that local definition and not to the global function definition.

function

(function *name arglist result-type1 result-type2 . . .*) is entirely equivalent to

(ftype (function *arglist result-type1 result-type2 . . .*) *name*)

but may be more convenient for some purposes. For example:

```
(declare (function nth (integer list) t)
         (function sin (number) float)
         (function cos (number) float))
```

The syntax mildly resembles that of `defun`: a function name, then an argument list, then a specification of results.

Note that rules of lexical scoping are observed; if one of the functions mentioned has a lexically apparent local definition (as made by `let` or `labels`), then the declaration applies to that local definition and not to the global function definition.

inline

(inline *function1 function2 . . .*) specifies that it is desirable for the compiler to open-code calls to the specified functions; that is, the code for a specified function should be integrated into the calling routine, appearing “in line” in place of a procedure call. This may achieve extra speed at the expense of debuggability (calls to functions compiled in-line cannot be traced, for example). This declaration is pervasive. Remember that a compiler is free to ignore this declaration. See `compile-file` for more information about `inline`.

Note that rules of lexical scoping are observed; if one of the functions mentioned has a lexically apparent local definition (as made by `let` or `labels`), then the declaration applies to that local definition and not to the global function definition.

notinline

(notinline *function1 function2 . . .*) specifies that it is *undesirable* to compile the specified functions in-line. This declaration is pervasive. A compiler is *not* free to ignore this declaration.

Note that the rules of lexical scoping are observed; if one of the functions mentioned has a lexically apparent local definition (as made by `let` or `labels`), then the declaration applies to that local definition and not to the global function definition.

ignore

(ignore *var1 var2 . . . varn*) affects only variable bindings and specifies that the bindings of the specified variables are never used. It is desirable for a compiler to issue a warning if a variable so declared is ever referred to or is also declared `special`, or if a variable is lexical, never referred to, and not declared to be ignored.

optimize

(optimize (*quality1 value1*) (*quality2 value2*) . . .) advises the compiler that each *quality* should be given attention according to the specified corresponding *value*. A quality is a symbol; standard qualities include speed (of the object code), space (both code size and run-time space), safety (run-time error checking), and compilation-speed (speed of the compilation process). Other qualities may be recognized by particular implementations. A *value* should be a non-negative integer, normally in the range 0 to 3. The value 0 means that the quality is totally unimportant, and 3 that the quality is extremely important; 1 and 2 are intermediate values, with 1 the “normal” or “usual” value. One may abbreviate (*quality 3*) to simply *quality*. This declaration is pervasive. For example:

```
(defun often-used-subroutine (x y)
  (declare (optimize (safety 2)))
  (error-check x y)
  (hairy-setup x)
  (do ((i 0 (+ i 1))
      (z x (cdr z)))
      ((null z) i)
    ;;This inner loop really needs to burn.
    (declare (optimize speed))
    (declare (fixnum i))
  )))
```

The compiler processes an optimize declaration by binding the following variables to the appropriate value (0-3):

optimize-safety	[Variable]
optimize-speed	[Variable]
optimize-space	[Variable]
optimize-compile-speed	[Variable]

The binding persists over the scope of the declaration, so a compiler-let or a macro expansion may refer to these variables. Users may test the values of these variables in writing their own optimizations, but should not set or rebound them.

declaration

(declaration *name1 name2* . . .) advises the compiler that each *namej* is a valid but non-standard declaration name. The purpose of this is to tell one compiler not to issue warnings for declarations meant for another compiler or other program processor. This kind of declaration may be used only as a proclamation. For example:

```
(proclaim '(declaration author
              target-language
              target-machine))

(proclaim '(target-language ada))

(proclaim '(target-machine IBM-650))

(defun strangep (x)
  (declare (author "Harry Tweeker"))
  (member x '(strange weird odd peculiar)))
```

An implementation is free to support other (implementation-dependent) declaration specifiers as well. On the other hand, a COMMON LISP compiler is free to ignore entire classes of declaration specifiers (for example, implementation-dependent declaration specifiers not supported by that compiler's implementation!), except for the `declaration` declaration specifier. Compiler implementors are encouraged, however, to program the compiler to issue by default a warning if the compiler finds a declaration specifier of a kind it never uses. Such a warning is required in any case if a declaration specifier is not one of those defined above and has not been declared in a `declaration` declaration.

Type Declaration for Forms

Frequently it is useful to declare that the value produced by the evaluation of some form will be of a particular type. Using `declare` one can declare the type of the value held by a bound variable, but there is no easy way to declare the type of the value of an unnamed form. For this purpose the special form is defined; (the *type form*) means that the value of *form* is declared to be of type *type*.

the *value-type form*

[*Special Form*]

The *form* is evaluated; whatever it produces is returned by the `the form`. In addition, it is an error if what is produced by the *form* does not conform to the data type specified by *value-type* (which is not evaluated). (A given implementation may or may not actually check for this error. Implementations are encouraged to make an explicit error check when running interpretively.) In effect, this declares that the user undertakes to guarantee that the values of the form will always be of the specified type. For example:

```
(the string (copy-seq x))           ;The result will be a string.
(the integer (+ x 3))              ;The result of + will be an integer.
(+ (the integer x) 3)              ;The value of x will be an integer.
(the (complex rational) (* z 3))
(the (unsigned-byte 8) (logand x mask))
```

The values type specifier may be used to indicate the types of multiple values:

```
(the (values integer integer) (floor x y))
(the (values string t)
    (gethash the-key the-string-table))
```

Compatibility note: This construct is borrowed from the INTERLISP DECL package; INTERLISP, however, allows an implicit `progn` after the type specifier rather than just a single form. The MACLISP `fixnum-identity` and `flonum-identity` constructs can be expressed as `(the fixnum x)` and `(the single-float x)`.

End of Chapter

Chapter 10

Symbols

A LISP symbol is a data object that has three user-visible components:

- The *property list* is a list that effectively provides each symbol with many modifiable named components.
- The *print name* must be a string, which is the sequence of characters used to identify the symbol. Symbols are of great use because a symbol can be located once its name is given (typed, say, on a keyboard). It is ordinarily not permitted to alter a symbol's print name.
- The *package cell* must refer to a package object. A package is a data structure used to locate a symbol once given the symbol's name. A symbol is uniquely identified by its name only when considered relative to a package. A symbol may appear in many packages, but it can be *owned* by at most one package. The package cell points to the owner, if any. Package cells are discussed along with packages in Chapter 11.

A symbol may actually have other components for use by the implementation. One of the more important uses of symbols is as names for program variables; it is frequently desirable for the implementor to use certain components of a symbol to implement the semantics of variables. See `symbol-value` and `symbol-function`. However, there are several possible implementation strategies, and so such possible components are not described here.

The Property List

Since its inception, LISP has associated with each symbol a kind of tabular data structure called a *property list* (*plist* for short). A property list contains zero or more entries; each entry associates with a key (called the *indicator*), which is typically a symbol, an arbitrary LISP object (called the *value* or, sometimes, the *property*). There are no duplications among the indicators; a property list may only have one property at a time with a given name. In this way, given a symbol and an indicator (another symbol), an associated value can be retrieved.

A property list is very similar in purpose 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 destructive operations that alter the property list rather than making a new one. Association lists, on the other hand, are normally augmented non-destructively (without side effects) by adding new entries to the front (see `acons` and `pairlis`).

A property list is implemented as a memory cell containing a list with an even number (possibly zero) of elements. (Usually this memory cell is the property-list cell of a symbol, but any memory cell acceptable to `setf` can be used if `getf` and `remf` are used.) Each pair of elements

in the list constitutes an entry; the first item is the indicator, and the second is the value. Because property-list functions are given the symbol and not the list itself, modifications to the property list can be recorded by storing back into the property-list cell of the symbol.

When a symbol is created, its property list is initially empty. Properties are created by using `get` within a `self` form.

COMMON LISP does not use a symbol's property list as extensively as earlier LISP implementations did. Less-used data, such as compiler, debugging, and documentation information, is kept on property lists in COMMON LISP.

Compatibility note: In older LISP implementations, the print name, value, and function definition of a symbol were kept on its property list. The value cell was introduced into MACLISP and INTERLISP to speed up access to variables; similarly for the print-name cell and function cell (MACLISP does not use a function cell). Recent LISP implementations such as SPICE LISP, ZETALISP, and NIL have introduced all of these cells plus the package cell. None of the MACLISP system property names (`expr`, `fexpr`, `macro`, `array`, `subr`, `lsubr`, `fsubr`, and in former times `value` and `pname`) exist in COMMON LISP.

In COMMON LISP, the notion of "disembodied property list" introduced in MACLISP is eliminated. It tended to be used for rather kludgy things, and in ZETALISP is often associated with the use of locatives (to make it "off by one" for searching alternating keyword lists). In COMMON LISP special `self`-like property-list functions are introduced: `getf` and `remf`.

`get symbol indicator &optional default` [Function]

`get` searches the property list of `symbol` for an indicator `eq` to `indicator`. The first argument must be a symbol. If one is found, then the corresponding value is returned; otherwise `default` is returned. If `default` is not specified, then `nil` is used for `default`. Note that there is no way to distinguish an absent property from one whose value is `default`.

```
(get x y) ≡ (getf (symbol-plist x) y)
```

Suppose that the property list of `foo` is `(bar t baz 3 hunoz "Huh?")`. Then, for example:

```
(get 'foo 'baz) ⇒ 3
(get 'foo 'hunoz) ⇒ "Huh?"
(get 'foo 'zoo) ⇒ nil
```

Compatibility note: In MACLISP, the first argument to `get` could be a list, in which case the `cdr` of the list was treated as a so-called "disembodied property list." The first argument to `get` could also be any other object, in which case `get` would always return `nil`. In COMMON LISP, it is an error to give anything but a symbol as the first argument to `get`.

What COMMON LISP calls `get`, INTERLISP calls `getprop`.

What MACLISP and INTERLISP call `putprop` is accomplished in COMMON LISP by using `get` with `self`.

`self` may be used with `get` to create a new property-value pair, possibly replacing an old pair with the same property name. For example:

```
(get 'clyde 'species) ⇒ nil
(self (get 'clyde 'species) 'elephant) ⇒ elephant
and now (get 'clyde 'species) ⇒ elephant
```

The *default* argument may be specified to `get` in this context; it is ignored by `self`, but may be useful in such macros as `push` that are related to `self`:

```
(push item (get sym 'token-stack '(initial-item)))
```

means approximately the same as

```
(self (get sym 'token-stack '(initial-item))
      (cons item (get sym 'token-stack '(initial-item))))
```

which in turn would be treated as simply

```
(self (get sym 'token-stack)
      (cons item (get sym 'token-stack '(initial-item))))
```

`remprop` *symbol indicator*

[Function]

This removes from *symbol* the property with an indicator *eq* to *indicator*. The property indicator and the corresponding value are removed by destructively splicing the property list. It returns `nil` if no such property was found, or non-`nil` if a property was found.

```
(remprop x y) ≡ (remf (symbol-plist x) y)
```

For example, if the property list of `foo` is initially

```
(color blue height 6.3 near-to bar)
```

then the call `(remprop 'foo 'height)`

returns a non-`nil` value after altering `foo`'s property list to be

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

`symbol-plist` *symbol*

[Function]

This returns the list that contains the property pairs of *symbol*; the contents of the property-list cell are extracted and returned.

Note that using `get` on the result of `symbol-plist` does *not* work. One must give the symbol itself to `get` or else use the function `getf`.

`self` may be used with `symbol-plist` to destructively replace the entire property list of a symbol. This is a relatively dangerous operation, as it may destroy important information that the implementation may happen to store in property lists. Also, care must be taken that the new property list is in fact a list of even length.

Compatibility note: In `MACLISP`, this function is called `plist`; in `INTERLISP`, it is called `getproplist`.

`getf` *place indicator* &optional *default* [Function]

`getf` searches the property list stored in *place* for an indicator `eq` to *indicator*. If one is found, then the corresponding value is returned; otherwise *default* is returned. If *default* is not specified, then `nil` is used for *default*. Note that there is no way to distinguish an absent property from one whose value is *default*. Often *place* is computed from a generalized variable acceptable to `self`.

`self` may be used with `getf`, in which case the *place* must indeed be acceptable as a *place* to `self`. The effect is to add a new property-value pair, or update an existing pair, in the property list in the *place*. The *default* argument may be specified to `getf` in this context; it is ignored by `self` but may be useful in such macros as `push` that are related to `self`. See the description of `get` for an example of this.

Compatibility note: The `INTERLISP` function `listget` is similar to `getf`. The `INTERLISP` function `listput` is similar to using `getf` with `self`.

`remf` *place indicator* [Macro]

This removes from the property list stored in *place* the property with an indicator `eq` to *indicator*. The property indicator and the corresponding value are removed by destructively splicing the property list. `remf` returns `nil` if no such property was found, or some non-`nil` value if a property was found. The form *place* may be any generalized variable acceptable to `self`. See `remprop`.

`get-properties` *place indicator-list* [Function]

`get-properties` is like `getf`, except that the second argument is a list of indicators. `get-properties` searches the property list stored in *place* for any of the indicators in *indicator-list* until it finds the first property in the property list whose indicator is one of the elements of *indicator-list*. Normally *place* is computed from a generalized variable acceptable to `self`.

`get-properties` returns three values. If any property was found, then the first two values are the indicator and value for the first property whose indicator was in *indicator-list*, and the third is that tail of the property list whose *car* was the indicator (and whose *cadr* is therefore the

value). If no property was found, all three values are `nil`. Thus the third value serves as a flag indicating success or failure and also allows the search to be restarted after the property found if desired.

The Print Name

Every symbol has an associated string called the *print name*. This string is used as the external representation of the symbol: if the characters in the string are typed in to `read` (with suitable escape conventions for certain characters), it is interpreted as a reference to that symbol (if it is interned); and if the symbol is printed, `print` types out the print name. For more information, see the sections on the *reader* (“What the Read Function Accepts,” Chapter 22) and *printer* (“What the Print Function Produces,” Chapter 22).

symbol-name *sym* [Function]

This returns the print name of the symbol *sym*. For example:

```
(symbol-name 'xyz) ⇒ "XYZ"
```

It is an extremely bad idea to modify a string being used as the print name of a symbol. Such a modification may tremendously confuse the function `read` and the package system.

Creating Symbols

Symbols can be used in two rather different ways. An *interned* symbol is one that is indexed by its print name in a catalogue called a *package*. Every time anyone asks for a symbol with that print name, he gets the same (`eq`) symbol. Every time input is read with the function `read`, and that print name appears, it is read as the same symbol. This property of symbols makes them appropriate to use as names for things and as hooks on which to hang permanent data objects (using the property list, for example).

Interned symbols are normally created automatically; the first time something (such as the function `read`) asks the package system for a symbol with a given print name, that symbol is automatically created. The function used to ask for an interned symbol is `intern`, or one of the functions related to `intern`.

Although interned symbols are the most commonly used, they will not be discussed further here. For more information, see Chapter 11.

An *uninterned* symbol is a symbol used simply as a data object, with no special cataloguing (it belongs to no particular package). An uninterned symbol is printed as `#:` followed by its print name. The following are some functions for creating uninterned symbols.

make-symbol *print-name* [Function]

(make-symbol *print-name*) creates a new uninterned symbol, whose print name is the string *print-name*. The value and function bindings will be unbound and the property list will be empty.

The string actually installed in the symbol's print-name component may be the given string *print-name* or may be a copy of it, at the implementation's discretion. The user should not

10-6

assume that (symbol-name (make-symbol x)) is eq to x, but also should not alter a string once it has been given as an argument to make-symbol.

Implementation note: An implementation might choose, for example, to copy the string to some read-only area, in the expectation that it will never be altered.

copy-symbol *sym* &optional *copy-props* [Function]

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

gensym &optional *x* [Function]

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

The invented print name consists of a prefix (which defaults to G), followed by the decimal representation of a number. The number is increased by one every time gensym is called.

If the argument *x* is present and is an integer, then *x* must be non-negative, and the internal counter is set to *x* for future use; otherwise the internal counter is incremented. If *x* is a string, then that string is made the default prefix for this and future calls to gensym. After handling the argument, gensym creates a symbol as it would with no argument. For example:

```
(gensym) ⇒ G7
(gensym "FOO-") ⇒ FOO-8
(gensym 32) ⇒ FOO-32
(gensym) ⇒ FOO-33
(gensym) "GARBAGE-" ⇒ GARBAGE-34
```

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."

Compatibility note: In earlier versions of LISP, such as MACLISP and INTERLISP, the print name of a gensym was of fixed length, consisting of a single letter and a fixed-length decimal representation with leading zeros if necessary, for example, G0007. This convention was motivated by an implementation consideration, namely that the name should fit into a single machine word, allowing a quick and clever implementation. Such considerations are less relevant in COMMON LISP. The consistent use of mnemonic prefixes can make it easier for the programmer, when

debugging, to determine what code generated a particular symbol. The elimination of the fixed-length decimal representation prevents the same name from being used twice unless the counter is explicitly reset.

If it is desirable for the generated symbols to be interned, and yet guaranteed to be symbols distinct from all others, then the function `gentemp` may be more appropriate to use.

`gentemp` &optional *prefix package* [Function]

`gentemp`, like `gensym`, creates and returns a new symbol. `gentemp` differs from `gensym` in that it interns the symbol (see `intern`) in the *package* (which defaults to the current package; see `*package*`). `gentemp` guarantees the symbol will be a new one not already existing in the package. It does this by using a counter as `gensym` does, but if the generated symbol is not really new, then the process is repeated until a new one is created. There is no provision for resetting the `gentemp` counter. Also, the prefix for `gentemp` is not remembered from one call to the next; if *prefix* is omitted, the default prefix `T` is used.

`symbol-package` *sym* [Function]

Given a symbol *sym*, `symbol-package` returns the contents of the package cell of that symbol. This will be a package object or `nil`.

`keywordp` *object* [Function]

The argument may be any LISP object. The predicate `keywordp` is true if the argument is a symbol and that symbol is a keyword (that is, belongs to the keyword package). Keywords are those symbols that are written with a leading colon. Every keyword is a constant, in the sense that it always evaluates to itself. See `constantp`.

End of Chapter

Chapter 11

Packages

One problem with earlier LISP systems is the use of a single name space for all symbols. In large LISP systems, with modules written by many different programmers, accidental name collisions become a serious problem. COMMON LISP addresses this problem through the *package system*, derived from an earlier package system developed for ZETALISP [21]. In addition to preventing name-space conflicts, the package system makes the modular structure of large LISP systems more explicit.

A *package* is a data structure that establishes a mapping from print names (strings) to symbols. The package thus replaces the “oblist” or “obarray” machinery of earlier LISP systems. At any given time one package is current, and this package is used by the LISP reader in translating strings into symbols. The current package is, by definition, the one that is the value of the global variable `*package*`. It is possible to refer to symbols in packages other than the current one through the use of *package qualifiers* in the printed representation of the symbol. For example, `foo:bar`, when seen by the reader, refers to the symbol whose name is `bar` in the package whose name is `foo`. (Actually, this is true only if `bar` is an external symbol of `foo`, that is, a symbol that is supposed to be visible outside of `foo`. A reference to an internal symbol requires the intentionally clumsier syntax `foo::bar`.)

The string-to-symbol mappings available in a given 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, then it is either external or internal in that package, but not both.

External symbols are part of the package’s public interface to other packages. External symbols are supposed to be chosen with some care and are advertised to 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 appear explicitly in an `export` command for the package.

A symbol may appear in many packages. It will always have the same name wherever it appears, but it may be external in some packages and internal in others. On the other hand, the same name (string) may refer to different symbols in different packages.

Normally, a symbol that appears in one or more packages will be *owned* by one particular package, called the *home package* of the symbol; that package is said to *own* the symbol. Every symbol has a component called the *package cell* that contains a pointer to its home package. A symbol that is owned by some package is said to be *interned*. Some symbols are not owned by any package; such a symbol is said to be *uninterned*, and its package cell contains `nil`.

Packages may be built up in layers. From the point of view of a package's user, the package is a single collection of mappings from strings into internal and external symbols. However, some of these mappings may be established within the package itself, while other mappings are inherited from other packages via the `use-package` construct. (The mechanisms responsible for this inheritance are described below.) In what follows, we will refer to a symbol as being *accessible* in a package if it can be referred to without a package qualifier when that package is current, regardless of whether the mapping occurs within that package or via inheritance. We will refer to a symbol as being *present* in a package if the mapping is in the package itself and is not inherited from somewhere else.

A symbol is said to be *interned in a package* if it is accessible in that package and also is owned (by either that package or some other package). Normally all the symbols accessible in a package will in fact be owned by some package, but the terminology is useful when discussing the pathological case of an accessible but unowned (uninterned) symbol.

As a verb, to *intern* a symbol in a package means to cause the symbol to be interned in the package if it was not already; this process is performed by the function `intern`. If the symbol was previously unowned, then the package it is being interned in becomes its owner (home package); but if the symbol was previously owned by another package, that other package continues to own the symbol.

To *unintern* a symbol from the package means to cause it to be not present and, additionally, to make the symbol uninterned if the package was the symbol's home package (owner). This process is performed by the function `unintern`.

Consistency Rules

Package-related bugs can be very subtle and confusing: things are not what they appear to be. The COMMON LISP 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 may seem over-protective, 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 the user or his code:

- *Read-read consistency*: Reading the same print name always results in 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 will be different sequences of characters.

These consistency rules remain true in spite of any amount of implicit interning caused by typing in LISP forms, loading files, etc. This has the important implication that, as long as the current package is not changed, results are reproducible regardless of the order of loading files or the exact history of what symbols were typed in when. The rules can only be violated by explicit action: changing the value of `*package*`, forcing some action by continuing from an error, or calling one of the "dangerous" functions `unintern`, `unexport`, `shadow`, `shadowing-import`, or `unuse-package`.

Package Names

Each package has a name (a string) 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, such as `ed`.

There is a single name space for packages. The function `find-package` translates a package name or nickname into the associated package. The function `package-name` returns the name of a package. The function `package-nicknames` returns a list of all nicknames for a package. The function `rename-package` removes a package's current name and nicknames and replaces them with new ones specified by the user. Package renaming is occasionally useful when, for development purposes, it is desirable to load two versions of a package into the same LISP. One can load the first version, rename it, and then load the other version, without getting a lot of name conflicts.

When the LISP reader sees a qualified symbol, it handles the package-name part in the same way as the symbol part with respect to capitalization. Lowercase characters in the package name are converted to corresponding uppercase characters unless preceded by the escape character `\` or surrounded by `|` characters. The lookup done by the `find-package` function is case-sensitive, like that done for symbols. 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 seven-character symbol that has a colon in its name (as well as two upper case letters and four lowercase letters) and is interned in the current package. Following the convention used in this manual for symbols, we will show ordinary package names using lowercase letters, even though the name string is internally represented with uppercase letters.

Most of the functions that require a package-name argument from the user accept either a symbol or a string. If the user supplies a symbol, its print name will be used; the print name will already have undergone case-conversion by the usual rules. If the user supplies a string, he must be careful to capitalize the string so as to match exactly the string that names the package.

Translating Strings to Symbols

The value of the special variable `*package*` must always be a package object (not a name). Whatever package object is currently the value of `*package*` is referred to as the *current package*.

When the LISP reader has, by parsing, obtained a string of characters thought to name a symbol, that name is looked up in the current package. This lookup may involve looking in other packages whose external symbols are inherited by the current package. If the name is found, the corresponding symbol is returned. If the name is not found (that is, there is no symbol of that name accessible in the current package), a new symbol is created for it and is placed in the current package as an internal symbol. Moreover, the current package becomes the owner (home package) of the symbol, and so the symbol becomes interned in the current package. If the name is later read again while this same package is current, the same symbol will then be found and returned.

Often it is desirable to refer to an external symbol in some package other than the current one. This is done 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` accessible in 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 accessible in `editor`, or if `buffer` is an internal symbol in `editor`, the LISP reader will signal a correctable error to ask the user what he really wants to do.

On rare occasions, a user may need to refer to an *internal* symbol of some package other than the current one. It is illegal to do this with the colon qualifier, since accessing an internal symbol of some other package is usually a mistake. However, this operation is legal if a doubled colon `::` is used as the separator in place of the usual single colon. If `editor::buffer` is seen, 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.

The package named `keyword` contains all keyword symbols used by the LISP system itself and by user-written code. Such symbols must be easily accessible from any package, and name conflicts are not an issue because these symbols are used only as labels and never to carry package-specific values or properties. Because keyword symbols are used so frequently, COMMON LISP provides a special reader syntax for them. Any symbol preceded by a colon but no package name (for example `:foo`) is added to (or looked up in) the `keyword` package as an *external* symbol. The `keyword` package is also treated specially in that whenever a symbol is added to the `keyword` package the symbol is always made external; the symbol is also automatically declared to be a constant (see `defconstant`) and made to have itself as its value. This is why every keyword evaluates to itself. As a matter of style, keywords should always be accessed using the leading-colon convention; the user should never import or inherit keywords into any other package. It is an error to try to apply `use-package` to the `keyword` package.

Each symbol contains a package cell that is used to record the home package of the symbol, or `nil` if the symbol is uninterned. This cell may be accessed by using the function `symbol-package`. When an interned symbol is printed, if it is a symbol in the `keyword` package, then it is printed with a preceding colon; otherwise, if it is accessible (directly or by inheritance) in the current package, it is printed without any qualification; otherwise, it is printed with the name of the home package as the qualifier, using `:` as the separator if the symbol is external and `::` if not.

A symbol whose package slot contains `nil` (that is, has no home package) is printed preceded by `#:`. It is possible, by the use of `import` and `unintern`, to create a symbol that has no recorded home package, but that in fact is accessible in some package. The system does not check for this pathological case, and such symbols will always be printed preceded by `#:`.

In summary, the following four uses of symbol qualifier syntax are defined.

`foo:bar`

When read, looks up `BAR` among the external symbols of the package named `FOO`. Printed when symbol `bar` is external in its home package `foo` and is not accessible in the current package.

`foo::bar`

When read, interns `BAR` as if `FOO` were the current package. Printed when symbol `bar` is internal in its home package `foo` and is not accessible in the current package.

:bar

When read, interns BAR as an external symbol in the `keyword` package, and makes it evaluate to itself. Printed when the home package of symbol `bar` is `keyword`.

#:bar

When read, creates a new uninterned symbol named BAR. Printed when the symbol `bar` is uninterned (has no home package), even in the pathological case that `bar` is uninterned but nevertheless somehow accessible in the current package.

All other uses of colons within names of symbols are not defined by COMMON LISP but are reserved for implementation-dependent use; this includes names that end in a colon, contain two or more colons, or consist of just a colon.

Exporting and Importing Symbols

Symbols from one package may be accessible in another package in two ways.

First, any individual symbol may be added to a package by use of the function `import`. The form `(import 'editor:buffer)` takes the external symbol named `buffer` in the `editor` package (this symbol was located when the form was read by the LISP reader) and adds it to the current package as an internal symbol. The symbol is then present in the current package. The imported symbol is not automatically exported from the current package, but if it is already present and external, then the fact that it is external is not changed. After the call to `import` it is possible to refer to `buffer` in the importing package without any qualifier. The status of `buffer` in the package named `editor` is unchanged, and `editor` remains the home package for this symbol. Once imported, a symbol is *present* in the importing package and can be removed only by calling `unintern`.

If the symbol is already present in the importing package, `import` has no effect. If a distinct symbol with the name `buffer` is accessible in the importing package (directly or by inheritance), then a correctable error is signalled, as described below in section "Name Conflicts," because `import` avoids letting one symbol shadow another.

A symbol is said to be *shadowed* by another symbol in some package if the first symbol would be accessible by inheritance if not for the presence of the second symbol. If the user really wants to import a symbol without the possibility of getting an error because of shadowing, he should use the function `shadowing-import`. This inserts the symbol into the specified package as an internal symbol, regardless of whether another symbol of the same name will be shadowed by this action. If a different symbol of the same name is already present in the package, that symbol will first be uninterned from the package (see `unintern`). The new symbol is added to the package's `shadowing-symbols` list. `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.

The second mechanism is provided by the function `use-package`. This causes a package to inherit all of the external symbols of some other package. These symbols become accessible as *internal* symbols of the using package. That is, they can be referred to without a qualifier while this package is current, but they are not passed along to any other package that uses this package.

Note that `use-package`, unlike `import`, does not cause any new symbols to be *present* in the current package but only makes them *accessible* by inheritance. `use-package` checks carefully for name conflicts between the newly imported symbols and those already accessible in the importing package. This is described in detail below in “Name Conflicts.”

Typically a user, working by default in the user package, will load a number of packages into his LISP to provide an augmented working environment; then he will call `use-package` on each of these packages so that he can easily access their external symbols. `unuse-package` undoes the effects of a previous `use-package`. The external symbols of the used package are no longer inherited. However, any symbols that have been imported into the using package continue to be present in that package.

There is no way to inherit the *internal* symbols of another package; to refer to an internal symbol, the user must either make that symbol's home package current, use a qualifier, or import that symbol into the current package.

When `intern` or some other function wants to look up a symbol in a given package, it first looks for the symbol among the external and internal symbols of the package itself; then it looks through the external symbols of the used packages in some unspecified order. The order does not matter; according to the rules for handling name conflicts (see below), if conflicting symbols appear in two or more packages inherited by package *X*, a symbol of this name must also appear in *X* itself as a shadowing symbol. Of course, implementations are free to choose other, more efficient ways of implementing this search, as long as the user-visible behavior is equivalent to what is described here.

The function `export` takes a symbol that is accessible in some specified package (directly or by inheritance) and makes it an external symbol of that package. If the symbol is already accessible as an external symbol in the package, `export` has no effect. If the symbol is directly present in the package as an internal symbol, it is simply changed to external status. If it is accessible as an internal symbol via `use-package`, the symbol is first imported into the package, then exported. (The symbol is then present in the specified package whether or not the package continues to use the package through which the symbol was originally inherited.) If the symbol is not accessible at all in the specified package, a correctable error is signalled that, upon continuing, asks the user whether the symbol should be imported.

The function `unexport` is provided mainly as a way to undo erroneous calls to `export`. It works only on symbols directly present in the current package, switching them back to internal status. If `unexport` is given a symbol already accessible as an internal symbol in the current package, it does nothing; if it is given a symbol not accessible in the package at all, it signals an error.

Name Conflicts

A fundamental invariant of the package system is that within one package any particular name can refer to at most one symbol. A *name conflict* is said to occur when there is more than one candidate symbol 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, and subsequently additional programs or data are read, then the two programs will 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 mapping from names to symbols different from what was expected by the user, causing programs to execute incorrectly. Therefore, any time a name conflict is about to occur, an error is signalled. The user may continue from the error and tell the package system how to resolve the conflict.

It may be that the same symbol is accessible to a package through more than one path. For example, the symbol might be an external symbol of more than one used package, or the symbol might be directly present in a package and also inherited from another package. In such cases there is no name conflict. The same identical symbol cannot conflict with itself. Name conflicts occur only between distinct symbols with the same name.

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 functions `shadow` and `shadowing-import` may be used to declare shadowing symbols.

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. `use-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. `import` adds a single symbol 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 a name conflict error even if the conflict is with a shadowing symbol, the rationale being that the user has given two explicit and inconsistent directives. `export` makes a single symbol 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 sp`) does (`find-symbol (symbol-name s) q`) for each package `q` in (`package-used-by-list p`). Note that in the usual case of an `export` during the initial definition of a package, the result of `package-used-by-list` will be `nil` and the name-conflict checking will take negligible time.

The function `intern`, which is the one used most frequently by the LISP reader for looking up names of symbols, 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.

`shadow` and `shadowing-import` never signal a name-conflict error because the user, by calling these functions, 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 uninterned if it was directly present in the package.

`unuse-package`, `unexport`, and `unintern` (when the symbol being uninterned 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.

Giving a shadowing symbol to `unintern` 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 removing the shadowing symbol x from A will reveal a name conflict between $b:x$ and $c:x$ if those two symbols are distinct. In this case `unintern` will signal an error.

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. When multiple changes are to be made, however, for example when `export` is given a list of symbols, it is permissible for the implementation to process each change separately, so that aborting from a name conflict caused by the second symbol in the list will not `unexport` the first symbol in the list. However, aborting from a name-conflict error caused by `export` of a single symbol will not leave that symbol accessible to some packages and inaccessible to others; with respect to each symbol processed, `export` behaves as if it were an atomic operation.

Continuing from a name-conflict error should offer the user a chance to resolve the name conflict in favor of either of the candidates. The package structure should be altered to reflect the resolution of the name conflict, via `shadowing-import`, `unintern`, or `unexport`.

A name conflict in `use-package` between a symbol directly present in the using package and an external symbol of the used package may be resolved in favor of the first symbol by making it a shadowing symbol, or in favor of the second symbol by `uninterning` the first symbol from the using package. The latter resolution is dangerous if the symbol to be `uninterned` is an external symbol of the using package, since it will cease to be an external symbol.

A name conflict in `use-package` between two external symbols inherited by the using package from other packages may 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 may be resolved in favor of the exported symbol by `uninterning` the other one, or in favor of the already-present symbol by making it a shadowing symbol.

A name conflict in `export` or `unintern` due to a package inheriting two distinct symbols with the same name from two other packages may 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 may 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 may be resolved by `uninterning` 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 the user. The user may then choose to resolve all of them wholesale or to resolve each of them individually, the latter requiring a lot of interaction but permitting different conflicts to be resolved different ways.

Implementations may offer other ways of resolving name conflicts. For instance, if the symbols that conflict are not being used as objects but only as names for functions, it may be possible to “merge” the two symbols by putting the function definition onto both symbols. References to either symbol for purposes of calling a function would be equivalent. A similar merging operation can be done for variable values and for things stored on the property list. In

ZETALISP, for example, one can also *forward* the value, function, and property cells so that future changes to either symbol will propagate to the other one. Some other implementations are able to do this with value cells but not with property lists. Only the user can know whether this way of resolving a name conflict is adequate, because it will work only if the use of two non-`eq` symbols with the same name will not prevent the correct operation of his program. The value of offering symbol-merging as a way of resolving name conflicts is that it can avoid the need to throw away the whole LISP world, correct the package-definition forms that caused the error, and start over from scratch.

Built-in Packages

The following packages, at least, are built into every COMMON LISP system:

`lisp`

The package named `lisp` contains the primitives of the COMMON LISP system. Its external symbols include all of the user-visible functions and global variables that are present in the COMMON LISP system, such as `car`, `cdr`, `*package*`, etc. Almost all other packages will want to use `lisp` so that these symbols will be accessible without qualification.

`user`

The user package is, by default, the current package at the time a COMMON LISP system starts up. This package uses the `lisp` package.

`keyword`

This package contains all of the keywords used by built-in or user-defined LISP functions. Printed symbol representations that start with a colon are interpreted as referring to symbols in this package, which are always external symbols. All symbols in this package are treated as constants that evaluate to themselves, so that the user can type `:foo` instead of `:foo.system`.

This package name is reserved to the implementation. Normally this is used to contain names of implementation-dependent system-interface functions. This package uses `lisp` and has the nickname `sys`.

Package System Functions and Variables

Some of the functions and variables in this section are described in previous sections but are included here for completeness.

It is up to each implementation's compiler to ensure that when a compiled file is loaded, all of the symbols in the file end up in the same packages that they would occupy if the LISP source file were loaded. In most compilers, this will be accomplished by treating certain package operations as though they are surrounded by `(eval-when (compile load eval) . . .)`; see `eval-when`. These operations are `make-package`, `in-package`, `shadow`, `shadowing-import`, `export`, `unexport`, `use-package`, `unuse-package`, and `import`. To guarantee proper compilation in all COMMON LISP implementations, these functions should appear only at top level within a file. As a matter

of style, it is suggested that each file contain only one package, and that all of the package setup forms appear near the start of the file. This is discussed in more detail below in "An Example."

Implementation note: In the past, some LISP compilers have read the entire file into LISP before processing any of the forms. Other compilers have arranged for the loader to do all of its intern operations before evaluating any of the top-level forms. Neither of these techniques will work in a straightforward way in COMMON LISP because of the presence of multiple packages.

For the functions described here, all optional arguments named *package* default to the current value of `*package*`. Where a function takes an argument that is either a symbol or a list of symbols, an argument of `nil` is treated as an empty list of symbols. Any argument described as a package name may be either a string or a symbol. If a symbol is supplied, its print name will be used as the package name; if a string is supplied, the user must take care to specify the same capitalization used in the package name, normally all capitals.

`*package*` [Variable]

The value of this variable must be a package; this package is said to be the current package. The initial value of `*package*` is the user package.

The function `load` rebinds `*package*` to its current value. If some form in the file changes the value of `*package*` during the loading, the old value will be restored when the loading is completed.

`make-package package-name &key :nicknames :use` [Function]

This creates and returns a new package with the specified package name. As described above, this argument may be either a string or a symbol. The `:nicknames` argument must be a list of strings to be used as alternative names for the package. Once again, the user may supply symbols in place of the strings, in which case the print names of the symbols are used. These names and nicknames must not conflict with any existing package names; if they do, a correctable error is signalled.

The `:use` argument is a list of packages or the names (strings or symbols) of packages whose external symbols are to be inherited by the new package. These packages must already exist. If not supplied, `:use` defaults to a list of one package, the `lisp` package.

`in-package package-name &key :nicknames :use` [Function]

The `in-package` function is intended to be placed at the start of a file containing a subsystem that is to be loaded into some package other than `user`. If there is not already a package named *package-name*, this function is similar to `make-package`, except that after the new package is created, `*package*` is set to it. This binding will remain in force until changed by the user (perhaps with another `in-package` call) or until the `*package*` variable reverts to its old value at the completion of a load operation.

If there is an existing package whose name is *package-name*, the assumption is that the user is reloading a file after making some changes. The existing package is augmented to reflect any new nicknames or new packages in the `:use` list (with the usual error checking), and `*package*` is then set to this package.

`find-package` *name* [Function]

The *name* must be a string that is the name or nickname for a package. This argument may also be a symbol, in which case the symbol's print name is used. The package with that name or nickname is returned; if no such package exists, `find-package` returns `nil`. The matching of names observes case (as in `string=`).

`package-name` *package* [Function]

The argument must be a package. This function returns the string that names that package.

`package-nicknames` *package* [Function]

The argument must be a package. This function returns the list of nickname strings for that package, not including the primary name.

`rename-package` *package new-name* &optional *new-nicknames* [Function]

The old name and all of the old nicknames of *package* are eliminated and are replaced by *new-name* and *new-nicknames*. The *new-name* argument is a string or symbol; the *new-nicknames* argument, which defaults to `nil`, is a list of strings or symbols.

`package-use-list` *package* [Function]

A list of other packages used by the argument package is returned.

`package-used-by-list` *package* [Function]

A list of other packages that use the argument package is returned.

`package-shadowing-symbols` *package* [Function]

A list is returned 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.

`list-all-packages` [Function]

This function returns a list of all packages that currently exist in the LISP system.

`intern string &optional package`

[Function]

The *package*, which defaults to the current package, is searched for a symbol with the name specified by the *string* argument. This search will include inherited symbols, as described above in “Exporting and Importing Symbols.” If a symbol with the specified name is found, it is returned. If no such symbol is found, one is created and is installed in the specified package as an internal symbol (as an external symbol if the package is the keyword package); the specified package becomes the home package of the created symbol.

Two values are returned. The first is the symbol that was found or created. The second value is `nil` if no pre-existing symbol was found, and takes on one of three values if a symbol was found:

<code>:internal</code>	The symbol was directly present in the package as an internal symbol.
<code>:external</code>	The symbol was directly present as an external symbol.
<code>:inherited</code>	The symbol was inherited via <code>use-package</code> (which implies that the symbol is internal).

Compatibility note: Conceptually, `intern` translates a string to a symbol. In `MACLISP` and several other dialects of `LISP`, `intern` can take either a string or a symbol as its argument; in the latter case, the symbol’s print name is extracted and used as the string. However, this leads to some confusing issues about what to do if `intern` finds a symbol that is not `eq` to the argument symbol. To avoid such confusion, `COMMON LISP` requires the argument to be a string.

`find-symbol string &optional package`

[Function]

This is identical to `intern`, but it never creates a new symbol. If a symbol with the specified name is found in the specified package, directly or by inheritance, the symbol found is returned as the first value and the second value is as specified for `intern`. If the symbol is not accessible in the specified package, both values are `nil`.

`unintern symbol &optional package`

[Function]

If the specified symbol is present in the specified *package*, it is removed from that package and also from the package’s shadowing-symbols list if it is present there. Moreover, if the *package* is the home package for the symbol, the symbol is made to have no home package. Note that in some circumstances the symbol may continue to be accessible in the specified package by inheritance. `unintern` returns `t` if it actually removed a symbol, and `nil` otherwise.

`unintern` 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.

Compatibility note: The equivalent of this in `MACLISP` is `remob`.

`export symbols &optional package` [Function]

The *symbols* argument should be a list of symbols, or possibly a single symbol. These symbols become accessible as external symbols in *package*. See “Exporting and Importing Symbols” above for details. `export` returns `t`.

By convention, a call to `export` listing all exported symbols is placed near the start of a file to advertise which of the symbols mentioned in the file are intended to be used by other programs.

`unexport symbols &optional package` [Function]

The argument should be a list of symbols, or possibly a single symbol. These symbols become internal symbols in *package*. It is an error to `unexport` a symbol from the keyword package. See “Exporting and Importing Symbols” above for details. `unexport` returns `t`.

`import symbols &optional package` [Function]

The argument should be a list of symbols, or possibly a single symbol. These symbols become internal symbols in *package* and can therefore be referred to without having to use qualified-name (colon) syntax. `import` signals a correctable error if any of the imported symbols has the same name as some distinct symbol already accessible in the package. See “Exporting and Importing Symbols” above for details. `import` 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 accessible in the package. In addition to being imported, the symbol is placed on the shadowing-symbols list of *package*. See “Name Conflicts” above for details. `shadowing-import` 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 argument should be a list of symbols, or possibly a single symbol. The print name of each symbol is extracted, and the specified package is searched for a symbol of that name. If such a symbol is present in this package (directly, not by inheritance), then nothing is done. Otherwise, a new symbol is created with this print name, and it is inserted in the specified package as an internal symbol. The symbol is also placed on the shadowing-symbols list of *package*. See “Name Conflicts” above for details. `shadow` 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.

Modules

A *module* is a COMMON LISP subsystem that is loaded from one or more files. A module is normally loaded as a single unit, regardless of how many files are involved. A module may consist of one package or several packages. The file-loading process is necessarily implementation-dependent, but COMMON LISP provides some very simple portable machinery for naming modules, for keeping track of which modules have been loaded, and for loading modules as a unit.

modules

[*Variable*]

The variable **modules** is a list of names of the modules that have been loaded into the LISP system so far. This list is used by the functions `provide` and `require`.

`provide` *module-name*

[*Function*]

`require` *module-name* &optional *pathname*

[*Function*]

Each module has a unique name (a string). The `provide` and `require` functions accept either a string or a symbol as the *module-name* argument. If a symbol is provided, its print name is used as the module name. If the module consists of a single package, it is customary for the package and module names to be the same.

The `provide` function adds a new module name to the list of modules maintained in the variable **modules**, thereby indicating that the module in question has been loaded.

The `require` function tests whether a module is already present (using a case-sensitive comparison); if the module is not present, `require` proceeds to load the appropriate file or set of files. The *pathname* argument, if present, is a single pathname or a list of pathnames whose files are to be loaded in order, left to right. If the *pathname* argument is `nil` or is not provided, the system will attempt to determine, in some system-dependent manner, which files to load. This will typically involve some central registry of module names and the associated file lists.

Implementation note: One way to implement such a registry on many operating systems is simply to use a distinguished "library" directory within the file system, where the name of each file is the same as the module it contains.

Table 11-1: An Initialization File

```
;;; Lisp init file for I. Newton.

;;; Set up the USER package the way I like it.

(require 'calculus)                ;I use CALCULUS a lot. Load it.
(use-package 'calculus)            ;Get easy access to its
                                   ;exported symbols.

(require 'newtonian-mechanics) ;Ditto for NEWTONIAN-MECHANICS.
(use-package 'newtonian-mechanics)

;;; I just want a few things from RELATIVITY,
;;; and other things conflict.
;;; Import only what I need into the USER package.

(require 'relativity)
(import '(relativity:speed-of-light
         relativity:ignore-small-errors))

;;; These are worth loading, but I will use qualified names,
;;; such as PHLOGISTON:MAKE-FIRE-BOTTLE, to get at any symbols
;;; I might need from these packages.

(require 'phlogiston)
(require 'alchemy)

;;; End of Lisp init file for I. Newton.
```

An Example

Most users will want to load and use packages but will never need to build one. Often a user will load a number of packages into the user package whenever he uses COMMON LISP. Typically an implementation might provide some sort of initialization file mechanism to make such setup automatic when the LISP starts up. Table 11-1 shows such an initialization file, one that simply causes other facilities to be loaded.

When each of two files uses some symbols from the other, one must be careful to arrange the contents of the file in the proper order. Typically each file contains a single package that is a complete module. The contents of such a file should include the following items, in order:

1. A call to `provide` that announces the module name.
2. A call to `in-package` that establishes the package.
3. A call to `shadow` that establishes any local symbols that will shadow symbols that would otherwise be inherited from packages that this package will use.

Table 11-2: File alchemy

```

;;; Alchemy functions, written and maintained by Merlin, Inc.

(provide 'alchemy)                ;The module is named ALCHEMY.
(in-package 'alchemy)            ;So is the package.

;;; There is nothing to shadow.

;;; Here is the external interface.

(export '(lead-to-gold gold-to-lead
        antimony-to-zinc elixir-of-life))

;;; This package/module needs a function from
;;; the PHLOGISTON package/module.

(require 'phlogiston)

;;; We don't frequently need most of the external symbols from
;;; PHLOGISTON, so it's not worth doing a USE-PACKAGE on it.
;;; We'll just use qualified names as needed. But we use
;;; one function, MAKE-FIRE-BOTTLE, a lot, so import it.
;;; It's external in PHLOGISTON, and so can be referred to
;;; here using ":" qualified-name syntax.

(import '(phlogiston:make-fire-bottle))

;;; Now for the real contents of this file.

(defun lead-to-gold (x)
  "Takes a quantity of lead and returns gold."
  (when (> (phlogiston:heat-flow x) 3) ;Using a qualified symbol.
    (make-fire-bottle x)) ;Using an imported symbol.
  (gold x))

;;; And so on . . .

```

4. A call to `export` that establishes all of this package's external symbols.
5. Any number of calls to `require` to load other modules that the contents of this file might want to use or refer to. (Because the calls to `require` follow the calls to `in-package`, `shadow`, and `export`, it is possible for the packages that may be loaded to refer to external symbols in this package.)
6. Any number of calls to `use-package`, to make external symbols from other packages accessible in this package.
7. Any number of calls to `import`, to make symbols from other packages present in this package.
8. Finally, the definitions making up the contents of this package/module.

Table 11-3: File phlogiston

```
;;; Phlogiston functions, by Thermofluidics, Ltd.

(provide 'phlogiston)           ;The module is named PHLOGISTON.
(in-package 'phlogiston)       ;So is the package.

;;; There is nothing to shadow.

;;; Here is the external interface.

(export '(heat-flow cold-flow mix-fluids separate-fluids
         burn make-fire-bottle))

;;; This file uses functions from the ALCHEMY package/module.

(require 'alchemy)

;;; We use alchemy functions a lot, so use the package.
;;; This will allow symbols exported from the ALCHEMY package
;;; to be referred to here without the need for qualified names.

(use-package 'alchemy)

;;; No calls to IMPORT are needed here.

;;; The real contents of this package/module.

(defun heat-flow (amount x y)
  "Make some amount of heat flow from x to y."
  (when feeling-weak
    (quaff (elixir-of-life))) ;No qualifier needed.
  (push-heat amount x y))

;;; And so on . . .
```

The following mnemonic sentence may be helpful in remembering the proper order of these calls:

Put in seven extremely random user interface commands.

Each word of the sentence corresponds to one item in the above ordering:

Put	Provide
IN	IN-package
Seven	Shadow
EXtremely	EXport
Random	Require
USER	USE-package
Interface	Import
COMmands	COntents of package/module

The sentence says what it helps you to do.

Now, suppose for the sake of example that the phlogiston and alchemy packages are single-file, single-package modules as described above. The phlogiston package needs to use the alchemy package, and the alchemy package needs to use several external symbols from the phlogiston package. The definitions in the alchemy and phlogiston files (see Tables 11-2 and 11-3) allow a user to specify require statements for either of these modules, or for both of them in either order, and all relevant information will be loaded automatically and in the correct order.

For very large modules whose contents are spread over several files (the `lisp` package is an example), it is recommended that the user create the package and declare all of the shadows and external symbols in a separate file, so that this can be loaded before anything that might use symbols from this package.

End of Chapter

