Common Lisp Reference

Table of Contents

Common Lisp HyperSpec

Credits

```
Principal Technical Editors:

Kent M. Pitman               Harlequin, Inc.               1993-present
                             Symbolics, Inc.               1990-1992
Kathy Chapman                Digital Equipment Corporation 1987-1989

Occasional Guest Editors:

Richard P. Gabriel           Lucid, Inc.
Sandra Loosemore             self

Financial Contributors to the Editing Process:

Digital Equipment Corporation
Harlequin, Ltd. and Harlequin, Inc.
Symbolics, Inc.
Apple, Inc.
Franz, Inc.
Lucid, Inc.
```

Special thanks to Guy L. Steele Jr. and Digital Press for producing *Common Lisp: The Language*, and for relaxing copyright restrictions enough to make it possible for that document's text to provide an early basis of this work.

Edit and Review History:

01-Jan-89	Chapman	Draft of Chapters 1.1 (scope).
01-Jan-89	Pitman	Draft of Chapters 5.1 (conditions).
01-May-89	Chapman	Draft of 1.2--1.6.
01-May-89	Gabriel	Rewrite of Chapters 1.1 and 5.1.
01-Jun-89	Loosemore	Review of Chapter 4.2.
01-Jun-89	Pitman	Review of Glossary
15-Jun-89	Gabriel	Rewrite of Glossary
16-Jun-89	Margolin	Comments on Chapters 2.1--2.4 (types, objects).
23-Jun-89	Gabriel	Rewrite of 4.2.
07-Jul-89	Moon	Review of Chapters 4.1, 4.3
12-Jul-89	Gabriel	Revision of 4.2.
15-Jul-89	Pitman	Review of Glossary
18-Jul-89	Gray	Comments on 5.1
25-Jul-89	Gabriel	Revision of Chapters 1.2--1.6, 2.2
26-Jul-89	Gabriel	Rewrite of 5.1
26-Jul-89	Gabriel	Rewrite of 4.1.
27-Jul-89	Pitman	Revision of 5.1
27-Jul-89	Gabriel	Revision of 5.1
28-Jul-89	Chapman	Draft of 2.2, 3.2, 3.3, 5.4
28-Jul-89	Gabriel	Revision of Glossary.
01-Oct-89	Margolin	Review of Dictionary from Jun-89 draft.
20-Jan-91	Pitman	Draft 8.81 (for X3J13 review). Document X3J13/91-101.
29-Jan-91	Waters	Review of 8.81/Chapter 23 (Printer).
01-Mar-91	Moon	Review of 8.81/Chapter 4 (Evaluation and Compilation).
01-Mar-91	Barrett	Review of 8.81/Chapter 4 (Evaluation and Compilation).
01-Mar-91	Moon	Review of 8.81/Glossary.
13-Mar-90	Wechsler	Review of 8.81/Glossary.
21-Mar-91	Kerns	Review of 8.81/Chapter 1.
26-Apr-91	Margolin	Review of 8.81/Chapters 1--12.
15-May-91	Barrett	Review of 8.81/Chapters 5 (Misc), 11 (Conditions).
04-Jun-91	Laddaga	Review of 9.60/Chapter 20 (Pathnames).
10-Jun-91	Pitman	Draft 9.126 (for X3J13 review). Document X3J13/91-102.
02-Sep-91	Barrett	Review of 9.28/Chapter 4 (Evaluation and Compilation).
02-Sep-91	Barrett	Review of 9.52/Chapter 4 (Evaluation and Compilation).
15-Sep-91	Barrett	Review of 9.126/Chapter 4 (Evaluation and Compilation) and Chapter 7 (Evaluation/Compilation). (some comments not yet merged)
18-Sep-91	Wechsler	Review of 9.126.
21-Sep-91	Barrett	Review of 10.16/Chapter 7 (Evaluation/Compilation). (some comments not yet merged)
28-Sep-91	Barrett	Review of 10.95/Chapter 25 (Printer). (some comments not yet merged)
13-Oct-91	Barrett	Review (and help editing) of 10.104/Chapter 4 (Evaluation and Compilation)
15-Oct-91	Waters	Review of 10.95/Chapter 25 (Printer).
24-Oct-91	Pitman	Draft 10.156 (for X3J13 review). Document X3J13/91-103.
04-Nov-91	Moon	Review of 10.156/Chapter 5 (Data and Control Flow) and Chapter 26 (Glossary).
11-Nov-91	Loosemore	Review of 10.156/Chapter 2 (Syntax), Chapter 3 (Evaluation and Compilation), Chapter 5 (Data and Control Flow), and Chapter 8 (Structures).
02-Dec-91	Barrett	Review of 10.156/Chapter 4 (Types and Classes), and Chapter 10 (Symbols).
02-Dec-91	Barrett	Review of 10.156/Chapter 3 (Evaluation and Compilation), Chapter 6 (Iteration), Chapter 9 (Conditions), and Chapter 14 (Conses). (some comments not yet merged)
09-Dec-91	Gabriel	Review of 10.156/Chapter 1 (Introduction), Chapter 2 (Syntax), and Chapter 3 (Evaluation and Compilation).
09-Dec-91	Ida	Light review of 10.156/Chapters 1-5.
09-Dec-91	Moon	Review of 10.156/Chapter 3 (Evaluation and Compilation). (some comments not yet merged)
10-Dec-91	Loosemore	Review of 10.156/Chapter 10 (Symbols), Chapter 20 (Files), and Chapter 13 (Characters).

10-Dec-91	Loosemore	Review of 10.156/Chapter 14 (Conses). (some comments not yet merged)
10-Dec-91	Laubsch	Review of 10.156/Chapters 1 (Introduction), Chapter 2 (Syntax), Chapter 3 (Evaluation and Compilation), Chapter 4 (Types and Classes), Chapter 5 (Data and Control Flow), Chapter 7 (Objects), Chapter 11 (Packages), Chapter 19 (Filenames), and Chapter 21 (Streams).
18-Dec-91	Margolin	Review of 10.156/Chapter 18 (Hash Tables).
04-Jan-92	White	Review of 10.156/Chapter 6 (Iteration), Chapter 11 (Packages), Chapter 18 (Hash Tables), and Chapter 23 (Reader).
04-Jan-92	White	Review of 10.156/Chapter 26 (Glossary). (some comments not yet merged)
04-Jan-92	Barrett	Review of 10.156/Chapter 18 (Hash Tables) and Chapter 16 (Strings).
04-Jan-92	Barrett	Review of 10.156/Chapter 15 (Arrays) and Chapter 21 (Streams). (some comments not yet merged)
06-Jan-92	Loosemore	Review of 10.156/Chapter 16 (Strings), Chapter 17 (Sequences), and Chapter 25 (Environment).
06-Jan-92	Loosemore	Review of 10.156/Chapter 21 (Streams) and Chapter 23 (Reader). (some comments not yet merged)
06-Jan-92	Margolin	Review of 10.156/Chapter 2 (Syntax).
07-Jan-92	Margolin	Review of 10.156/Chapter 4 (Types and Classes).
03-Feb-92	Aspinall	Review of 10.156/Chapter 12 (Numbers).
16-Feb-92	Pitman	Draft 11.82 (for X3J13 letter ballot). Document X3J13/92-101.
16-Mar-92	Loosemore	Review of 11.82/Chapter 1, 3-5, 7-12, 18, and 22-26.
16-Feb-92	Pitman	Draft 12.24 (for X3 consideration). Document X3J13/92-102.
09-Sep-92	Samson	Public Review Comments (#1). Documents X3J13/92-1001 to 92-1003.
22-Oct-92	Rose, Yen	Public Review Comments (#2). Documents X3J13/92-1101 to 92-1103.
23-Oct-92	Staley	Public Review Comments (#3). Documents X3J13/92-1201 to 92-1204.
09-Nov-92	Barrett	Public Review Comments (#4). Documents X3J13/92-3101 to 92-3110.
11-Nov-92	Moon	Public Review Comments (#5). Documents X3J13/92-3201 to 92-3248.
17-Nov-92	Loosemore	Public Review Comments (#6). Documents X3J13/92-1301 to 92-1335.
23-Nov-92	Margolin	Public Review Comments (#7). Documents X3J13/92-1401 to 92-1419.
23-Nov-92	Withington	Public Review Comments (#8a). Documents X3J13/92-1501 to 92-1512.
23-Nov-92	Feinberg	Public Review Comments (#8b). Documents X3J13/92-1601 to 92-1603.
23-Nov-92	Wechsler	Public Review Comments (#8c). Documents X3J13/92-1701 to 92-1703.
23-Nov-92	Moore	Public Review Comments (#9). Documents X3J13/92-1801 to 92-1802.
23-Nov-92	Flanagan	Public Review Comments (#10). Documents X3J13/92-1901 to 92-1910.
23-Nov-92	Dalton	Public Review Comments (#11). Documents X3J13/92-2001 to 92-2012.
23-Nov-92	Gallagher	Public Review Comments (#12). Documents X3J13/92-2101 to 92-2103.
23-Nov-92	Norvig	Public Review Comments (#13). Documents X3J13/92-2201 to 92-2208.
24-Nov-92	Robertson	Public Review Comments (#14). Document X3J13/92-2301.
23-Nov-92	Kawabe	Public Review Comments (#15). Documents X3J13/92-2401 to 92-2403.
23-Nov-92	Barrett	Public Review Comments (#16). Documents X3J13/92-2511 to X3J13/92-2531.
23-Nov-92	Wertheimer	Public Review Comments (#17). Document X3J13/92-2601.
24-Nov-92	Pitman	Public Review Comments (#18). Documents X3J13/92-2701 to 92-2742.
24-Nov-92	Mato Mira	Public Review Comments (#19). Documents X3J13/92-2801 to 92-2805.
24-Nov-92	Philpot	Public Review Comments (#20). Document X3J13/92-2901.
23-Nov-92	Cerys	Public Review Comments (#21). Document X3J13/92-3001.
30-Aug-93	Pitman	Draft 13.65 (for X3J13 consideration). Document X3J13/93-101.
04-Oct-93	X3J13	Minor fixes to Draft 13.65 before sending to X3.
05-Oct-93	Pitman	Draft 14.10 (for X3 consideration). Document X3J13/93-102.
08-Nov-93	Dalton	"reply to reply to pr comments". Document X3J13/94-311.
04-Apr-94	Boyer, Kaufmann, Moore	Public Review Comments (#1). Document X3J13/94-305.
05-Apr-94	Pitman	Public Review Comments (#2). Document X3J13/94-306.
14-Mar-94	Schulenburg	Public Review Comments (#3). Document X3J13/94-307.
04-Apr-94	Shepard	Late commentary. Document X3J13/94-309.
05-May-94	X3J13	Editorial-only changes to Draft 14.10 in response to comments.
10-May-94	Pitman	Draft 15.17 (for X3 consideration). Document X3J13/94-101.
12-Aug-94	X3J13	Letter ballot to make specific corrections to Credits. Drafts 15.17 and 15.17R are identical except for: Changes to document date and version number. Disclaimer added to back of cover page.

```
                        Changes to this Edit and Review History, page Credits iv.
                        Changes to names and headings, pages Credits v-vii.
12-Aug-94  Pitman       Draft 15.17R (for X3 consideration). Document ANSI X3.266-1994.
01-Feb-94  Pitman       Pre-publication changes per ANSI. This is ANSI X3.226-1994!
```

The following lists of information are almost certainly incomplete, but it was felt that it was better to risk publishing incomplete information than to fail to acknowledge important contributions by the many people and organizations who have contributed to this effort.

Mention here of any individual or organization does not imply endorsement of this document by that individual or organization.

Ad Hoc Group Chairs:

Characters	Linden, Thom
Charter	Ennis, Susan P.
Compiler Specification	Haflich, Steven M.
	Loosemore, Sandra
Editorial	Chapman, Kathy
	van Roggen, Walter
Error and Condition System	Pitman, Kent M.
Graphics &Windows	Douglas Rand
	Schoen, Eric
Iteration Facility	White, JonL
Language Cleanup	Masinter, Larry
	Fahlman, Scott
Lisp1/Lisp2	Gabriel, Richard P.
Macros	Haflich, Steven M.
	Pitman, Kent M.
	Wegman, Mark
Object System	Bobrow, Daniel G.
Presentation of Standard	Brown, Gary L.
Pretty Printer	Waters, Richard C.
Public Review	Ida, Masayuki
Types &Declarations	Scherlis, William L.
Validation	Berman, Richard

Major Administrative Contributions:

Barrett, Kim	Mathis, Robert
Brown, Gary L.	Pitman, Kent M.
Eiron, Hanoch	Steele, Guy L., Jr.
Gabriel, Richard P.	Tyson, Mabry
Haflich, Steven M.	Van Deusen, Mary
Ida, Masayuki	White, JonL
Loeffler, David D.	Whittemore, Susan
Loosemore, Sandra	Woodyatt, Anne
Masinter, Larry	Zubkoff, Jan L.

Major Technical Contributions:

Barrett, Kim A.	Keene, Sonya	Moon, David A.
Bobrow, Daniel G.	Kempf, James	Perdue, Crispin
Daniels, Andy	Kerns, Robert W.	Pitman, Kent M.
DeMichiel, Linda G.	Kiczales, Gregor	Steele, Guy L., Jr.
Dussud, Patrick H.	Loosemore, Sandra	Waters, Richard C.
Fahlman, Scott	Margolin, Barry	Weinreb, Daniel
Gabriel, Richard P.	Masinter, Larry	White, JonL
Ida, Masayuki	Mlynarik, Richard	

Participating Companies and Organizations:

AI Architects, Inc.	Lucid, Inc.
Amoco Production Co.	MCC

Aoyama Gakuin University
Apple Computer
Boeing Advanced Technology Center
Carnegie-Mellon University
Chestnut Software
Computer Sciences
Computer & Business Equipment Manufacturing Association (X3 Secretariat)
CONTEL
CSC
DARPA
Digital Equipment Corporation
Encore
Evans & Sutherland
Franz, Inc.
Gigamos
GMD
Gold Hill
Grumman Data Systems Corporation
Harlequin, Ltd.
Hewlett-Packard
Honeywell
IBM
Ibuki
Integrated Inference Machines
International LISP Associates
Johnson Controls, Inc.
LMI
MIT
MITRE Corporation
MSC
NASA Ames Research Center
National Bureau of Standards
Nihon Symbolics
ParcPlace Systems, Inc.
Prime Computer
Siemens
Southern Illinois University
Sperry
SRI International
Sun Microsystems
Symbolics
Tektronix
Texas Instruments
The Aerospace Corporation
Thinking Machines Corporation
Unisys
University of Bath
University of Edinburgh
University of Maryland
University of Utah
US Army
USC/ISI
Xerox

Individual Participants:

Adler, Annette
Allen, Stanley
Antonisse, Jim
Arbaugh, Bill
Balzer, Bob
Barrett, Kim
Bartley, David H.
Beckerle, Michael
Beiser, Paul
Benson, Eric
Berman, Richard
Bobrow, Daniel G.
Boelk, Mary P.
Brittain, Skona
Brown, Gary L.
Chailloux, Jerome
Chapman, Kathy
Clinger, Will
Coffee, Peter C.
Cugini, John
Curtis, Pavel
Dabrowski, Christopher
Daessler, Klaus
Dalton, Jeff
Daniels, Andy
DeMichiel, Linda G.
Doi, Takumi
Drescher, Gary
Duggan, Jerry
Dussud, Patrick H.
Ennis, Susan P.
Fahlman, Scott
Frayman, Felix
Gabriel, Richard P.
Giansiracusa, Bob
Haflich, Steven M.
Harris, Richard M.
Hendler, Jim
Hewitt, Carl
Hornig, Charles
Ida, Masayuki
Kachurik, Catherine A.
Kahn, Ken
Keene, Sonya
Keller, Shaun
Kempf, James
Kerns, Robert W.
Kiczales, Gregor
Kolb, Dieter
Koschmann, Timothy
Kosinski, Paul
Larson, Aaron
Latto, Andy
Laubsch, Joachim
Layer, Kevin
Linden, Thom
Loeffler, David D.
Loosemore, Sandra
Magataca, Mituhiro
Margolin, Barry
Masinter, Larry
Mathis, Robert
Matthews, David C.
McCarthy, John
Mikelsons, Martin
Mlynarik, Richard
Moon, David A.
Moore, Timothy
Nicoud, Stephen
Nilsson, Jarl
Peck, Jeff
Pellegrino, Bob
Perdue, Crispin
Philipp, Christopher
Pierson, Dan
Pitman, Kent M.
Posner, Dave
Raghavan, B.
Rand, Douglas
Rininger, Jeff
Rosenking, Jeffrey P.
Scherlis, William L.
Shiota, Eiji
Sizer, Andy
Slater, David
Sodan, Angela
Soley, Richard M.
Squires, Stephen L.
St. Clair, Bill
Stanhope, Philip
Steele, Guy L., Jr.
Tucker, Paul
Turba, Thomas
Unietis, Dave
Van Deusen, Mary
van Roggen, Walter
Waldrum, Ellen
Waters, Richard C.
Wechsler, Allan
Wegman, Mark
Weinreb, Daniel
Weyhrauch, Richard
White, JonL
Wieland, Alexis
Withington, P. Tucker

Goldstein, Brad	O'Dell, Jim	Wright, Whitman
Gray, David	Ohlander, Ron	York, Bill
Greenblatt, Richard	Padget, Julian	Zacharias, Gail
Hadden, George D.	Palter, Gary	Zubkoff, Jan L.

1. Introduction

1.1 Scope, Purpose, and History

1.1.2 History

Lisp is a family of languages with a long history. Early key ideas in Lisp were developed by John McCarthy during the 1956 Dartmouth Summer Research Project on Artificial Intelligence. McCarthy's motivation was to develop an algebraic list processing language for artificial intelligence work. Implementation efforts for early dialects of Lisp were undertaken on the IBM 704, the IBM 7090, the Digital Equipment Corporation (DEC) PDP-1, the DEC PDP-6, and the PDP-10. The primary dialect of Lisp between 1960 and 1965 was Lisp 1.5. By the early 1970's there were two predominant dialects of Lisp, both arising from these early efforts: MacLisp and Interlisp. For further information about very early Lisp dialects, see *The Anatomy of Lisp* or *Lisp 1.5 Programmer's Manual*.

MacLisp improved on the Lisp 1.5 notion of special variables and error handling. MacLisp also introduced the concept of functions that could take a variable number of arguments, macros, arrays, non-local dynamic exits, fast arithmetic, the first good Lisp compiler, and an emphasis on execution speed. By the end of the 1970's, MacLisp was in use at over 50 sites. For further information about Maclisp, see *Maclisp Reference Manual, Revision 0* or *The Revised Maclisp Manual*.

Interlisp introduced many ideas into Lisp programming environments and methodology. One of the Interlisp ideas that influenced Common Lisp was an iteration construct implemented by Warren Teitelman that inspired the **loop** macro used both on the Lisp Machines and in MacLisp, and now in Common Lisp. For further information about Interlisp, see *Interlisp Reference Manual*.

Although the first implementations of Lisp were on the IBM 704 and the IBM 7090, later work focussed on the DEC PDP-6 and, later, PDP-10 computers, the latter being the mainstay of Lisp and artificial intelligence work at such places as Massachusetts Institute of Technology (MIT), Stanford University, and Carnegie Mellon University (CMU) from the mid-1960's through much of the 1970's. The PDP-10 computer and its predecessor the PDP-6 computer were, by design, especially well-suited to Lisp because they had 36-bit words and 18-bit addresses. This architecture allowed a *cons* cell to be stored in one word; single instructions could extract the *car* and *cdr* parts. The PDP-6 and PDP-10 had fast, powerful stack instructions that enabled fast function calling. But the limitations of the PDP-10 were evident by 1973: it supported a small number of researchers using Lisp, and the small, 18-bit address space (2^18 = 262,144 words) limited the size of a single program. One response to the address space problem was the Lisp Machine, a special-purpose computer designed to run Lisp programs. The other response was to use general-purpose computers with address spaces larger than 18 bits, such as the DEC VAX and the S-1 Mark IIA. For further information about S-1 Common Lisp, see "S-1 Common Lisp Implementation."

The Lisp machine concept was developed in the late 1960's. In the early 1970's, Peter Deutsch, working with Daniel Bobrow, implemented a Lisp on the Alto, a single-user minicomputer, using microcode to interpret a byte-code implementation language. Shortly thereafter, Richard Greenblatt began work on a different hardware and instruction set design at MIT. Although the Alto was not a total success as a Lisp machine, a dialect of Interlisp known as Interlisp-D became available on the D-series machines manufactured by Xerox---the Dorado, Dandelion, Dandetiger, and Dove (or Daybreak). An upward-compatible extension of MacLisp called Lisp Machine Lisp became available on the early MIT Lisp Machines. Commercial Lisp machines from Xerox, Lisp Machines (LMI), and Symbolics were on the market by 1981. For further information about Lisp Machine Lisp, see *Lisp Machine Manual*.

During the late 1970's, Lisp Machine Lisp began to expand towards a much fuller language. Sophisticated lambda lists, `setf`, multiple values, and structures like those in Common Lisp are the results of early experimentation with programming styles by the Lisp Machine group. Jonl White and others migrated these features to MacLisp. Around 1980, Scott Fahlman and others at CMU began work on a Lisp to run on the Scientific Personal Integrated Computing Environment (SPICE) workstation. One of the goals of the project was to design a simpler dialect than Lisp Machine Lisp.

The Macsyma group at MIT began a project during the late 1970's called the New Implementation of Lisp (NIL) for the VAX, which was headed by White. One of the stated goals of the NIL project was to fix many of the historic, but annoying, problems with Lisp while retaining significant compatibility with MacLisp. At about the same time, a research group at Stanford University and Lawrence Livermore National Laboratory headed by Richard P. Gabriel began the design of a Lisp to run on the S-1 Mark IIA supercomputer. S-1 Lisp, never completely functional, was the test bed for adapting advanced compiler techniques to Lisp implementation. Eventually the S-1 and NIL groups collaborated. For further information about the NIL project, see "NIL---A Perspective."

The first effort towards Lisp standardization was made in 1969, when Anthony Hearn and Martin Griss at the University of Utah defined Standard Lisp---a subset of Lisp 1.5 and other dialects---to transport REDUCE, a symbolic algebra system. During the 1970's, the Utah group implemented first a retargetable optimizing compiler for Standard Lisp, and then an extended implementation known as Portable Standard Lisp (PSL). By the mid 1980's, PSL ran on about a dozen kinds of computers. For further information about Standard Lisp, see "Standard LISP Report."

PSL and Franz Lisp---a MacLisp-like dialect for Unix machines---were the first examples of widely available Lisp dialects on multiple hardware platforms.

One of the most important developments in Lisp occurred during the second half of the 1970's: *Scheme*. Scheme, designed by Gerald J. Sussman and Guy L. Steele Jr., is a simple dialect of Lisp whose design brought to Lisp some of the ideas from programming language semantics developed in the 1960's. Sussman was one of the prime innovators behind many other advances in Lisp technology from the late 1960's through the 1970's. The major contributions of Scheme were lexical scoping, lexical closures, first-class continuations, and simplified syntax (no separation of value cells and function cells). Some of these contributions made a large impact on the design of Common Lisp. For further information about Scheme, see *IEEE Standard for the Scheme Programming Language* or "*Revised^3 Report on the Algorithmic Language Scheme*."

In the late 1970's object-oriented programming concepts started to make a strong impact on Lisp. At MIT, certain ideas from Smalltalk made their way into several widely used programming systems. Flavors, an object-oriented programming system with multiple inheritance, was developed at MIT for the Lisp machine community by Howard Cannon and others. At Xerox, the experience with Smalltalk and Knowledge Representation Language (KRL) led to the development of Lisp Object Oriented Programming System (LOOPS) and later Common LOOPS. For further information on Smalltalk, see *Smalltalk-80: The Language and its Implementation*. For further information on Flavors, see *Flavors: A Non-Hierarchical Approach to Object-Oriented Programming*.

These systems influenced the design of the Common Lisp Object System (CLOS). CLOS was developed specifically for this standardization effort, and was separately written up in "Common Lisp Object System Specification." However, minor details of its design have changed slightly since that publication, and that paper should not be taken as an authoritative reference to the semantics of the object system as described in this document.

In 1980 Symbolics and LMI were developing Lisp Machine Lisp; stock-hardware implementation groups were developing NIL, Franz Lisp, and PSL; Xerox was developing Interlisp; and the SPICE project at CMU was developing a MacLisp-like dialect of Lisp called SpiceLisp.

In April 1981, after a DARPA-sponsored meeting concerning the splintered Lisp community, Symbolics, the SPICE project, the NIL project, and the S-1 Lisp project joined together to define Common Lisp. Initially spearheaded by White and Gabriel, the driving force behind this grassroots effort was provided by Fahlman, Daniel Weinreb, David Moon, Steele, and Gabriel. Common Lisp was designed as a description of a family of languages. The primary influences on Common Lisp were Lisp Machine Lisp, MacLisp, NIL, S-1 Lisp, Spice Lisp, and Scheme. *Common Lisp: The Language* is a description of that design. Its semantics were intentionally underspecified in places where it was felt that a tight specification would overly constrain Common Lisp esearch and use.

In 1986 X3J13 was formed as a technical working group to produce a draft for an ANSI Common Lisp standard. Because of the acceptance of Common Lisp, the goals of this group differed from those of the original designers. These new goals included stricter standardization for portability, an object-oriented programming system, a condition system, iteration facilities, and a way to handle large character sets. To accommodate those goals, a new language specification, this document, was developed.

1.2 Organization of the Document

This is a reference document, not a tutorial document. Where possible and convenient, the order of presentation has been chosen so that the more primitive topics precede those that build upon them; however, linear readability has not been a priority.

This document is divided into chapters by topic. Any given chapter might contain conceptual material, dictionary entries, or both.

Defined names within the dictionary portion of a chapter are grouped in a way that brings related topics into physical proximity. Many such groupings were possible, and no deep significance should be inferred from the particular grouping that was chosen. To see *defined names* grouped alphabetically, consult the index. For a complete list of *defined names*, see Section 1.9 (Symbols in the COMMON-LISP Package).

In order to compensate for the sometimes-unordered portions of this document, a glossary has been provided; see Section 26 (Glossary). The glossary provides connectivity by providing easy access to definitions of terms, and in some cases by providing examples or cross references to additional conceptual material.

For information about notational conventions used in this document, see Section 1.4 (Definitions).

For information about conformance, see Section 1.5 (Conformance).

For information about extensions and subsets, see Section 1.6 (Language Extensions) nd Section 1.7 (Language Subsets).

For information about how *programs* in the language are parsed by the *Lisp reader*, see Section 2 (Syntax).

For information about how *programs* in the language are *compiled* and *executed*, see Section 3 (Evaluation and Compilation).

For information about data types, see Section 4 (Types and Classes). Not all *types* and *classes* are defined in this chapter; many are defined in chapter corresponding to their topic--for example, the numeric types are defined in Section 12 (Numbers). For a complete list of *standardized types*, see Figure 4-2.

1.3 Referenced Publications

- *The Anatomy of Lisp*, John Allen, McGraw-Hill, Inc., 1978.
- *The Art of Computer Programming, Volume 3*, Donald E. Knuth, Addison-Wesley Company (Reading, MA), 1973.

- *The Art of the Metaobject Protocol*, Kiczales et al., MIT Press (Cambridge, MA), 1991.
- "Common Lisp Object System Specification," D. Bobrow, L. DiMichiel, R.P. Gabriel, S. Keene, G. Kiczales, D. Moon, *SIGPLAN Notices* V23, September, 1988.
- *Common Lisp: The Language*, Guy L. Steele Jr., Digital Press (Burlington, MA), 1984.
- *Common Lisp: The Language, Second Edition*, Guy L. Steele Jr., Digital Press (Bedford, MA), 1990.
- *Exceptional Situations in Lisp*, Kent M. Pitman, *Proceedings of the First European Conference on the Practical Application of LISP* (EUROPAL '90), Churchill College, Cambridge, England, March 27-29, 1990.
- *Flavors: A Non-Hierarchical Approach to Object-Oriented Programming*, Howard I. Cannon, 1982.
- *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985, Institute of Electrical and Electronics Engineers, Inc. (New York), 1985.
- *IEEE Standard for the Scheme Programming Language*, IEEE Std 1178-1990, Institute of Electrical and Electronic Engineers, Inc. (New York), 1991.
- *Interlisp Reference Manual*, Third Revision, Teitelman, Warren, et al, Xerox Palo Alto Research Center (Palo Alto, CA), 1978.
- ISO 6937/2, *Information processing---Coded character sets for text communication---Part 2: Latin alphabetic and non-alphabetic graphic characters*, ISO, 1983.
- *Lisp 1.5 Programmer's Manual*, John McCarthy, MIT Press (Cambridge, MA), August, 1962.
- *Lisp Machine Manual*, D.L. Weinreb and D.A. Moon, Artificial Intelligence Laboratory, MIT (Cambridge, MA), July, 1981.
- *Maclisp Reference Manual, Revision 0*, David A. Moon, Project MAC (Laboratory for Computer Science), MIT (Cambridge, MA), March, 1974.
- "NIL---A Perspective," JonL White, *Macsyma User's Conference*, 1979.
- *Performance and Evaluation of Lisp Programs*, Richard P. Gabriel, MIT Press (Cambridge, MA), 1985.
- "Principal Values and Branch Cuts in Complex APL," Paul Penfield Jr., *APL 81 Conference Proceedings*, ACM SIGAPL (San Francisco, September 1981), 248-256. Proceedings published as *APL Quote Quad 12*, 1 (September 1981).
- *The Revised Maclisp Manual*, Kent M. Pitman, Technical Report 295, Laboratory for Computer Science, MIT (Cambridge, MA), May 1983.
- "*Revised^3 Report on the Algorithmic Language Scheme*," Jonathan Rees and William Clinger (editors), *SIGPLAN Notices* V21, #12, December, 1986.
- "S-1 Common Lisp Implementation," R.A. Brooks, R.P. Gabriel, and G.L. Steele, *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, 108-113, 1982.
- *Smalltalk-80: The Language and its Implementation*, A. Goldberg and D. Robson, Addison-Wesley Company, 1983.
- "Standard LISP Report," J.B. Marti, A.C. Hearn, M.L. Griss, and C. Griss, *SIGPLAN Notices* V14, #10, October, 1979.
- *Webster's Third New International Dictionary the English Language, Unabridged*, Merriam Webster (Springfield, MA), 1986.
- *XP: A Common Lisp Pretty Printing System*, R.C. Waters, Memo 1102a, Artificial Intelligence Laboratory, MIT (Cambridge, MA), September 1989.

1.4 Definitions

This section contains notational conventions and definitions of terms used in this manual.

1.4.1 Notational Conventions

The following notational conventions are used throughout this document.

1.4.1.1 Font Key

Fonts are used in this document to convey information.

name
: Denotes a formal term whose meaning is defined in the Glossary. When this font is used, the Glossary definition takes precedence over normal English usage.

  Sometimes a glossary term appears subscripted, as in "*whitespace*[2]." Such a notation selects one particular Glossary definition out of several, in this case the second. The subscript notation for Glossary terms is generally used where the context might be insufficient to disambiguate among the available definitions.

name
: Denotes the introduction of a formal term locally to the current text. There is still a corresponding glossary entry, and is formally equivalent to a use of "*name*," but the hope is that making such uses conspicuous will save the reader a trip to the glossary in some cases.

name
: Denotes a symbol in the `COMMON-LISP` package. For information about *case* conventions, see Section 1.4.1.4.1 (Case in Symbols).

`name`
: Denotes a sample *name* or piece of *code* that a programmer might write in Common Lisp.

  This font is also used for certain *standardized* names that are not names of *external symbols* of the `COMMON-LISP` package, such as *keywords*[1], *package names*, and *loop keywords*.

name
: Denotes the name of a *parameter* or *value*.

  In some situations the notation "<<*name*>>" (i.e., the same font, but with surrounding "angle brackets") is used instead in order to provide better visual separation from surrounding characters. These "angle brackets" are metasyntactic, and never actually appear in program input or output.

1.4.1.2 Modified BNF Syntax

This specification uses an extended Backus Normal Form (BNF) to describe the syntax of Common Lisp *macro forms* and *special forms*. This section discusses the syntax of BNF expressions.

1.4.1.2.1 Splicing in Modified BNF Syntax

The primary extension used is the following:

[[O]]

An expression of this form appears whenever a list of elements is to be spliced into a larger structure and the elements can appear in any order. The symbol O represents a description of the syntax of some number of syntactic elements to be spliced; that description must be of the form

O1 | ... | Ol

where each Oi can be of the form S or of the form S* or of the form {S}1 . The expression [[O]] means that a list of the form

(Oi1...Oij) 1<=j

is spliced into the enclosing expression, such that if n /=m and 1<=n,m<=j, then either Oin/=Oim or Oin = Oim = Qk, where for some 1<=k <=n, Ok is of the form Qk*. Furthermore, for each Oin that is of the form {Qk}1 , that element is required to appear somewhere in the list to be spliced.

For example, the expression

```
(x [[A | B* | C]] y)
```

means that at most one A, any number of B's, and at most one C can occur in any order. It is a description of any of these:

```
(x y)
(x B A C y)
(x A B B B B B C y)
(x C B A B B B y)
```

but not any of these:

```
(x B B A A C C y)
(x C B C y)
```

In the first case, both A and C appear too often, and in the second case C appears too often.

The notation [[O1 | O2 | ...]]+ adds the additional restriction that at least one item from among the possible choices must be used. For example:

```
(x [[A | B* | C]]+ y)
```

means that at most one A, any number of B's, and at most one C can occur in any order, but that in any case at least one of these options must be selected. It is a description of any of these:

```
(x B y)
(x B A C y)
(x A B B B B B C y)
(x C B A B B B y)
```

but not any of these:

```
(x y)
(x B B A A C C y)
(x C B C y)
```

In the first case, no item was used; in the second case, both A and C appear too often; and in the third case C appears too often.

Also, the expression:

```
(x [[{A}1 | {B}1 | C]] y)
```

can generate exactly these and no others:

```
(x A B C y)
(x A C B y)
(x A B y)
(x B A C y)
(x B C A y)
(x B A y)
(x C A B y)
(x C B A y)
```

1.4.1.2.2 Indirection in Modified BNF Syntax

An indirection extension is introduced in order to make this new syntax more readable:

O

If *O* is a non-terminal symbol, the right-hand side of its definition is substituted for the entire expression *O*. For example, the following BNF is equivalent to the BNF in the previous example:

`(x [[`*O*`]] y)`

```
O::= A | B* | C
```

1.4.1.2.3 Additional Uses for Indirect Definitions in Modified BNF Syntax

In some cases, an auxiliary definition in the BNF might appear to be unused within the BNF, but might still be useful elsewhere. For example, consider the following definitions:

case *keyform* {normal-clause}***** [*otherwise-clause*] => *result******

ccase *keyplace* {normal-clause}***** => *result******

ecase *keyform* {normal-clause}***** => *result******

```
normal-clause::= (keys form*)

otherwise-clause::= ({otherwise | t} form*)

clause::= normal-clause | otherwise-clause
```

Here the term "*clause*" might appear to be "dead" in that it is not used in the BNF. However, the purpose of the BNF is not just to guide parsing, but also to define useful terms for reference in the descriptive text which follows. As such, the term "*clause*" might appear in text that follows, as shorthand for "*normal-clause* or *otherwise-clause*."

1.4.1.3 Special Symbols

The special symbols described here are used as a notational convenience within this document, and are part of neither the Common Lisp language nor its environment.

=>

This indicates evaluation. For example:

```
(+ 4 5) =>  9
```

This means that the result of evaluating the *form* `(+ 4 5)` is `9`.

If a *form* returns *multiple values*, those values might be shown separated by spaces, line breaks, or commas. For example:

```
 (truncate 7 5)
=>  1 2
 (truncate 7 5)
=>  1
    2
 (truncate 7 5)
=>  1, 2
```

Each of the above three examples is equivalent, and specifies that `(truncate 7 5)` returns two values, which are `1` and `2`.

Some *conforming implementations* actually type an arrow (or some other indicator) before showing return values, while others do not.

OR=>

The notation "OR=> " is used to denote one of several possible alternate results. The example

```
 (char-name #\a)
=>  NIL
OR=>  "LOWERCASE-a"
OR=>  "Small-A"
OR=>  "LA01"
```

indicates that **nil**, `"LOWERCASE-a"`, `"Small-A"`, `"LA01"` are among the possible results of `(char-name #\a)`---each with equal preference. Unless explicitly specified otherwise, it should not be assumed that the set of possible results shown is exhaustive. Formally, the above example is equivalent to

```
 (char-name #\a) =>  implementation-dependent
```

but it is intended to provide additional information to illustrate some of the ways in which it is permitted for implementations to diverge.

NOT=>

The notation "NOT=> " is used to denote a result which is not possible. This might be used, for example, in order to emphasize a situation where some anticipated misconception might lead the reader to falsely believe that the result might be possible. For example,

```
 (function-lambda-expression
    (funcall #'(lambda (x) #'(lambda () x)) nil))
=>  NIL, true, NIL
OR=>  (LAMBDA () X), true, NIL
NOT=>  NIL, false, NIL
NOT=>  (LAMBDA () X), false, NIL
```

==

This indicates code equivalence. For example:

```
 (gcd x (gcd y z)) ==  (gcd (gcd x y) z)
```

This means that the results and observable side-effects of evaluating the *form* `(gcd x (gcd y z))` are always the same as the results and observable side-effects of `(gcd (gcd x y) z)` for any `x`, `y`, and `z`.

>>

Common Lisp specifies input and output with respect to a non-interactive stream model. The specific details of how interactive input and output are mapped onto that non-interactive model are *implementation-defined*.

For example, *conforming implementations* are permitted to differ in issues of how interactive input is terminated. For example, the *function* **read** terminates when the final delimiter is typed on a non-interactive stream. In some *implementations*, an interactive call to **read** returns as soon as the final delimiter is typed, even if that delimiter is not a *newline*. In other *implementations*, a final *newline* is always required. In still other *implementations*, there might be a command which "activates" a buffer full of input without the command itself being visible on the program's input stream.

In the examples in this document, the notation ">> " precedes lines where interactive input and output occurs. Within such a scenario, "this notation" notates user input.

For example, the notation

```
 (+ 1 (print (+ (sqrt (read)) (sqrt (read)))))
>>  9 16
>>  7
=>  8
```

shows an interaction in which "`(+ 1 (print (+ (sqrt (read)) (sqrt (read)))))`" is a *form* to be *evaluated*, "`9 16` " is interactive input, "`7`" is interactive output, and "`8`" is the *value yielded* from the *evaluation*.

The use of this notation is intended to disguise small differences in interactive input and output behavior between *implementations*.

Sometimes, the non-interactive stream model calls for a *newline*. How that *newline* character is interactively entered is an *implementation-defined* detail of the user interface, but in that case, either the notation "<Newline>" or "<NEWLINE>" might be used.

```
 (progn (format t "~&Who? ") (read-line))
>>  Who? Fred, Mary, and Sally<NEWLINE>
=>  "Fred, Mary, and Sally", false
```

1.4.1.4 Objects with Multiple Notations

Some *objects* in Common Lisp can be notated in more than one way. In such situations, the choice of which notation to use is technically arbitrary, but conventions may exist which convey a "point of view" or "sense of intent."

1.4.1.4.1 Case in Symbols

While *case* is significant in the process of *interning* a *symbol*, the *Lisp reader*, by default, attempts to canonicalize the case of a *symbol* prior to interning; see Section 23.1.2 (Effect of Readtable Case on the Lisp Reader). As such, case in *symbols* is not, by default, significant. Throughout this document, except as explicitly noted otherwise, the case in which a *symbol* appears is not significant; that is, `HELLO`, `Hello`, `hElLo`, and `hello` are all equivalent ways to denote a symbol whose name is `"HELLO"`.

The characters *backslash* and *vertical-bar* are used to explicitly quote the *case* and other parsing-related aspects of characters. As such, the notations |`hello`| and `\h\e\l\l\o` are equivalent ways to refer to a symbol whose name is `"hello"`, and which is *distinct* from any symbol whose name is `"HELLO"`.

The *symbols* that correspond to Common Lisp *defined names* have *uppercase* names even though their names generally appear in *lowercase* in this document.

1.4.1.4.2 Numbers

Although Common Lisp provides a variety of ways for programs to manipulate the input and output radix for rational numbers, all numbers in this document are in decimal notation unless explicitly noted otherwise.

1.4.1.4.3 Use of the Dot Character

The dot appearing by itself in an *expression* such as

(*item1 item2* . *tail*)

means that *tail* represents a *list* of *objects* at the end of a list. For example,

```
(A B C . (D E F))
```

is notationally equivalent to:

```
(A B C D E F)
```

Although *dot* is a valid constituent character in a symbol, no *standardized symbols* contain the character *dot*, so a period that follows a reference to a *symbol* at the end of a sentence in this document should always be interpreted as a period and never as part of the *symbol*'s *name*. For example, within this document, a sentence such as "This sample sentence refers to the symbol **car**." refers to a symbol whose name is `"CAR"` (with three letters), and never to a four-letter symbol `"CAR."`

1.4.1.4.4 NIL

nil has a variety of meanings. It is a *symbol* in the `COMMON-LISP` package with the *name* `"NIL"`, it is *boolean* (and *generalized boolean*) *false*, it is the *empty list*, and it is the *name* of the *empty type* (a *subtype* of all *types*).

Within Common Lisp, **nil** can be notated interchangeably as either `NIL` or `()`. By convention, the choice of notation offers a hint as to which of its many roles it is playing.

```
For Evaluation?  Notation  Typically Implied Role
----------

Yes              nil       use as a boolean.
Yes              'nil      use as a symbol.
Yes              '()       use as an empty list
No               nil       use as a symbol or boolean.
No               ()        use as an empty list.
```

Figure 1-1. Notations for NIL

Within this document only, **nil** is also sometimes notated as *false* to emphasize its role as a *boolean*.

For example:

```
(print ())                          ;avoided
(defun three nil 3)                 ;avoided
'(nil nil)                          ;list of two symbols
'(() ())                            ;list of empty lists
(defun three () 3)                  ;Emphasize empty parameter list.
(append '() '()) =>  ()              ;Emphasize use of empty lists
(not nil) =>  true                   ;Emphasize use as Boolean false
(get 'nil 'color)                   ;Emphasize use as a symbol
```

A *function* is sometimes said to "be *false*" or "be *true*" in some circumstance. Since no *function* object can be the same as **nil** and all *function objects* represent *true* when viewed as *booleans*, it would be meaningless to say that the *function* was literally *false* and uninteresting to say that it was literally *true*. Instead, these phrases are just traditional alternative ways of saying that the *function* "returns *false*" or "returns *true*," respectively.

1.4.1.5 Designators

A *designator* is an *object* that denotes another *object*.

Where a *parameter* of an *operator* is described as a *designator*, the description of the *operator* is written in a way that assumes that the value of the *parameter* is the denoted *object*; that is, that the *parameter* is already of the denoted *type*. (The specific nature of the *object* denoted by a "<<*type*>> *designator*" or a "*designator* for a <<*type*>>" can be found in the Glossary entry for "<<*type*>> *designator*.")

For example, "**nil**" and "the *value* of ***standard-output***" are operationally indistinguishable as *stream designators*. Similarly, the *symbol* `FOO` and the *string* `"FOO"` are operationally indistinguishable as *string designators*.

Except as otherwise noted, in a situation where the denoted *object* might be used multiple times, it is *implementation-dependent* whether the *object* is coerced only once or whether the coercion occurs each time the *object* must be used.

For example, **mapcar** receives a *function designator* as an argument, and its description is written as if this were simply a function. In fact, it is *implementation-dependent* whether the *function designator* is coerced right away or whether it is carried around internally in the form that it was given as an *argument* and re-coerced each time it is needed. In most cases, *conforming programs* cannot detect the distinction, but there are some pathological situations (particularly those involving self-redefining or mutually-redefining functions) which do conform and which can detect this difference. The following program is a *conforming program*, but might or might not have portably correct results, depending on whether its correctness depends on one or the other of the results:

```
 (defun add-some (x)
   (defun add-some (x) (+ x 2))
   (+ x 1)) =>  ADD-SOME
 (mapcar 'add-some '(1 2 3 4))
=>  (2 3 4 5)
OR=>  (2 4 5 6)
```

In a few rare situations, there may be a need in a dictionary entry to refer to the *object* that was the original *designator* for a *parameter*. Since naming the *parameter* would refer to the denoted *object*, the phrase "the <<*parameter-name*>> *designator*" can be used to refer to the *designator* which was the *argument* from which the *value* of <<*parameter-name*>> was computed.

1.4.1.6 Nonsense Words

When a word having no pre-attached semantics is required (e.g., in an example), it is common in the Lisp community to use one of the words "foo," "bar," "baz," and "quux." For example, in

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

the use of the name `FOO` is just a shorthand way of saying "please substitute your favorite name here."

These nonsense words have gained such prevalance of usage, that it is commonplace for newcomers to the community to begin to wonder if there is an attached semantics which they are overlooking---there is not.

1.4.2 Error Terminology

Situations in which errors might, should, or must be signaled are described in the standard. The wording used to describe such situations is intended to have precise meaning. The following list is a glossary of those meanings.

Safe code
: This is *code* processed with the **safety** optimization at its highest setting (3). **safety** is a lexical property of code. The phrase "the function F should signal an error" means that if F is invoked from code processed with the highest **safety** optimization, an error is signaled. It is *implementation-dependent* whether F or the calling code signals the error.

Unsafe code
: This is code processed with lower safety levels.

  Unsafe code might do error checking. Implementations are permitted to treat all code as safe code all the time.

An error is signaled
: This means that an error is signaled in both safe and unsafe code. *Conforming code* may rely on the fact that the error is signaled in both safe and unsafe code. Every implementation is required to detect the error in both safe and unsafe code. For example, "an error is signaled if **unexport** is given a *symbol* not *accessible* in the *current package*."

  If an explicit error type is not specified, the default is **error**.

An error should be signaled
: This means that an error is signaled in safe code, and an error might be signaled in unsafe code. *Conforming code* may rely on the fact that the error is signaled in safe code. Every implementation is required to detect the error at least in safe code. When the error is not signaled, the "consequences are undefined" (see below). For example, "+ should signal an error of *type* **type-error** if any argument is not of *type* **number**."

Should be prepared to signal an error
: This is similar to "should be signaled" except that it does not imply that ‘extra effort’ has to be taken on the part of an *operator* to discover an erroneous situation if the normal action of that *operator* can be performed successfully with only ‘lazy’ checking. An *implementation* is always permitted to signal an error, but even in *safe code*, it is only required to signal the error when failing to signal it might lead to incorrect results. In *unsafe code*, the consequences are undefined.

  For example, defining that "**find** should be prepared to signal an error of *type* **type-error** if its second *argument* is not a *proper list*" does not imply that an error is always signaled. The *form*

```
(find 'a '(a b . c))
```

  must either signal an error of *type* **type-error** in *safe code*, else return A. In *unsafe code*, the consequences are undefined. By contrast,

```
(find 'd '(a b . c))
```

  must signal an error of *type* **type-error** in *safe code*. In *unsafe code*, the consequences are undefined. Also,

```
(find 'd '#1=(a b . #1#))
```

in *safe code* might return **nil** (as an *implementation-defined* extension), might never return, or might signal an error of *type* **type-error**. In *unsafe code*, the consequences are undefined.

Typically, the "should be prepared to signal" terminology is used in type checking situations where there are efficiency considerations that make it impractical to detect errors that are not relevant to the correct operation of the *operator*.

The consequences are unspecified
This means that the consequences are unpredictable but harmless. Implementations are permitted to specify the consequences of this situation. No *conforming code* may depend on the results or effects of this situation, and all *conforming code* is required to treat the results and effects of this situation as unpredictable but harmless. For example, "if the second argument to **shared-initialize** specifies a name that does not correspond to any *slots accessible* in the *object*, the results are unspecified."

The consequences are undefined
This means that the consequences are unpredictable. The consequences may range from harmless to fatal. No *conforming code* may depend on the results or effects. *Conforming code* must treat the consequences as unpredictable. In places where the words "must," "must not," or "may not" are used, then "the consequences are undefined" if the stated requirement is not met and no specific consequence is explicitly stated. An implementation is permitted to signal an error in this case.

For example: "Once a name has been declared by **defconstant** to be constant, any further assignment or binding of that variable has undefined consequences."

An error might be signaled
This means that the situation has undefined consequences; however, if an error is signaled, it is of the specified *type*. For example, "**open** might signal an error of *type* **file-error**."

The return values are unspecified
This means that only the number and nature of the return values of a *form* are not specified. However, the issue of whether or not any side-effects or transfer of control occurs is still well-specified.

A program can be well-specified even if it uses a function whose returns values are unspecified. For example, even if the return values of some function `F` are unspecified, an expression such as `(length (list (F)))` is still well-specified because it does not rely on any particular aspect of the value or values returned by `F`.

Implementations may be extended to cover this situation
This means that the *situation* has undefined consequences; however, a *conforming implementation* is free to treat the situation in a more specific way. For example, an *implementation* might define that an error is signaled, or that an error should be signaled, or even that a certain well-defined non-error behavior occurs.

No *conforming code* may depend on the consequences of such a *situation*; all *conforming code* must treat the consequences of the situation as undefined. *Implementations* are required to document how the situation is treated.

For example, "implementations may be extended to define other type specifiers to have a corresponding *class*."

Implementations are free to extend the syntax
This means that in this situation implementations are permitted to define unambiguous extensions to the syntax of the *form* being described. No *conforming code* may depend on this extension. Implementations are required to document each such extension. All *conforming code* is required to treat the syntax as meaningless. The standard might disallow certain extensions while allowing others. For example, "no implementation is free to extend the syntax of **defclass**."

A warning might be issued

This means that *implementations* are encouraged to issue a warning if the context is appropriate (e.g., when compiling). However, a *conforming implementation* is not required to issue a warning.

1.4.3 Sections Not Formally Part Of This Standard

Front matter and back matter, such as the "Table of Contents," "Index," "Figures," "Credits," and "Appendix" are not considered formally part of this standard, so that we retain the flexibility needed to update these sections even at the last minute without fear of needing a formal vote to change those parts of the document. These items are quite short and very useful, however, and it is not recommended that they be removed even in an abridged version of this document.

Within the concept sections, subsections whose names begin with the words "Note" or "Notes" or "Example" or "Examples" are provided for illustration purposes only, and are not considered part of the standard.

An attempt has been made to place these sections last in their parent section, so that they could be removed without disturbing the contiguous numbering of the surrounding sections in order to produce a document of smaller size.

Likewise, the "Examples" and "Notes" sections in a dictionary entry are not considered part of the standard and could be removed if necessary.

Nevertheless, the examples provide important clarifications and consistency checks for the rest of the material, and such abridging is not recommended unless absolutely unavoidable.

1.4.4 Interpreting Dictionary Entries

The dictionary entry for each *defined name* is partitioned into sections. Except as explicitly indicated otherwise below, each section is introduced by a label identifying that section. The omission of a section implies that the section is either not applicable, or would provide no interesting information.

This section defines the significance of each potential section in a dictionary entry.

1.4.4.1 The "Affected By" Section of a Dictionary Entry

For an *operator*, anything that can affect the side effects of or *values* returned by the *operator*.

For a *variable*, anything that can affect the *value* of the *variable* including *functions* that bind or assign it.

1.4.4.2 The "Arguments" Section of a Dictionary Entry

This information describes the syntax information of entries such as those for *declarations* and special *expressions* which are never *evaluated* as *forms*, and so do not return *values*.

1.4.4.3 The "Arguments and Values" Section of a Dictionary Entry

An English language description of what *arguments* the *operator* accepts and what *values* it returns, including information about defaults for *parameters* corresponding to omittable *arguments* (such as *optional parameters* and *keyword parameters*). For *special operators* and *macros*, their *arguments* are not *evaluated* unless it is explicitly stated in their descriptions that they are *evaluated*.

Except as explicitly specified otherwise, the consequences are undefined if these type restrictions are violated.

1.4.4.4 The "Binding Types Affected" Section of a Dictionary Entry

This information alerts the reader to the kinds of *bindings* that might potentially be affected by a declaration. Whether in fact any particular such *binding* is actually affected is dependent on additional factors as well. See the "Description" section of the declaration in question for details.

1.4.4.5 The "Class Precedence List" Section of a Dictionary Entry

This appears in the dictionary entry for a *class*, and contains an ordered list of the *classes* defined by Common Lisp that must be in the *class precedence list* of this *class*.

It is permissible for other (*implementation-defined*) *classes* to appear in the *implementation*'s *class precedence list* for the *class*.

It is permissible for either **standard-object** or **structure-object** to appear in the *implementation*'s *class precedence list*; for details, see Section 4.2.2 (Type Relationships).

Except as explicitly indicated otherwise somewhere in this specification, no additional *standardized classes* may appear in the *implementation*'s *class precedence list*.

By definition of the relationship between *classes* and *types*, the *classes* listed in this section are also *supertypes* of the *type* denoted by the *class*.

1.4.4.6 Dictionary Entries for Type Specifiers

The *atomic type specifiers* are those *defined names* listed in Figure 4-2. Such dictionary entries are of kind "Class," "Condition Type," "System Class," or "Type." A description of how to interpret a *symbol* naming one of these *types* or *classes* as an *atomic type specifier* is found in the "Description" section of such dictionary entries.

The *compound type specifiers* are those *defined names* listed in Figure 4-3. Such dictionary entries are of kind "Class," "System Class," "Type," or "Type Specifier." A description of how to interpret as a *compound type specifier* a *list* whose *car* is such a *symbol* is found in the "Compound Type Specifier Kind," "Compound Type Specifier Syntax," "Compound Type Specifier Arguments," and "Compound Type Specifier Description" sections of such dictionary entries.

1.4.4.6.1 The "Compound Type Specifier Kind" Section of a Dictionary Entry

An "abbreviating" *type specifier* is one that describes a *subtype* for which it is in principle possible to enumerate the *elements*, but for which in practice it is impractical to do so.

A "specializing" *type specifier* is one that describes a *subtype* by restricting the *type* of one or more components of the *type*, such as *element type* or *complex part type*.

A "predicating" *type specifier* is one that describes a *subtype* containing only those *objects* that satisfy a given *predicate*.

A "combining" *type specifier* is one that describes a *subtype* in a compositional way, using combining operations (such as "and," "or," and "not") on other *types*.

1.4.4.6.2 The "Compound Type Specifier Syntax" Section of a Dictionary Entry

This information about a *type* describes the syntax of a *compound type specifier* for that *type*.

Whether or not the *type* is acceptable as an *atomic type specifier* is not represented here; see Section 1.4.4.6 (Dictionary Entries for Type Specifiers).

1.4.4.6.3 The "Compound Type Specifier Arguments" Section of a Dictionary Entry

This information describes *type* information for the structures defined in the "Compound Type Specifier Syntax" section.

1.4.4.6.4 The "Compound Type Specifier Description" Section of a Dictionary Entry

This information describes the meaning of the structures defined in the "Compound Type Specifier Syntax" section.

1.4.4.7 The "Constant Value" Section of a Dictionary Entry

This information describes the unchanging *type* and *value* of a *constant variable*.

1.4.4.8 The "Description" Section of a Dictionary Entry

A summary of the *operator* and all intended aspects of the *operator*, but does not necessarily include all the fields referenced below it ("Side Effects," "Exceptional Situations," *etc*.)

1.4.4.9 The "Examples" Section of a Dictionary Entry

Examples of use of the *operator*. These examples are not considered part of the standard; see Section 1.4.3 (Sections Not Formally Part Of This Standard).

1.4.4.10 The "Exceptional Situations" Section of a Dictionary Entry

Three kinds of information may appear here:

- Situations that are detected by the *function* and formally signaled.
- Situations that are handled by the *function*.
- Situations that may be detected by the *function*.

This field does not include conditions that could be signaled by *functions* passed to and called by this *operator* as arguments or through dynamic variables, nor by executing subforms of this operator if it is a *macro* or *special operator*.

1.4.4.11 The "Initial Value" Section of a Dictionary Entry

This information describes the initial *value* of a *dynamic variable*. Since this variable might change, see *type* restrictions in the "Value Type" section.

1.4.4.12 The "Argument Precedence Order" Section of a Dictionary Entry

This information describes the *argument precedence order*. If it is omitted, the *argument precedence order* is the default (left to right).

1.4.4.13 The "Method Signature" Section of a Dictionary Entry

The description of a *generic function* includes descriptions of the *methods* that are defined on that *generic function* by the standard. A method signature is used to describe the *parameters* and *parameter specializers* for each *method*. *Methods* defined for the *generic function* must be of the form described by the *method signature*.

F *(*x *class*) (*y* t) `&optional` *z* `&key` *k*

This *signature* indicates that this method on the *generic function* **F** has two *required parameters*: *x*, which must be a *generalized instance* of the *class class*; and *y*, which can be any *object* (i.e., a *generalized instance* of the *class* **t**). In addition, there is an *optional parameter z* and a *keyword parameter k*. This *signature* also indicates that this method on `F` is a *primary method* and has no *qualifiers*.

For each *parameter*, the *argument* supplied must be in the intersection of the *type* specified in the description of the corresponding *generic function* and the *type* given in the *signature* of some *method* (including not only those *methods* defined in this specification, but also *implementation-defined* or user-defined *methods* in situations where the definition of such *methods* is permitted).

1.4.4.14 The "Name" Section of a Dictionary Entry

This section introduces the dictionary entry. It is not explicitly labeled. It appears preceded and followed by a horizontal bar.

In large print at left, the *defined name* appears; if more than one *defined name* is to be described by the entry, all such *names* are shown separated by commas.

In somewhat smaller italic print at right is an indication of what kind of dictionary entry this is. Possible values are:

Accessor
: This is an *accessor function*.

Class
: This is a *class*.

Condition Type
: This is a *subtype* of *type* **condition**.

Constant Variable
: This is a *constant variable*.

Declaration
: This is a *declaration identifier*.

Function
: This is a *function*.

Local Function
: This is a *function* that is defined only lexically within the scope of some other *macro form*.

Local Macro
: This is a *macro* that is defined only lexically within the scope of some other *macro form*.

Macro
: This is a *macro*.

Restart
: This is a *restart*.

Special Operator
: This is a *special operator*.

Standard Generic Function
: This is a *standard generic function*.

Symbol
: This is a *symbol* that is specially recognized in some particular situation, such as the syntax of a *macro*.

System Class
: This is like *class*, but it identifies a *class* that is potentially a *built-in class*. (No *class* is actually required to be a *built-in class*.)

Type
: This is an *atomic type specifier*, and depending on information for each particular entry, may subject to form other *type specifiers*.

Type Specifier
: This is a *defined name* that is not an *atomic type specifier*, but that can be used in constructing valid *type specifiers*.

Variable
: This is a *dynamic variable*.

1.4.4.15 The "Notes" Section of a Dictionary Entry

Information not found elsewhere in this description which pertains to this *operator*. Among other things, this might include cross reference information, code equivalences, stylistic hints, implementation hints, typical uses. This information is not considered part of the standard; any *conforming implementation* or *conforming program* is permitted to ignore the presence of this information.

1.4.4.16 The "Pronunciation" Section of a Dictionary Entry

This offers a suggested pronunciation for *defined names* so that people not in verbal communication with the original designers can figure out how to pronounce words that are not in normal English usage. This information is advisory only, and is not considered part of the standard. For brevity, it is only provided for entries with names that are specific to Common Lisp and would not be found in *Webster's Third New International Dictionary the English Language, Unabridged*.

1.4.4.17 The "See Also" Section of a Dictionary Entry

List of references to other parts of this standard that offer information relevant to this *operator*. This list is not part of the standard.

1.4.4.18 The "Side Effects" Section of a Dictionary Entry

Anything that is changed as a result of the evaluation of the *form* containing this *operator*.

1.4.4.19 The "Supertypes" Section of a Dictionary Entry

This appears in the dictionary entry for a *type*, and contains a list of the *standardized types* that must be *supertypes* of this *type*.

In *implementations* where there is a corresponding *class*, the order of the *classes* in the *class precedence list* is consistent with the order presented in this section.

1.4.4.20 The "Syntax" Section of a Dictionary Entry

This section describes how to use the *defined name* in code. The "Syntax" description for a *generic function* describes the *lambda list* of the *generic function* itself, while the "Method Signatures" describe the *lambda lists* of the defined *methods*. The "Syntax" description for an *ordinary function*, a *macro*, or a *special operator* describes its *parameters*.

For example, an *operator* description might say:

F *x y* `&optional` *z* `&key` *k*

This description indicates that the function **F** has two required parameters, *x* and *y*. In addition, there is an optional parameter *z* and a keyword parameter *k*.

For *macros* and *special operators*, syntax is given in modified BNF notation; see Section 1.4.1.2 (Modified BNF Syntax). For *functions* a *lambda list* is given. In both cases, however, the outermost parentheses are omitted, and default value information is omitted.

1.4.4.20.1 Special "Syntax" Notations for Overloaded Operators

If two descriptions exist for the same operation but with different numbers of arguments, then the extra arguments are to be treated as optional. For example, this pair of lines:

file-position *stream* => *position*

file-position *stream position-spec* => *success-p*

is operationally equivalent to this line:

file-position *stream* `&optional` *position-spec* => *result*

and differs only in that it provides on opportunity to introduce different names for *parameter* and *values* for each case. The separated (multi-line) notation is used when an *operator* is overloaded in such a way that the *parameters* are used in different ways depending on how many *arguments* are supplied (e.g., for the *function* /) or the return values are different in the two cases (e.g., for the *function* **file-position**).

1.4.4.20.2 Naming Conventions for Rest Parameters

Within this specification, if the name of a *rest parameter* is chosen to be a plural noun, use of that name in *parameter* font refers to the *list* to which the *rest parameter* is bound. Use of the singular form of that name in *parameter* font refers to an *element* of that *list*.

For example, given a syntax description such as:

F `&rest` arguments

it is appropriate to refer either to the *rest parameter* named *arguments* by name, or to one of its elements by speaking of "an *argument*," "some *argument*," "each *argument*" *etc*.

1.4.4.20.3 Requiring Non-Null Rest Parameters in the "Syntax" Section

In some cases it is useful to refer to all arguments equally as a single aggregation using a *rest parameter* while at the same time requiring at least one argument. A variety of imperative and declarative means are available in *code* for expressing such a restriction, however they generally do not manifest themselves in a *lambda list*. For descriptive purposes within this specification,

F `&rest` *arguments+*

means the same as

F `&rest` *arguments*

but introduces the additional requirement that there be at least one *argument*.

1.4.4.20.4 Return values in the "Syntax" Section

An evaluation arrow "=> " precedes a list of *values* to be returned. For example:

F *a b c* => *x*

indicates that F is an operator that has three *required parameters* (i.e., *a*, *b*, and *c*) and that returns one *value* (i.e., *x*). If more than one *value* is returned by an operator, the *names* of the *values* are separated by commas, as in:

F *a b c* => *x, y, z*

1.4.4.20.4.1 No Arguments or Values in the "Syntax" Section

If no *arguments* are permitted, or no *values* are returned, a special notation is used to make this more visually apparent. For example,

F <*no arguments*> => <*no values*>

indicates that F is an operator that accepts no *arguments* and returns no *values*.

1.4.4.20.4.2 Unconditional Transfer of Control in the "Syntax" Section

Some *operators* perform an unconditional transfer of control, and so never have any return values. Such *operators* are notated using a notation such as the following:

F *a b c* =>|

1.4.4.21 The "Valid Context" Section of a Dictionary Entry

This information is used by dictionary entries such as "Declarations" in order to restrict the context in which the declaration may appear.

A given "Declaration" might appear in a *declaration* (i.e., a **declare** *expression*), a *proclamation* (i.e., a **declaim** or **proclaim** *form*), or both.

1.4.4.22 The "Value Type" Section of a Dictionary Entry

This information describes any *type* restrictions on a *dynamic variable*.

Except as explicitly specified otherwise, the consequences are undefined if this type restriction is violated.

1.5 Conformance

This standard presents the syntax and semantics to be implemented by a *conforming implementation* (and its accompanying documentation). In addition, it imposes requirements on *conforming programs*.

1.5.1 Conforming Implementations

A *conforming implementation* shall adhere to the requirements outlined in this section.

1.5.1.1 Required Language Features

A *conforming implementation* shall accept all features (including deprecated features) of the language specified in this standard, with the meanings defined in this standard.

A *conforming implementation* shall not require the inclusion of substitute or additional language elements in code in order to accomplish a feature of the language that is specified in this standard.

1.5.1.2 Documentation of Implementation-Dependent Features

A *conforming implementation* shall be accompanied by a document that provides a definition of all *implementation-defined* aspects of the language defined by this specification.

In addition, a *conforming implementation* is encouraged (but not required) to document items in this standard that are identified as *implementation-dependent*, although in some cases such documentation might simply identify the item as "undefined."

1.5.1.3 Documentation of Extensions

A *conforming implementation* shall be accompanied by a document that separately describes any features accepted by the *implementation* that are not specified in this standard, but that do not cause any ambiguity or contradiction when added to the language standard. Such extensions shall be described as being "extensions to Common Lisp as specified by ANSI <<*standard number*>>."

1.5.1.4 Treatment of Exceptional Situations

A *conforming implementation* shall treat exceptional situations in a manner consistent with this specification.

1.5.1.4.1 Resolution of Apparent Conflicts in Exceptional Situations

If more than one passage in this specification appears to apply to the same situation but in conflicting ways, the passage that appears to describe the situation in the most specific way (not necessarily the passage that provides the most constrained kind of error detection) takes precedence.

1.5.1.4.1.1 Examples of Resolution of Apparent Conflicts in Exceptional Situations

Suppose that function `foo` is a member of a set S of *functions* that operate on numbers. Suppose that one passage states that an error must be signaled if any *function* in S is ever given an argument of `17`. Suppose that an apparently conflicting passage states that the consequences are undefined if `foo` receives an argument of `17`. Then the second passage (the one specifically about `foo`) would dominate because the description of the situational context is the most specific, and it would not be required that `foo` signal an error on an argument of `17` even though other functions in the set S would be required to do so.

1.5.1.5 Conformance Statement

A *conforming implementation* shall produce a conformance statement as a consequence of using the implementation, or that statement shall be included in the accompanying documentation. If the implementation conforms in all respects with this standard, the conformance statement shall be

"<<*Implementation*>> conforms with the requirements of ANSI <<*standard number*>>"

If the *implementation* conforms with some but not all of the requirements of this standard, then the conformance statement shall be

"<<*Implementation*>> conforms with the requirements of ANSI <<*standard number*>> with the following exceptions: <<*reference to or complete list of the requirements of the standard with which the implementation does not conform*>>."

1.5.2 Conforming Programs

Code conforming with the requirements of this standard shall adhere to the following:

1. *Conforming code* shall use only those features of the language syntax and semantics that are either specified in this standard or defined using the extension mechanisms specified in the standard.
2. *Conforming code* may use *implementation-dependent* features and values, but shall not rely upon any particular interpretation of these features and values other than those that are discovered by the execution of *code*.
3. *Conforming code* shall not depend on the consequences of undefined or unspecified situations.
4. *Conforming code* does not use any constructions that are prohibited by the standard.
5. *Conforming code* does not depend on extensions included in an implementation.

1.5.2.1 Use of Implementation-Defined Language Features

Note that *conforming code* may rely on particular *implementation-defined* values or features. Also note that the requirements for *conforming code* and *conforming implementations* do not require that the results produced by conforming code always be the same when processed by a *conforming implementation*. The results may be the same, or they may differ.

Conforming code may run in all conforming implementations, but might have allowable *implementation-defined* behavior that makes it non-portable code. For example, the following are examples of *forms* that are conforming, but that might return different *values* in different implementations:

```
(evenp most-positive-fixnum) =>  implementation-dependent
(random) =>  implementation-dependent
(> lambda-parameters-limit 93) =>  implementation-dependent
(char-name #\A) =>  implementation-dependent
```

1.5.2.1.1 Use of Read-Time Conditionals

Use of `#+` and `#-` does not automatically disqualify a program from being conforming. A program which uses `#+` and `#-` is considered conforming if there is no set of *features* in which the program would not be conforming. Of course, *conforming programs* are not necessarily working programs. The following program is conforming:

```
(defun foo ()
  #+ACME (acme:initialize-something)
  (print 'hello-there))
```

However, this program might or might not work, depending on whether the presence of the feature `ACME` really implies that a function named `acme:initialize-something` is present in the environment. In effect, using `#+` or `#-` in a *conforming program* means that the variable ***features*** becomes just one more piece of input data to that program. Like any other data coming into a program, the programmer is responsible for assuring that the program does not make unwarranted assumptions on the basis of input data.

1.5.2.2 Character Set for Portable Code

Portable code is written using only *standard characters*.

1.6 Language Extensions

A language extension is any documented *implementation-defined* behavior of a *defined name* in this standard that varies from the behavior described in this standard, or a documented consequence of a situation that the standard specifies as undefined, unspecified, or extendable by the implementation. For example, if this standard says that "the results are unspecified," an extension would be to specify the results.

If the correct behavior of a program depends on the results provided by an extension, only implementations with the same extension will execute the program correctly. Note that such a program might be non-conforming. Also, if this standard says that "an implementation may be extended," a conforming, but possibly non-portable, program can be written using an extension.

An implementation can have extensions, provided they do not alter the behavior of conforming code and provided they are not explicitly prohibited by this standard.

The term "extension" refers only to extensions available upon startup. An implementation is free to allow or prohibit redefinition of an extension.

The following list contains specific guidance to implementations concerning certain types of extensions.

Extra return values
An implementation must return exactly the number of return values specified by this standard unless the standard specifically indicates otherwise.

Unsolicited messages

No output can be produced by a function other than that specified in the standard or due to the signaling of *conditions* detected by the function.

Unsolicited output, such as garbage collection notifications and autoload heralds, should not go directly to the *stream* that is the value of a *stream* variable defined in this standard, but can go indirectly to *terminal I/O* by using a *synonym stream* to ***terminal-io***.

Progress reports from such functions as **load** and **compile** are considered solicited, and are not covered by this prohibition.

Implementation of macros and special forms

Macros and *special operators* defined in this standard must not be *functions*.

1.7 Language Subsets

The language described in this standard contains no subsets, though subsets are not forbidden.

For a language to be considered a subset, it must have the property that any valid *program* in that language has equivalent semantics and will run directly (with no extralingual pre-processing, and no special compatibility packages) in any *conforming implementation* of the full language.

1.8 Deprecated Language Features

Deprecated language features are not expected to appear in future Common Lisp tandards, but are required to be implemented for conformance with this standard; see Section 1.5.1.1 (Required Language Features).

Conforming programs can use deprecated features; however, it is considered good programming style to avoid them. It is permissible for the compiler to produce *style warnings* about the use of such features at compile time, but there should be no such warnings at program execution time.

1.8.1 Deprecated Functions

The *functions* in the next figure are deprecated.

```
assoc-if-not   nsubst-if-not       require
count-if-not   nsubstitute-if-not  set
delete-if-not  position-if-not     subst-if-not
find-if-not    provide             substitute-if-not
gentemp        rassoc-if-not
member-if-not  remove-if-not
```

Figure 1-2. Deprecated Functions

1.8.2 Deprecated Argument Conventions

The ability to pass a numeric *argument* to **gensym** has been deprecated.

The `:test-not` *argument* to the *functions* in the next figure are deprecated.

```
adjoin              nset-difference     search
assoc               nset-exclusive-or   set-difference
count               nsublis             set-exclusive-or
delete              nsubst              sublis
delete-duplicates   nsubstitute         subsetp
find                nunion              subst
intersection        position            substitute
member              rassoc              tree-equal
mismatch            remove              union
nintersection       remove-duplicates
```

Figure 1-3. Functions with Deprecated :TEST-NOT Arguments

The use of the situation names **compile**, **load**, and **eval** in **eval-when** is deprecated.

1.8.3 Deprecated Variables

The *variable* ***modules*** is deprecated.

1.8.4 Deprecated Reader Syntax

The #S *reader macro* forces keyword names into the `KEYWORD` package; see Section 2.4.8.13 (Sharpsign S). This feature is deprecated; in the future, keyword names will be taken in the package they are read in, so *symbols* that are actually in the `KEYWORD` package should be used if that is what is desired.

1.9 Symbols in the COMMON-LISP Package

The figures on the next twelve pages contain a complete enumeration of the 978 *external symbols* in the `COMMON-LISP` package.

```
&allow-other-keys            *print-miser-width*
&aux                         *print-pprint-dispatch*
&body                        *print-pretty*
&environment                 *print-radix*
&key                         *print-readably*
&optional                    *print-right-margin*
&rest                        *query-io*
&whole                       *random-state*
*                            *read-base*
**                           *read-default-float-format*
***                          *read-eval*
*break-on-signals*           *read-suppress*
*compile-file-pathname*      *readtable*
*compile-file-truename*      *standard-input*
*compile-print*              *standard-output*
*compile-verbose*            *terminal-io*
*debug-io*                   *trace-output*
*debugger-hook*              +
*default-pathname-defaults*  ++
```

```
*error-output*              +++
*features*                  -
*gensym-counter*            /
*load-pathname*             //
*load-print*                ///
*load-truename*             /=
*load-verbose*              1+
*macroexpand-hook*          1-
*modules*                   <
*package*                   <=
*print-array*               =
*print-base*                >
*print-case*                >=
*print-circle*              abort
*print-escape*              abs
*print-gensym*              acons
*print-length*              acos
*print-level*               acosh
*print-lines*               add-method
```

Figure 1-4. Symbols in the COMMON-LISP package (part one of twelve).

```
adjoin                      atom          boundp
adjust-array                base-char     break
adjustable-array-p          base-string   broadcast-stream
allocate-instance           bignum        broadcast-stream-streams
alpha-char-p                bit           built-in-class
alphanumericp               bit-and       butlast
and                         bit-andc1     byte
append                      bit-andc2     byte-position
apply                       bit-eqv       byte-size
apropos                     bit-ior       caaaar
apropos-list                bit-nand      caaadr
aref                        bit-nor       caaar
arithmetic-error            bit-not       caadar
arithmetic-error-operands   bit-orc1      caaddr
arithmetic-error-operation  bit-orc2      caadr
array                       bit-vector    caar
array-dimension             bit-vector-p  cadaar
array-dimension-limit       bit-xor       cadadr
array-dimensions            block         cadar
array-displacement          boole         caddar
array-element-type          boole-1       cadddr
array-has-fill-pointer-p    boole-2       caddr
array-in-bounds-p           boole-and     cadr
array-rank                  boole-andc1   call-arguments-limit
array-rank-limit            boole-andc2   call-method
array-row-major-index       boole-c1      call-next-method
array-total-size            boole-c2      car
array-total-size-limit      boole-clr     case
arrayp                      boole-eqv     catch
ash                         boole-ior     ccase
asin                        boole-nand    cdaaar
asinh                       boole-nor     cdaadr
assert                      boole-orc1    cdaar
assoc                       boole-orc2    cdadar
assoc-if                    boole-set     cdaddr
assoc-if-not                boole-xor     cdadr
atan                        boolean       cdar
atanh                       both-case-p   cddaar
```

Figure 1-5. Symbols in the COMMON-LISP package (part two of twelve).

```
cddadr              clear-input                 copy-tree
cddar               clear-output                cos
cdddar              close                       cosh
cddddr              clrhash                     count
cdddr               code-char                   count-if
cddr                coerce                      count-if-not
cdr                 compilation-speed           ctypecase
ceiling             compile                     debug
cell-error          compile-file                decf
cell-error-name     compile-file-pathname       declaim
cerror              compiled-function           declaration
change-class        compiled-function-p         declare
char                compiler-macro              decode-float
char-code           compiler-macro-function     decode-universal-time
char-code-limit     complement                  defclass
char-downcase       complex                     defconstant
char-equal          complexp                    defgeneric
char-greaterp       compute-applicable-methods  define-compiler-macro
char-int            compute-restarts            define-condition
char-lessp          concatenate                 define-method-combination
char-name           concatenated-stream         define-modify-macro
char-not-equal      concatenated-stream-streams define-setf-expander
char-not-greaterp   cond                        define-symbol-macro
char-not-lessp      condition                   defmacro
char-upcase         conjugate                   defmethod
char/=              cons                        defpackage
char<               consp                       defparameter
char<=              constantly                  defsetf
char=               constantp                   defstruct
char>               continue                    deftype
char>=              control-error               defun
character           copy-alist                  defvar
characterp          copy-list                   delete
check-type          copy-pprint-dispatch        delete-duplicates
cis                 copy-readtable              delete-file
class               copy-seq                    delete-if
class-name          copy-structure              delete-if-not
class-of            copy-symbol                 delete-package
```

Figure 1-6. Symbols in the COMMON-LISP package (part three of twelve).

```
denominator                   eq
deposit-field                 eql
describe                      equal
describe-object               equalp
destructuring-bind            error
digit-char                    etypecase
digit-char-p                  eval
directory                     eval-when
directory-namestring          evenp
disassemble                   every
division-by-zero              exp
do                            export
do*                           expt
do-all-symbols                extended-char
do-external-symbols           fboundp
do-symbols                    fceiling
documentation                 fdefinition
dolist                        ffloor
dotimes                       fifth
double-float                  file-author
double-float-epsilon          file-error
double-float-negative-epsilon file-error-pathname
dpb                           file-length
dribble                       file-namestring
```

```
dynamic-extent                file-position
ecase                         file-stream
echo-stream                   file-string-length
echo-stream-input-stream      file-write-date
echo-stream-output-stream     fill
ed                            fill-pointer
eighth                        find
elt                           find-all-symbols
encode-universal-time         find-class
end-of-file                   find-if
endp                          find-if-not
enough-namestring             find-method
ensure-directories-exist      find-package
ensure-generic-function       find-restart
```

Figure 1-7. Symbols in the COMMON-LISP package (part four of twelve).

```
find-symbol                         get-internal-run-time
finish-output                       get-macro-character
first                               get-output-stream-string
fixnum                              get-properties
flet                                get-setf-expansion
float                               get-universal-time
float-digits                        getf
float-precision                     gethash
float-radix                         go
float-sign                          graphic-char-p
floating-point-inexact              handler-bind
floating-point-invalid-operation    handler-case
floating-point-overflow             hash-table
floating-point-underflow            hash-table-count
floatp                              hash-table-p
floor                               hash-table-rehash-size
fmakunbound                         hash-table-rehash-threshold
force-output                        hash-table-size
format                              hash-table-test
formatter                           host-namestring
fourth                              identity
fresh-line                          if
fround                              ignorable
ftruncate                           ignore
ftype                               ignore-errors
funcall                             imagpart
function                            import
function-keywords                   in-package
function-lambda-expression          incf
functionp                           initialize-instance
gcd                                 inline
generic-function                    input-stream-p
gensym                              inspect
gentemp                             integer
get                                 integer-decode-float
get-decoded-time                    integer-length
get-dispatch-macro-character        integerp
get-internal-real-time              interactive-stream-p
```

Figure 1-8. Symbols in the COMMON-LISP package (part five of twelve).

```
intern                                lisp-implementation-type
internal-time-units-per-second        lisp-implementation-version
intersection                          list
invalid-method-error                  list*
invoke-debugger                       list-all-packages
invoke-restart                        list-length
invoke-restart-interactively          listen
```

```
isqrt                                 listp
keyword                               load
keywordp                              load-logical-pathname-translations
labels                                load-time-value
lambda                                locally
lambda-list-keywords                  log
lambda-parameters-limit               logand
last                                  logandc1
lcm                                   logandc2
ldb                                   logbitp
ldb-test                              logcount
ldiff                                 logeqv
least-negative-double-float           logical-pathname
least-negative-long-float             logical-pathname-translations
least-negative-normalized-double-float logior
least-negative-normalized-long-float  lognand
least-negative-normalized-short-float lognor
least-negative-normalized-single-float lognot
least-negative-short-float            logorc1
least-negative-single-float           logorc2
least-positive-double-float           logtest
least-positive-long-float             logxor
least-positive-normalized-double-float long-float
least-positive-normalized-long-float  long-float-epsilon
least-positive-normalized-short-float long-float-negative-epsilon
least-positive-normalized-single-float long-site-name
least-positive-short-float            loop
least-positive-single-float           loop-finish
length                                lower-case-p
let                                   machine-instance
let*                                  machine-type
```

Figure 1-9. Symbols in the COMMON-LISP package (part six of twelve).

```
machine-version                 mask-field
macro-function                  max
macroexpand                     member
macroexpand-1                   member-if
macrolet                        member-if-not
make-array                      merge
make-broadcast-stream           merge-pathnames
make-concatenated-stream        method
make-condition                  method-combination
make-dispatch-macro-character   method-combination-error
make-echo-stream                method-qualifiers
make-hash-table                 min
make-instance                   minusp
make-instances-obsolete         mismatch
make-list                       mod
make-load-form                  most-negative-double-float
make-load-form-saving-slots     most-negative-fixnum
make-method                     most-negative-long-float
make-package                    most-negative-short-float
make-pathname                   most-negative-single-float
make-random-state               most-positive-double-float
make-sequence                   most-positive-fixnum
make-string                     most-positive-long-float
make-string-input-stream        most-positive-short-float
make-string-output-stream       most-positive-single-float
make-symbol                     muffle-warning
make-synonym-stream             multiple-value-bind
make-two-way-stream             multiple-value-call
makunbound                      multiple-value-list
map                             multiple-value-prog1
map-into                        multiple-value-setq
```

```
mapc                          multiple-values-limit
mapcan                        name-char
mapcar                        namestring
mapcon                        nbutlast
maphash                       nconc
mapl                          next-method-p
maplist                       nil
```

Figure 1-10. Symbols in the COMMON-LISP package (part seven of twelve).

```
nintersection         package-error
ninth                 package-error-package
no-applicable-method  package-name
no-next-method        package-nicknames
not                   package-shadowing-symbols
notany                package-use-list
notevery              package-used-by-list
notinline             packagep
nreconc               pairlis
nreverse              parse-error
nset-difference       parse-integer
nset-exclusive-or     parse-namestring
nstring-capitalize    pathname
nstring-downcase      pathname-device
nstring-upcase        pathname-directory
nsublis               pathname-host
nsubst                pathname-match-p
nsubst-if             pathname-name
nsubst-if-not         pathname-type
nsubstitute           pathname-version
nsubstitute-if        pathnamep
nsubstitute-if-not    peek-char
nth                   phase
nth-value             pi
nthcdr                plusp
null                  pop
number                position
numberp               position-if
numerator             position-if-not
nunion                pprint
oddp                  pprint-dispatch
open                  pprint-exit-if-list-exhausted
open-stream-p         pprint-fill
optimize              pprint-indent
or                    pprint-linear
otherwise             pprint-logical-block
output-stream-p       pprint-newline
package               pprint-pop
```

Figure 1-11. Symbols in the COMMON-LISP package (part eight of twelve).

```
pprint-tab                 read-char
pprint-tabular             read-char-no-hang
prin1                      read-delimited-list
prin1-to-string            read-from-string
princ                      read-line
princ-to-string            read-preserving-whitespace
print                      read-sequence
print-not-readable         reader-error
print-not-readable-object  readtable
print-object               readtable-case
print-unreadable-object    readtablep
probe-file                 real
proclaim                   realp
prog                       realpart
```

```
prog*                       reduce
prog1                       reinitialize-instance
prog2                       rem
progn                       remf
program-error               remhash
progv                       remove
provide                     remove-duplicates
psetf                       remove-if
psetq                       remove-if-not
push                        remove-method
pushnew                     remprop
quote                       rename-file
random                      rename-package
random-state                replace
random-state-p              require
rassoc                      rest
rassoc-if                   restart
rassoc-if-not               restart-bind
ratio                       restart-case
rational                    restart-name
rationalize                 return
rationalp                   return-from
read                        revappend
read-byte                   reverse
```

Figure 1-12. Symbols in the COMMON-LISP package (part nine of twelve).

```
room                            simple-bit-vector
rotatef                         simple-bit-vector-p
round                           simple-condition
row-major-aref                  simple-condition-format-arguments
rplaca                          simple-condition-format-control
rplacd                          simple-error
safety                          simple-string
satisfies                       simple-string-p
sbit                            simple-type-error
scale-float                     simple-vector
schar                           simple-vector-p
search                          simple-warning
second                          sin
sequence                        single-float
serious-condition               single-float-epsilon
set                             single-float-negative-epsilon
set-difference                  sinh
set-dispatch-macro-character    sixth
set-exclusive-or                sleep
set-macro-character             slot-boundp
set-pprint-dispatch             slot-exists-p
set-syntax-from-char            slot-makunbound
setf                            slot-missing
setq                            slot-unbound
seventh                         slot-value
shadow                          software-type
shadowing-import                software-version
shared-initialize               some
shiftf                          sort
short-float                     space
short-float-epsilon             special
short-float-negative-epsilon    special-operator-p
short-site-name                 speed
signal                          sqrt
signed-byte                     stable-sort
signum                          standard
simple-array                    standard-char
simple-base-string              standard-char-p
```

Figure 1-13. Symbols in the COMMON-LISP package (part ten of twelve).

```
standard-class              sublis
standard-generic-function   subseq
standard-method             subsetp
standard-object             subst
step                        subst-if
storage-condition           subst-if-not
store-value                 substitute
stream                      substitute-if
stream-element-type         substitute-if-not
stream-error                subtypep
stream-error-stream         svref
stream-external-format      sxhash
streamp                     symbol
string                      symbol-function
string-capitalize           symbol-macrolet
string-downcase             symbol-name
string-equal                symbol-package
string-greaterp             symbol-plist
string-left-trim            symbol-value
string-lessp                symbolp
string-not-equal            synonym-stream
string-not-greaterp         synonym-stream-symbol
string-not-lessp            t
string-right-trim           tagbody
string-stream               tailp
string-trim                 tan
string-upcase               tanh
string/=                    tenth
string<                     terpri
string<=                    the
string=                     third
string>                     throw
string>=                    time
stringp                     trace
structure                   translate-logical-pathname
structure-class             translate-pathname
structure-object            tree-equal
style-warning               truename
```

Figure 1-14. Symbols in the COMMON-LISP package (part eleven of twelve).

```
truncate                            values-list
two-way-stream                      variable
two-way-stream-input-stream         vector
two-way-stream-output-stream        vector-pop
type                                vector-push
type-error                          vector-push-extend
type-error-datum                    vectorp
type-error-expected-type            warn
type-of                             warning
typecase                            when
typep                               wild-pathname-p
unbound-slot                        with-accessors
unbound-slot-instance               with-compilation-unit
unbound-variable                    with-condition-restarts
undefined-function                  with-hash-table-iterator
unexport                            with-input-from-string
unintern                            with-open-file
union                               with-open-stream
unless                              with-output-to-string
unread-char                         with-package-iterator
unsigned-byte                       with-simple-restart
untrace                             with-slots
```

```
unuse-package                         with-standard-io-syntax
unwind-protect                        write
update-instance-for-different-class   write-byte
update-instance-for-redefined-class   write-char
upgraded-array-element-type           write-line
upgraded-complex-part-type            write-sequence
upper-case-p                          write-string
use-package                           write-to-string
use-value                             y-or-n-p
user-homedir-pathname                 yes-or-no-p
values                                zerop
```

2. Syntax

2.1 Character Syntax

The *Lisp reader* takes *characters* from a *stream*, interprets them as a printed representation of an *object*, constructs that *object*, and returns it.

The syntax described by this chapter is called the *standard syntax*. Operations are provided by Common Lisp so that various aspects of the syntax information represented by a *readtable* can be modified under program control; see Section 23 (Reader). Except as explicitly stated otherwise, the syntax used throughout this document is *standard syntax*.

2.1.1 Readtables

Syntax information for use by the *Lisp reader* is embodied in an *object* called a *readtable*. Among other things, the *readtable* contains the association between *characters* and *syntax types*.

The next figure lists some *defined names* that are applicable to *readtables*.

```
*readtable*                    readtable-case
copy-readtable                 readtablep
get-dispatch-macro-character   set-dispatch-macro-character
get-macro-character            set-macro-character
make-dispatch-macro-character  set-syntax-from-char
```

Figure 2-1. Readtable defined names

2.1.1.1 The Current Readtable

Several *readtables* describing different syntaxes can exist, but at any given time only one, called the *current readtable*, affects the way in which *expressions*[2] are parsed into *objects* by the *Lisp reader*. The *current readtable* in a given *dynamic environment* is the *value* of ***readtable*** in that *environment*. To make a different *readtable* become the *current readtable*, ***readtable*** can be *assigned* or *bound*.

2.1.1.2 The Standard Readtable

The *standard readtable* conforms to *standard syntax*. The consequences are undefined if an attempt is made to modify the *standard readtable*. To achieve the effect of altering or extending *standard syntax*, a copy of the *standard readtable* can be created; see the *function* **copy-readtable**.

The *readtable case* of the *standard readtable* is `:upcase`.

2.1.1.3 The Initial Readtable

The *initial readtable* is the *readtable* that is the *current readtable* at the time when the *Lisp image* starts. At that time, it conforms to *standard syntax*. The *initial readtable* is *distinct* from the *standard readtable*. It is permissible for a *conforming program* to modify the *initial readtable*.

2.1.2 Variables that affect the Lisp Reader

The *Lisp reader* is influenced not only by the *current readtable*, but also by various *dynamic variables*. The next figure lists the *variables* that influence the behavior of the *Lisp reader*.

```
*package*    *read-default-float-format*  *readtable*
*read-base*  *read-suppress*
```

Figure 2-2. Variables that influence the Lisp reader.

2.1.3 Standard Characters

All *implementations* must support a *character repertoire* called **standard-char**; *characters* that are members of that *repertoire* are called *standard characters*.

The **standard-char** *repertoire* consists of the *non-graphic character newline*, the *graphic character space*, and the following additional ninety-four *graphic characters* or their equivalents:

Graphic ID	Glyph	Description	Graphic ID	Glyph	Description
LA01	a	small a	LN01	n	small n
LA02	A	capital A	LN02	N	capital N
LB01	b	small b	LO01	o	small o
LB02	B	capital B	LO02	O	capital O
LC01	c	small c	LP01	p	small p
LC02	C	capital C	LP02	P	capital P
LD01	d	small d	LQ01	q	small q
LD02	D	capital D	LQ02	Q	capital Q
LE01	e	small e	LR01	r	small r
LE02	E	capital E	LR02	R	capital R
LF01	f	small f	LS01	s	small s
LF02	F	capital F	LS02	S	capital S
LG01	g	small g	LT01	t	small t
LG02	G	capital G	LT02	T	capital T
LH01	h	small h	LU01	u	small u
LH02	H	capital H	LU02	U	capital U
LI01	i	small i	LV01	v	small v
LI02	I	capital I	LV02	V	capital V
LJ01	j	small j	LW01	w	small w
LJ02	J	capital J	LW02	W	capital W
LK01	k	small k	LX01	x	small x
LK02	K	capital K	LX02	X	capital X
LL01	l	small l	LY01	y	small y
LL02	L	capital L	LY02	Y	capital Y
LM01	m	small m	LZ01	z	small z
LM02	M	capital M	LZ02	Z	capital Z

Figure 2-3. Standard Character Subrepertoire (Part 1 of 3: Latin Characters)

Graphic ID	Glyph	Description	Graphic ID	Glyph	Description
ND01	1	digit 1	ND06	6	digit 6
ND02	2	digit 2	ND07	7	digit 7
ND03	3	digit 3	ND08	8	digit 8
ND04	4	digit 4	ND09	9	digit 9
ND05	5	digit 5	ND10	0	digit 0

Figure 2-4. Standard Character Subrepertoire (Part 2 of 3: Numeric Characters)

Graphic ID	Glyph	Description
SP02	!	exclamation mark
SC03	$	dollar sign
SP04	"	quotation mark, or double quote
SP05	’	apostrophe, or [single] quote
SP06	(	left parenthesis, or open parenthesis
SP07	)	right parenthesis, or close parenthesis
SP08	,	comma
SP09	_	low line, or underscore
SP10	-	hyphen, or minus [sign]
SP11	.	full stop, period, or dot
SP12	/	solidus, or slash
SP13	:	colon
SP14	;	semicolon
SP15	?	question mark
SA01	+	plus [sign]
SA03	<	less-than [sign]
SA04	=	equals [sign]
SA05	>	greater-than [sign]
SM01	#	number sign, or sharp[sign]
SM02	%	percent [sign]
SM03	&	ampersand
SM04	*	asterisk, or star
SM05	@	commercial at, or at-sign
SM06	[	left [square] bracket
SM07	\	reverse solidus, or backslash
SM08	]	right [square] bracket
SM11	{	left curly bracket, or left brace
SM13	|	vertical bar
SM14	}	right curly bracket, or right brace
SD13	‘	grave accent, or backquote
SD15	^	circumflex accent
SD19	~	tilde

Figure 2-5. Standard Character Subrepertoire (Part 3 of 3: Special Characters)

The graphic IDs are not used within Common Lisp, but are provided for cross reference purposes with ISO 6937/2. Note that the first letter of the graphic ID categorizes the character as follows: L---Latin, N---Numeric, S---Special.

2.1.4 Character Syntax Types

The *Lisp reader* constructs an *object* from the input text by interpreting each *character* according to its *syntax type*. The *Lisp reader* cannot accept as input everything that the *Lisp printer* produces, and the *Lisp reader* has features that are not used by the *Lisp printer*. The *Lisp reader* can be used as a lexical analyzer for a more general user-written parser.

When the *Lisp reader* is invoked, it reads a single character from the *input stream* and dispatches according to the *syntax type* of that *character*. Every *character* that can appear in the *input stream* is of one of the *syntax types* shown in Figure 2-6.

constituent	macro character	single escape
invalid	multiple escape	whitespace[2]

Figure 2-6. Possible Character Syntax Types

The *syntax type* of a *character* in a *readtable* determines how that character is interpreted by the *Lisp reader* while that *readtable* is the *current readtable*. At any given time, every character has exactly one *syntax type*.

Figure 2-7 lists the *syntax type* of each *character* in *standard syntax*.

character	syntax type	character	syntax type
Backspace	constituent	0--9	constituent
Tab	whitespace[2]	:	constituent
Newline	whitespace[2]	;	terminating macro char
Linefeed	whitespace[2]	<	constituent
Page	whitespace[2]	=	constituent
Return	whitespace[2]	>	constituent
Space	whitespace[2]	?	constituent*
!	constituent*	@	constituent
"	terminating macro char	A--Z	constituent
#	non-terminating macro char	[	constituent*
$	constituent	\	single escape
%	constituent	]	constituent*
&	constituent	^	constituent
’	terminating macro char	_	constituent
(	terminating macro char	‘	terminating macro char
)	terminating macro char	a--z	constituent
*	constituent	{	constituent*
+	constituent	|	multiple escape
,	terminating macro char	}	constituent*
-	constituent	~	constituent
.	constituent	Rubout	constituent
/	constituent		

Figure 2-7. Character Syntax Types in Standard Syntax

The characters marked with an asterisk (*) are initially *constituents*, but they are not used in any standard Common Lisp notations. These characters are explicitly reserved to the *programmer*. ~ is not used in Common Lisp, and reserved to implementors. $ and % are *alphabetic*[2] *characters*, but are not used in the names of any standard Common Lisp *defined names*.

Whitespace[2] characters serve as separators but are otherwise ignored. *Constituent* and *escape characters* are accumulated to make a *token*, which is then interpreted as a *number* or *symbol*. *Macro characters* trigger the invocation of *functions* (possibly user-supplied) that can perform arbitrary parsing actions. *Macro characters* are divided into two kinds, *terminating* and *non-terminating*, depending on whether or not they terminate a *token*. The following are descriptions of each kind of *syntax type*.

2.1.4.1 Constituent Characters

Constituent characters are used in *tokens*. A *token* is a representation of a *number* or a *symbol*. Examples of *constituent characters* are letters and digits.

Letters in symbol names are sometimes converted to letters in the opposite *case* when the name is read; see Section 23.1.2 (Effect of Readtable Case on the Lisp Reader). *Case* conversion can be suppressed by the use of *single escape* or *multiple escape* characters.

2.1.4.2 Constituent Traits

Every *character* has one or more *constituent traits* that define how the *character* is to be interpreted by the *Lisp reader* when the *character* is a *constituent character*. These *constituent traits* are *alphabetic*[2], digit, *package marker*, plus sign, minus sign, dot, decimal point, *ratio marker*, *exponent marker*, and *invalid*. Figure 2-8 shows the *constituent traits* of the *standard characters* and of certain *semi-standard characters*; no mechanism is provided for changing the *constituent trait* of a *character*. Any *character* with the alphadigit *constituent trait* in that figure is a digit if the *current input base* is greater than that character’s digit value, otherwise the *character* is *alphabetic*[2]. Any *character* quoted by a *single escape* is treated as an *alphabetic*[2] constituent, regardless of its normal syntax.

constituent character	traits	constituent character	traits
Backspace	invalid	{	alphabetic[2]
Tab	invalid*	}	alphabetic[2]
Newline	invalid*	+	alphabetic[2], plus sign
Linefeed	invalid*	-	alphabetic[2], minus sign
Page	invalid*	.	alphabetic[2], dot, decimal point
Return	invalid*	/	alphabetic[2], ratio marker
Space	invalid*	A, a	alphadigit
!	alphabetic[2]	B, b	alphadigit
"	alphabetic[2]*	C, c	alphadigit
#	alphabetic[2]*	D, d	alphadigit, double-float exponent marker
$	alphabetic[2]	E, e	alphadigit, float exponent marker
%	alphabetic[2]	F, f	alphadigit, single-float exponent marker
&	alphabetic[2]	G, g	alphadigit
'	alphabetic[2]*	H, h	alphadigit
(	alphabetic[2]*	I, i	alphadigit
)	alphabetic[2]*	J, j	alphadigit
*	alphabetic[2]	K, k	alphadigit
,	alphabetic[2]*	L, l	alphadigit, long-float exponent marker
0-9	alphadigit	M, m	alphadigit
:	package marker	N, n	alphadigit
;	alphabetic[2]*	O, o	alphadigit
<	alphabetic[2]	P, p	alphadigit
=	alphabetic[2]	Q, q	alphadigit
>	alphabetic[2]	R, r	alphadigit
?	alphabetic[2]	S, s	alphadigit, short-float exponent marker
@	alphabetic[2]	T, t	alphadigit
[	alphabetic[2]	U, u	alphadigit
\	alphabetic[2]*	V, v	alphadigit
]	alphabetic[2]	W, w	alphadigit
^	alphabetic[2]	X, x	alphadigit
_	alphabetic[2]	Y, y	alphadigit
`	alphabetic[2]*	Z, z	alphadigit
|	alphabetic[2]*	Rubout	invalid
~	alphabetic[2]		

Figure 2-8. Constituent Traits of Standard Characters and Semi-Standard Characters

The interpretations in this table apply only to *characters* whose *syntax type* is *constituent*. Entries marked with an asterisk (*) are normally *shadowed*[2] because the indicated *characters* are of *syntax type whitespace*[2], *macro character*, *single escape*, or *multiple escape*; these *constituent traits* apply to them only if their *syntax types* are changed to *constituent*.

2.1.4.3 Invalid Characters

Characters with the *constituent trait invalid* cannot ever appear in a *token* except under the control of a *single escape character*. If an *invalid character* is encountered while an *object* is being read, an error of *type* **reader-error** is signaled. If an *invalid character* is preceded by a *single escape character*, it is treated as an *alphabetic*[2] *constituent* instead.

2.1.4.4 Macro Characters

When the *Lisp reader* encounters a *macro character* on an *input stream*, special parsing of subsequent *characters* on the *input stream* is performed.

A *macro character* has an associated *function* called a *reader macro function* that implements its specialized parsing behavior. An association of this kind can be established or modified under control of a *conforming program* by using the *functions* **set-macro-character** and **set-dispatch-macro-character**.

Upon encountering a *macro character*, the *Lisp reader* calls its *reader macro function*, which parses one specially formatted object from the *input stream*. The *function* either returns the parsed *object*, or else it returns no *values* to indicate that the characters scanned by the *function* are being ignored (e.g., in the case of a comment). Examples of *macro characters* are *backquote*, *single-quote*, *left-parenthesis*, and *right-parenthesis*.

A *macro character* is either *terminating* or *non-terminating*. The difference between *terminating* and *non-terminating macro characters* lies in what happens when such characters occur in the middle of a *token*. If a *non-terminating macro character* occurs in the middle of a *token*, the *function* associated with the *non-terminating macro character* is not called, and the *non-terminating macro character* does not terminate the *token*'s name; it becomes part of the name as if the *macro character* were really a constituent character. A *terminating macro character* terminates any *token*, and its associated *reader macro function* is called no matter where the *character* appears. The only *non-terminating macro character* in *standard syntax* is *sharpsign*.

If a *character* is a *dispatching macro character* C1, its *reader macro function* is a *function* supplied by the *implementation*. This *function* reads decimal *digit characters* until a non-*digit* C2 is read. If any *digits* were read, they are converted into a corresponding *integer* infix parameter P; otherwise, the infix parameter P is **nil**. The terminating non-*digit* C2 is a *character* (sometimes called a "sub-character" to emphasize its subordinate role in the dispatching) that is looked up in the dispatch table associated with the *dispatching macro character* C1. The *reader macro function* associated with the sub-character C2 is invoked with three arguments: the *stream*, the sub-character C2, and the infix parameter P. For more information about dispatch characters, see the *function* **set-dispatch-macro-character**.

For information about the *macro characters* that are available in *standard syntax*, see Section 2.4 (Standard Macro Characters).

2.1.4.5 Multiple Escape Characters

A pair of *multiple escape characters* is used to indicate that an enclosed sequence of characters, including possible *macro characters* and *whitespace*[2] *characters*, are to be treated as *alphabetic*[2] *characters* with *case* preserved. Any *single escape* and *multiple escape characters* that are to appear in the sequence must be preceded by a *single escape character*.

Vertical-bar is a *multiple escape character* in *standard syntax*.

2.1.4.5.1 Examples of Multiple Escape Characters

```
;; The following examples assume the readtable case of *readtable*
;; and *print-case* are both :upcase.
(eq 'abc 'ABC) =>  true
(eq 'abc '|ABC|) =>  true
(eq 'abc 'a|B|c) =>  true
(eq 'abc '|abc|) =>  false
```

2.1.4.6 Single Escape Character

A *single escape* is used to indicate that the next *character* is to be treated as an *alphabetic*[2] *character* with its *case* preserved, no matter what the *character* is or which *constituent traits* it has.

Backslash is a *single escape character* in *standard syntax*.

2.1.4.6.1 Examples of Single Escape Characters

```
;; The following examples assume the readtable case of *readtable*
;; and *print-case* are both :upcase.
(eq 'abc '\A\B\C) =>  true
(eq 'abc 'a\Bc) =>  true
(eq 'abc '\ABC) =>  true
(eq 'abc '\abc) =>  false
```

2.1.4.7 Whitespace Characters

Whitespace[2] *characters* are used to separate *tokens*.

Space and *newline* are *whitespace*[2] *characters* in *standard syntax*.

2.1.4.7.1 Examples of Whitespace Characters

```
(length '(this-that)) =>  1
(length '(this - that)) =>  3
(length '(a
          b)) =>  2
(+ 34) =>  34
(+ 3 4) =>  7
```

2.2 Reader Algorithm

This section describes the algorithm used by the *Lisp reader* to parse *objects* from an *input character stream*, including how the *Lisp reader* processes *macro characters*.

When dealing with *tokens*, the reader's basic function is to distinguish representations of *symbols* from those of *numbers*. When a *token* is accumulated, it is assumed to represent a *number* if it satisfies the syntax for numbers listed in Figure 2-9. If it does not represent a *number*, it is then assumed to be a *potential number* if it satisfies the rules governing the syntax for a *potential number*. If a valid *token* is neither a representation of a *number* nor a *potential number*, it represents a *symbol*.

The algorithm performed by the *Lisp reader* is as follows:

1. If at end of file, end-of-file processing is performed as specified in **read**. Otherwise, one *character*, *x*, is read from the *input stream*, and dispatched according to the *syntax type* of *x* to one of steps 2 to 7.
2. If *x* is an *invalid character*, an error of *type* **reader-error** is signaled.
3. If *x* is a *whitespace*[2] *character*, then it is discarded and step 1 is re-entered.
4. If *x* is a *terminating* or *non-terminating macro character* then its associated *reader macro function* is called with two *arguments*, the *input stream* and *x*.

   The *reader macro function* may read *characters* from the *input stream*; if it does, it will see those *characters* following the *macro character*. The *Lisp reader* may be invoked recursively from the *reader macro function*.

   The *reader macro function* must not have any side effects other than on the *input stream*; because of backtracking and restarting of the **read** operation, front ends to the *Lisp reader* (e.g., "editors" and "rubout handlers") may cause the *reader macro function* to be called repeatedly during the reading of a single *expression* in which *x* only appears once.

The *reader macro function* may return zero values or one value. If one value is returned, then that value is returned as the result of the read operation; the algorithm is done. If zero values are returned, then step 1 is re-entered.

5. If *x* is a *single escape character* then the next *character*, *y*, is read, or an error of *type* **end-of-file** is signaled if at the end of file. *y* is treated as if it is a *constituent* whose only *constituent trait* is *alphabetic*[2]. *y* is used to begin a *token*, and step 8 is entered.
6. If *x* is a *multiple escape character* then a *token* (initially containing no *characters*) is begun and step 9 is entered.
7. If *x* is a *constituent character*, then it begins a *token*. After the *token* is read in, it will be interpreted either as a Lisp *object* or as being of invalid syntax. If the *token* represents an *object*, that *object* is returned as the result of the read operation. If the *token* is of invalid syntax, an error is signaled. If *x* is a *character* with *case*, it might be replaced with the corresponding *character* of the opposite *case*, depending on the *readtable case* of the *current readtable*, as outlined in Section 23.1.2 (Effect of Readtable Case on the Lisp Reader). *X* is used to begin a *token*, and step 8 is entered.
8. At this point a *token* is being accumulated, and an even number of *multiple escape characters* have been encountered. If at end of file, step 10 is entered. Otherwise, a *character*, *y*, is read, and one of the following actions is performed according to its *syntax type*:

* If *y* is a *constituent* or *non-terminating macro character*:
  -- If *y* is a *character* with *case*, it might be replaced with the corresponding *character* of the opposite *case*, depending on the *readtable case* of the *current readtable*, as outlined in Section 23.1.2 (Effect of Readtable Case on the Lisp Reader).
  -- *Y* is appended to the *token* being built.
  -- Step 8 is repeated.
* If *y* is a *single escape character*, then the next *character*, *z*, is read, or an error of *type* **end-of-file** is signaled if at end of file. *Z* is treated as if it is a *constituent* whose only *constituent trait* is *alphabetic*[2]. *Z* is appended to the *token* being built, and step 8 is repeated.
* If *y* is a *multiple escape character*, then step 9 is entered.
* If *y* is an *invalid character*, an error of *type* **reader-error** is signaled.
* If *y* is a *terminating macro character*, then it terminates the *token*. First the *character y* is unread (see **unread-char**), and then step 10 is entered.
* If *y* is a *whitespace*[2] *character*, then it terminates the *token*. First the *character y* is unread if appropriate (see **read-preserving-whitespace**), and then step 10 is entered.

9. At this point a *token* is being accumulated, and an odd number of *multiple escape characters* have been encountered. If at end of file, an error of *type* **end-of-file** is signaled. Otherwise, a *character*, *y*, is read, and one of the following actions is performed according to its *syntax type*:

* If *y* is a *constituent*, macro, or *whitespace*[2] *character*, *y* is treated as a *constituent* whose only *constituent trait* is *alphabetic*[2]. *Y* is appended to the *token* being built, and step 9 is repeated.
* If *y* is a *single escape character*, then the next *character*, *z*, is read, or an error of *type* **end-of-file** is signaled if at end of file. *Z* is treated as a *constituent* whose only *constituent trait* is *alphabetic*[2]. *Z* is appended to the *token* being built, and step 9 is repeated.
* If *y* is a *multiple escape character*, then step 8 is entered.
* If *y* is an *invalid character*, an error of *type* **reader-error** is signaled.

10. An entire *token* has been accumulated. The *object* represented by the *token* is returned as the result of the read operation, or an error of *type* **reader-error** is signaled if the *token* is not of valid syntax.

2.3 Interpretation of Tokens

2.3.1 Numbers as Tokens

When a *token* is read, it is interpreted as a *number* or *symbol*. The *token* is interpreted as a *number* if it satisfies the syntax for numbers specified in the next figure.

```
numeric-token  ::=  integer |
                                ratio   |
                                float
integer        ::=  [sign]
                                decimal-digit+
                                decimal-point |
                                [sign]
                                digit+
ratio          ::=  [sign]
                                {digit}+
                                slash
                                {digit}+
float          ::=  [sign]
                                {decimal-digit}*
                                decimal-point
                                {decimal-digit}+
                                [exponent]
                     |
                                [sign]
                                {decimal-digit}+
                                [decimal-point
                                        {decimal-digit}*]
                                exponent
exponent       ::=  exponent-marker
                                [sign]
                                {digit}+

sign---a sign.
slash---a slash
decimal-point---a dot.
exponent-marker---an exponent marker.
decimal-digit---a digit in radix 10.
digit---a digit in the current input radix.
```

Figure 2-9. Syntax for Numeric Tokens

2.3.1.1 Potential Numbers as Tokens

To allow implementors and future Common Lisp standards to extend the syntax of numbers, a syntax for *potential numbers* is defined that is more general than the syntax for numbers. A *token* is a *potential number* if it satisfies all of the following requirements:

1. The *token* consists entirely of *digits*, *signs*, *ratio markers*, decimal points (`.`), extension characters (^ or _), and number markers. A number marker is a letter. Whether a letter may be treated as a number marker depends on context, but no letter that is adjacent to another letter may ever be treated as a number marker. *Exponent markers* are number markers.
2. The *token* contains at least one digit. Letters may be considered to be digits, depending on the *current input base*, but only in *tokens* containing no decimal points.
3. The *token* begins with a *digit*, *sign*, decimal point, or extension character, but not a *package marker*. The syntax involving a leading *package marker* followed by a *potential number* is not well-defined. The consequences of the use of notation such as `:1`, `:1/2`, and `:2^3` in a position where an expression appropriate for **read** is expected are unspecified.
4. The *token* does not end with a sign.

If a *potential number* has number syntax, a *number* of the appropriate type is constructed and returned, if the *number* is representable in an implementation. A *number* will not be representable in an implementation if it is outside the boundaries set by the *implementation-dependent* constants for *numbers*. For example, specifying too large or too small an exponent for a *float* may make the *number* impossible to represent in the implementation. A *ratio* with denominator zero (such as `-35/000`) is not represented in any implementation. When a *token* with the syntax of a number cannot be converted to an internal *number*, an error of *type* **reader-error** is signaled. An error

must not be signaled for specifying too many significant digits for a *float*; a truncated or rounded value should be produced.

If there is an ambiguity as to whether a letter should be treated as a digit or as a number marker, the letter is treated as a digit.

2.3.1.1.1 Escape Characters and Potential Numbers

A *potential number* cannot contain any *escape characters*. An *escape character* robs the following *character* of all syntactic qualities, forcing it to be strictly *alphabetic*[2] and therefore unsuitable for use in a *potential number*. For example, all of the following representations are interpreted as *symbols*, not *numbers*:

```
\256   25\64   1.0\E6   |100|   3\.14159   |3/4|   3\/4   5||
```

In each case, removing the *escape character* (or *characters*) would cause the token to be a *potential number*.

2.3.1.1.2 Examples of Potential Numbers

As examples, the *tokens* in the next figure are *potential numbers*, but they are not actually numbers, and so are reserved *tokens*; a *conforming implementation* is permitted, but not required, to define their meaning.

```
1b5000                      777777q               1.7J  -3/4+6.7J  12/25/83
27^19                       3^4/5                 6//7  3.1.2.6    ^-43^
3.141_592_653_589_793_238_4  -3.7+2.6i-6.17j+19.6k
```

Figure 2-10. Examples of reserved tokens

The *tokens* in the next figure are not *potential numbers*; they are always treated as *symbols*:

```
/     /5     +  1+  1-
foo+  ab.cd  _  ^   ^/-
```

Figure 2-11. Examples of symbols

The *tokens* in the next figure are *potential numbers* if the *current input base* is `16`, but they are always treated as *symbols* if the *current input base* is `10`.

```
bad-face  25-dec-83  a/b  fad_cafe  f^
```

Figure 2-12. Examples of symbols or potential numbers

2.3.2 Constructing Numbers from Tokens

A *real* is constructed directly from a corresponding numeric *token*; see Figure 2-9.

A *complex* is notated as a `#C` (or `#c`) followed by a *list* of two *reals*; see Section 2.4.8.11 (Sharpsign C).

The *reader macros* `#B`, `#O`, `#X`, and `#R` may also be useful in controlling the input *radix* in which *rationals* are parsed; see Section 2.4.8.7 (Sharpsign B), Section 2.4.8.8 (Sharpsign O), Section 2.4.8.9 (Sharpsign X), and Section 2.4.8.10 (Sharpsign R).

This section summarizes the full syntax for *numbers*.

2.3.2.1 Syntax of a Rational

2.3.2.1.1 Syntax of an Integer

Integers can be written as a sequence of *digits*, optionally preceded by a *sign* and optionally followed by a decimal point; see Figure 2-9. When a decimal point is used, the *digits* are taken to be in *radix* `10`; when no decimal point is used, the *digits* are taken to be in radix given by the *current input base*.

For information on how *integers* are printed, see Section 22.1.3.1.1 (Printing Integers).

2.3.2.1.2 Syntax of a Ratio

Ratios can be written as an optional *sign* followed by two non-empty sequences of *digits* separated by a *slash*; see Figure 2-9. The second sequence may not consist entirely of zeros. Examples of *ratios* are in the next figure.

```
2/3                 ;This is in canonical form
4/6                 ;A non-canonical form for 2/3
-17/23              ;A ratio preceded by a sign
-30517578125/32768  ;This is (-5/2)^15
10/5                ;The canonical form for this is 2
#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
```

Figure 2-13. Examples of Ratios

For information on how *ratios* are printed, see Section 22.1.3.1.2 (Printing Ratios).

2.3.2.2 Syntax of a Float

Floats can be 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. If no exponent specifier is present, or if the *exponent marker* `e` (or `E`) is used, then the format specified by ***read-default-float-format*** is used. See Figure 2-9.

An implementation may provide one or more kinds of *float* that collectively make up the *type* **float**. The letters `s`, `f`, `d`, and `l` (or their respective uppercase equivalents) explicitly specify the use of the *types* **short-float**, **single-float**, **double-float**, and **long-float**, respectively.

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.

The next figure contains examples of notations for *floats*:

```
0.0       ;Floating-point zero in default format
0E0       ;As input, this is also floating-point zero in default format.
          ;As output, this would appear as 0.0.
0e0       ;As input, this is also floating-point zero in default format.
          ;As output, this would appear as 0.0.
-.0       ;As input, this might be a zero or a minus zero,
          ; depending on whether the implementation supports
          ; a distinct minus zero.
          ;As output, 0.0 is zero and -0.0 is minus zero.
0.        ;On input, the integer zero---not a floating-point number!
```

```
          ;Whether this appears as 0 or 0. on output depends
          ;on the value of *print-radix*.
0.0s0     ;A floating-point zero in short format
0s0       ;As input, this is a floating-point zero in short format.
          ;As output, such a zero would appear as 0.0s0
          ; (or as 0.0 if short-float was the default format).
6.02E+23  ;Avogadro's number, in default format
602E+21   ;Also Avogadro's number, in default format
```

Figure 2-14. Examples of Floating-point numbers

For information on how *floats* are printed, see Section 22.1.3.1.3 (Printing Floats).

2.3.2.3 Syntax of a Complex

A *complex* has a Cartesian structure, with a real part and an imaginary part each of which is a *real*. The parts of a *complex* are not necessarily *floats* but both parts must be of the same *type*: either both are *rationals*, or both are of the same *float subtype*. When constructing a *complex*, if the specified parts are not the same *type*, the parts are converted to be the same *type* internally (i.e., the *rational* part is converted to a *float*). An *object* of type `(complex rational)` is converted internally and represented thereafter as a *rational* if its imaginary part is an *integer* whose value is 0.

For further information, see Section 2.4.8.11 (Sharpsign C) and Section 22.1.3.1.4 (Printing Complexes).

2.3.3 The Consing Dot

If a *token* consists solely of dots (with no escape characters), then an error of *type* **reader-error** is signaled, except in one circumstance: if the *token* is a single *dot* and appears in a situation where *dotted pair* notation permits a *dot*, then it is accepted as part of such syntax and no error is signaled. See Section 2.4.1 (Left-Parenthesis).

2.3.4 Symbols as Tokens

Any *token* that is not a *potential number*, does not contain a *package marker*, and does not consist entirely of dots will always be interpreted as a *symbol*. Any *token* that is a *potential number* but does not fit the number syntax is a reserved *token* and has an *implementation-dependent* interpretation. In all other cases, the *token* is construed to be the name of a *symbol*.

Examples of the printed representation of *symbols* are in the next figure. For presentational simplicity, these examples assume that the *readtable case* of the *current readtable* is `:upcase`.

```
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 hyphen 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.
file.rel.43     This symbol has periods in its name.
\(              The symbol whose name is (.
\+1             The symbol whose name is +1.
+\1             Also the symbol whose name is +1.
\frobboz        The symbol whose name is fROBBOZ.
3.14159265\s0   The symbol whose name is 3.14159265s0.
3.14159265\S0   A different symbol, whose name is 3.14159265S0.
3.14159265s0    A possible short float approximation to <PI>.
```

Figure 2-15. Examples of the printed representation of symbols (Part 1 of 2)

```
APL\\360            The symbol whose name is APL\360.
apl\\360            Also the symbol whose name is APL\360.
\(b^2\)\-\4*a*c     The name is (B^2) - 4*A*C.
                    Parentheses and two spaces in it.
\(\b^2\)\-\4*\a*\c  The name is (b^2) - 4*a*c.
                    Letters explicitly lowercase.
|"|                 The same as writing \".
|(b^2) - 4*a*c|     The name is (b^2) - 4*a*c.
|frobboz|           The name is frobboz, not FROBBOZ.
|APL\360|           The name is APL360.
|APL\\360|          The name is APL\360.
|apl\\360|          The name is apl\360.
|\|\||              Same as \|\| ---the name is ||.
|(B^2) - 4*A*C|     The name is (B^2) - 4*A*C.
                    Parentheses and two spaces in it.
|(b^2) - 4*a*c|     The name is (b^2) - 4*a*c.
```

Figure 2-16. Examples of the printed representation of symbols (Part 2 of 2)

In the process of parsing a *symbol*, it is *implementation-dependent* which *implementation-defined attributes* are removed from the *characters* forming a *token* that represents a *symbol*.

When parsing the syntax for a *symbol*, the *Lisp reader* looks up the *name* of that *symbol* 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 and is placed in the *current package* as an *internal symbol*. The *current package* becomes the owner (*home package*) of the *symbol*, and 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 be found and returned.

2.3.5 Valid Patterns for Tokens

The valid patterns for *tokens* are summarized in the next figure.

```
nnnnn              a number
xxxxx              a symbol in the current package
:xxxxx             a symbol in the the KEYWORD package
ppppp:xxxxx        an external symbol in the ppppp package
ppppp::xxxxx       a (possibly internal) symbol in the ppppp package
:nnnnn             undefined
ppppp:nnnnn        undefined
ppppp::nnnnn       undefined
::aaaaa            undefined
aaaaa:             undefined
aaaaa:aaaaa:aaaaa  undefined
```

Figure 2-17. Valid patterns for tokens

Note that *nnnnn* has number syntax, neither *xxxxx* nor *ppppp* has number syntax, and *aaaaa* has any syntax.

A summary of rules concerning *package markers* follows. In each case, examples are offered to illustrate the case; for presentational simplicity, the examples assume that the *readtable case* of the *current readtable* is `:upcase`.

1. If there is a single *package marker*, and it occurs at the beginning of the *token*, then the *token* is interpreted as a *symbol* in the KEYWORD package. It also sets the **symbol-value** of the newly-created *symbol* to that same *symbol* so that the *symbol* will self-evaluate.

For example, `:bar`, when read, interns BAR as an *external symbol* in the KEYWORD package.

2. If there is a single *package marker* not at the beginning or end of the *token*, then it divides the *token* into two parts. The first part specifies a *package*; the second part is the name of an *external symbol* available in that package.

For example, `foo:bar`, when read, looks up BAR among the *external symbols* of the *package* named FOO.

3. If there are two adjacent *package markers* not at the beginning or end of the *token*, then they divide the *token* into two parts. The first part specifies a *package*; the second part is the name of a *symbol* within that *package* (possibly an *internal symbol*).

For example, `foo::bar`, when read, interns BAR in the *package* named FOO.

4. If the *token* contains no *package markers*, and does not have *potential number* syntax, then the entire *token* is the name of the *symbol*. The *symbol* is looked up in the *current package*.

For example, `bar`, when read, interns BAR in the *current package*.

5. The consequences are unspecified if any other pattern of *package markers* in a *token* is used. All other uses of *package markers* within names of *symbols* are not defined by this standard but are reserved for *implementation-dependent* use.

For example, assuming the *readtable case* of the *current readtable* is `:upcase`, `editor:buffer` refers to the *external symbol* named BUFFER present 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 present in `editor`, or if BUFFER is not exported by `editor`, the reader signals a correctable error. If `editor::buffer` is seen, the effect is exactly the same as reading `buffer` with the EDITOR package being the *current package*.

2.3.6 Package System Consistency Rules

The following rules apply to the package system as long as the *value* of ***package*** is not changed:

Read-read consistency
Reading the same *symbol name* always results in the *same symbol*.

Print-read consistency
An *interned symbol* always prints as a sequence of characters that, when read back in, yields the *same symbol*.

For information about how the *Lisp printer* treats *symbols*, see Section 22.1.3.3 (Printing Symbols).

Print-print consistency
If two interned *symbols* are not the *same*, then their printed representations will be different sequences of characters.

These rules are true regardless of any implicit interning. 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. If the *value* of ***package*** is changed and then changed back to the previous value, consistency is maintained. The rules can be violated by changing the *value* of ***package***, forcing a change to *symbols* or to *packages* or to both by continuing from an error, or calling one of the following *functions*: **unintern**, **unexport**, **shadow**, **shadowing-import**, or **unuse-package**.

An inconsistency only applies if one of the restrictions is violated between two of the named *symbols*. **shadow**, **unexport**, **unintern**, and **shadowing-import** can only affect the consistency of *symbols* with the same *names* (under **string=**) as the ones supplied as arguments.

2.4 Standard Macro Characters

If the reader encounters a *macro character*, then its associated *reader macro function* is invoked and may produce an *object* to be returned. This *function* may read the *characters* following the *macro character* in the *stream* in any syntax and return the *object* represented by that syntax.

Any *character* can be made to be a *macro character*. The *macro characters* defined initially in a *conforming implementation* include the following:

2.4.1 Left-Parenthesis

The *left-parenthesis* initiates reading of a *list*. **read** is called recursively to read successive *objects* until a right parenthesis is found in the input *stream*. A *list* of the *objects* read is returned. Thus

```
(a b c)
```

is read as a *list* of three *objects* (the *symbols* `a`, `b`, and `c`). The right parenthesis need not immediately follow the printed representation of the last *object*; *whitespace*[2] characters and comments may precede it.

If no *objects* precede the right parenthesis, it reads as a *list* of zero *objects* (the *empty list*).

If a *token* that is just a dot not immediately preceded by an escape character is read after some *object* then exactly one more *object* must follow the dot, possibly preceded or followed by *whitespace*[2] or a comment, followed by the right parenthesis:

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

This means that the *cdr* of the last *cons* in the *list* is not **nil**, but rather the *object* whose representation followed the dot. The above example might have been the result of evaluating

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

Similarly,

```
(cons 'this-one 'that-one) =>  (this-one . that-one)
```

It is permissible for the *object* following the dot to be a *list*:

```
(a b c d . (e f . (g))) ==  (a b c d e f g)
```

For information on how the *Lisp printer* prints *lists* and *conses*, see Section 22.1.3.5 (Printing Lists and Conses).

2.4.2 Right-Parenthesis

The *right-parenthesis* is invalid except when used in conjunction with the left parenthesis character. For more information, see Section 2.2 (Reader Algorithm).

2.4.3 Single-Quote

Syntax: `'`<<*exp*>>

A *single-quote* introduces an *expression* to be "quoted." *Single-quote* followed by an *expression exp* is treated by the *Lisp reader* as an abbreviation for and is parsed identically to the *expression* `(quote` *exp*`)`. See the *special operator* **quote**.

2.4.3.1 Examples of Single-Quote

```
'foo =>  FOO
"foo =>  (QUOTE FOO)
(car "foo) =>  QUOTE
```

2.4.4 Semicolon

Syntax: ;<<*text*>>

A *semicolon* introduces *characters* that are to be ignored, such as comments. The *semicolon* and all *characters* up to and including the next *newline* or end of file are ignored.

2.4.4.1 Examples of Semicolon

```
 (+ 3 ; three
    4)
=>  7
```

2.4.4.2 Notes about Style for Semicolon

Some text editors make assumptions about desired indentation based on the number of *semicolons* that begin a comment. The following style conventions are common, although not by any means universal.

2.4.4.2.1 Use of Single Semicolon

Comments that begin with a single *semicolon* are all aligned to the same column at the right (sometimes called the "comment column"). The text of such a comment generally applies only to the line on which it appears. Occasionally two or three contain a single sentence together; this is sometimes indicated by indenting all but the first with an additional space (after the *semicolon*).

2.4.4.2.2 Use of Double Semicolon

Comments that begin with a double *semicolon* are all aligned to the same level of indentation as a *form* would be at that same position in the *code*. The text of such a comment usually describes the state of the *program* at the point where the comment occurs, the *code* which follows the comment, or both.

2.4.4.2.3 Use of Triple Semicolon

Comments that begin with a triple *semicolon* are all aligned to the left margin. Usually they are used prior to a definition or set of definitions, rather than within a definition.

2.4.4.2.4 Use of Quadruple Semicolon

Comments that begin with a quadruple *semicolon* are all aligned to the left margin, and generally contain only a short piece of text that serve as a title for the code which follows, and might be used in the header or footer of a program that prepares code for presentation as a hardcopy document.

2.4.4.2.5 Examples of Style for Semicolon

```
;;;; Math Utilities

;;; FIB computes the the Fibonacci function in the traditional
;;; recursive way.

(defun fib (n)
  (check-type n integer)
  ;; At this point we're sure we have an integer argument.
  ;; Now we can get down to some serious computation.
```

```
(cond ((< n 0)
       ;; Hey, this is just supposed to be a simple example.
       ;; Did you really expect me to handle the general case?
       (error "FIB got ~D as an argument." n))
      ((< n 2) n)            ;fib[0]=0 and fib[1]=1
      ;; The cheap cases didn't work.
      ;; Nothing more to do but recurse.
      (t (+ (fib (- n 1))    ;The traditional formula
            (fib (- n 2))))) ; is fib[n-1]+fib[n-2].
```

2.4.5 Double-Quote

Syntax: "<<*text*>>"

The *double-quote* is used to begin and end a *string*. When a *double-quote* is encountered, *characters* are read from the *input stream* and accumulated until another *double-quote* is encountered. If a *single escape character* is seen, the *single escape character* is discarded, the next *character* is accumulated, and accumulation continues. The accumulated *characters* up to but not including the matching *double-quote* are made into a *simple string* and returned. It is *implementation-dependent* which *attributes* of the accumulated characters are removed in this process.

Examples of the use of the *double-quote* character are in the next figure.

```
"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
```

Figure 2-18. Examples of the use of double-quote

Note that to place a single escape character or a *double-quote* into a string, such a character must be preceded by a single escape character. Note, too, that a multiple escape character need not be quoted by a single escape character within a string.

For information on how the *Lisp printer* prints *strings*, see Section 22.1.3.4 (Printing Strings).

2.4.6 Backquote

The *backquote* introduces a template of a data structure to be built. For example, writing

```
‘(cond ((numberp ,x) ,@y) (t (print ,x) ,@y))
```

is roughly equivalent to writing

```
(list ’cond
      (cons (list ’numberp x) y)
      (list* ’t (list ’print x) y))
```

Where a comma occurs in the template, the *expression* following the comma is to be evaluated to produce an *object* to be inserted at that point. Assume `b` has the value 3, for example, then evaluating the *form* denoted by `‘(a b ,b ,(+ b 1) b)` produces the result `(a b 3 4 b)`.

If a comma is immediately followed by an *at-sign*, then the *form* following the *at-sign* is evaluated to produce a *list* of *objects*. These *objects* are then "spliced" into place in the template. For example, if `x` has the value `(a b c)`, then

```
 ‘(x ,x ,@x foo ,(cadr x) bar ,(cdr x) baz ,@(cdr x))
=>  (x (a b c) a b c foo b bar (b c) baz b c)
```

The backquote syntax can be summarized formally as follows.

* ‘*basic* is the same as ’*basic*, that is, (quote *basic*), for any *expression basic* that is not a *list* or a general *vector*.
* ‘,*form* is the same as *form*, for any *form*, provided that the representation of *form* does not begin with *at-sign* or *dot*. (A similar caveat holds for all occurrences of a form after a *comma*.)
* ‘,@*form* has undefined consequences.
* ‘(x1 x2 x3 ... xn . atom) may be interpreted to mean

```
(append [ x1] [ x2] [ x3] ... [ xn] (quote atom))
```

  where the brackets are used to indicate a transformation of an *xj* as follows:
  -- [*form*] is interpreted as (list ‘*form*), which contains a backquoted form that must then be further interpreted.
  -- [,*form*] is interpreted as (list *form*).
  -- [,@*form*] is interpreted as *form*.
* ‘(x1 x2 x3 ... xn) may be interpreted to mean the same as the backquoted form ‘(x1 x2 x3 ... xn . **nil**), thereby reducing it to the previous case.
* ‘(x1 x2 x3 ... xn . ,form) may be interpreted to mean

```
(append [ x1] [ x2] [ x3] ... [ xn] form)
```

  where the brackets indicate a transformation of an xj as described above.
* ‘(x1 x2 x3 ... xn . ,@form) has undefined consequences.
* ‘#(x1 x2 x3 ... xn) may be interpreted to mean (apply #’vector ‘(x1 x2 x3 ... xn)).

Anywhere ",@" may be used, the syntax ",." may be used instead to indicate that it is permissible to operate *destructively* on the *list structure* produced by the form following the ",." (in effect, to use **nconc** instead of **append**).

If the backquote syntax is nested, the innermost backquoted form should be expanded first. This means that if several commas occur in a row, the leftmost one belongs to the innermost *backquote*.

An *implementation* is free to interpret a backquoted *form* F1 as any *form* F2 that, when evaluated, will produce a result that is the *same* under **equal** as the result implied by the above definition, provided that the side-effect behavior of the substitute *form* F2 is also consistent with the description given above. The constructed copy of the template might or might not share *list* structure with the template itself. As an example, the above definition implies that

```
‘((,a b) ,c ,@d)
```

will be interpreted as if it were

```
(append (list (append (list a) (list ’b) ’nil)) (list c) d ’nil)
```

but it could also be legitimately interpreted to mean any of the following:

```
(append (list (append (list a) (list ’b))) (list c) d)
(append (list (append (list a) ’(b))) (list c) d)
(list* (cons a ’(b)) c d)
(list* (cons a (list ’b)) c d)
(append (list (cons a ’(b))) (list c) d)
(list* (cons a ’(b)) c (copy-list d))
```

2.4.6.1 Notes about Backquote

Since the exact manner in which the *Lisp reader* will parse an *expression* involving the *backquote reader macro* is not specified, an *implementation* is free to choose any representation that preserves the semantics described.

Often an *implementation* will choose a representation that facilitates pretty printing of the expression, so that `(pprint '(a ,b))` will display `'(a ,b)` and not, for example, `(list 'a b)`. However, this is not a requirement.

Implementors who have no particular reason to make one choice or another might wish to refer to *IEEE Standard for the Scheme Programming Language*, which identifies a popular choice of representation for such expressions that might provide useful to be useful compatibility for some user communities. There is no requirement, however, that any *conforming implementation* use this particular representation. This information is provided merely for cross-reference purposes.

2.4.7 Comma

The *comma* is part of the backquote syntax; see Section 2.4.6 (Backquote). *Comma* is invalid if used other than inside the body of a backquote *expression* as described above.

2.4.8 Sharpsign

Sharpsign is a *non-terminating dispatching macro character*. It reads an optional sequence of digits and then one more character, and uses that character to select a *function* to run as a *reader macro function*.

The *standard syntax* includes constructs introduced by the # character. The syntax of these constructs is as follows: a character that identifies the type of construct is followed by arguments in some form. If the character is a letter, its *case* is not important; `#O` and `#o` are considered to be equivalent, for example.

Certain # constructs allow an unsigned decimal number to appear between the # and the character.

The *reader macros* associated with the *dispatching macro character* # are described later in this section and summarized in the next figure.

dispatch char	purpose	dispatch char	purpose
Backspace	signals error	{	undefined*
Tab	signals error	}	undefined*
Newline	signals error	+	read-time conditional
Linefeed	signals error	-	read-time conditional
Page	signals error	.	read-time evaluation
Return	signals error	/	undefined
Space	signals error	A, a	array
!	undefined*	B, b	binary rational
"	undefined	C, c	complex number
#	reference to = label	D, d	undefined
$	undefined	E, e	undefined
%	undefined	F, f	undefined
&	undefined	G, g	undefined
'	function abbreviation	H, h	undefined
(	simple vector	I, i	undefined
)	signals error	J, j	undefined
*	bit vector	K, k	undefined
,	undefined	L, l	undefined
:	uninterned symbol	M, m	undefined
;	undefined	N, n	undefined
<	signals error	O, o	octal rational
=	labels following object	P, p	pathname
>	undefined	Q, q	undefined

?	undefined*	R, r	radix-n rational
@	undefined	S, s	structure
[	undefined*	T, t	undefined
\	character object	U, u	undefined
]	undefined*	V, v	undefined
^	undefined	W, w	undefined
_	undefined	X, x	hexadecimal rational
‘	undefined	Y, y	undefined
|	balanced comment	Z, z	undefined
~	undefined	Rubout	undefined

Figure 2-19. Standard #Dispatching Macro Character Syntax

The combinations marked by an asterisk (*) are explicitly reserved to the user. No *conforming implementation* defines them.

Note also that *digits* do not appear in the preceding table. This is because the notations `#0`, `#1`, ..., `#9` are reserved for another purpose which occupies the same syntactic space. When a *digit* follows a *sharpsign*, it is not treated as a dispatch character. Instead, an unsigned integer argument is accumulated and passed as an *argument* to the *reader macro* for the *character* that follows the digits. For example, `#2A((1 2) (3 4))` is a use of `#A` with an argument of `2`.

2.4.8.1 Sharpsign Backslash

Syntax: #\<<*x*>>

When the *token x* is a single *character* long, this parses as the literal *character char*. *Uppercase* and *lowercase* letters are distinguished after #\; `#\A` and `#\a` denote different *character objects*. Any single *character* works after #\, even those that are normally special to **read**, such as *left-parenthesis* and *right-parenthesis*.

In the single *character* case, the *x* must be followed by a non-constituent *character*. After #\ is read, the reader backs up over the *slash* and then reads a *token*, treating the initial *slash* as a *single escape character* (whether it really is or not in the *current readtable*).

When the *token x* is more than one *character* long, the *x* must have the syntax of a *symbol* with no embedded *package markers*. In this case, the *sharpsign backslash* notation parses as the *character* whose *name* is `(string-upcase` *x*`)`; see Section 13.1.7 (Character Names).

For information about how the *Lisp printer* prints *character objects*, see Section 22.1.3.2 (Printing Characters).

2.4.8.2 Sharpsign Single-Quote

Any *expression* preceded by `#'` (*sharpsign* followed by *single-quote*), as in `#'`*expression*, is treated by the *Lisp reader* as an abbreviation for and parsed identically to the *expression* `(function` *expression*`)`. See **function**. For example,

```
(apply #'+ l) ==  (apply (function +) l)
```

2.4.8.3 Sharpsign Left-Parenthesis

`#(` and `)` are used to notate a *simple vector*.

If an unsigned decimal integer appears between the # and `(`, it specifies explicitly the length of the *vector*. The consequences are undefined if the number of *objects* specified before the closing `)` exceeds the unsigned decimal integer. If the number of *objects* supplied before the closing `)` is less than the unsigned decimal integer but greater than zero, the last *object* is used to fill all remaining elements of the *vector*. The consequences are undefined if the

unsigned decimal integer is non-zero and number of *objects* supplied before the closing `)` is zero. For example,

```
#(a b c c c c)
#6(a b c c c c)
#6(a b c)
#6(a b c c)
```

all mean the same thing: a *vector* of length `6` with *elements* `a`, `b`, and four occurrences of `c`. Other examples follow:

```
#(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
```

The notation `#()` denotes an empty *vector*, as does `#0()`.

For information on how the *Lisp printer* prints *vectors*, see Section 22.1.3.4 (Printing Strings), Section 22.1.3.6 (Printing Bit Vectors), or Section 22.1.3.7 (Printing Other Vectors).

2.4.8.4 Sharpsign Asterisk

Syntax: `#*<<`*bits*`>>`

A *simple bit vector* is constructed containing the indicated *bits* (`0`'s and `1`'s), where the leftmost *bit* has index zero and the subsequent *bits* have increasing indices.

Syntax: `#<<`*n*`>>*<<`*bits*`>>`

With an argument *n*, the *vector* to be created is of *length n*. If the number of *bits* is less than *n* but greater than zero, the last bit is used to fill all remaining bits of the *bit vector*.

The notations `#*` and `#0*` each denote an empty *bit vector*.

Regardless of whether the optional numeric argument *n* is provided, the *token* that follows the *asterisk* is delimited by a normal *token* delimiter. However, (unless the *value* of ***read-suppress*** is *true*) an error of *type* **reader-error** is signaled if that *token* is not composed entirely of `0`'s and `1`'s, or if *n* was supplied and the *token* is composed of more than *n bits*, or if *n* is greater than one, but no *bits* were specified. Neither a *single escape* nor a *multiple escape* is permitted in this *token*.

For information on how the *Lisp printer* prints *bit vectors*, see Section 22.1.3.6 (Printing Bit Vectors).

2.4.8.4.1 Examples of Sharpsign Asterisk

For example,

```
  #*101111
 #6*101111
 #6*101
 #6*1011
```

all mean the same thing: a *vector* of length `6` with *elements* `1`, `0`, `1`, `1`, `1`, and `1`.

For example:

```
#*         ;An empty bit-vector
```

2.4.8.5 Sharpsign Colon

Syntax: #:<<*symbol-name*>>

#: introduces an *uninterned symbol* whose *name* is *symbol-name*. Every time this syntax is encountered, a *distinct uninterned symbol* is created. The *symbol-name* must have the syntax of a *symbol* with no *package prefix*.

For information on how the *Lisp reader* prints *uninterned symbols*, see Section 22.1.3.3 (Printing Symbols).

2.4.8.6 Sharpsign Dot

#.*foo* is read as the *object* resulting from the evaluation of the *object* represented by *foo*. The evaluation is done during the **read** process, when the #. notation is encountered. The #. syntax therefore performs a read-time evaluation of *foo*.

The normal effect of #. is inhibited when the *value* of ***read-eval*** is *false*. In that situation, an error of *type* **reader-error** is signaled.

For an *object* that does not have a convenient printed representation, a *form* that computes the *object* can be given using the #. notation.

2.4.8.7 Sharpsign B

#B*rational* reads *rational* in binary (radix 2). For example,

```
#B1101 ==  13 ;11012
#b101/11 ==  5/3
```

The consequences are undefined if the token immediately following the #B does not have the syntax of a binary (i.e., radix 2) *rational*.

2.4.8.8 Sharpsign O

#O*rational* reads *rational* in octal (radix 8). For example,

```
#o37/15 ==  31/13
#o777 ==  511
#o105 ==  69 ;1058
```

The consequences are undefined if the token immediately following the #O does not have the syntax of an octal (i.e., radix 8) *rational*.

2.4.8.9 Sharpsign X

#X*rational* reads *rational* in hexadecimal (radix 16). The digits above 9 are the letters A through F (the lowercase letters a through f are also acceptable). For example,

```
#xF00 ==  3840
#x105 ==  261 ;10516
```

The consequences are undefined if the token immediately following the #X does not have the syntax of a hexadecimal (i.e., radix 16) *rational*.

2.4.8.10 Sharpsign R

#*n*R

#*radix*R*rational* reads *rational* in radix *radix*. *radix* must consist of only digits that are interpreted as an *integer* in decimal radix; its value must be between 2 and 36 (inclusive). Only valid digits for the specified radix may be used.

For example, #3r102 is another way of writing 11 (decimal), and #11R32 is another way of writing 35 (decimal). For radices larger than 10, letters of the alphabet are used in order for the digits after 9. No alternate # notation exists for the decimal radix since a decimal point suffices.

The next figure contains examples of the use of #B, #O, #X, and #R.

```
#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
#xACCEDED    ;181202413, in hexadecimal radix
```

Figure 2-20. Radix Indicator Example

The consequences are undefined if the token immediately following the #*n*R does not have the syntax of a *rational* in radix *n*.

2.4.8.11 Sharpsign C

#C reads a following *object*, which must be a *list* of length two whose *elements* are both *reals*. These *reals* denote, respectively, the real and imaginary parts of a *complex* number. If the two parts as notated are not of the same data type, then they are converted according to the rules of floating-point *contagion* described in Section 12.1.1.2 (Contagion in Numeric Operations).

#C(*real imag*) is equivalent to #.(complex (quote *real*) (quote *imag*)), except that #C is not affected by ***read-eval***. See the *function* **complex**.

The next figure contains examples of the use of #C.

```
#C(3.0s1 2.0s-1)  ;A complex with small float parts.
#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.
```

Figure 2-21. Complex Number Example

For further information, see Section 22.1.3.1.4 (Printing Complexes) and Section 2.3.2.3 (Syntax of a Complex).

2.4.8.12 Sharpsign A

#*n*A

#*n*A*object* constructs an *n*-dimensional *array*, using *object* as the value of the `:initial-contents` argument to **make-array**.

For example, `#2A((0 1 5) (foo 2 (hot dog)))` represents a 2-by-3 matrix:

```
0       1       5
foo     2       (hot dog)
```

In contrast, `#1A((0 1 5) (foo 2 (hot dog)))` represents a *vector* of *length* `2` whose *elements* are *lists*:

```
(0 1 5) (foo 2 (hot dog))
```

`#0A((0 1 5) (foo 2 (hot dog)))` represents a zero-dimensional *array* whose sole element is a *list*:

```
((0 1 5) (foo 2 (hot dog)))
```

`#0A foo` represents a zero-dimensional *array* whose sole element is the *symbol* `foo`. The notation `#1A foo` is not valid because `foo` is not a *sequence*.

If some *dimension* of the *array* whose representation is being parsed is found to be `0`, all *dimensions* to the right (i.e., the higher numbered *dimensions*) are also considered to be `0`.

For information on how the *Lisp printer* prints *arrays*, see Section 22.1.3.4 (Printing Strings), Section 22.1.3.6 (Printing Bit Vectors), Section 22.1.3.7 (Printing Other Vectors), or Section 22.1.3.8 (Printing Other Arrays).

2.4.8.13 Sharpsign S

`#s(name slot1 value1 slot2 value2 ...)` denotes a *structure*. This is valid only if *name* is the name of a *structure type* already defined by **defstruct** and if the *structure type* has a standard constructor function. Let *cm* stand for the name of this constructor function; then this syntax is equivalent to

```
#.(cm keyword1 'value1 keyword2 'value2 ...)
```

where each *keywordj* is the result of computing

```
(intern (string slotj) (find-package 'keyword))
```

The net effect is that the constructor function is called with the specified slots having the specified values. (This coercion feature is deprecated; in the future, keyword names will be taken in the package they are read in, so *symbols* that are actually in the `KEYWORD` package should be used if that is what is desired.)

Whatever *object* the constructor function returns is returned by the `#S` syntax.

For information on how the *Lisp printer* prints *structures*, see Section 22.1.3.12 (Printing Structures).

2.4.8.14 Sharpsign P

`#P` reads a following *object*, which must be a *string*.

`#P<<`*expression*`>>` is equivalent to `#.(parse-namestring '<<`*expression*`>>)`, except that `#P` is not affected by ***read-eval***.

For information on how the *Lisp printer* prints *pathnames*, see Section 22.1.3.11 (Printing Pathnames).

2.4.8.15 Sharpsign Equal-Sign

#*n*=

#*n*=*object* reads as whatever *object* has *object* as its printed representation. However, that *object* is labeled by *n*, a required unsigned decimal integer, for possible reference by the syntax #*n*#. The scope of the label is the *expression* being read by the outermost call to **read**; within this *expression*, the same label may not appear twice.

2.4.8.16 Sharpsign Sharpsign

#*n*#

#*n*#, where *n* is a required unsigned decimal *integer*, provides a reference to some *object* labeled by #*n*=; that is, #*n*# represents a pointer to the same (**eq**) *object* labeled by #*n*=. For example, a structure created in the variable `y` by this code:

```
(setq x (list 'p 'q))
(setq y (list (list 'a 'b) x 'foo x))
(rplacd (last y) (cdr y))
```

could be represented in this way:

```
((a b) . #1=(#2=(p q) foo #2# . #1#))
```

Without this notation, but with ***print-length*** set to `10` and ***print-circle*** set to **nil**, the structure would print in this way:

```
((a b) (p q) foo (p q) (p q) foo (p q) (p q) foo (p q) ...)
```

A reference #*n*# may only occur after a label #*n*=; forward references are not permitted. The reference may not appear as the labeled object itself (that is, #*n*=#*n*#) may not be written because the *object* labeled by #*n*= is not well defined in this case.

2.4.8.17 Sharpsign Plus

#+ provides a read-time conditionalization facility; the syntax is #+*test expression*. If the *feature expression test* succeeds, then this textual notation represents an *object* whose printed representation is *expression*. If the *feature expression test* fails, then this textual notation is treated as *whitespace*[2]; that is, it is as if the "#+ *test expression*" did not appear and only a *space* appeared in its place.

For a detailed description of success and failure in *feature expressions*, see Section 24.1.2.1 (Feature Expressions).

#+ operates by first reading the *feature expression* and then skipping over the *form* if the *feature expression* fails. While reading the *test*, the *current package* is the `KEYWORD` package. Skipping over the *form* is accomplished by *binding* ***read-suppress*** to *true* and then calling **read**.

For examples, see Section 24.1.2.1.1 (Examples of Feature Expressions).

2.4.8.18 Sharpsign Minus

#- is like #+ except that it skips the *expression* if the *test* succeeds; that is,

```
#-test expression ==  #+(not test) expression
```

For examples, see Section 24.1.2.1.1 (Examples of Feature Expressions).

2.4.8.19 Sharpsign Vertical-Bar

#|...|# is treated as a comment by the reader. It must be balanced with respect to other occurrences of #| and |#, but otherwise may contain any characters whatsoever.

2.4.8.19.1 Examples of Sharpsign Vertical-Bar

The following are some examples that exploit the #|...|# notation:

```
;;; In this example, some debugging code is commented out with #|...|#
;;; Note that this kind of comment can occur in the middle of a line
;;; (because a delimiter marks where the end of the comment occurs)
;;; where a semicolon comment can only occur at the end of a line
;;; (because it comments out the rest of the line).
 (defun add3 (n) #|(format t "~&Adding 3 to ~D." n)|# (+ n 3))

;;; The examples that follow show issues related to #| ... |# nesting.

;;; In this first example, #| and |# always occur properly paired,
;;; so nesting works naturally.
 (defun mention-fun-fact-1a ()
   (format t "CL uses ; and #|...|# in comments."))
=>  MENTION-FUN-FACT-1A
 (mention-fun-fact-1a)
>>  CL uses ; and #|...|# in comments.
=>  NIL
 #| (defun mention-fun-fact-1b ()
      (format t "CL uses ; and #|...|# in comments.")) |#
 (fboundp 'mention-fun-fact-1b) =>  NIL

;;; In this example, vertical-bar followed by sharpsign needed to appear
;;; in a string without any matching sharpsign followed by vertical-bar
;;; having preceded this.  To compensate, the programmer has included a
;;; slash separating the two characters.  In case 2a, the slash is
;;; unnecessary but harmless, but in case 2b, the slash is critical to
;;; allowing the outer #| ... |# pair match.  If the slash were not present,
;;; the outer comment would terminate prematurely.
 (defun mention-fun-fact-2a ()
   (format t "Don't use |\# unmatched or you'll get in trouble!"))
=>  MENTION-FUN-FACT-2A
 (mention-fun-fact-2a)
>>  Don't use |# unmatched or you'll get in trouble!
=>  NIL
 #| (defun mention-fun-fact-2b ()
      (format t "Don't use |\# unmatched or you'll get in trouble!") |#
 (fboundp 'mention-fun-fact-2b) =>  NIL

;;; In this example, the programmer attacks the mismatch problem in a
;;; different way.  The sharpsign vertical bar in the comment is not needed
;;; for the correct parsing of the program normally (as in case 3a), but
;;; becomes important to avoid premature termination of a comment when such
;;; a program is commented out (as in case 3b).
 (defun mention-fun-fact-3a () ; #|
   (format t "Don't use |# unmatched or you'll get in trouble!"))
=>  MENTION-FUN-FACT-3A
 (mention-fun-fact-3a)
>>  Don't use |# unmatched or you'll get in trouble!
=>  NIL
```

```
#|
(defun mention-fun-fact-3b () ; #|
  (format t "Don't use |# unmatched or you'll get in trouble!"))
|#
(fboundp 'mention-fun-fact-3b) =>  NIL
```

2.4.8.19.2 Notes about Style for Sharpsign Vertical-Bar

Some text editors that purport to understand Lisp syntax treat any |...| as balanced pairs that cannot nest (as if they were just balanced pairs of the multiple escapes used in notating certain symbols). To compensate for this deficiency, some programmers use the notation #||...#||...||#...||# instead of #|...#|...|#...|#. Note that this alternate usage is not a different *reader macro*; it merely exploits the fact that the additional vertical-bars occur within the comment in a way that tricks certain text editor into better supporting nested comments. As such, one might sometimes see code like:

```
#|| (+ #|| 3 ||# 4 5) ||#
```

Such code is equivalent to:

```
#| (+ #| 3 |# 4 5) |#
```

2.4.8.20 Sharpsign Less-Than-Sign

#< is not valid reader syntax. The *Lisp reader* will signal an error of *type* **reader-error** on encountering #<. This syntax is typically used in the printed representation of *objects* that cannot be read back in.

2.4.8.21 Sharpsign Whitespace

followed immediately by *whitespace*[1] is not valid reader syntax. The *Lisp reader* will signal an error of *type* **reader-error** if it encounters the reader macro notation #<Newline> or #<Space>.

2.4.8.22 Sharpsign Right-Parenthesis

This is not valid reader syntax.

The *Lisp reader* will signal an error of *type* **reader-error** upon encountering #).

2.4.9 Re-Reading Abbreviated Expressions

Note that the *Lisp reader* will generally signal an error of *type* **reader-error** when reading an *expression*[2] that has been abbreviated because of length or level limits (see ***print-level***, ***print-length***, and ***print-lines***) due to restrictions on "..", "...", "#" followed by *whitespace*[1], and "#)".

3. Evaluation and Compilation

3.1 Evaluation

Execution of *code* can be accomplished by a variety of means ranging from direct interpretation of a *form* representing a *program* to invocation of *compiled code* produced by a *compiler*.

Evaluation is the process by which a *program* is *executed* in Common Lisp. The mechanism of *evaluation* is manifested both implicitly through the effect of the *Lisp read-eval-print loop*, and explicitly through the presence of the *functions* **eval**, **compile**, **compile-file**, and **load**. Any of these facilities might share the same execution strategy, or each might use a different one.

The behavior of a *conforming program* processed by **eval** and by **compile-file** might differ; see Section 3.2.2.3 (Semantic Constraints).

Evaluation can be understood in terms of a model in which an interpreter recursively traverses a *form* performing each step of the computation as it goes. This model, which describes the semantics of Common Lisp *programs*, is described in Section 3.1.2 (The Evaluation Model).

3.1.1 Introduction to Environments

A *binding* is an association between a *name* and that which the name denotes. *Bindings* are *established* in a *lexical environment* or a *dynamic environment* by particular *special operators*.

An *environment* is a set of *bindings* and other information used during evaluation (e.g., to associate meanings with names).

Bindings in an *environment* are partitioned into *namespaces*. A single *name* can simultaneously have more than one associated *binding* per *environment*, but can have only one associated *binding* per *namespace*.

3.1.1.1 The Global Environment

The *global environment* is that part of an *environment* that contains *bindings* with both *indefinite scope* and *indefinite extent*. The *global environment* contains, among other things, the following:

- *bindings* of *dynamic variables* and *constant variables*.
- *bindings* of *functions*, *macros*, and *special operators*.
- *bindings* of *compiler macros*.
- *bindings* of *type* and *class names*
- information about *proclamations*.

3.1.1.2 Dynamic Environments

A *dynamic environment* for *evaluation* is that part of an *environment* that contains *bindings* whose duration is bounded by points of *establishment* and *disestablishment* within the execution of the *form* that established the *binding*. A *dynamic environment* contains, among other things, the following:

- *bindings* for *dynamic variables*.
- information about *active catch tags*.
- information about *exit points* established by **unwind-protect**.
- information about *active handlers* and *restarts*.

The *dynamic environment* that is active at any given point in the *execution* of a *program* is referred to by definite reference as "the current *dynamic environment*," or sometimes as just "the *dynamic environment*."

Within a given *namespace*, a *name* is said to be *bound* in a *dynamic environment* if there is a *binding* associated with its *name* in the *dynamic environment* or, if not, there is a *binding* associated with its name in the *global environment*.

3.1.1.3 Lexical Environments

A *lexical environment* for *evaluation* at some position in a *program* is that part of the *environment* that contains information having *lexical scope* within the *forms* containing that position. A *lexical environment* contains, among other things, the following:

- *bindings* of *lexical variables* and *symbol macros*.
- *bindings* of *functions* and *macros*. (Implicit in this is information about those *compiler macros* that are locally disabled.)
- *bindings* of *block tags*.
- *bindings* of *go tags*.
- information about *declarations*.

The *lexical environment* that is active at any given position in a *program* being semantically processed is referred to by definite reference as "the current *lexical environment*," or sometimes as just "the *lexical environment*."

Within a given *namespace*, a *name* is said to be *bound* in a *lexical environment* if there is a *binding* associated with its *name* in the *lexical environment* or, if not, there is a *binding* associated with its name in the *global environment*.

3.1.1.3.1 The Null Lexical Environment

The *null lexical environment* is equivalent to the *global environment*.

Although in general the representation of an *environment object* is *implementation-dependent*, **nil** can be used in any situation where an *environment object* is called for in order to denote the *null lexical environment*.

3.1.1.4 Environment Objects

Some *operators* make use of an *object*, called an *environment object*, that represents the set of *lexical bindings* needed to perform semantic analysis on a *form* in a given *lexical environment*. The set of *bindings* in an *environment object* may be a subset of the *bindings* that would be needed to actually perform an *evaluation*; for example, *values* associated with *variable names* and *function names* in the corresponding *lexical environment* might not be available in an *environment object*.

The *type* and nature of an *environment object* is *implementation-dependent*. The *values* of *environment parameters* to *macro functions* are examples of *environment objects*.

The *object* **nil** when used as an *environment object* denotes the *null lexical environment*; see Section 3.1.1.3.1 (The Null Lexical Environment).

3.1.2 The Evaluation Model

A Common Lisp system evaluates *forms* with respect to lexical, dynamic, and global *environments*. The following sections describe the components of the Common Lisp evaluation model.

3.1.2.1 Form Evaluation

Forms fall into three categories: *symbols*, *conses*, and *self-evaluating objects*. The following sections explain these categories.

3.1.2.1.1 Symbols as Forms

If a *form* is a *symbol*, then it is either a *symbol macro* or a *variable*.

The *symbol* names a *symbol macro* if there is a *binding* of the *symbol* as a *symbol macro* in the current *lexical environment* (see **define-symbol-macro** and **symbol-macrolet**). If the *symbol* is a *symbol macro*, its expansion function is obtained. The expansion function is a function of two arguments, and is invoked by calling the *macroexpand hook* with the expansion function as its first argument, the *symbol* as its second argument, and an *environment object* (corresponding to the current *lexical environment*) as its third argument. The *macroexpand hook*, in turn, calls the expansion function with the *form* as its first argument and the *environment* as its second argument. The *value* of the expansion function, which is passed through by the *macroexpand hook*, is a *form*. This resulting *form* is processed in place of the original *symbol*.

If a *form* is a *symbol* that is not a *symbol macro*, then it is the *name* of a *variable*, and the *value* of that *variable* is returned. There are three kinds of variables: *lexical variables*, *dynamic variables*, and *constant variables*. A *variable* can store one *object*. The main operations on a *variable* are to *read*[1] and to *write*[1] its *value*.

An error of *type* **unbound-variable** should be signaled if an *unbound variable* is referenced.

Non-constant variables can be *assigned* by using **setq** or *bound*[3] by using **let**. The next figure lists some *defined names* that are applicable to assigning, binding, and defining *variables*.

```
boundp        let                 progv
defconstant   let*                psetq
defparameter  makunbound          set
defvar        multiple-value-bind setq
lambda        multiple-value-setq symbol-value
```

Figure 3-1. Some Defined Names Applicable to Variables

The following is a description of each kind of variable.

3.1.2.1.1.1 Lexical Variables

A *lexical variable* is a *variable* that can be referenced only within the *lexical scope* of the *form* that establishes that *variable*; *lexical variables* have *lexical scope*. Each time a *form* creates a *lexical binding* of a *variable*, a *fresh binding* is *established*.

Within the *scope* of a *binding* for a *lexical variable name*, uses of that *name* as a *variable* are considered to be references to that *binding* except where the *variable* is *shadowed*[2] by a *form* that *establishes* a *fresh binding* for that *variable name*, or by a *form* that locally *declares* the *name* **special**.

A *lexical variable* always has a *value*. There is no *operator* that introduces a *binding* for a *lexical variable* without giving it an initial *value*, nor is there any *operator* that can make a *lexical variable* be *unbound*.

Bindings of *lexical variables* are found in the *lexical environment*.

3.1.2.1.1.2 Dynamic Variables

A *variable* is a *dynamic variable* if one of the following conditions hold:

- It is locally declared or globally proclaimed **special**.
- It occurs textually within a *form* that creates a *dynamic binding* for a *variable* of the *same name*, and the *binding* is not *shadowed*[2] by a *form* that creates a *lexical binding* of the same *variable name*.

A *dynamic variable* can be referenced at any time in any *program*; there is no textual limitation on references to *dynamic variables*. At any given time, all *dynamic variables* with a given name refer to exactly one *binding*, either in the *dynamic environment* or in the *global environment*.

The *value* part of the *binding* for a *dynamic variable* might be empty; in this case, the *dynamic variable* is said to have no *value*, or to be *unbound*. A *dynamic variable* can be made *unbound* by using **makunbound**.

The effect of *binding* a *dynamic variable* is to create a new *binding* to which all references to that *dynamic variable* in any *program* refer for the duration of the *evaluation* of the *form* that creates the *dynamic binding*.

A *dynamic variable* can be referenced outside the *dynamic extent* of a *form* that *binds* it. Such a *variable* is sometimes called a "global variable" but is still in all respects just a *dynamic variable* whose *binding* happens to exist in the *global environment* rather than in some *dynamic environment*.

A *dynamic variable* is *unbound* unless and until explicitly assigned a value, except for those variables whose initial value is defined in this specification or by an *implementation*.

3.1.2.1.1.3 Constant Variables

Certain variables, called *constant variables*, are reserved as "named constants." The consequences are undefined if an attempt is made to assign a value to, or create a *binding* for a *constant variable*, except that a ‘compatible’ redefinition of a *constant variable* using **defconstant** is permitted; see the *macro* **defconstant**.

Keywords, *symbols* defined by Common Lisp or the *implementation* as constant (such as **nil**, **t**, and **pi**), and *symbols* declared as constant using **defconstant** are *constant variables*.

3.1.2.1.1.4 Symbols Naming Both Lexical and Dynamic Variables

The same *symbol* can name both a *lexical variable* and a *dynamic variable*, but never in the same *lexical environment*.

In the following example, the *symbol* `X` is used, at different times, as the *name* of a *lexical variable* and as the *name* of a *dynamic variable*.

```
 (let ((x 1))            ;Binds a special variable X
   (declare (special x))
   (let ((x 2))          ;Binds a lexical variable X
     (+ x                ;Reads a lexical variable X
        (locally (declare (special x))
                 x))))   ;Reads a special variable X
=>  3
```

3.1.2.1.2 Conses as Forms

A *cons* that is used as a *form* is called a *compound form*.

If the *car* of that *compound form* is a *symbol*, that *symbol* is the *name* of an *operator*, and the *form* is either a *special form*, a *macro form*, or a *function form*, depending on the *function binding* of the *operator* in the current *lexical environment*. If the *operator* is neither a *special operator* nor a *macro name*, it is assumed to be a *function name* (even if there is no definition for such a *function*).

If the *car* of the *compound form* is not a *symbol*, then that *car* must be a *lambda expression*, in which case the *compound form* is a *lambda form*.

How a *compound form* is processed depends on whether it is classified as a *special form*, a *macro form*, a *function form*, or a *lambda form*.

3.1.2.1.2.1 Special Forms

A *special form* is a *form* with special syntax, special evaluation rules, or both, possibly manipulating the evaluation environment, control flow, or both. A *special operator* has access to the current *lexical environment* and the current *dynamic environment*. Each *special operator* defines the manner in which its *subexpressions* are treated---which are *forms*, which are special syntax, *etc*.

Some *special operators* create new lexical or dynamic *environments* for use during the *evaluation* of *subforms* of the *special form*. For example, **block** creates a new *lexical environment* that is the same as the one in force at the point of evaluation of the **block** *form* with the addition of a *binding* of the **block** name to an *exit point* from the **block**.

The set of *special operator names* is fixed in Common Lisp; no way is provided for the user to define a *special operator*. The next figure lists all of the Common Lisp *symbols* that have definitions as *special operators*.

```
block      let*                 return-from
catch      load-time-value      setq
eval-when  locally              symbol-macrolet
flet       macrolet             tagbody
function   multiple-value-call  the
go         multiple-value-prog1 throw
if         progn                unwind-protect
labels     progv
let        quote
```

Figure 3-2. Common Lisp Special Operators

3.1.2.1.2.2 Macro Forms

If the *operator* names a *macro*, its associated *macro function* is applied to the entire *form* and the result of that application is used in place of the original *form*.

Specifically, a *symbol* names a *macro* in a given *lexical environment* if **macro-function** is *true* of the *symbol* and that *environment*. The *function* returned by **macro-function** is a *function* of two arguments, called the expansion function. The expansion function is invoked by calling the *macroexpand hook* with the expansion function as its first argument, the entire *macro form* as its second argument, and an *environment object* (corresponding to the current *lexical environment*) as its third argument. The *macroexpand hook*, in turn, calls the expansion function with the *form* as its first argument and the *environment* as its second argument. The *value* of the expansion function, which is passed through by the *macroexpand hook*, is a *form*. The returned *form* is *evaluated* in place of the original *form*.

The consequences are undefined if a *macro function* destructively modifies any part of its *form* argument.

A *macro name* is not a *function designator*, and cannot be used as the *function* argument to *functions* such as **apply**, **funcall**, or **map**.

An *implementation* is free to implement a Common Lisp *special operator* as a *macro*. An *implementation* is free to implement any *macro operator* as a *special operator*, but only if an equivalent definition of the *macro* is also provided.

The next figure lists some *defined names* that are applicable to *macros*.

```
*macroexpand-hook*  macro-function  macroexpand-1
defmacro            macroexpand     macrolet
```

Figure 3-3. Defined names applicable to macros

3.1.2.1.2.3 Function Forms

If the *operator* is a *symbol* naming a *function*, the *form* represents a *function form*, and the *cdr* of the list contains the *forms* which when evaluated will supply the arguments passed to the *function*.

When a *function name* is not defined, an error of *type* **undefined-function** should be signaled at run time; see Section 3.2.2.3 (Semantic Constraints).

A *function form* is evaluated as follows:

The *subforms* in the *cdr* of the original *form* are evaluated in left-to-right order in the current lexical and dynamic *environments*. The *primary value* of each such *evaluation* becomes an *argument* to the named *function*; any additional *values* returned by the *subforms* are discarded.

The *functional value* of the *operator* is retrieved from the *lexical environment*, and that *function* is invoked with the indicated arguments.

Although the order of *evaluation* of the *argument subforms* themselves is strictly left-to-right, it is not specified whether the definition of the *operator* in a *function form* is looked up before the *evaluation* of the *argument subforms*, after the *evaluation* of the *argument subforms*, or between the *evaluation* of any two *argument subforms* if there is more than one such *argument subform*. For example, the following might return 23 or 24.

```
(defun foo (x) (+ x 3))
(defun bar () (setf (symbol-function 'foo) #'(lambda (x) (+ x 4))))
(foo (progn (bar) 20))
```

A *binding* for a *function name* can be *established* in one of several ways. A *binding* for a *function name* in the *global environment* can be *established* by **defun**, **setf** of **fdefinition**, **setf** of **symbol-function**, **ensure-generic-function**, **defmethod** (implicitly, due to **ensure-generic-function**), or **defgeneric**. A *binding* for a *function name* in the *lexical environment* can be *established* by **flet** or **labels**.

The next figure lists some *defined names* that are applicable to *functions*.

```
apply                 fdefinition  mapcan
call-arguments-limit  flet         mapcar
complement            fmakunbound  mapcon
constantly            funcall      mapl
defgeneric            function     maplist
defmethod             functionp    multiple-value-call
defun                 labels       reduce
fboundp               map          symbol-function
```

Figure 3-4. Some function-related defined names

3.1.2.1.2.4 Lambda Forms

A *lambda form* is similar to a *function form*, except that the *function name* is replaced by a *lambda expression*.

A *lambda form* is equivalent to using *funcall* of a *lexical closure* of the *lambda expression* on the given *arguments*. (In practice, some compilers are more likely to produce inline code for a *lambda form* than for an arbitrary named function that has been declared **inline**; however, such a difference is not semantic.)

For further information, see Section 3.1.3 (Lambda Expressions).

3.1.2.1.3 Self-Evaluating Objects

A *form* that is neither a *symbol* nor a *cons* is defined to be a *self-evaluating object*. *Evaluating* such an *object yields* the *same object* as a result.

Certain specific *symbols* and *conses* might also happen to be "self-evaluating" but only as a special case of a more general set of rules for the *evaluation* of *symbols* and *conses*; such *objects* are not considered to be *self-evaluating objects*.

The consequences are undefined if *literal objects* (including *self-evaluating objects*) are destructively modified.

3.1.2.1.3.1 Examples of Self-Evaluating Objects

Numbers, *pathnames*, and *arrays* are examples of *self-evaluating objects*.

```
3 =>  3
#c(2/3 5/8) =>  #C(2/3 5/8)
#p"S:[BILL]OTHELLO.TXT" =>  #P"S:[BILL]OTHELLO.TXT"
#(a b c) =>  #(A B C)
"fred smith" =>  "fred smith"
```

3.1.3 Lambda Expressions

In a *lambda expression*, the body is evaluated in a lexical *environment* that is formed by adding the *binding* of each *parameter* in the *lambda list* with the corresponding *value* from the *arguments* to the current lexical *environment*.

For further discussion of how *bindings* are *established* based on the *lambda list*, see Section 3.4 (Lambda Lists).

The body of a *lambda expression* is an *implicit progn*; the *values* it returns are returned by the *lambda expression*.

3.1.4 Closures and Lexical Binding

A *lexical closure* is a *function* that can refer to and alter the values of *lexical bindings established* by *binding forms* that textually include the function definition.

Consider this code, where `x` is not declared **special**:

```
(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** *special form* coerces a *lambda expression* into a *closure* in which the *lexical environment* in effect when the *special form* is evaluated is captured along with the *lambda expression*.

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. This variable has the value `6` initially, but **setq** can alter this *binding*. The *lexical closure* created for the first *lambda expression* does not "snapshot" the *value* `6` for `x` when the *closure* is created; rather it captures the *binding* of `x`. The second *function* can be used to alter the *value* in the same (captured) *binding* (to `43`, in the example), and this altered variable binding then affects the value returned by 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 *identical*, at the discretion of the *implementation*. That is, two *functions* that are behaviorally indistinguishable might or might not be *identical*. Two *functions* that are behaviorally distinguishable are *distinct*. 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 *form* is a *list* of ten *closures*. Each requires only the *binding* of `x`. It is the same *binding* in each case, but the ten *closure objects* might or might not be *identical*. On the other hand, the result of the *form*

```
(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 *closure objects* can be *identical* because each *closure* is closed over a distinct *binding* of `x`, and these *bindings* can be behaviorally distinguished because of the use of **setq**.

The result of the *form*

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

is a *list* of ten *closure objects* that might or might not be *identical*. A different *binding* of `x` is involved for each *closure*, but the *bindings* cannot be distinguished because their values are the *same* and immutable (there being no occurrence of **setq** on `x`). A compiler could internally transform the *form* to

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

where the *closures* may be *identical*.

It is possible that a *closure* does not close over any variable bindings. In the code fragment

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

the function `(lambda (x) (+ x 2))` contains no references to any outside object. In this case, the same *closure* might be returned for all evaluations of the **function** *form*.

3.1.5 Shadowing

If two *forms* that *establish lexical bindings* with the same *name* N are textually nested, then references to N within the inner *form* refer to the *binding* established by the inner *form*; the inner *binding* for N *shadows* the outer *binding* for N. Outside the inner *form* but inside the outer one, references to N refer to the *binding* established by the outer

form. For example:

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

The *binding* of the variable `z` by **let** 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 `test` refers to the *parameter* named `z`.

Constructs that are lexically scoped act as if new names were generated for each *object* on each execution. Therefore, dynamic shadowing cannot occur. 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 `4`. During the course of execution, there are three calls to `contorted-example`, interleaved with two blocks:

```
(contorted-example nil nil 2)
  (block here1 ...)
    (contorted-example nil #'(lambda () (return-from here1 4)) 1)
      (block here2 ...)
        (contorted-example #'(lambda () (return-from here1 4))
                           #'(lambda () (return-from here2 4))
                           0)
            (funcall f)
                   where f =>  #'(lambda () (return-from here1 4))
                (return-from here1 4)
```

At the time the `funcall` is executed there are two **block** *exit points* outstanding, each apparently named `here`. 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). It refers to that *exit point* textually visible at the point of execution of **function** (here abbreviated by the `#'` syntax) that resulted in creation of the *function object* actually invoked by **funcall**.

If, in this example, one were to change the `(funcall f)` to `(funcall g)`, then the value of the call `(contorted-example nil nil 2)` would be `9`. The value would change because **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 environment that is packaged up with a *lambda expression* when **function** is executed.

3.1.6 Extent

`Contorted-example` works only because the *function* named by `f` is invoked during the *extent* of the *exit point*. Once the flow of execution has left the block, the *exit point* is *disestablished*. For example:

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

One might expect the call `(invalid-example)` to produce `5` by the following incorrect reasoning: **let** binds `y` to the value of **block**; 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 `invalid-example`.

The argument fails only because *exit points* have *dynamic extent*. The argument is correct up to the execution of **return-from**. The execution of **return-from** should signal an error of *type* **control-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*.

A reference by name to a dynamic *exit point* binding such as a *catch tag* refers to the most recently *established binding* of that name that has not 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 is `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 **catch** 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 *exit points* would have different *names*, and therefore the one in `fun1` would not be shadowed. The result would then have been `7`.

3.1.7 Return Values

Ordinarily the result of calling a *function* is a single *object*. Sometimes, however, it is convenient for a function to compute several *objects* and return them.

In order to receive other than exactly one value from a *form*, one of several *special forms* or *macros* must be used to request those values. If a *form* produces *multiple values* which were not requested in this way, then the first value is given to the caller and all others are discarded; if the *form* produces zero values, then the caller receives **nil** as a value.

The next figure lists some *operators* for receiving *multiple values*[2]. These *operators* can be used to specify one or more *forms* to *evaluate* and where to put the *values* returned by those *forms*.

```
multiple-value-bind  multiple-value-prog1  return-from
multiple-value-call  multiple-value-setq   throw
multiple-value-list  return
```

Figure 3-5. Some operators applicable to receiving multiple values

The *function* **values** can produce *multiple values*[2]. `(values)` returns zero values; `(values` *form*`)` returns the *primary value* returned by *form*; `(values` *form1 form2*`)` returns two values, the *primary value* of *form1* and the *primary value* of *form2*; and so on.

See **multiple-values-limit** and **values-list**.

3.2 Compilation

3.2.1 Compiler Terminology

The following terminology is used in this section.

The *compiler* is a utility that translates code into an *implementation-dependent* form that might be represented or executed efficiently. The term *compiler* refers to both of the *functions* **compile** and **compile-file**.

The term *compiled code* refers to *objects* representing compiled programs, such as *objects* constructed by **compile** or by **load** when *loading* a *compiled file*.

The term *implicit compilation* refers to *compilation* performed during *evaluation*.

The term *literal object* refers to a quoted *object* or a *self-evaluating object* or an *object* that is a substructure of such an *object*. A *constant variable* is not itself a *literal object*.

The term *coalesce* is defined as follows. Suppose A and B are two *literal constants* in the *source code*, and that A′ and B′ are the corresponding *objects* in the *compiled code*. If A′ and B′ are **eql** but A and B are not **eql**, then it is said that A and B have been coalesced by the compiler.

The term *minimal compilation* refers to actions the compiler must take at *compile time*. These actions are specified in Section 3.2.2 (Compilation Semantics).

The verb *process* refers to performing *minimal compilation*, determining the time of evaluation for a *form*, and possibly *evaluating* that *form* (if required).

The term *further compilation* refers to *implementation-dependent* compilation beyond *minimal compilation*. That is, *processing* does not imply complete compilation. Block compilation and generation of machine-specific instructions are examples of further compilation. Further compilation is permitted to take place at *run time*.

Four different *environments* relevant to compilation are distinguished: the *startup environment*, the *compilation environment*, the *evaluation environment*, and the *run-time environment*.

The *startup environment* is the *environment* of the *Lisp image* from which the *compiler* was invoked.

The *compilation environment* is maintained by the compiler and is used to hold definitions and declarations to be used internally by the compiler. Only those parts of a definition needed for correct compilation are saved. The *compilation environment* is used as the *environment argument* to macro expanders called by the compiler. It is unspecified whether a definition available in the *compilation environment* can be used in an *evaluation* initiated in the *startup environment* or *evaluation environment*.

The *evaluation environment* is a *run-time environment* in which macro expanders and code specified by **eval-when** to be evaluated are evaluated. All evaluations initiated by the *compiler* take place in the *evaluation environment*.

The *run-time environment* is the *environment* in which the program being compiled will be executed.

The *compilation environment* inherits from the *evaluation environment*, and the *compilation environment* and *evaluation environment* might be *identical*. The *evaluation environment* inherits from the *startup environment*, and the *startup environment* and *evaluation environment* might be *identical*.

The term *compile time* refers to the duration of time that the compiler is processing *source code*. At *compile time*, only the *compilation environment* and the *evaluation environment* are available.

The term *compile-time definition* refers to a definition in the *compilation environment*. For example, when compiling a file, the definition of a function might be retained in the *compilation environment* if it is declared **inline**. This definition might not be available in the *evaluation environment*.

The term *run time* refers to the duration of time that the loader is loading compiled code or compiled code is being executed. At run time, only the *run-time environment* is available.

The term *run-time definition* refers to a definition in the *run-time environment*.

The term *run-time compiler* refers to the *function* **compile** or *implicit compilation*, for which the compilation and run-time *environments* are maintained in the same *Lisp image*. Note that when the *run-time compiler* is used, the *run-time environment* and *startup environment* are the same.

3.2.2 Compilation Semantics

Conceptually, compilation is a process that traverses code, performs certain kinds of syntactic and semantic analyses using information (such as proclamations and *macro* definitions) present in the *compilation environment*, and produces equivalent, possibly more efficient code.

3.2.2.1 Compiler Macros

A *compiler macro* can be defined for a *name* that also names a *function* or *macro*. That is, it is possible for a *function name* to name both a *function* and a *compiler macro*.

A *function name* names a *compiler macro* if **compiler-macro-function** is *true* of the *function name* in the *lexical environment* in which it appears. Creating a *lexical binding* for the *function name* not only creates a new local *function* or *macro* definition, but also *shadows*[2] the *compiler macro*.

The *function* returned by **compiler-macro-function** is a *function* of two arguments, called the expansion function. To expand a *compiler macro*, the expansion function is invoked by calling the *macroexpand hook* with the expansion function as its first argument, the entire compiler macro *form* as its second argument, and the current compilation *environment* (or with the current lexical *environment*, if the *form* is being processed by something other than **compile-file**) as its third argument. The *macroexpand hook*, in turn, calls the expansion function with the *form* as its first argument and the *environment* as its second argument. The return value from the expansion function, which is passed through by the *macroexpand hook*, might either be the *same form*, or else a form that can, at the discretion of the *code* doing the expansion, be used in place of the original *form*.

`*macroexpand-hook*  compiler-macro-function  define-compiler-macro`

Figure 3-6. Defined names applicable to compiler macros

3.2.2.1.1 Purpose of Compiler Macros

The purpose of the *compiler macro* facility is to permit selective source code transformations as optimization advice to the *compiler*. When a *compound form* is being processed (as by the compiler), if the *operator* names a *compiler macro* then the *compiler macro function* may be invoked on the form, and the resulting expansion recursively processed in preference to performing the usual processing on the original *form* according to its normal interpretation as a *function form* or *macro form*.

A *compiler macro function*, like a *macro function*, is a *function* of two *arguments*: the entire call *form* and the *environment*. Unlike an ordinary *macro function*, a *compiler macro function* can decline to provide an expansion merely by returning a value that is the *same* as the original *form*. The consequences are undefined if a *compiler macro function* destructively modifies any part of its *form* argument.

The *form* passed to the compiler macro function can either be a *list* whose *car* is the function name, or a *list* whose *car* is **funcall** and whose *cadr* is a list `(function` *name*`)`; note that this affects destructuring of the form argument by the *compiler macro function*. **define-compiler-macro** arranges for destructuring of arguments to be performed correctly for both possible formats.

When **compile-file** chooses to expand a *top level form* that is a *compiler macro form*, the expansion is also treated as a *top level form* for the purposes of **eval-when** processing; see Section 3.2.3.1 (Processing of Top Level Forms).

3.2.2.1.2 Naming of Compiler Macros

Compiler macros may be defined for *function names* that name *macros* as well as *functions*.

Compiler macro definitions are strictly global. There is no provision for defining local *compiler macros* in the way that **macrolet** defines local *macros*. Lexical bindings of a function name shadow any compiler macro definition associated with the name as well as its global *function* or *macro* definition.

Note that the presence of a compiler macro definition does not affect the values returned by functions that access *function* definitions (e.g., **fboundp**) or *macro* definitions (e.g., **macroexpand**). Compiler macros are global, and the function **compiler-macro-function** is sufficient to resolve their interaction with other lexical and global definitions.

3.2.2.1.3 When Compiler Macros Are Used

The presence of a *compiler macro* definition for a *function* or *macro* indicates that it is desirable for the *compiler* to use the expansion of the *compiler macro* instead of the original *function form* or *macro form*. However, no language processor (compiler, evaluator, or other code walker) is ever required to actually invoke *compiler macro functions*, or to make use of the resulting expansion if it does invoke a *compiler macro function*.

When the *compiler* encounters a *form* during processing that represents a call to a *compiler macro name* (that is not declared **notinline**), the *compiler* might expand the *compiler macro*, and might use the expansion in place of the original *form*.

When **eval** encounters a *form* during processing that represents a call to a *compiler macro name* (that is not declared **notinline**), **eval** might expand the *compiler macro*, and might use the expansion in place of the original *form*.

There are two situations in which a *compiler macro* definition must not be applied by any language processor:

- The global function name binding associated with the compiler macro is shadowed by a lexical binding of the function name.
- The function name has been declared or proclaimed **notinline** and the call form appears within the scope of the declaration.

It is unspecified whether *compiler macros* are expanded or used in any other situations.

3.2.2.1.3.1 Notes about the Implementation of Compiler Macros

Although it is technically permissible, as described above, for **eval** to treat *compiler macros* in the same situations as *compiler* might, this is not necessarily a good idea in *interpreted implementations*.

Compiler macros exist for the purpose of trading compile-time speed for run-time speed. Programmers who write *compiler macros* tend to assume that the *compiler macros* can take more time than normal *functions* and *macros* in order to produce code which is especially optimal for use at run time. Since **eval** in an *interpreted implementation* might perform semantic analysis of the same form multiple times, it might be inefficient in general for the

implementation to choose to call *compiler macros* on every such *evaluation*.

Nevertheless, the decision about what to do in these situations is left to each *implementation*.

3.2.2.2 Minimal Compilation

Minimal compilation is defined as follows:

- All *compiler macro* calls appearing in the *source code* being compiled are expanded, if at all, at compile time; they will not be expanded at run time.
- All *macro* and *symbol macro* calls appearing in the source code being compiled are expanded at compile time in such a way that they will not be expanded again at run time. **macrolet** and **symbol-macrolet** are effectively replaced by *forms* corresponding to their bodies in which calls to *macros* are replaced by their expansions.
- The first *argument* in a **load-time-value** *form* in *source code* processed by **compile** is *evaluated* at *compile time*; in *source code* processed by **compile-file**, the compiler arranges for it to be *evaluated* at *load time*. In either case, the result of the *evaluation* is remembered and used later as the value of the **load-time-value** *form* at *execution time*.

3.2.2.3 Semantic Constraints

All *conforming programs* must obey the following constraints, which are designed to minimize the observable differences between compiled and interpreted programs:

- Definitions of any referenced *macros* must be present in the *compilation environment*. Any *form* that is a *list* beginning with a *symbol* that does not name a *special operator* or a *macro* defined in the *compilation environment* is treated by the compiler as a function call.
- **Special** proclamations for *dynamic variables* must be made in the *compilation environment*. Any *binding* for which there is no **special** declaration or proclamation in the *compilation environment* is treated by the compiler as a *lexical binding*.
- The definition of a function that is defined and declared **inline** in the *compilation environment* must be the same at run time.
- Within a *function* named F, the compiler may (but is not required to) assume that an apparent recursive call to a *function* named F refers to the same definition of F, unless that function has been declared **notinline**. The consequences of redefining such a recursively defined *function* F while it is executing are undefined.
- A call within a file to a named function that is defined in the same file refers to that function, unless that function has been declared **notinline**. The consequences are unspecified if functions are redefined individually at run time or multiply defined in the same file.
- The argument syntax and number of return values for all functions whose **ftype** is declared at compile time must remain the same at run time.
- *Constant variables* defined in the *compilation environment* must have a *similar* value at run time. A reference to a *constant variable* in *source code* is equivalent to a reference to a *literal object* that is the *value* of the *constant variable*.
- Type definitions made with **deftype** or **defstruct** in the *compilation environment* must retain the same definition at run time. Classes defined by **defclass** in the *compilation environment* must be defined at run time to have the same *superclasses* and same *metaclass*.

  This implies that *subtype*/*supertype* relationships of *type specifiers* must not change between *compile time* and *run time*.

- Type declarations present in the compilation *environment* must accurately describe the corresponding values at run time; otherwise, the consequences are undefined. It is permissible for an unknown *type* to appear in a declaration at compile time, though a warning might be signaled in such a case.
- Except in the situations explicitly listed above, a *function* defined in the *evaluation environment* is permitted

to have a different definition or a different *signature* at run time, and the run-time definition prevails.

Conforming programs should not be written using any additional assumptions about consistency between the run-time *environment* and the startup, evaluation, and compilation *environments*.

Except where noted, when a compile-time and a run-time definition are different, one of the following occurs at run time:

- an error of *type* **error** is signaled
- the compile-time definition prevails
- the run-time definition prevails

If the *compiler* processes a *function form* whose *operator* is not defined at compile time, no error is signaled at compile time.

3.2.3 File Compilation

The *function* **compile-file** performs compilation of *forms* in a file following the rules specified in Section 3.2.2 (Compilation Semantics), and produces an output file that can be loaded by using **load**.

Normally, the *top level forms* appearing in a file compiled with **compile-file** are evaluated only when the resulting compiled file is loaded, and not when the file is compiled. However, it is typically the case that some forms in the file need to be evaluated at compile time so the remainder of the file can be read and compiled correctly.

The **eval-when** *special form* can be used to control whether a *top level form* is evaluated at compile time, load time, or both. It is possible to specify any of three situations with **eval-when**, denoted by the symbols `:compile-toplevel`, `:load-toplevel`, and `:execute`. For top level **eval-when** forms, `:compile-toplevel` specifies that the compiler must evaluate the body at compile time, and `:load-toplevel` specifies that the compiler must arrange to evaluate the body at load time. For non-top level **eval-when** forms, `:execute` specifies that the body must be executed in the run-time *environment*.

The behavior of this *form* can be more precisely understood in terms of a model of how **compile-file** processes forms in a file to be compiled. There are two processing modes, called "not-compile-time" and "compile-time-too".

Successive forms are read from the file by **compile-file** and processed in not-compile-time mode; in this mode, **compile-file** arranges for forms to be evaluated only at load time and not at compile time. When **compile-file** is in compile-time-too mode, forms are evaluated both at compile time and load time.

3.2.3.1 Processing of Top Level Forms

Processing of *top level forms* in the file compiler is defined as follows:

1. If the *form* is a *compiler macro form* (not disabled by a **notinline** *declaration*), the *implementation* might or might not choose to compute the *compiler macro expansion* of the *form* and, having performed the expansion, might or might not choose to process the result as a *top level form* in the same processing mode (compile-time-too or not-compile-time). If it declines to obtain or use the expansion, it must process the original *form*.
2. If the form is a *macro form*, its *macro expansion* is computed and processed as a *top level form* in the same processing mode (compile-time-too or not-compile-time).
3. If the form is a **progn** form, each of its body *forms* is sequentially processed as a *top level form* in the same processing mode.
4. If the form is a **locally**, **macrolet**, or **symbol-macrolet**, **compile-file** establishes the appropriate bindings and processes the body forms as *top level forms* with those bindings in effect in the same processing mode. (Note that this implies that the lexical *environment* in which *top level forms* are processed is not necessarily the *null*

lexical environment.)

5. If the form is an **eval-when** form, it is handled according to the next figure.

   plus .5 fil

CT	LT	E	Mode	Action	New Mode
Yes	Yes	---	---	Process	compile-time-too
No	Yes	Yes	CTT	Process	compile-time-too
No	Yes	Yes	NCT	Process	not-compile-time
No	Yes	No	---	Process	not-compile-time
Yes	No	---	---	Evaluate	---
No	No	Yes	CTT	Evaluate	---
No	No	Yes	NCT	Discard	---
No	No	No	---	Discard	---

   Figure 3-7. EVAL-WHEN processing

   Column **CT** indicates whether `:compile-toplevel` is specified. Column **LT** indicates whether `:load-toplevel` is specified. Column **E** indicates whether `:execute` is specified. Column **Mode** indicates the processing mode; a dash (---) indicates that the processing mode is not relevant.

   The **Action** column specifies one of three actions:

   Process: process the body as *top level forms* in the specified mode.
   Evaluate: evaluate the body in the dynamic execution context of the compiler, using the *evaluation environment* as the global environment and the *lexical environment* in which the **eval-when** appears.
   Discard: ignore the *form*.

   The **New Mode** column indicates the new processing mode. A dash (---) indicates the compiler remains in its current mode.

6. Otherwise, the form is a *top level form* that is not one of the special cases. In compile-time-too mode, the compiler first evaluates the form in the evaluation *environment* and then minimally compiles it. In not-compile-time mode, the *form* is simply minimally compiled. All *subforms* are treated as *non-top-level forms*.

   Note that *top level forms* are processed in the order in which they textually appear in the file and that each *top level form* read by the compiler is processed before the next is read. However, the order of processing (including macro expansion) of *subforms* that are not *top level forms* and the order of further compilation is unspecified as long as Common Lisp semantics are preserved.

eval-when forms cause compile-time evaluation only at top level. Both `:compile-toplevel` and `:load-toplevel` situation specifications are ignored for *non-top-level forms*. For *non-top-level forms*, an **eval-when** specifying the `:execute` situation is treated as an *implicit progn* including the *forms* in the body of the **eval-when** *form*; otherwise, the *forms* in the body are ignored.

3.2.3.1.1 Processing of Defining Macros

Defining *macros* (such as **defmacro** or **defvar**) appearing within a file being processed by **compile-file** normally have compile-time side effects which affect how subsequent *forms* in the same *file* are compiled. A convenient model for explaining how these side effects happen is that the defining macro expands into one or more **eval-when** *forms*, and that the calls which cause the compile-time side effects to happen appear in the body of an `(eval-when (:compile-toplevel) ...)` *form*.

The compile-time side effects may cause information about the definition to be stored differently than if the defining macro had been processed in the ‘normal’ way (either interpretively or by loading the compiled file).

In particular, the information stored by the defining *macros* at compile time might or might not be available to the interpreter (either during or after compilation), or during subsequent calls to the *compiler*. For example, the following code is nonportable because it assumes that the *compiler* stores the macro definition of `foo` where it is available to the interpreter:

```
(defmacro foo (x) '(car ,x))
(eval-when (:execute :compile-toplevel :load-toplevel)
  (print (foo '(a b c))))
```

A portable way to do the same thing would be to include the macro definition inside the **eval-when** *form*, as in:

```
(eval-when (:execute :compile-toplevel :load-toplevel)
  (defmacro foo (x) '(car ,x))
  (print (foo '(a b c))))
```

The next figure lists macros that make definitions available both in the compilation and run-time *environments*. It is not specified whether definitions made available in the *compilation environment* are available in the evaluation *environment*, nor is it specified whether they are available in subsequent compilation units or subsequent invocations of the compiler. As with **eval-when**, these compile-time side effects happen only when the defining macros appear at top level.

`declaim`	`define-modify-macro`	`defsetf`
`defclass`	`define-setf-expander`	`defstruct`
`defconstant`	`defmacro`	`deftype`
`define-compiler-macro`	`defpackage`	`defvar`
`define-condition`	`defparameter`	

Figure 3-8. Defining Macros That Affect the Compile-Time Environment

3.2.3.1.2 Constraints on Macros and Compiler Macros

Except where explicitly stated otherwise, no *macro* defined in the Common Lisp standard produces an expansion that could cause any of the *subforms* of the *macro form* to be treated as *top level forms*. If an *implementation* also provides a *special operator* definition of a Common Lisp *macro*, the *special operator* definition must be semantically equivalent in this respect.

Compiler macro expansions must also have the same top level evaluation semantics as the *form* which they replace. This is of concern both to *conforming implementations* and to *conforming programs*.

3.2.4 Literal Objects in Compiled Files

The functions **eval** and **compile** are required to ensure that *literal objects* referenced within the resulting interpreted or compiled code objects are the *same* as the corresponding *objects* in the *source code*. **compile-file**, on the other hand, must produce a *compiled file* that, when loaded with **load**, constructs the *objects* defined by the *source code* and produces references to them.

In the case of **compile-file**, *objects* constructed by **load** of the *compiled file* cannot be spoken of as being the *same* as the *objects* constructed at compile time, because the *compiled file* may be loaded into a different *Lisp image* than the one in which it was compiled. This section defines the concept of *similarity* which relates *objects* in the *evaluation environment* to the corresponding *objects* in the *run-time environment*.

The constraints on *literal objects* described in this section apply only to **compile-file**; **eval** and **compile** do not copy or coalesce constants.

3.2.4.1 Externalizable Objects

The fact that the *file compiler* represents *literal objects* externally in a *compiled file* and must later reconstruct suitable equivalents of those *objects* when that *file* is loaded imposes a need for constraints on the nature of the *objects* that can be used as *literal objects* in *code* to be processed by the *file compiler*.

An *object* that can be used as a *literal object* in *code* to be processed by the *file compiler* is called an *externalizable object*.

We define that two *objects* are *similar* if they satisfy a two-place conceptual equivalence predicate (defined below), which is independent of the *Lisp image* so that the two *objects* in different *Lisp images* can be understood to be equivalent under this predicate. Further, by inspecting the definition of this conceptual predicate, the programmer can anticipate what aspects of an *object* are reliably preserved by *file compilation*.

The *file compiler* must cooperate with the *loader* in order to assure that in each case where an *externalizable object* is processed as a *literal object*, the *loader* will construct a *similar object*.

The set of *objects* that are *externalizable objects* are those for which the new conceptual term "*similar*" is defined, such that when a *compiled file* is *loaded*, an *object* can be constructed which can be shown to be *similar* to the original *object* which existed at the time the *file compiler* was operating.

3.2.4.2 Similarity of Literal Objects

3.2.4.2.1 Similarity of Aggregate Objects

Of the *types* over which *similarity* is defined, some are treated as aggregate objects. For these types, *similarity* is defined recursively. We say that an *object* of these types has certain "basic qualities" and to satisfy the *similarity* relationship, the values of the corresponding qualities of the two *objects* must also be similar.

3.2.4.2.2 Definition of Similarity

Two *objects* S (in *source code*) and C (in *compiled code*) are defined to be *similar* if and only if they are both of one of the *types* listed here (or defined by the *implementation*) and they both satisfy all additional requirements of *similarity* indicated for that *type*.

number
Two *numbers* S and C are *similar* if they are of the same *type* and represent the same mathematical value.

character
Two *simple characters* S and C are *similar* if they have *similar code attributes*.

Implementations providing additional, *implementation-defined attributes* must define whether and how *non-simple characters* can be regarded as *similar*.

symbol
Two *apparently uninterned symbols* S and C are *similar* if their *names* are *similar*.

Two *interned* symbols S and C are *similar* if their *names* are *similar*, and if either S is accessible in the *current package* at compile time and C is accessible in the *current package* at load time, or C is accessible in the *package* that is *similar* to the *home package* of S.

(Note that *similarity* of *symbols* is dependent on neither the *current readtable* nor how the *function* **read** would parse the *characters* in the *name* of the *symbol*.)

package

Two *packages* S and C are *similar* if their *names* are *similar*.

Note that although a *package object* is an *externalizable object*, the programmer is responsible for ensuring that the corresponding *package* is already in existence when code referencing it as a *literal object* is *loaded*. The *loader* finds the corresponding *package object* as if by calling **find-package** with that *name* as an *argument*. An error is signaled by the *loader* if no *package* exists at load time.

random-state

Two *random states* S and C are *similar* if S would always produce the same sequence of pseudo-random numbers as a *copy*[5] of C when given as the *random-state argument* to the *function* **random**, assuming equivalent *limit arguments* in each case.

(Note that since C has been processed by the *file compiler*, it cannot be used directly as an *argument* to **random** because **random** would perform a side effect.)

cons

Two *conses*, S and C, are *similar* if the *car*[2] of S is *similar* to the *car*[2] of C, and the *cdr*[2] of S is *similar* to the *cdr*[2] of C.

array

Two one-dimensional *arrays*, S and C, are *similar* if the *length* of S is *similar* to the *length* of C, the *actual array element type* of S is *similar* to the *actual array element type* of C, and each *active element* of S is *similar* to the corresponding *element* of C.

Two *arrays* of *rank* other than one, S and C, are *similar* if the *rank* of S is *similar* to the *rank* of C, each *dimension*[1] of S is *similar* to the corresponding *dimension*[1] of C, the *actual array element type* of S is *similar* to the *actual array element type* of C, and each *element* of S is *similar* to the corresponding *element* of C.

In addition, if S is a *simple array*, then C must also be a *simple array*. If S is a *displaced array*, has a *fill pointer*, or is *actually adjustable*, C is permitted to lack any or all of these qualities.

hash-table

Two *hash tables* S and C are *similar* if they meet the following three requirements:

1. They both have the same test (e.g., they are both **eql** *hash tables*).
2. There is a unique one-to-one correspondence between the keys of the two *hash tables*, such that the corresponding keys are *similar*.
3. For all keys, the values associated with two corresponding keys are *similar*.

If there is more than one possible one-to-one correspondence between the keys of S and C, the consequences are unspecified. A *conforming program* cannot use a table such as S as an *externalizable constant*.

pathname

Two *pathnames* S and C are *similar* if all corresponding *pathname components* are *similar*.

function

Functions are not *externalizable objects*.

structure-object and **standard-object**

A general-purpose concept of *similarity* does not exist for *structures* and *standard objects*. However, a *conforming program* is permitted to define a **make-load-form** *method* for any *class* K defined by that *program* that is a *subclass* of either **structure-object** or **standard-object**. The effect of such a *method* is to define that an *object* S of *type* K in *source code* is *similar* to an *object* C of *type* K in *compiled code* if C was constructed from *code* produced by calling **make-load-form** on S.

3.2.4.3 Extensions to Similarity Rules

Some *objects*, such as *streams*, **readtables**, and **methods** are not *externalizable objects* under the definition of similarity given above. That is, such *objects* may not portably appear as *literal objects* in *code* to be processed by the *file compiler*.

An *implementation* is permitted to extend the rules of similarity, so that other kinds of *objects* are *externalizable objects* for that *implementation*.

If for some kind of *object*, *similarity* is neither defined by this specification nor by the *implementation*, then the *file compiler* must signal an error upon encountering such an *object* as a *literal constant*.

3.2.4.4 Additional Constraints on Externalizable Objects

If two *literal objects* appearing in the source code for a single file processed with the *file compiler* are the *identical*, the corresponding *objects* in the *compiled code* must also be the *identical*. With the exception of *symbols* and *packages*, any two *literal objects* in *code* being processed by the *file compiler* may be *coalesced* if and only if they are *similar*; if they are either both *symbols* or both *packages*, they may only be *coalesced* if and only if they are *identical*.

Objects containing circular references can be *externalizable objects*. The *file compiler* is required to preserve **eql**ness of substructures within a *file*. Preserving **eql**ness means that subobjects that are the *same* in the *source code* must be the *same* in the corresponding *compiled code*.

In addition, the following are constraints on the handling of *literal objects* by the *file compiler*:

array: If an *array* in the source code is a *simple array*, then the corresponding *array* in the compiled code will also be a *simple array*. If an *array* in the source code is displaced, has a *fill pointer*, or is *actually adjustable*, the corresponding *array* in the compiled code might lack any or all of these qualities. If an *array* in the source code has a fill pointer, then the corresponding *array* in the compiled code might be only the size implied by the fill pointer.
packages: The loader is required to find the corresponding *package object* as if by calling **find-package** with the package name as an argument. An error of *type* **package-error** is signaled if no *package* of that name exists at load time.
random-state: A constant *random state* object cannot be used as the state argument to the *function* **random** because **random** modifies this data structure.
structure, standard-object: *Objects* of *type* **structure-object** and **standard-object** may appear in compiled constants if there is an appropriate **make-load-form** method defined for that *type*.

The *file compiler* calls **make-load-form** on any *object* that is referenced as a *literal object* if the *object* is a *generalized instance* of **standard-object**, **structure-object**, **condition**, or any of a (possibly empty) *implementation-dependent* set of other *classes*. The *file compiler* only calls **make-load-form** once for any given *object* within a single *file*.

symbol: In order to guarantee that *compiled files* can be *loaded* correctly, users must ensure that the *packages* referenced in those *files* are defined consistently at compile time and load time. *Conforming programs* must satisfy the following requirements:

1. The *current package* when a *top level form* in the *file* is processed by **compile-file** must be the same as the *current package* when the *code* corresponding to that *top level form* in the *compiled file* is executed by **load**. In particular:
   a. Any *top level form* in a *file* that alters the *current package* must change it to a *package* of the same *name* both at compile time and at load time.
   b. If the first *non-atomic top level form* in the *file* is not an **in-package** *form*, then the *current package* at the time **load** is called must be a *package* with the same *name* as the package that was the *current package* at the time **compile-file** was called.
2. For all *symbols* appearing lexically within a *top level form* that were *accessible* in the *package* that was

the *current package* during processing of that *top level form* at compile time, but whose *home package* was another *package*, at load time there must be a *symbol* with the same *name* that is *accessible* in both the load-time *current package* and in the *package* with the same *name* as the compile-time *home package*.

3. For all *symbols* represented in the *compiled file* that were *external symbols* in their *home package* at compile time, there must be a *symbol* with the same *name* that is an *external symbol* in the *package* with the same *name* at load time.

If any of these conditions do not hold, the *package* in which the *loader* looks for the affected *symbols* is unspecified. *Implementations* are permitted to signal an error or to define this behavior.

3.2.5 Exceptional Situations in the Compiler

compile and **compile-file** are permitted to signal errors and warnings, including errors due to compile-time processing of `(eval-when (:compile-toplevel) ...)` forms, macro expansion, and conditions signaled by the compiler itself.

Conditions of *type* **error** might be signaled by the compiler in situations where the compilation cannot proceed without intervention.

In addition to situations for which the standard specifies that *conditions* of *type* **warning** must or might be signaled, warnings might be signaled in situations where the compiler can determine that the consequences are undefined or that a run-time error will be signaled. Examples of this situation are as follows: violating type declarations, altering or assigning the value of a constant defined with **defconstant**, calling built-in Lisp functions with a wrong number of arguments or malformed keyword argument lists, and using unrecognized declaration specifiers.

The compiler is permitted to issue warnings about matters of programming style as conditions of *type* **style-warning**. Examples of this situation are as follows: redefining a function using a different argument list, calling a function with a wrong number of arguments, not declaring **ignore** of a local variable that is not referenced, and referencing a variable declared **ignore**.

Both **compile** and **compile-file** are permitted (but not required) to *establish* a *handler* for *conditions* of *type* **error**. For example, they might signal a warning, and restart compilation from some *implementation-dependent* point in order to let the compilation proceed without manual intervention.

Both **compile** and **compile-file** return three values, the second two indicating whether the source code being compiled contained errors and whether style warnings were issued.

Some warnings might be deferred until the end of compilation. See **with-compilation-unit**.

3.3 Declarations

Declarations provide a way of specifying information for use by program processors, such as the evaluator or the compiler.

Local declarations can be embedded in executable code using **declare**. *Global declarations*, or *proclamations*, are established by **proclaim** or **declaim**.

The **the** *special form* provides a shorthand notation for making a *local declaration* about the *type* of the *value* of a given *form*.

The consequences are undefined if a program violates a *declaration* or a *proclamation*.

3.3.1 Minimal Declaration Processing Requirements

In general, an *implementation* is free to ignore *declaration specifiers* except for the **declaration**, **notinline**, **safety**, and **special** *declaration specifiers*.

A **declaration** *declaration* must suppress warnings about unrecognized *declarations* of the kind that it declares. If an *implementation* does not produce warnings about unrecognized declarations, it may safely ignore this *declaration*.

A **notinline** *declaration* must be recognized by any *implementation* that supports inline functions or *compiler macros* in order to disable those facilities. An *implementation* that does not use inline functions or *compiler macros* may safely ignore this *declaration*.

A **safety** *declaration* that increases the current safety level must always be recognized. An *implementation* that always processes code as if safety were high may safely ignore this *declaration*.

A **special** *declaration* must be processed by all *implementations*.

3.3.2 Declaration Specifiers

A *declaration specifier* is an *expression* that can appear at top level of a **declare** expression or a **declaim** form, or as the argument to **proclaim**. It is a *list* whose *car* is a *declaration identifier*, and whose *cdr* is data interpreted according to rules specific to the *declaration identifier*.

3.3.3 Declaration Identifiers

The next figure shows a list of all *declaration identifiers* defined by this standard.

```
declaration     ignore     special
dynamic-extent  inline     type
ftype           notinline
ignorable       optimize
```

Figure 3-9. Common Lisp Declaration Identifiers

An implementation is free to support other (*implementation-defined*) *declaration identifiers* as well. A warning might be issued if a *declaration identifier* is not among those defined above, is not defined by the *implementation*, is not a *type name*, and has not been declared in a **declaration** *proclamation*.

3.3.3.1 Shorthand notation for Type Declarations

A *type specifier* can be used as a *declaration identifier*. (*type-specifier var**) is taken as shorthand for (type *type-specifier var**).

3.3.4 Declaration Scope

Declarations can be divided into two kinds: those that apply to the *bindings* of *variables* or *functions*; and those that do not apply to *bindings*.

A *declaration* that appears at the head of a binding *form* and applies to a *variable* or *function binding* made by that *form* is called a *bound declaration*; such a *declaration* affects both the *binding* and any references within the *scope* of the *declaration*.

Declarations that are not *bound declarations* are called *free declarations*.

A *free declaration* in a *form* F1 that applies to a *binding* for a *name* N *established* by some *form* F2 of which F1 is a *subform* affects only references to N within F1; it does not to apply to other references to N outside of F1, nor does it affect the manner in which the *binding* of N by F2 is *established*.

Declarations that do not apply to *bindings* can only appear as *free declarations*.

The *scope* of a *bound declaration* is the same as the *lexical scope* of the *binding* to which it applies; for *special variables*, this means the *scope* that the *binding* would have had had it been a *lexical binding*.

Unless explicitly stated otherwise, the *scope* of a *free declaration* includes only the body *subforms* of the *form* at whose head it appears, and no other *subforms*. The *scope* of *free declarations* specifically does not include *initialization forms* for *bindings* established by the *form* containing the *declarations*.

Some *iteration forms* include step, end-test, or result *subforms* that are also included in the *scope* of *declarations* that appear in the *iteration form*. Specifically, the *iteration forms* and *subforms* involved are:

- **do**, **do***: *step-forms*, *end-test-form*, and *result-forms*.
- **dolist**, **dotimes**: *result-form*
- **do-all-symbols**, **do-external-symbols**, **do-symbols**: *result-form*

3.3.4.1 Examples of Declaration Scope

Here is an example illustrating the *scope* of *bound declarations*.

```
(let ((x 1))                ;[1] 1st occurrence of x
  (declare (special x))     ;[2] 2nd occurrence of x
  (let ((x 2))              ;[3] 3rd occurrence of x
    (let ((old-x x)         ;[4] 4th occurrence of x
          (x 3))            ;[5] 5th occurrence of x
      (declare (special x)) ;[6] 6th occurrence of x
      (list old-x x))))     ;[7] 7th occurrence of x
=>  (2 3)
```

The first occurrence of `x` *establishes* a *dynamic binding* of `x` because of the **special** *declaration* for `x` in the second line. The third occurrence of `x` *establishes* a *lexical binding* of `x` (because there is no **special** *declaration* in the corresponding **let** *form*). The fourth occurrence of `x` *x* is a reference to the *lexical binding* of `x` established in the third line. The fifth occurrence of `x` *establishes* a *dynamic binding* of *x* for the body of the **let** *form* that begins on that line because of the **special** *declaration* for `x` in the sixth line. The reference to `x` in the fourth line is not affected by the **special** *declaration* in the sixth line because that reference is not within the "would-be *lexical scope*" of the *variable* `x` in the fifth line. The reference to `x` in the seventh line is a reference to the *dynamic binding* of *x established* in the fifth line.

Here is another example, to illustrate the *scope* of a *free declaration*. In the following:

```
(lambda (&optional (x (foo 1))) ;[1]
  (declare (notinline foo))     ;[2]
  (foo x))                      ;[3]
```

the *call* to `foo` in the first line might be compiled inline even though the *call* to `foo` in the third line must not be. This is because the **notinline** *declaration* for `foo` in the second line applies only to the body on the third line. In order to suppress inlining for both *calls*, one might write:

```
(locally (declare (notinline foo)) ;[1]
  (lambda (&optional (x (foo 1)))  ;[2]
    (foo x)))                      ;[3]
```

or, alternatively:

```
 (lambda (&optional                                ;[1]
           (x (locally (declare (notinline foo)) ;[2]
                (foo 1))))                       ;[3]
   (declare (notinline foo))                      ;[4]
   (foo x))                                       ;[5]
```

Finally, here is an example that shows the *scope* of *declarations* in an *iteration form*.

```
 (let ((x  1))                    ;[1]
   (declare (special x))          ;[2]
     (let ((x 2))                 ;[3]
       (dotimes (i x x)           ;[4]
         (declare (special x)))))  ;[5]
=>  1
```

In this example, the first reference to x on the fourth line is to the *lexical binding* of x established on the third line. However, the second occurrence of x on the fourth line lies within the *scope* of the *free declaration* on the fifth line (because this is the *result-form* of the **dotimes**) and therefore refers to the *dynamic binding* of x.

3.4 Lambda Lists

A *lambda list* is a *list* that specifies a set of *parameters* (sometimes called *lambda variables*) and a protocol for receiving *values* for those *parameters*.

There are several kinds of *lambda lists*.

Context	Kind of Lambda List
defun form	ordinary lambda list
defmacro form	macro lambda list
lambda expression	ordinary lambda list
flet local function definition	ordinary lambda list
labels local function definition	ordinary lambda list
handler-case clause specification	ordinary lambda list
restart-case clause specification	ordinary lambda list
macrolet local macro definition	macro lambda list
define-method-combination	ordinary lambda list
define-method-combination :arguments option	define-method-combination arguments lambda list
defstruct :constructor option	boa lambda list
defgeneric form	generic function lambda list
defgeneric method clause	specialized lambda list
defmethod form	specialized lambda list
defsetf form	defsetf lambda list
define-setf-expander form	macro lambda list
deftype form	deftype lambda list
destructuring-bind form	destructuring lambda list
define-compiler-macro form	macro lambda list
define-modify-macro form	define-modify-macro lambda list

Figure 3-10. What Kind of Lambda Lists to Use

The next figure lists some *defined names* that are applicable to *lambda lists*.

```
lambda-list-keywords  lambda-parameters-limit
```

Figure 3-11. Defined names applicable to lambda lists

3.4.1 Ordinary Lambda Lists

An *ordinary lambda list* is used to describe how a set of *arguments* is received by an *ordinary function*. The *defined names* in the next figure are those which use *ordinary lambda lists*:

```
define-method-combination  handler-case  restart-case
defun                      labels
flet                       lambda
```

Figure 3-12. Standardized Operators that use Ordinary Lambda Lists

An *ordinary lambda list* can contain the *lambda list keywords* shown in the next figure.

```
&allow-other-keys  &key       &rest
&aux               &optional
```

Figure 3-13. Lambda List Keywords used by Ordinary Lambda Lists

Each *element* of a *lambda list* is either a parameter specifier or a *lambda list keyword*. Implementations are free to provide additional *lambda list keywords*. For a list of all *lambda list keywords* used by the implementation, see **lambda-list-keywords**.

The syntax for *ordinary lambda lists* is as follows:

```
lambda-list::= (var*
                [&optional {var | (var [init-form [supplied-p-parameter]])}*]
                [&rest var]
                [&key {var | ({var | (keyword-name var)} [init-form [supplied-p-parameter]])}* [&allow-other-keys]]
                [&aux {var | (var [init-form])}*])
```

A *var* or *supplied-p-parameter* must be a *symbol* that is not the name of a *constant variable*.

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

A *keyword-name* can be any *symbol*, but by convention is normally a *keyword*[1]; all *standardized functions* follow that convention.

An *ordinary lambda list* has five parts, any or all of which may be empty. For information about the treatment of argument mismatches, see Section 3.5 (Error Checking in Function Calls).

3.4.1.1 Specifiers for the required parameters

These are all the parameter specifiers up to the first *lambda list keyword*; if there are no *lambda list keywords*, then all the specifiers are for required parameters. Each required parameter is specified by a parameter variable *var*. *var* is bound as a lexical variable unless it is declared **special**.

If there are n required parameters (n may be zero), there must be at least n passed arguments, and the required parameters are bound to the first n passed arguments; see Section 3.5 (Error Checking in Function Calls). The other parameters are then processed using any remaining arguments.

3.4.1.2 Specifiers for optional parameters

If `&optional` is present, the optional parameter specifiers are those following `&optional` up to the next *lambda list keyword* or the end of the list. 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 *init-form* is evaluated, and the parameter variable is bound to the resulting value (or to **nil** if no *init-form* appears in the parameter specifier). If another variable name *supplied-p-parameter* appears in the specifier, it is bound to *true* if an argument had been available, and to *false* if no argument remained (and therefore *init-form* had to be evaluated). *Supplied-p-parameter* is bound not to an argument but to a value indicating whether or not an argument had been supplied for the corresponding *var*.

3.4.1.3 A specifier for a rest parameter

`&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*. 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 an error should be signaled if any unprocessed arguments remain; see Section 3.5 (Error Checking in Function Calls). The value of a *rest parameter* is permitted, but not required, to share structure with the last argument to **apply**.

3.4.1.4 Specifiers for keyword parameters

If `&key` is present, all specifiers up to the next *lambda list keyword* or the end of the *list* are keyword parameter specifiers. When keyword parameters are processed, the same arguments are processed that would be made into a *list* for a *rest parameter*. 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. If `&key` is specified, there must remain an even number of arguments; see Section 3.5.1.6 (Odd Number of Keyword Arguments). These arguments are considered as pairs, the first argument in each pair being interpreted as a name and the second as the corresponding value. The first *object* of each pair must be a *symbol*; see Section 3.5.1.5 (Invalid Keyword Arguments). The keyword parameter specifiers may optionally be followed by the *lambda list keyword* `&allow-other-keys`.

In each keyword parameter specifier must be a name *var* for the parameter variable. If the *var* appears alone or in a `(`*var init-form*`)` combination, the keyword name used when matching *arguments* to *parameters* is a *symbol* in the `KEYWORD` package whose *name* is the *same* (under **string=**) as *var*'s. If the notation `((`*keyword-name var*`)` *init-form*`)` is used, then the keyword name used to match *arguments* to *parameters* is *keyword-name*, which may be a *symbol* in any *package*. (Of course, if it is not a *symbol* in the `KEYWORD` package, it does not necessarily self-evaluate, so care must be taken when calling the function to make sure that normal evaluation still yields the keyword name.) 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 name matches that specifier's 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, the leftmost argument pair is used. If no such argument pair exists, then the *init-form* for that specifier is evaluated and the parameter variable is bound to that value (or to **nil** if no *init-form* was specified). *supplied-p-parameter* is treated as for `&optional` parameters: it is

bound to *true* if there was a matching argument pair, and to *false* otherwise.

Unless keyword argument checking is suppressed, an argument pair must a name matched by a parameter specifier; see Section 3.5.1.4 (Unrecognized Keyword Arguments).

If keyword argument checking is suppressed, then it is permitted for an argument pair to match no parameter specifier, and the argument pair is ignored, but such an argument pair is accessible through the *rest parameter* if one was supplied. The purpose of these mechanisms is to allow sharing of argument lists among several *lambda expressions* and to allow either the caller or the called *lambda expression* to specify that such sharing may be taking place.

Note that if `&key` is present, a keyword argument of `:allow-other-keys` is always permitted---regardless of whether the associated value is *true* or *false*. However, if the value is *false*, other non-matching keywords are not tolerated (unless `&allow-other-keys` was used).

Furthermore, if the receiving argument list specifies a regular argument which would be flagged by `:allow-other-keys`, then `:allow-other-keys` has both its special-cased meaning (identifying whether additional keywords are permitted) and its normal meaning (data flow into the function in question).

3.4.1.4.1 Suppressing Keyword Argument Checking

If `&allow-other-keys` was specified in the *lambda list* of a *function*, *keyword*[2] *argument* checking is suppressed in calls to that *function*.

If the `:allow-other-keys` *argument* is *true* in a call to a *function*, *keyword*[2] *argument* checking is suppressed in that call.

The `:allow-other-keys` *argument* is permissible in all situations involving *keyword*[2] *arguments*, even when its associated *value* is *false*.

3.4.1.4.1.1 Examples of Suppressing Keyword Argument Checking

```
;;; The caller can supply :ALLOW-OTHER-KEYS T to suppress checking.
 ((lambda (&key x) x) :x 1 :y 2 :allow-other-keys t) =>  1
;;; The callee can use &ALLOW-OTHER-KEYS to suppress checking.
 ((lambda (&key x &allow-other-keys) x) :x 1 :y 2) =>  1
;;; :ALLOW-OTHER-KEYS NIL is always permitted.
 ((lambda (&key) t) :allow-other-keys nil) =>  T
;;; As with other keyword arguments, only the left-most pair
;;; named :ALLOW-OTHER-KEYS has any effect.
 ((lambda (&key x) x)
  :x 1 :y 2 :allow-other-keys t :allow-other-keys nil)
=>  1
;;; Only the left-most pair named :ALLOW-OTHER-KEYS has any effect,
;;; so in safe code this signals a PROGRAM-ERROR (and might enter the
;;; debugger).  In unsafe code, the consequences are undefined.
 ((lambda (&key x) x)                   ;This call is not valid
  :x 1 :y 2 :allow-other-keys nil :allow-other-keys t)
```

3.4.1.5 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. After all parameter specifiers have been processed, the auxiliary variable specifiers (those following `&aux`) are processed from left to right. For each one, *init-form* is evaluated and *var* is bound to that value (or to **nil** if no *init-form* was specified). `&aux` variable processing is analogous to **let*** processing.

```
 (lambda (x y &aux (a (car x)) (b 2) c) (list x y a b c))
    ==  (lambda (x y) (let* ((a (car x)) (b 2) c) (list x y a b c)))
```

3.4.1.6 Examples of Ordinary Lambda Lists

Here are some examples involving *optional parameters* 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))
```

Here are some examples involving *keyword 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)
 ((lambda (a b &key ((:sea c)) d) (list a b c d)) 1 2 :sea 6) =>  (1 2 6 NIL)
 ((lambda (a b &key ((c c)) d) (list a b c d)) 1 2 'c 6) =>  (1 2 6 NIL)
```

Here are some examples involving *optional parameters*, *rest parameters*, and *keyword parameters* together:

```
 ((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))
```

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

```
(defun array-of-strings (str dims &rest named-pairs
                         &key (start 0) end &allow-other-keys)
  (apply #'make-array dims
         :initial-element (subseq str start end)
         :allow-other-keys t
         named-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` named arguments may be used 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 supply additional named arguments; the *rest parameter* provides access to them. These additional named arguments are passed to **make-array**. The *function* **make-array** normally does not allow the named arguments `:start` and `:end` to be used, and an error should be signaled if such named arguments are supplied to **make-array**. However, the presence in the call to **make-array** of the named argument `:allow-other-keys` with a *true* value causes any extraneous named arguments, including `:start` and `:end`, to be acceptable and ignored.

3.4.2 Generic Function Lambda Lists

A *generic function lambda list* is used to describe the overall shape of the argument list to be accepted by a *generic function*. Individual *method signatures* might contribute additional *keyword parameters* to the *lambda list* of the *effective method*.

A *generic function lambda list* is used by **defgeneric**.

A *generic function lambda list* has the following syntax:

```
lambda-list::= (var*
                [&optional {var | (var)}*]
                [&rest var]
                [&key {var | ({var | (keyword-name var)})}* [&allow-other-keys]])
```

A *generic function lambda list* can contain the *lambda list keywords* shown in the next figure.

```
&allow-other-keys  &optional
&key               &rest
```

Figure 3-14. Lambda List Keywords used by Generic Function Lambda Lists

A *generic function lambda list* differs from an *ordinary lambda list* in the following ways:

Required arguments
: Zero or more *required parameters* must be specified.

Optional and keyword arguments
: *Optional parameters* and *keyword parameters* may not have default initial value forms nor use supplied-p parameters.

Use of `&aux`
: The use of `&aux` is not allowed.

3.4.3 Specialized Lambda Lists

A *specialized lambda list* is used to *specialize* a *method* for a particular *signature* and to describe how *arguments* matching that *signature* are received by the *method*. The *defined names* in the next figure use *specialized lambda lists* in some way; see the dictionary entry for each for information about how.

```
defmethod  defgeneric
```

Figure 3-15. Standardized Operators that use Specialized Lambda Lists

A *specialized lambda list* can contain the *lambda list keywords* shown in the next figure.

```
&allow-other-keys  &key       &rest
&aux               &optional
```

Figure 3-16. Lambda List Keywords used by Specialized Lambda Lists

A *specialized lambda list* is syntactically the same as an *ordinary lambda list* except that each *required parameter* may optionally be associated with a *class* or *object* for which that *parameter* is *specialized*.

```
lambda-list::= ({var | (var [specializer])}*
               [&optional {var | (var [init-form [supplied-p-parameter]])}*]
               [&rest var]
               [&key {var | ({var | (keyword-name var)} [init-form [supplied-p-parameter]])}* [&allow-other-keys]]
               [&aux {var | (var [init-form])}*])
```

3.4.4 Macro Lambda Lists

A *macro lambda list* is used in describing *macros* defined by the *operators* in the next figure.

```
define-compiler-macro  defmacro  macrolet
define-setf-expander
```

Figure 3-17. Operators that use Macro Lambda Lists

With the additional restriction that an *environment parameter* may appear only once (at any of the positions indicated), a *macro lambda list* has the following syntax:

```
reqvars::= var*

optvars::= [&optional {var | (var [init-form [supplied-p-parameter]])}*]

restvar::= [{&rest | &body} var]

keyvars::= [&key {var | ({var | (keyword-name var)} [init-form [supplied-p-parameter]])}*
            [&allow-other-keys]]

auxvars::= [&aux {var | (var [init-form])}*]

envvar::= [&environment var]

wholevar::= [&whole var]

lambda-list::= (wholevar envvar  reqvars envvar  optvars envvar
                restvar envvar  keyvars envvar  auxvars envvar) |
               (wholevar envvar  reqvars envvar  optvars envvar .  var)

pattern::= (wholevar reqvars optvars restvar keyvars auxvars) |
           (wholevar reqvars optvars . var)
```

A *macro lambda list* can contain the *lambda list keywords* shown in the next figure.

```
&allow-other-keys  &environment  &rest
&aux               &key          &whole
&body              &optional
```

Figure 3-18. Lambda List Keywords used by Macro Lambda Lists

Optional parameters (introduced by `&optional`) and *keyword parameters* (introduced by `&key`) can be supplied in a *macro lambda list*, just as in an *ordinary lambda list*. Both may contain default initialization forms and *supplied-p parameters*.

`&body` is identical in function to `&rest`, but it can be used to inform 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` can be used at any particular level; see Section 3.4.4.1 (Destructuring by Lambda Lists). `&body` can appear at any level of a *macro lambda list*; for details, see Section 3.4.4.1 (Destructuring by Lambda Lists).

`&whole` is followed by a single variable that is bound to the entire macro-call form; this is the value that the *macro function* receives as its first argument. If `&whole` and a following variable appear, they must appear first in *lambda-list*, before any other parameter or *lambda list keyword*. `&whole` can appear at any level of a *macro lambda list*. At inner levels, the `&whole` variable is bound to the corresponding part of the argument, as with `&rest`, but unlike `&rest`, other arguments are also allowed. The use of `&whole` does not affect the pattern of arguments specified.

`&environment` 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* should be used with **macro-function**, **get-setf-expansion**, **compiler-macro-function**, and **macroexpand** (for example) in computing the expansion of the macro, to ensure that any *lexical bindings* or definitions established in the *compilation environment* are taken into account. `&environment` can only appear at the top level of a *macro lambda list*, and can only appear once, but can appear anywhere in that list; the `&environment` *parameter* is *bound* along with `&whole` before any other *variables* in the *lambda list*, regardless of where `&environment` appears in the *lambda list*. The *object* that is bound to the *environment parameter* has *dynamic extent*.

Destructuring allows a *macro lambda list* to express the structure of a macro call syntax. If no *lambda list keywords* appear, then the *macro lambda list* is a *tree* containing parameter names at the leaves. The pattern and the *macro form* must have compatible *tree structure*; that is, their *tree structure* must be equivalent, or it must differ only in that some *leaves* of the pattern match *non-atomic objects* of the *macro form*. For information about error detection in this *situation*, see Section 3.5.1.7 (Destructuring Mismatch).

A destructuring *lambda list* (whether at top level or embedded) can 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`.

It is permissible for a *macro form* (or a *subexpression* of a *macro form*) to be a *dotted list* only when `(... &rest var)` or `(... . var)` is used to match it. It is the responsibility of the *macro* to recognize and deal with such situations.

3.4.4.1 Destructuring by Lambda Lists

Anywhere in a *macro lambda list* where a parameter name can appear, and where *ordinary lambda list* syntax (as described in Section 3.4.1 (Ordinary Lambda Lists)) does not otherwise allow a *list*, a *destructuring lambda list* can appear in place of the parameter name. When this is done, then the argument that would match the parameter is treated as a (possibly dotted) *list*, to be used as an argument list for satisfying the parameters in the embedded *lambda list*. This is known as destructuring.

Destructuring is the process of decomposing a compound *object* into its component parts, using an abbreviated, declarative syntax, rather than writing it out by hand using the primitive component-accessing functions. Each component part is bound to a variable.

A destructuring operation requires an *object* to be decomposed, a pattern that specifies what components are to be extracted, and the names of the variables whose values are to be the components.

3.4.4.1.1 Data-directed Destructuring by Lambda Lists

In data-directed destructuring, the pattern is a sample *object* of the *type* to be decomposed. Wherever a component is to be extracted, a *symbol* appears in the pattern; this *symbol* is the name of the variable whose value will be that component.

3.4.4.1.1.1 Examples of Data-directed Destructuring by Lambda Lists

An example pattern is

```
(a b c)
```

which destructures a list of three elements. The variable `a` is assigned to the first element, `b` to the second, etc. A more complex example is

```
((first . rest) . more)
```

The important features of data-directed destructuring are its syntactic simplicity and the ability to extend it to lambda-list-directed destructuring.

3.4.4.1.2 Lambda-list-directed Destructuring by Lambda Lists

An extension of data-directed destructuring of *trees* is lambda-list-directed destructuring. This derives from the analogy between the three-element destructuring pattern

```
(first second third)
```

and the three-argument *lambda list*

```
(first second third)
```

Lambda-list-directed destructuring is identical to data-directed destructuring if no *lambda list keywords* appear in the pattern. Any list in the pattern (whether a sub-list or the whole pattern itself) that contains a *lambda list keyword* is interpreted specially. Elements of the list to the left of the first *lambda list keyword* are treated as destructuring patterns, as usual, but the remaining elements of the list are treated like a function's *lambda list* except that where a variable would normally be required, an arbitrary destructuring pattern is allowed. Note that in case of ambiguity, *lambda list* syntax is preferred over destructuring syntax. Thus, after `&optional` a list of elements is a list of a destructuring pattern and a default value form.

The detailed behavior of each *lambda list keyword* in a lambda-list-directed destructuring pattern is as follows:

`&optional`
: Each following element is a variable or a list of a destructuring pattern, a default value form, and a supplied-p variable. The default value and the supplied-p variable can be omitted. If the list being destructured ends early, so that it does not have an element to match against this destructuring (sub)-pattern, the default form is evaluated and destructured instead. The supplied-p variable receives the value **nil** if the default form is used, **t** otherwise.

`&rest`, `&body`
: The next element is a destructuring pattern that matches the rest of the list. `&body` is identical to `&rest` but declares that what is being matched is a list of forms that constitutes the body of *form*. This next element must be the last unless a *lambda list keyword* follows it.

`&aux`
: The remaining elements are not destructuring patterns at all, but are auxiliary variable bindings.

`&whole`
: The next element is a destructuring pattern that matches the entire form in a macro, or the entire *subexpression* at inner levels.

`&key`
: Each following element is one of
a *variable*,
or a list of a variable, an optional initialization form, and an optional supplied-p variable.
or a list of a list of a keyword and a destructuring pattern, an optional initialization form, and an optional supplied-p variable.
The rest of the list being destructured is taken to be alternating keywords and values and is taken apart appropriately.

`&allow-other-keys`
: Stands by itself.

3.4.5 Destructuring Lambda Lists

A *destructuring lambda list* is used by **destructuring-bind**.

Destructuring lambda lists are closely related to *macro lambda lists*; see Section 3.4.4 (Macro Lambda Lists). A *destructuring lambda list* can contain all of the *lambda list keywords* listed for *macro lambda lists* except for `&environment`, and supports destructuring in the same way. Inner *lambda lists* nested within a *macro lambda list* have the syntax of *destructuring lambda lists*.

A *destructuring lambda list* has the following syntax:

```
reqvars::= var*

optvars::= [&optional {var | (var [init-form [supplied-p-parameter]])}*]

restvar::= [{&rest | &body} var]

keyvars::= [&key {var | ({var | (keyword-name var)} [init-form [supplied-p-parameter]])}*
            [&allow-other-keys]]

auxvars::= [&aux {var | (var [init-form])}*]

envvar::= [&environment var]

wholevar::= [&whole var]

lambda-list::= (wholevar reqvars optvars restvar keyvars auxvars) |
               (wholevar reqvars optvars . var)
```

3.4.6 Boa Lambda Lists

A *boa lambda list* is a *lambda list* that is syntactically like an *ordinary lambda list*, but that is processed in "**b**y **o**rder of **a**rgument" style.

A *boa lambda list* is used only in a **defstruct** *form*, when explicitly specifying the *lambda list* of a constructor *function* (sometimes called a "boa constructor").

The `&optional`, `&rest`, `&aux`, `&key`, and `&allow-other-keys` *lambda list keywords* are recognized in a *boa lambda list*. The way these *lambda list keywords* differ from their use in an *ordinary lambda list* follows.

Consider this example, which describes how **destruct** processes its `:constructor` option.

```
 (:constructor create-foo
         (a &optional b (c 'sea) &rest d &aux e (f 'eff)))
```

This defines `create-foo` to be a constructor of one or more arguments. The first argument is used to initialize the `a` slot. The second argument is used to initialize the `b` slot. If there isn't any second argument, then the default value given in the body of the **defstruct** (if given) is used instead. The third argument is used to initialize the `c` slot. If there isn't any third argument, then the symbol `sea` is used instead. Any arguments following the third argument are collected into a *list* and used to initialize the `d` slot. If there are three or fewer arguments, then **nil** is placed in the `d` slot. The `e` slot is not initialized; its initial value is *implementation-defined*. Finally, the `f` slot is initialized to contain the symbol `eff`. `&key` and `&allow-other-keys` arguments default in a manner similar to that of `&optional` arguments: if no default is supplied in the *lambda list* then the default value given in the body of the **defstruct** (if given) is used instead. For example:

```
 (defstruct (foo (:constructor CREATE-FOO (a &optional b (c 'sea)
                                             &key (d 2)
                                             &aux e (f 'eff))))
   (a 1) (b 2) (c 3) (d 4) (e 5) (f 6))

 (create-foo 10) =>  #S(FOO A 10 B 2 C SEA D 2 E implemention-dependent F EFF)
 (create-foo 10 'bee 'see :d 'dee)
=>  #S(FOO A 10 B BEE C SEE D DEE E implemention-dependent F EFF)
```

If keyword arguments of the form `((`*key var*`)` [*default* [*svar*]]`)` are specified, the *slot name* is matched with *var* (not *key*).

The actions taken in the `b` and `e` cases were carefully chosen to allow the user to specify all possible behaviors. The `&aux` variables can be used to completely override the default initializations given in the body.

If no default value is supplied for an *aux variable* variable, the consequences are undefined if an attempt is later made to read the corresponding *slot*'s value before a value is explicitly assigned. If such a *slot* has a `:type` option specified, this suppressed initialization does not imply a type mismatch situation; the declared type is only required to apply when the *slot* is finally assigned.

With this definition, the following can be written:

```
 (create-foo 1 2)
```

instead of

```
 (make-foo :a 1 :b 2)
```

and `create-foo` provides defaulting different from that of `make-foo`.

Additional arguments that do not correspond to slot names but are merely present to supply values used in subsequent initialization computations are allowed. For example, in the definition

```
 (defstruct (frob (:constructor create-frob
                  (a &key (b 3 have-b) (c-token 'c)
                          (c (list c-token (if have-b 7 2))))))
         a b c)
```

the `c-token` argument is used merely to supply a value used in the initialization of the `c` slot. The *supplied-p parameters* associated with *optional parameters* and *keyword parameters* might also be used this way.

3.4.7 Defsetf Lambda Lists

A *defsetf lambda list* is used by **defsetf**.

A *defsetf lambda list* has the following syntax:

```
lambda-list::= (var*
                [&optional {var | (var [init-form [supplied-p-parameter]])}*]
                [&rest var]
                [&key {var | ({var | (keyword-name var)} [init-form [supplied-p-parameter]])}* [&allow-other-keys]]
                [&environment var]
```

A *defsetf lambda list* can contain the *lambda list keywords* shown in the next figure.

```
&allow-other-keys  &key       &rest
&environment       &optional
```

Figure 3-19. Lambda List Keywords used by Defsetf Lambda Lists

A *defsetf lambda list* differs from an *ordinary lambda list* only in that it does not permit the use of `&aux`, and that it permits use of `&environment`, which introduces an *environment parameter*.

3.4.8 Deftype Lambda Lists

A *deftype lambda list* is used by **deftype**.

A *deftype lambda list* has the same syntax as a *macro lambda list*, and can therefore contain the *lambda list keywords* as a *macro lambda list*.

A *deftype lambda list* differs from a *macro lambda list* only in that if no *init-form* is supplied for an *optional parameter* or *keyword parameter* in the *lambda-list*, the default *value* for that *parameter* is the *symbol* ***** (rather than **nil**).

3.4.9 Define-modify-macro Lambda Lists

A *define-modify-macro lambda list* is used by **define-modify-macro**.

A *define-modify-macro lambda list* can contain the *lambda list keywords* shown in the next figure.

```
&optional  &rest
```

Figure 3-20. Lambda List Keywords used by Define-modify-macro Lambda Lists

Define-modify-macro lambda lists are similar to *ordinary lambda lists*, but do not support keyword arguments. **define-modify-macro** has no need match keyword arguments, and a *rest parameter* is sufficient. *Aux variables* are also not supported, since **define-modify-macro** has no body *forms* which could refer to such *bindings*. See the *macro* **define-modify-macro**.

3.4.10 Define-method-combination Arguments Lambda Lists

A *define-method-combination arguments lambda list* is used by the `:arguments` option to **define-method-combination**.

A *define-method-combination arguments lambda list* can contain the *lambda list keywords* shown in the next figure.

```
&allow-other-keys  &key       &rest
&aux               &optional  &whole
```

Figure 3-21. Lambda List Keywords used by Define-method-combination arguments Lambda Lists

Define-method-combination arguments lambda lists are similar to *ordinary lambda lists*, but also permit the use of `&whole`.

3.4.11 Syntactic Interaction of Documentation Strings and Declarations

In a number of situations, a *documentation string* can appear amidst a series of **declare** *expressions* prior to a series of *forms*.

In that case, if a *string* S appears where a *documentation string* is permissible and is not followed by either a **declare** *expression* or a *form* then S is taken to be a *form*; otherwise, S is taken as a *documentation string*. The consequences are unspecified if more than one such *documentation string* is present.

3.5 Error Checking in Function Calls

3.5.1 Argument Mismatch Detection

3.5.1.1 Safe and Unsafe Calls

A *call* is a *safe call* if each of the following is either *safe code* or *system code* (other than *system code* that results from *macro expansion* of *programmer code*):

* the *call*.
* the definition of the *function* being *called*.
* the point of *functional evaluation*

The following special cases require some elaboration:

* If the *function* being called is a *generic function*, it is considered *safe* if all of the following are *safe code* or *system code*:
    -- its definition (if it was defined explicitly).
    -- the *method* definitions for all *applicable methods*.
    -- the definition of its *method combination*.
* For the form `(coerce` *x* `'function)`, where *x* is a *lambda expression*, the value of the *optimize quality* **safety** in the global environment at the time the **coerce** is *executed* applies to the resulting *function*.
* For a call to the *function* **ensure-generic-function**, the value of the *optimize quality* **safety** in the *environment object* passed as the `:environment` *argument* applies to the resulting *generic function*.
* For a call to **compile** with a *lambda expression* as the *argument*, the value of the *optimize quality* **safety** in the *global environment* at the time **compile** is *called* applies to the resulting *compiled function*.
* For a call to **compile** with only one argument, if the original definition of the *function* was *safe*, then the resulting *compiled function* must also be *safe*.
* A *call* to a *method* by **call-next-method** must be considered *safe* if each of the following is *safe code* or *system code*:
    -- the definition of the *generic function* (if it was defined explicitly).
    -- the *method* definitions for all *applicable methods*.
    -- the definition of the *method combination*.

-- the point of entry into the body of the *method defining form*, where the *binding* of **call-next-method** is established.
-- the point of *functional evaluation* of the name **call-next-method**.

An *unsafe call* is a *call* that is not a *safe call*.

The informal intent is that the *programmer* can rely on a *call* to be *safe*, even when *system code* is involved, if all reasonable steps have been taken to ensure that the *call* is *safe*. For example, if a *programmer* calls **mapcar** from *safe code* and supplies a *function* that was *compiled* as *safe*, the *implementation* is required to ensure that **mapcar** makes a *safe call* as well.

3.5.1.1.1 Error Detection Time in Safe Calls

If an error is signaled in a *safe call*, the exact point of the *signal* is *implementation-dependent*. In particular, it might be signaled at compile time or at run time, and if signaled at run time, it might be prior to, during, or after *executing* the *call*. However, it is always prior to the execution of the body of the *function* being *called*.

3.5.1.2 Too Few Arguments

It is not permitted to supply too few *arguments* to a *function*. Too few arguments means fewer *arguments* than the number of *required parameters* for the *function*.

If this *situation* occurs in a *safe call*, an error of *type* **program-error** must be signaled; and in an *unsafe call* the *situation* has undefined consequences.

3.5.1.3 Too Many Arguments

It is not permitted to supply too many *arguments* to a *function*. Too many arguments means more *arguments* than the number of *required parameters* plus the number of *optional parameters*; however, if the *function* uses `&rest` or `&key`, it is not possible for it to receive too many arguments.

If this *situation* occurs in a *safe call*, an error of *type* **program-error** must be signaled; and in an *unsafe call* the *situation* has undefined consequences.

3.5.1.4 Unrecognized Keyword Arguments

It is not permitted to supply a keyword argument to a *function* using a name that is not recognized by that *function* unless keyword argument checking is suppressed as described in Section 3.4.1.4.1 (Suppressing Keyword Argument Checking).

If this *situation* occurs in a *safe call*, an error of *type* **program-error** must be signaled; and in an *unsafe call* the *situation* has undefined consequences.

3.5.1.5 Invalid Keyword Arguments

It is not permitted to supply a keyword argument to a *function* using a name that is not a *symbol*.

If this *situation* occurs in a *safe call*, an error of *type* **program-error** must be signaled unless keyword argument checking is suppressed as described in Section 3.4.1.4.1 (Suppressing Keyword Argument Checking); and in an *unsafe call* the *situation* has undefined consequences.

3.5.1.6 Odd Number of Keyword Arguments

An odd number of *arguments* must not be supplied for the *keyword parameters*.

If this *situation* occurs in a *safe call*, an error of *type* **program-error** must be signaled unless keyword argument checking is suppressed as described in Section 3.4.1.4.1 (Suppressing Keyword Argument Checking); and in an *unsafe call* the *situation* has undefined consequences.

3.5.1.7 Destructuring Mismatch

When matching a *destructuring lambda list* against a *form*, the pattern and the *form* must have compatible *tree structure*, as described in Section 3.4.4 (Macro Lambda Lists).

Otherwise, in a *safe call*, an error of *type* **program-error** must be signaled; and in an *unsafe call* the *situation* has undefined consequences.

3.5.1.8 Errors When Calling a Next Method

If **call-next-method** is called with *arguments*, the ordered set of *applicable methods* for the changed set of *arguments* for **call-next-method** must be the same as the ordered set of *applicable methods* for the original *arguments* to the *generic function*, or else an error should be signaled.

The comparison between the set of methods applicable to the new arguments and the set applicable to the original arguments is insensitive to order differences among methods with the same specializers.

If **call-next-method** is called with *arguments* that specify a different ordered set of *applicable* methods and there is no *next method* available, the test for different methods and the associated error signaling (when present) takes precedence over calling **no-next-method**.

3.6 Traversal Rules and Side Effects

The consequences are undefined when *code* executed during an *object-traversing* operation destructively modifies the *object* in a way that might affect the ongoing traversal operation. In particular, the following rules apply.

List traversal
: For *list* traversal operations, the *cdr* chain of the *list* is not allowed to be destructively modified.

Array traversal
: For *array* traversal operations, the *array* is not allowed to be adjusted and its *fill pointer*, if any, is not allowed to be changed.

Hash-table traversal
: For *hash table* traversal operations, new elements may not be added or deleted except that the element corresponding to the current hash key may be changed or removed.

Package traversal
: For *package* traversal operations (e.g., **do-symbols**), new *symbols* may not be *interned* in or *uninterned* from the *package* being traversed or any *package* that it uses except that the current *symbol* may be *uninterned* from the *package* being traversed.

3.7 Destructive Operations

3.7.1 Modification of Literal Objects

The consequences are undefined if *literal objects* are destructively modified. For this purpose, the following operations are considered *destructive*:

random-state
Using it as an *argument* to the *function* **random**.

cons
Changing the *car*[1] or *cdr*[1] of the *cons*, or performing a *destructive* operation on an *object* which is either the *car*[2] or the *cdr*[2] of the *cons*.

array
Storing a new value into some element of the *array*, or performing a *destructive* operation on an *object* that is already such an *element*.

Changing the *fill pointer*, *dimensions*, or displacement of the *array* (regardless of whether the *array* is *actually adjustable*).

Performing a *destructive* operation on another *array* that is displaced to the *array* or that otherwise shares its contents with the *array*.

hash-table
Performing a *destructive* operation on any *key*.

Storing a new *value*[4] for any *key*, or performing a *destructive* operation on any *object* that is such a *value*.

Adding or removing entries from the *hash table*.

structure-object
Storing a new value into any slot, or performing a *destructive* operation on an *object* that is the value of some slot.

standard-object
Storing a new value into any slot, or performing a *destructive* operation on an *object* that is the value of some slot.

Changing the class of the *object* (e.g., using the *function* **change-class**).

readtable
Altering the *readtable case*.

Altering the syntax type of any character in this readtable.

Altering the *reader macro function* associated with any *character* in the *readtable*, or altering the *reader macro functions* associated with *characters* defined as *dispatching macro characters* in the *readtable*.

stream
Performing I/O operations on the *stream*, or *closing* the *stream*.

All other standardized types
[This category includes, for example, **character**, **condition**, **function**, **method-combination**, **method**, **number**, **package**, **pathname**, **restart**, and **symbol**.]

There are no *standardized destructive* operations defined on *objects* of these *types*.

3.7.2 Transfer of Control during a Destructive Operation

Should a transfer of control out of a *destructive* operation occur (e.g., due to an error) the state of the *object* being modified is *implementation-dependent*.

3.7.2.1 Examples of Transfer of Control during a Destructive Operation

The following examples illustrate some of the many ways in which the *implementation-dependent* nature of the modification can manifest itself.

```
 (let ((a (list 2 1 4 3 7 6 'five)))
   (ignore-errors (sort a #'<))
   a)
=>  (1 2 3 4 6 7 FIVE)
OR=>  (2 1 4 3 7 6 FIVE)
OR=>  (2)

 (prog foo ((a (list 1 2 3 4 5 6 7 8 9 10)))
   (sort a #'(lambda (x y) (if (zerop (random 5)) (return-from foo a) (> x y)))))
=>  (1 2 3 4 5 6 7 8 9 10)
OR=>  (3 4 5 6 2 7 8 9 10 1)
OR=>  (1 2 4 3)
```

4. Types and Classes

4.1 Introduction

A *type* is a (possibly infinite) set of *objects*. An *object* can belong to more than one *type*. *Types* are never explicitly represented as *objects* by Common Lisp. Instead, they are referred to indirectly by the use of *type specifiers*, which are *objects* that denote *types*.

New *types* can be defined using **deftype**, **defstruct**, **defclass**, and **define-condition**.

The *function* **typep**, a set membership test, is used to determine whether a given *object* is of a given *type*. The function **subtypep**, a subset test, is used to determine whether a given *type* is a *subtype* of another given *type*. The function **type-of** returns a particular *type* to which a given *object* belongs, even though that *object* must belong to one or more other *types* as well. (For example, every *object* is of *type* **t**, but **type-of** always returns a *type specifier* for a *type* more specific than **t**.)

Objects, not *variables*, have *types*. Normally, any *variable* can have any *object* as its *value*. It is possible to declare that a *variable* takes on only values of a given *type* by making an explicit *type declaration*. *Types* are arranged in a directed acyclic graph, except for the presence of equivalences.

Declarations can be made about *types* using **declare**, **proclaim**, **declaim**, or **the**. For more information about *declarations*, see Section 3.3 (Declarations).

Among the fundamental *objects* of the object system are *classes*. A *class* determines the structure and behavior of a set of other *objects*, which are called its *instances*. Every *object* is a *direct instance* of a *class*. The *class* of an *object* determines the set of operations that can be performed on the *object*. For more information, see Section 4.3 (Classes).

It is possible to write *functions* that have behavior *specialized* to the class of the *objects* which are their *arguments*. For more information, see Section 7.6 (Generic Functions and Methods).

4.2 Types

4.2.1 Data Type Definition

Information about *type* usage is located in the sections specified in Figure 4-1. Figure 4-7 lists some *classes* that are particularly relevant to the object system. Figure 9-1 lists the defined *condition types*.

Section	Data Type

Figure 4-1. Cross-References to Data Type Information

4.2.2 Type Relationships

* The *types* **cons**, **symbol**, **array**, **number**, **character**, **hash-table**, **function**, **readtable**, **package**, **pathname**, **stream**, **random-state**, **condition**, **restart**, and any single other *type* created by **defstruct**, **define-condition**, or **defclass** are *pairwise disjoint*, except for type relations explicitly established by specifying *superclasses* in **defclass** or **define-condition** or the `:include` option of **destruct**.
* Any two *types* created by **defstruct** are *disjoint* unless one is a *supertype* of the other by virtue of the **defstruct** `:include` option.
* Any two *distinct classes* created by **defclass** or **define-condition** are *disjoint* unless they have a common *subclass* or one *class* is a *subclass* of the other.
* An implementation may be extended to add other *subtype* relationships between the specified *types*, as long as they do not violate the type relationships and disjointness requirements specified here. An implementation may define additional *types* that are *subtypes* or *supertypes* of any specified *types*, as long as each additional *type* is a *subtype* of *type* **t** and a *supertype* of *type* **nil** and the disjointness requirements are not violated.

    At the discretion of the implementation, either **standard-object** or **structure-object** might appear in any class precedence list for a *system class* that does not already specify either **standard-object** or **structure-object**. If it does, it must precede the *class* **t** and follow all other *standardized classes*.

4.2.3 Type Specifiers

Type specifiers can be *symbols*, *classes*, or *lists*. Figure 4-2 lists *symbols* that are *standardized atomic type specifiers*, and Figure 4-3 lists *standardized compound type specifier names*. For syntax information, see the dictionary entry for the corresponding *type specifier*. It is possible to define new *type specifiers* using **defclass**, **define-condition**, **defstruct**, or **deftype**.

arithmetic-error	function	simple-condition
array	generic-function	simple-error
atom	hash-table	simple-string
base-char	integer	simple-type-error
base-string	keyword	simple-vector
bignum	list	simple-warning
bit	logical-pathname	single-float
bit-vector	long-float	standard-char
broadcast-stream	method	standard-class
built-in-class	method-combination	standard-generic-function
cell-error	nil	standard-method
character	null	standard-object
class	number	storage-condition
compiled-function	package	stream
complex	package-error	stream-error
concatenated-stream	parse-error	string

```
condition                           pathname            string-stream
cons                                print-not-readable  structure-class
control-error                       program-error       structure-object
division-by-zero                    random-state        style-warning
double-float                        ratio               symbol
echo-stream                         rational            synonym-stream
end-of-file                         reader-error        t
error                               readtable           two-way-stream
extended-char                       real                type-error
file-error                          restart             unbound-slot
file-stream                         sequence            unbound-variable
fixnum                              serious-condition   undefined-function
float                               short-float         unsigned-byte
floating-point-inexact              signed-byte         vector
floating-point-invalid-operation    simple-array        warning
floating-point-overflow             simple-base-string
floating-point-underflow            simple-bit-vector
```

Figure 4-2. Standardized Atomic Type Specifiers

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. Such a *type specifier* is called a *compound type specifier*. Except as explicitly stated otherwise, the subsidiary items can be unspecified. The unspecified subsidiary items are indicated by writing `*`. For example, to completely specify a *vector*, the *type* of the elements and the length of the *vector* must be present.

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

The following leaves the length unspecified:

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

The following leaves the element type unspecified:

```
(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.

If a *list* has one or more unspecified items at the end, those items can be dropped. If dropping all occurrences of `*` results in a *singleton list*, then the parentheses can be dropped as well (the list can be replaced by the *symbol* in its *car*). For example, `(vector double-float *)` can be abbreviated to `(vector double-float)`, and `(vector * *)` can be abbreviated to `(vector)` and then to `vector`.

```
and           long-float    simple-base-string
array         member        simple-bit-vector
base-string   mod           simple-string
bit-vector    not           simple-vector
complex       or            single-float
cons          rational      string
double-float  real          unsigned-byte
eql           satisfies     values
float         short-float   vector
function      signed-byte
integer       simple-array
```

Figure 4-3. Standardized Compound Type Specifier Names

The next figure show the *defined names* that can be used as *compound type specifier names* but that cannot be used as *atomic type specifiers*.

```
and     mod  satisfies
eql     not  values
member  or
```

Figure 4-4. Standardized Compound-Only Type Specifier Names

New *type specifiers* can come into existence in two ways.

* Defining a structure by using **defstruct** without using the `:type` specifier or defining a *class* by using **defclass** or **define-condition** automatically causes the name of the structure or class to be a new *type specifier symbol*.
* **deftype** can be used to define *derived type specifiers*, which act as 'abbreviations' for other *type specifiers*.

A *class object* can be used as a *type specifier*. When used this way, it denotes the set of all members of that *class*.

The next figure shows some *defined names* relating to *types* and *declarations*.

```
coerce            defstruct  subtypep
declaim           deftype    the
declare           ftype      type
defclass          locally    type-of
define-condition  proclaim   typep
```

Figure 4-5. Defined names relating to types and declarations.

The next figure shows all *defined names* that are *type specifier names*, whether for *atomic type specifiers* or *compound type specifiers*; this list is the union of the lists in Figure 4-2 and Figure 4-3.

```
and                                 function            simple-array
arithmetic-error                    generic-function    simple-base-string
array                               hash-table          simple-bit-vector
atom                                integer             simple-condition
base-char                           keyword             simple-error
base-string                         list                simple-string
bignum                              logical-pathname    simple-type-error
bit                                 long-float          simple-vector
bit-vector                          member              simple-warning
broadcast-stream                    method              single-float
built-in-class                      method-combination  standard-char
cell-error                          mod                 standard-class
character                           nil                 standard-generic-function
class                               not                 standard-method
compiled-function                   null                standard-object
complex                             number              storage-condition
concatenated-stream                 or                  stream
condition                           package             stream-error
cons                                package-error       string
control-error                       parse-error         string-stream
division-by-zero                    pathname            structure-class
double-float                        print-not-readable  structure-object
echo-stream                         program-error       style-warning
end-of-file                         random-state        symbol
eql                                 ratio               synonym-stream
error                               rational            t
extended-char                       reader-error        two-way-stream
file-error                          readtable           type-error
file-stream                         real                unbound-slot
fixnum                              restart             unbound-variable
float                               satisfies           undefined-function
floating-point-inexact              sequence            unsigned-byte
floating-point-invalid-operation    serious-condition   values
floating-point-overflow             short-float         vector
floating-point-underflow            signed-byte         warning
```

Figure 4-6. Standardized Type Specifier Names

4.3 Classes

While the object system is general enough to describe all *standardized classes* (including, for example, **number**, **hash-table**, and **symbol**), the next figure contains a list of *classes* that are especially relevant to understanding the object system.

```
built-in-class    method-combination         standard-object
class             standard-class             structure-class
generic-function  standard-generic-function  structure-object
method            standard-method
```

Figure 4-7. Object System Classes

4.3.1 Introduction to Classes

A *class* is an *object* that determines the structure and behavior of a set of other *objects*, which are called its *instances*.

A *class* can inherit structure and behavior from other *classes*. A *class* whose definition refers to other *classes* for the purpose of inheriting from them is said to be a *subclass* of each of those *classes*. The *classes* that are designated for purposes of inheritance are said to be *superclasses* of the inheriting *class*.

A *class* can have a *name*. The *function* **class-name** takes a *class object* and returns its *name*. The *name* of an anonymous *class* is **nil**. A *symbol* can *name* a *class*. The *function* **find-class** takes a *symbol* and returns the *class* that the *symbol* names. A *class* has a *proper name* if the *name* is a *symbol* and if the *name* of the *class* names that *class*. That is, a *class* C has the *proper name* S if S= `(class-name C)` and C= `(find-class S)`. Notice that it is possible for `(find-class S1)` = `(find-class S2)` and S1/=S2. If C= `(find-class S)`, we say that C is the *class named* S.

A *class* C1 is a *direct superclass* of a *class* C2 if C2 explicitly designates C1 as a *superclass* in its definition. In this case C2 is a *direct subclass* of C1. A *class* Cn is a *superclass* of a *class* C1 if there exists a series of *classes* C2,...,Cn-1 such that Ci+1 is a *direct superclass* of Ci for 1 <=i<n. In this case, C1 is a *subclass* of Cn. A *class* is considered neither a *superclass* nor a *subclass* of itself. That is, if C1 is a *superclass* of C2, then C1 /=C2. The set of *classes* consisting of some given *class* C along with all of its *superclasses* is called "C and its superclasses."

Each *class* has a *class precedence list*, which is a total ordering on the set of the given *class* and its *superclasses*. The total ordering is expressed as a list ordered from most specific to least specific. The *class precedence list* is used in several ways. In general, more specific *classes* can *shadow*[1] features that would otherwise be inherited from less specific *classes*. The *method* selection and combination process uses the *class precedence list* to order *methods* from most specific to least specific.

When a *class* is defined, the order in which its direct *superclasses* are mentioned in the defining form is important. Each *class* has a *local precedence order*, which is a *list* consisting of the *class* followed by its *direct superclasses* in the order mentioned in the defining *form*.

A *class precedence list* is always consistent with the *local precedence order* of each *class* in the list. The *classes* in each *local precedence order* appear within the *class precedence list* in the same order. If the *local precedence orders* are inconsistent with each other, no *class precedence list* can be constructed, and an error is signaled. The *class precedence list* and its computation is discussed in Section 4.3.5 (Determining the Class Precedence List).

classes are organized into a directed acyclic graph. There are two distinguished *classes*, named **t** and **standard-object**. The *class* named **t** has no *superclasses*. It is a *superclass* of every *class* except itself. The *class* named **standard-object** is an *instance* of the *class* **standard-class** and is a *superclass* of every *class* that is an

instance of the *class* **standard-class** except itself.

There is a mapping from the object system *class* space into the *type* space. Many of the standard *types* specified in this document have a corresponding *class* that has the same *name* as the *type*. Some *types* do not have a corresponding *class*. The integration of the *type* and *class* systems is discussed in Section 4.3.7 (Integrating Types and Classes).

Classes are represented by *objects* that are themselves *instances* of *classes*. The *class* of the *class* of an *object* is termed the *metaclass* of that *object*. When no misinterpretation is possible, the term *metaclass* is used to refer to a *class* that has *instances* that are themselves *classes*. The *metaclass* determines the form of inheritance used by the *classes* that are its *instances* and the representation of the *instances* of those *classes*. The object system provides a default *metaclass*, **standard-class**, that is appropriate for most programs.

Except where otherwise specified, all *classes* mentioned in this standard are *instances* of the *class* **standard-class**, all *generic functions* are *instances* of the *class* **standard-generic-function**, and all *methods* are *instances* of the *class* **standard-method**.

4.3.1.1 Standard Metaclasses

The object system provides a number of predefined *metaclasses*. These include the *classes* **standard-class**, **built-in-class**, and **structure-class**:

* The *class* **standard-class** is the default *class* of *classes* defined by **defclass**.
* The *class* **built-in-class** is the *class* whose *instances* are *classes* that have special implementations with restricted capabilities. Any *class* that corresponds to a standard *type* might be an *instance* of **built-in-class**. The predefined *type* specifiers that are required to have corresponding *classes* are listed in Figure 4-8. It is *implementation-dependent* whether each of these *classes* is implemented as a *built-in class*.
* All *classes* defined by means of **defstruct** are *instances* of the *class* **structure-class**.

4.3.2 Defining Classes

The macro **defclass** is used to define a new named *class*.

The definition of a *class* includes:

* The *name* of the new *class*. For newly-defined *classes* this *name* is a *proper name*.
* The list of the direct *superclasses* of the new *class*.
* A set of *slot specifiers*. Each *slot specifier* includes the *name* of the *slot* and zero or more *slot* options. A *slot* option pertains only to a single *slot*. If a *class* definition contains two *slot specifiers* with the same *name*, an error is signaled.
* A set of *class* options. Each *class* option pertains to the *class* as a whole.

The *slot* options and *class* options of the **defclass** form provide mechanisms for the following:

* Supplying a default initial value *form* for a given *slot*.
* Requesting that *methods* for *generic functions* be automatically generated for reading or writing *slots*.
* Controlling whether a given *slot* is shared by all *instances* of the *class* or whether each *instance* of the *class* has its own *slot*.
* Supplying a set of initialization arguments and initialization argument defaults to be used in *instance* creation.
* Indicating that the *metaclass* is to be other than the default. The `:metaclass` option is reserved for future use; an implementation can be extended to make use of the `:metaclass` option.
* Indicating the expected *type* for the value stored in the *slot*.

* Indicating the *documentation string* for the *slot*.

4.3.3 Creating Instances of Classes

The generic function **make-instance** creates and returns a new *instance* of a *class*. The object system provides several mechanisms for specifying how a new *instance* is to be initialized. For example, it is possible to specify the initial values for *slots* in newly created *instances* either by giving arguments to **make-instance** or by providing default initial values. Further initialization activities can be performed by *methods* written for *generic functions* that are part of the initialization protocol. The complete initialization protocol is described in Section 7.1 (Object Creation and Initialization).

4.3.4 Inheritance

A *class* can inherit *methods*, *slots*, and some **defclass** options from its *superclasses*. Other sections describe the inheritance of *methods*, the inheritance of *slots* and *slot* options, and the inheritance of *class* options.

4.3.4.1 Examples of Inheritance

```
(defclass C1 ()
    ((S1 :initform 5.4 :type number)
     (S2 :allocation :class)))

(defclass C2 (C1)
    ((S1 :initform 5 :type integer)
     (S2 :allocation :instance)
     (S3 :accessor C2-S3)))
```

Instances of the class `C1` have a *local slot* named `S1`, whose default initial value is 5.4 and whose *value* should always be a *number*. The class `C1` also has a *shared slot* named `S2`.

There is a *local slot* named `S1` in *instances* of `C2`. The default initial value of `S1` is 5. The value of `S1` should always be of type `(and integer number)`. There are also *local slots* named `S2` and `S3` in *instances* of `C2`. The class `C2` has a *method* for `C2-S3` for reading the value of slot `S3`; there is also a *method* for `(setf C2-S3)` that writes the value of `S3`.

4.3.4.2 Inheritance of Class Options

The `:default-initargs` class option is inherited. The set of defaulted initialization arguments for a *class* is the union of the sets of initialization arguments supplied in the `:default-initargs` class options of the *class* and its *superclasses*. When more than one default initial value *form* is supplied for a given initialization argument, the default initial value *form* that is used is the one supplied by the *class* that is most specific according to the *class precedence list*.

If a given `:default-initargs` class option specifies an initialization argument of the same *name* more than once, an error of *type* **program-error** is signaled.

4.3.5 Determining the Class Precedence List

The **defclass** form for a *class* provides a total ordering on that *class* and its direct *superclasses*. This ordering is called the *local precedence order*. It is an ordered list of the *class* and its direct *superclasses*. The *class precedence list* for a class C is a total ordering on C and its *superclasses* that is consistent with the *local precedence orders* for each of C and its *superclasses*.

A *class* precedes its direct *superclasses*, and a direct *superclass* precedes all other direct *superclasses* specified to its right in the *superclasses* list of the **defclass** form. For every class C, define

RC={(C,C1),(C1,C2),...,(Cn-1,Cn)}

where C1,...,Cn are the direct *superclasses* of C in the order in which they are mentioned in the **defclass** form. These ordered pairs generate the total ordering on the class C and its direct *superclasses*.

Let SC be the set of C and its *superclasses*. Let R be

R=Uc<ELEMENT-OF>SCRc

.

The set R might or might not generate a partial ordering, depending on whether the Rc, c<ELEMENT-OF>SC, are consistent; it is assumed that they are consistent and that R generates a partial ordering. When the Rc are not consistent, it is said that R is inconsistent.

To compute the *class precedence list* for C, topologically sort the elements of SC with respect to the partial ordering generated by R. When the topological sort must select a *class* from a set of two or more *classes*, none of which are preceded by other *classes* with respect to R, the *class* selected is chosen deterministically, as described below.

If R is inconsistent, an error is signaled.

4.3.5.1 Topological Sorting

Topological sorting proceeds by finding a class C in SC such that no other *class* precedes that element according to the elements in R. The class C is placed first in the result. Remove C from SC, and remove all pairs of the form (C,D), D<ELEMENT-OF>SC, from R. Repeat the process, adding *classes* with no predecessors to the end of the result. Stop when no element can be found that has no predecessor.

If SC is not empty and the process has stopped, the set R is inconsistent. If every *class* in the finite set of *classes* is preceded by another, then R contains a loop. That is, there is a chain of classes C1,...,Cn such that Ci precedes Ci+1, 1<=i<n, and Cn precedes C1.

Sometimes there are several *classes* from SC with no predecessors. In this case select the one that has a direct *subclass* rightmost in the *class precedence list* computed so far. (If there is no such candidate *class*, R does not generate a partial ordering---the Rc, c<ELEMENT-OF>SC, are inconsistent.)

In more precise terms, let {N1,...,Nm}, m>=2, be the *classes* from SC with no predecessors. Let (C1...Cn), n>=1, be the *class precedence list* constructed so far. C1 is the most specific *class*, and Cn is the least specific. Let 1<=j<=n be the largest number such that there exists an i where 1<=i<=m and Ni is a direct *superclass* of Cj; Ni is placed next.

The effect of this rule for selecting from a set of *classes* with no predecessors is that the *classes* in a simple *superclass* chain are adjacent in the *class precedence list* and that *classes* in each relatively separated subgraph are adjacent in the *class precedence list*. For example, let T1 and T2 be subgraphs whose only element in common is the class J. Suppose that no superclass of J appears in either T1 or T2, and that J is in the superclass chain of every class in both T1 and T2. Let C1 be the bottom of T1; and let C2 be the bottom of T2. Suppose C is a *class* whose direct *superclasses* are C1 and C2 in that order, then the *class precedence list* for C starts with C and is followed by all *classes* in T1 except J. All the *classes* of T2 are next. The *class* J and its *superclasses* appear last.

4.3.5.2 Examples of Class Precedence List Determination

This example determines a *class precedence list* for the class `pie`. The following *classes* are defined:

```
(defclass pie (apple cinnamon) ())

(defclass apple (fruit) ())

(defclass cinnamon (spice) ())

(defclass fruit (food) ())

(defclass spice (food) ())

(defclass food () ())
```

The set Spie = {`pie, apple, cinnamon, fruit, spice, food, standard-object, t`}. The set R = {`(pie, apple), (apple, cinnamon), (apple, fruit), (cinnamon, spice), (fruit, food), (spice, food), (food, standard-object), (standard-object, t)`}.

The class `pie` is not preceded by anything, so it comes first; the result so far is `(pie)`. Remove `pie` from S and pairs mentioning `pie` from R to get S = {`apple, cinnamon, fruit, spice, food, standard-object, t`} and R = {`(apple, cinnamon), (apple, fruit), (cinnamon, spice), (fruit, food), (spice, food), (food, standard-object), (standard-object, t)`}.

The class `apple` is not preceded by anything, so it is next; the result is `(pie apple)`. Removing `apple` and the relevant pairs results in S = {`cinnamon, fruit, spice, food, standard-object, t`} and R = {`(cinnamon, spice), (fruit, food), (spice, food), (food, standard-object), (standard-object, t)`}.

The classes `cinnamon` and `fruit` are not preceded by anything, so the one with a direct *subclass* rightmost in the *class precedence list* computed so far goes next. The class `apple` is a direct *subclass* of `fruit`, and the class `pie` is a direct *subclass* of `cinnamon`. Because `apple` appears to the right of `pie` in the *class precedence list*, `fruit` goes next, and the result so far is `(pie apple fruit)`. S = {`cinnamon, spice, food, standard-object, t`}; R = {`(cinnamon, spice), (spice, food), (food, standard-object), (standard-object, t)`}.

The class `cinnamon` is next, giving the result so far as `(pie apple fruit cinnamon)`. At this point S = {`spice, food, standard-object, t`}; R = {`(spice, food), (food, standard-object), (standard-object, t)`}.

The classes `spice`, `food`, **standard-object**, and **t** are added in that order, and the *class precedence list* is `(pie apple fruit cinnamon spice food standard-object t)`.

It is possible to write a set of *class* definitions that cannot be ordered. For example:

```
(defclass new-class (fruit apple) ())

(defclass apple (fruit) ())
```

The class `fruit` must precede `apple` because the local ordering of *superclasses* must be preserved. The class `apple` must precede `fruit` because a *class* always precedes its own *superclasses*. When this situation occurs, an error is signaled, as happens here when the system tries to compute the *class precedence list* of `new-class`.

The following might appear to be a conflicting set of definitions:

```
(defclass pie (apple cinnamon) ())

(defclass pastry (cinnamon apple) ())

(defclass apple () ())

(defclass cinnamon () ())
```

The *class precedence list* for `pie` is `(pie apple cinnamon standard-object t)`.

The *class precedence list* for `pastry` is `(pastry cinnamon apple standard-object t)`.

It is not a problem for `apple` to precede `cinnamon` in the ordering of the *superclasses* of `pie` but not in the ordering for `pastry`. However, it is not possible to build a new *class* that has both `pie` and `pastry` as *superclasses*.

4.3.6 Redefining Classes

A *class* that is a *direct instance* of **standard-class** can be redefined if the new *class* is also a *direct instance* of **standard-class**. Redefining a *class* modifies the existing *class object* to reflect the new *class* definition; it does not create a new *class object* for the *class*. Any *method object* created by a `:reader`, `:writer`, or `:accessor` option specified by the old **defclass** form is removed from the corresponding *generic function*. *Methods* specified by the new **defclass** form are added.

When the class C is redefined, changes are propagated to its *instances* and to *instances* of any of its *subclasses*. Updating such an *instance* occurs at an *implementation-dependent* time, but no later than the next time a *slot* of that *instance* is read or written. Updating an *instance* does not change its identity as defined by the *function* **eq**. The updating process may change the *slots* of that particular *instance*, but it does not create a new *instance*. Whether updating an *instance* consumes storage is *implementation-dependent*.

Note that redefining a *class* may cause *slots* to be added or deleted. If a *class* is redefined in a way that changes the set of *local slots accessible* in *instances*, the *instances* are updated. It is *implementation-dependent* whether *instances* are updated if a *class* is redefined in a way that does not change the set of *local slots accessible* in *instances*.

The value of a *slot* that is specified as shared both in the old *class* and in the new *class* is retained. If such a *shared slot* was unbound in the old *class*, it is unbound in the new *class*. *Slots* that were local in the old *class* and that are shared in the new *class* are initialized. Newly added *shared slots* are initialized.

Each newly added *shared slot* is set to the result of evaluating the *captured initialization form* for the *slot* that was specified in the **defclass** *form* for the new *class*. If there was no *initialization form*, the *slot* is unbound.

If a *class* is redefined in such a way that the set of *local slots accessible* in an *instance* of the *class* is changed, a two-step process of updating the *instances* of the *class* takes place. The process may be explicitly started by invoking the generic function **make-instances-obsolete**. This two-step process can happen in other circumstances in some implementations. For example, in some implementations this two-step process is triggered if the order of *slots* in storage is changed.

The first step modifies the structure of the *instance* by adding new *local slots* and discarding *local slots* that are not defined in the new version of the *class*. The second step initializes the newly-added *local slots* and performs any other user-defined actions. These two steps are further specified in the next two sections.

4.3.6.1 Modifying the Structure of Instances

The first step modifies the structure of *instances* of the redefined *class* to conform to its new *class* definition. *Local slots* specified by the new *class* definition that are not specified as either local or shared by the old *class* are added, and *slots* not specified as either local or shared by the new *class* definition that are specified as local by the old *class* are discarded. The *names* of these added and discarded *slots* are passed as arguments to **update-instance-for-redefined-class** as described in the next section.

The values of *local slots* specified by both the new and old *classes* are retained. If such a *local slot* was unbound, it remains unbound.

The value of a *slot* that is specified as shared in the old *class* and as local in the new *class* is retained. If such a *shared slot* was unbound, the *local slot* is unbound.

4.3.6.2 Initializing Newly Added Local Slots

The second step initializes the newly added *local slots* and performs any other user-defined actions. This step is implemented by the generic function **update-instance-for-redefined-class**, which is called after completion of the first step of modifying the structure of the *instance*.

The generic function **update-instance-for-redefined-class** takes four required arguments: the *instance* being updated after it has undergone the first step, a list of the names of *local slots* that were added, a list of the names of *local slots* that were discarded, and a property list containing the *slot* names and values of *slots* that were discarded and had values. Included among the discarded *slots* are *slots* that were local in the old *class* and that are shared in the new *class*.

The generic function **update-instance-for-redefined-class** also takes any number of initialization arguments. When it is called by the system to update an *instance* whose *class* has been redefined, no initialization arguments are provided.

There is a system-supplied primary *method* for **update-instance-for-redefined-class** whose *parameter specializer* for its *instance* argument is the *class* **standard-object**. First this *method* checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared as valid. (For more information, see Section 7.1.2 (Declaring the Validity of Initialization Arguments).) Then it calls the generic function **shared-initialize** with the following arguments: the *instance*, the list of *names* of the newly added *slots*, and the initialization arguments it received.

4.3.6.3 Customizing Class Redefinition

Methods for **update-instance-for-redefined-class** may be defined to specify actions to be taken when an *instance* is updated. If only *after methods* for **update-instance-for-redefined-class** are defined, they will be run after the system-supplied primary *method* for initialization and therefore will not interfere with the default behavior of **update-instance-for-redefined-class**. Because no initialization arguments are passed to **update-instance-for-redefined-class** when it is called by the system, the *initialization forms* for *slots* that are filled by *before methods* for **update-instance-for-redefined-class** will not be evaluated by **shared-initialize**.

Methods for **shared-initialize** may be defined to customize *class* redefinition. For more information, see Section 7.1.5 (Shared-Initialize).

4.3.7 Integrating Types and Classes

The object system maps the space of *classes* into the space of *types*. Every *class* that has a proper name has a corresponding *type* with the same *name*.

The proper name of every *class* is a valid *type specifier*. In addition, every *class object* is a valid *type specifier*. Thus the expression `(typep` *object class*`)` evaluates to *true* if the *class* of *object* is *class* itself or a *subclass* of *class*. The evaluation of the expression `(subtypep class1 class2)` returns the values *true* and *true* if `class1` is a subclass of `class2` or if they are the same *class*; otherwise it returns the values *false* and *true*. If I is an *instance* of some *class* C named S and C is an *instance* of **standard-class**, the evaluation of the expression `(type-of I)` returns S if S is the *proper name* of C; otherwise, it returns C.

Because the names of *classes* and *class objects* are *type specifiers*, they may be used in the special form **the** and in type declarations.

Many but not all of the predefined *type specifiers* have a corresponding *class* with the same proper name as the *type*. These type specifiers are listed in Figure 4-8. For example, the *type* **array** has a corresponding *class* named **array**. No *type specifier* that is a list, such as `(vector double-float 100)`, has a corresponding *class*. The *operator* **deftype** does not create any *classes*.

Each *class* that corresponds to a predefined *type specifier* can be implemented in one of three ways, at the discretion of each implementation. It can be a *standard class*, a *structure class*, or a *system class*.

A *built-in class* is one whose *generalized instances* have restricted capabilities or special representations. Attempting to use **defclass** to define *subclasses* of a **built-in-class** signals an error. Calling **make-instance** to create a *generalized instance* of a *built-in class* signals an error. Calling **slot-value** on a *generalized instance* of a *built-in class* signals an error. Redefining a *built-in class* or using **change-class** to change the *class* of an *object* to or from a *built-in class* signals an error. However, *built-in classes* can be used as *parameter specializers* in *methods*.

It is possible to determine whether a *class* is a *built-in class* by checking the *metaclass*. A *standard class* is an *instance* of the *class* **standard-class**, a *built-in class* is an *instance* of the *class* **built-in-class**, and a *structure class* is an *instance* of the *class* **structure-class**.

Each *structure type* created by **defstruct** without using the `:type` option has a corresponding *class*. This *class* is a *generalized instance* of the *class* **structure-class**. The `:include` option of **defstruct** creates a direct *subclass* of the *class* that corresponds to the included *structure type*.

It is *implementation-dependent* whether *slots* are involved in the operation of *functions* defined in this specification on *instances* of *classes* defined in this specification, except when *slots* are explicitly defined by this specification.

If in a particular *implementation* a *class* defined in this specification has *slots* that are not defined by this specfication, the names of these *slots* must not be *external symbols* of *packages* defined in this specification nor otherwise *accessible* in the `CL-USER` package.

The purpose of specifying that many of the standard *type specifiers* have a corresponding *class* is to enable users to write *methods* that discriminate on these *types*. *Method* selection requires that a *class precedence list* can be determined for each *class*.

The hierarchical relationships among the *type specifiers* are mirrored by relationships among the *classes* corresponding to those *types*.

Figure 4-8 lists the set of *classes* that correspond to predefined *type specifiers*.

```
arithmetic-error                  generic-function    simple-error
array                             hash-table          simple-type-error
bit-vector                        integer             simple-warning
broadcast-stream                  list                standard-class
built-in-class                    logical-pathname    standard-generic-function
cell-error                        method              standard-method
character                         method-combination  standard-object
class                             null                storage-condition
complex                           number              stream
concatenated-stream               package             stream-error
condition                         package-error       string
cons                              parse-error         string-stream
control-error                     pathname            structure-class
division-by-zero                  print-not-readable  structure-object
echo-stream                       program-error       style-warning
end-of-file                       random-state        symbol
error                             ratio               synonym-stream
file-error                        rational            t
file-stream                       reader-error        two-way-stream
float                             readtable           type-error
floating-point-inexact            real                unbound-slot
floating-point-invalid-operation  restart             unbound-variable
floating-point-overflow           sequence            undefined-function
floating-point-underflow          serious-condition   vector
function                          simple-condition    warning
```

Figure 4-8. Classes that correspond to pre-defined type specifiers

The *class precedence list* information specified in the entries for each of these *classes* are those that are required by the object system.

Individual implementations may be extended to define other type specifiers to have a corresponding *class*. Individual implementations may be extended to add other *subclass* relationships and to add other *elements* to the *class precedence lists* as long as they do not violate the type relationships and disjointness requirements specified by this standard. A standard *class* defined with no direct *superclasses* is guaranteed to be disjoint from all of the *classes* in the table, except for the class named **t**.

5. Data and Control Flow

5.1 Generalized Reference

5.1.1 Overview of Places and Generalized Reference

A *generalized reference* is the use of a *form*, sometimes called a *place*, as if it were a *variable* that could be read and written. The *value* of a *place* is the *object* to which the *place form* evaluates. The *value* of a *place* can be changed by using **setf**. The concept of binding a *place* is not defined in Common Lisp, but an *implementation* is permitted to extend the language by defining this concept.

The next figure contains examples of the use of **setf**. Note that the values returned by evaluating the *forms* in column two are not necessarily the same as those obtained by evaluating the *forms* in column three. In general, the exact *macro expansion* of a **setf** *form* is not guaranteed and can even be *implementation-dependent*; all that is guaranteed is that the expansion is an update form that works for that particular *implementation*, that the left-to-right evaluation of *subforms* is preserved, and that the ultimate result of evaluating **setf** is the value or values being stored.

```
Access function   Update Function   Update using setf
x                 (setq x datum)    (setf x datum)
(car x)           (rplaca x datum)  (setf (car x) datum)
(symbol-value x)  (set x datum)     (setf (symbol-value x) datum)
```

Figure 5-1. Examples of setf

The next figure shows *operators* relating to *places* and *generalized reference*.

```
assert                defsetf             push
ccase                 get-setf-expansion  remf
ctypecase             getf                rotatef
decf                  incf                setf
define-modify-macro   pop                 shiftf
define-setf-expander  psetf
```

Figure 5-2. Operators relating to places and generalized reference.

Some of the *operators* above manipulate *places* and some manipulate *setf expanders*. A *setf expansion* can be derived from any *place*. New *setf expanders* can be defined by using **defsetf** and **define-setf-expander**.

5.1.1.1 Evaluation of Subforms to Places

The following rules apply to the *evaluation* of *subforms* in a *place*:

1. The evaluation ordering of *subforms* within a *place* is determined by the order specified by the second value returned by **get-setf-expansion**. For all *places* defined by this specification (e.g., **getf**, **ldb**, ...), this order of evaluation is left-to-right. When a *place* is derived from a macro expansion, this rule is applied after the macro is expanded to find the appropriate *place*.

   Places defined by using **defmacro** or **define-setf-expander** use the evaluation order defined by those definitions. For example, consider the following:

   ```
   (defmacro wrong-order (x y) '(getf ,y ,x))
   ```

   This following *form* evaluates `place2` first and then `place1` because that is the order they are evaluated in the macro expansion:

   ```
   (push value (wrong-order place1 place2))
   ```

2. For the *macros* that manipulate *places* (**push**, **pushnew**, **remf**, **incf**, **decf**, **shiftf**, **rotatef**, **psetf**, **setf**, **pop**, and those defined by **define-modify-macro**) the *subforms* of the macro call are evaluated exactly once in left-to-right order, with the *subforms* of the *places* evaluated in the order specified in (1).

   push, **pushnew**, **remf**, **incf**, **decf**, **shiftf**, **rotatef**, **psetf**, **pop** evaluate all *subforms* before modifying any of the *place* locations. **setf** (in the case when **setf** has more than two arguments) performs its operation on each pair in sequence. For example, in

   ```
   (setf place1 value1 place2 value2 ...)
   ```

   the *subforms* of `place1` and `value1` are evaluated, the location specified by `place1` is modified to contain the value returned by `value1`, and then the rest of the **setf** form is processed in a like manner.

3. For **check-type**, **ctypecase**, and **ccase**, *subforms* of the *place* are evaluated once as in (1), but might be evaluated again if the type check fails in the case of **check-type** or none of the cases hold in **ctypecase** and **ccase**.

4. For **assert**, the order of evaluation of the generalized references is not specified.

Rules 2, 3 and 4 cover all *standardized macros* that manipulate *places*.

5.1.1.1.1 Examples of Evaluation of Subforms to Places

```
 (let ((ref2 (list '())))
   (push (progn (princ "1") 'ref-1)
         (car (progn (princ "2") ref2))))
>>  12
=>  (REF1)

 (let (x)
    (push (setq x (list 'a))
          (car (setq x (list 'b))))
     x)
=>  (((A) . B))
```

push first evaluates `(setq x (list 'a)) => (a)`, then evaluates `(setq x (list 'b)) => (b)`, then modifies the *car* of this latest value to be `((a) . b)`.

5.1.1.2 Setf Expansions

Sometimes it is possible to avoid evaluating *subforms* of a *place* multiple times or in the wrong order. A *setf expansion* for a given access form can be expressed as an ordered collection of five *objects*:

List of temporary variables
: a list of symbols naming temporary variables to be bound sequentially, as if by **let***, to *values* resulting from value forms.

List of value forms
: a list of forms (typically, *subforms* of the *place*) which when evaluated yield the values to which the corresponding temporary variables should be bound.

List of store variables
: a list of symbols naming temporary store variables which are to hold the new values that will be assigned to the *place*.

Storing form
: a form which can reference both the temporary and the store variables, and which changes the *value* of the *place* and guarantees to return as its values the values of the store variables, which are the correct values for **setf** to return.

Accessing form
: a *form* which can reference the temporary variables, and which returns the *value* of the *place*.

The value returned by the accessing form is affected by execution of the storing form, but either of these forms might be evaluated any number of times.

It is possible to do more than one **setf** in parallel via **psetf**, **shiftf**, and **rotatef**. Because of this, the *setf expander* must produce new temporary and store variable names every time. For examples of how to do this, see **gensym**.

For each *standardized* accessor function *F*, unless it is explicitly documented otherwise, it is *implementation-dependent* whether the ability to use an *F form* as a **setf** *place* is implemented by a *setf expander* or a *setf function*. Also, it follows from this that it is *implementation-dependent* whether the name `(setf` *F*`)` is *fbound*.

5.1.1.2.1 Examples of Setf Expansions

Examples of the contents of the constituents of *setf expansions* follow.

For a variable *x*:

```
()              ;list of temporary variables
()              ;list of value forms
(g0001)         ;list of store variables
(setq x g0001)  ;storing form
x               ;accessing form
```

Figure 5-3. Sample Setf Expansion of a Variable

For `(car` *exp*`)`:

```
(g0002)                           ;list of temporary variables
(exp)                             ;list of value forms
(g0003)                           ;list of store variables
(progn (rplaca g0002 g0003) g0003)  ;storing form
(car g0002)                       ;accessing form
```

Figure 5-4. Sample Setf Expansion of a CAR Form

For `(subseq` *seq s e*`)`:

```
(g0004 g0005 g0006)         ;list of temporary variables
(seq s e)                   ;list of value forms
(g0007)                     ;list of store variables
(progn (replace g0004 g0007 :start1 g0005 :end1 g0006) g0007)
                            ;storing form
(subseq g0004 g0005 g0006)  ; accessing form
```

Figure 5-5. Sample Setf Expansion of a SUBSEQ Form

In some cases, if a *subform* of a *place* is itself a *place*, it is necessary to expand the *subform* in order to compute some of the values in the expansion of the outer *place*. For `(ldb` *bs* `(car` *exp*`))`:

```
(g0001 g0002)           ;list of temporary variables
(bs exp)                ;list of value forms
(g0003)                 ;list of store variables
(progn (rplaca g0002 (dpb g0003 g0001 (car g0002))) g0003)
                        ;storing form
(ldb g0001 (car g0002))  ; accessing form
```

Figure 5-6. Sample Setf Expansion of a LDB Form

5.1.2 Kinds of Places

Several kinds of *places* are defined by Common Lisp; this section enumerates them. This set can be extended by *implementations* and by *programmer code*.

5.1.2.1 Variable Names as Places

The name of a *lexical variable* or *dynamic variable* can be used as a *place*.

5.1.2.2 Function Call Forms as Places

A *function form* can be used as a *place* if it falls into one of the following categories:

* A function call form whose first element is the name of any one of the functions in the next figure.

```
aref    cdadr                    get
bit     cdar                     gethash
caaaar  cddaar                   logical-pathname-translations
caaadr  cddadr                   macro-function
caaar   cddar                    ninth
caadar  cdddar                   nth
caaddr  cddddr                   readtable-case
caadr   cdddr                    rest
caar    cddr                     row-major-aref
cadaar  cdr                      sbit
cadadr  char                     schar
cadar   class-name               second
caddar  compiler-macro-function  seventh
cadddr  documentation            sixth
caddr   eighth                   slot-value
cadr    elt                      subseq
car     fdefinition              svref
cdaaar  fifth                    symbol-function
cdaadr  fill-pointer             symbol-plist
cdaar   find-class               symbol-value
cdadar  first                    tenth
cdaddr  fourth                   third
```

Figure 5-7. Functions that setf can be used with---1

In the case of **subseq**, the replacement value must be a *sequence* whose elements might be contained by the sequence argument to **subseq**, but does not have to 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 **replace**.

* A function call form whose first element is the name of a selector function constructed by **defstruct**. The function name must refer to the global function definition, rather than a locally defined *function*.
* A function call form whose first element is the name of any one of the functions in the next figure, provided that the supplied 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 supplied "update" function.

```
Function name  Argument that is a place  Update function used
ldb            second                   dpb
mask-field     second                   deposit-field
getf           first                    implementation-dependent
```

Figure 5-8. Functions that setf can be used with---2 During the **setf** expansion of these *forms*, it is necessary to call **get-setf-expansion** in order to figure out how the inner, nested generalized variable must be treated.

The information from **get-setf-expansion** is used as follows.

ldb

In a form such as:

`(setf (ldb` *byte-spec place-form*`)` *value-form*`)`

the place referred to by the *place-form* must always be both *read* and *written*; note that the update is to the generalized variable specified by *place-form*, not to any object of *type* **integer**.

Thus this **setf** should generate code to do the following:

1. Evaluate *byte-spec* (and bind it into a temporary variable).
2. Bind the temporary variables for *place-form*.
3. Evaluate *value-form* (and bind its value or values into the store variable).
4. Do the *read* from *place-form*.
5. Do the *write* into *place-form* with the given bits of the *integer* fetched in step 4 replaced with the value from step 3.

If the evaluation of *value-form* in step 3 alters what is found in *place-form*, such as setting different bits of *integer*, then the change of the bits denoted by *byte-spec* is to that altered *integer*, because step 4 is done after the *value-form* evaluation. Nevertheless, the evaluations required for *binding* the temporary variables are done in steps 1 and 2, and thus the expected left-to-right evaluation order is seen. For example:

```
(setq integer #x69) =>  #x69
(rotatef (ldb (byte 4 4) integer)
         (ldb (byte 4 0) integer))
integer =>  #x96
;;; This example is trying to swap two independent bit fields
;;; in an integer.  Note that the generalized variable of
;;; interest here is just the (possibly local) program variable
;;; integer.
```

mask-field

This case is the same as **ldb** in all essential aspects.

getf

In a form such as:

`(setf (getf` *place-form ind-form*`)` *value-form*`)`

the place referred to by *place-form* must always be both *read* and *written*; note that the update is to the generalized variable specified by *place-form*, not necessarily to the particular *list* that is the property list in question.

Thus this **setf** should generate code to do the following:

1. Bind the temporary variables for *place-form*.
2. Evaluate *ind-form* (and bind it into a temporary variable).
3. Evaluate *value-form* (and bind its value or values into the store variable).
4. Do the *read* from *place-form*.
5. Do the *write* into *place-form* with a possibly-new property list obtained by combining the values from steps 2, 3, and 4. (Note that the phrase "possibly-new property list" can mean that the former property list is somehow destructively re-used, or it can mean partial or full copying of it. Since either copying or destructive re-use can occur, the treatment of the resultant value for the possibly-new property list must proceed as if it were a different copy needing to be stored back into the generalized variable.)

If the evaluation of *value-form* in step 3 alters what is found in *place-form*, such as setting a different named property in the list, then the change of the property denoted by *ind-form* is to that altered list, because step 4 is done after the *value-form* evaluation. Nevertheless, the evaluations required for *binding* the temporary variables are done in steps 1 and 2, and thus the expected left-to-right evaluation order is seen.

For example:

```
(setq s (setq r (list (list 'a 1 'b 2 'c 3)))) =>  ((a 1 b 2 c 3))
(setf (getf (car r) 'b)
      (progn (setq r nil) 6)) =>  6
r =>  NIL
s =>  ((A 1 B 6 C 3))
;;; Note that the (setq r nil) does not affect the actions of
;;; the SETF because the value of R had already been saved in
;;; a temporary variable as part of the step 1. Only the CAR
;;; of this value will be retrieved, and subsequently modified
;;; after the value computation.
```

5.1.2.3 VALUES Forms as Places

A **values** *form* can be used as a *place*, provided that each of its *subforms* is also a *place* form.

A form such as

`(setf (values` *place-1* `...`*place-n*`)` *values-form*`)`

does the following:

1. The *subforms* of each nested *place* are evaluated in left-to-right order.
2. The *values-form* is evaluated, and the first store variable from each *place* is bound to its return values as if by **multiple-value-bind**.
3. If the *setf expansion* for any *place* involves more than one store variable, then the additional store variables are bound to **nil**.
4. The storing forms for each *place* are evaluated in left-to-right order.

The storing form in the *setf expansion* of **values** returns as *multiple values*[2] the values of the store variables in step 2. That is, the number of values returned is the same as the number of *place* forms. This may be more or fewer values than are produced by the *values-form*.

5.1.2.4 THE Forms as Places

A **the** *form* can be used as a *place*, in which case the declaration is transferred to the *newvalue* form, and the resulting **setf** is analyzed. For example,

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

is processed as if it were

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

5.1.2.5 APPLY Forms as Places

The following situations involving **setf** of **apply** must be supported:

* `(setf (apply #'aref` *array subscript*`*` *more-subscripts*`)` *new-element*`)`
* `(setf (apply #'bit` *array subscript*`*` *more-subscripts*`)` *new-element*`)`
* `(setf (apply #'sbit` *array subscript*`*` *more-subscripts*`)` *new-element*`)`

In all three cases, the *element* of *array* designated by the concatenation of *subscripts* and *more-subscripts* (i.e., the same *element* which would be *read* by the call to *apply* if it were not part of a **setf** *form*) is changed to have the *value* given by *new-element*. For these usages, the function name (**aref**, **bit**, or **sbit**) must refer to the global function definition, rather than a locally defined *function*.

No other *standardized function* is required to be supported, but an *implementation* may define such support. An *implementation* may also define support for *implementation-defined operators*.

If a user-defined *function* is used in this context, the following equivalence is true, except that care is taken to preserve proper left-to-right evaluation of argument *subforms*:

```
(setf (apply #'name arg*) val)
==  (apply #'(setf name) val arg*)
```

5.1.2.6 Setf Expansions and Places

Any *compound form* for which the *operator* has a *setf expander* defined can be used as a *place*. The *operator* must refer to the global function definition, rather than a locally defined *function* or *macro*.

5.1.2.7 Macro Forms as Places

A *macro form* can be used as a *place*, in which case Common Lisp expands the *macro form* as if by **macroexpand-1** and then uses the *macro expansion* in place of the original *place*. Such *macro expansion* is attempted only after exhausting all other possibilities other than expanding into a call to a function named `(setf` *reader*`)`.

5.1.2.8 Symbol Macros as Places

A reference to a *symbol* that has been *established* as a *symbol macro* can be used as a *place*. In this case, **setf** expands the reference and then analyzes the resulting *form*.

5.1.2.9 Other Compound Forms as Places

For any other *compound form* for which the *operator* is a *symbol f*, the **setf** *form* expands into a call to the *function* named `(setf` *f*`)`. The first *argument* in the newly constructed *function form* is *newvalue* and the remaining *arguments* are the remaining *elements* of *place*. This expansion occurs regardless of whether *f* or `(setf` *f*`)` is defined as a *function* locally, globally, or not at all. For example,

`(setf (`*f arg1 arg2* `...)` *new-value*`)`

expands into a form with the same effect and value as

```
(let ((#:temp-1 arg1)          ;force correct order of evaluation
      (#:temp-2 arg2)
      ...
      (#:temp-0 new-value))
  (funcall (function (setf f)) #:temp-0 #:temp-1 #:temp-2...))
```

A *function* named `(setf` *f*`)` must return its first argument as its only value in order to preserve the semantics of **setf**.

5.1.3 Treatment of Other Macros Based on SETF

For each of the "read-modify-write" *operators* in the next figure, and for any additional *macros* defined by the *programmer* using **define-modify-macro**, an exception is made to the normal rule of left-to-right evaluation of arguments. Evaluation of *argument forms* occurs in left-to-right order, with the exception that for the *place argument*, the actual *read* of the "old value" from that *place* happens after all of the *argument form evaluations*, and just before a "new value" is computed and *written* back into the *place*.

Specifically, each of these *operators* can be viewed as involving a *form* with the following general syntax:

```
(operator preceding-form* place following-form*)
```

The evaluation of each such *form* proceeds like this:

1. *Evaluate* each of the *preceding-forms*, in left-to-right order.
2. *Evaluate* the *subforms* of the *place*, in the order specified by the second value of the *setf expansion* for that *place*.
3. *Evaluate* each of the *following-forms*, in left-to-right order.
4. *Read* the old value from *place*.
5. Compute the new value.
6. Store the new value into *place*.

```
decf  pop   pushnew
incf  push  remf
```

Figure 5-9. Read-Modify-Write Macros

5.2 Transfer of Control to an Exit Point

When a transfer of control is initiated by **go**, **return-from**, or **throw** the following events occur in order to accomplish the transfer of control. Note that for **go**, the *exit point* is the *form* within the **tagbody** that is being executed at the time the **go** is performed; for **return-from**, the *exit point* is the corresponding **block** *form*; and for **throw**, the *exit point* is the corresponding **catch** *form*.

1. Intervening *exit points* are "abandoned" (i.e., their *extent* ends and it is no longer valid to attempt to transfer control through them).
2. The cleanup clauses of any intervening **unwind-protect** clauses are evaluated.
3. Intervening dynamic *bindings* of **special** variables, *catch tags*, *condition handlers*, and *restarts* are undone.
4. The *extent* of the *exit point* being invoked ends, and control is passed to the target.

The extent of an exit being "abandoned" because it is being passed over ends as soon as the transfer of control is initiated. That is, event 1 occurs at the beginning of the initiation of the transfer of control. The consequences are undefined if an attempt is made to transfer control to an *exit point* whose *dynamic extent* has ended.

Events 2 and 3 are actually performed interleaved, in the order corresponding to the reverse order in which they were established. The effect of this is that the cleanup clauses of an **unwind-protect** see the same dynamic *bindings* of variables and *catch tags* as were visible when the **unwind-protect** was entered.

Event 4 occurs at the end of the transfer of control.

6. Iteration

6.1 The LOOP Facility

6.1.1 Overview of the Loop Facility

The **loop** *macro* performs iteration.

6.1.1.1 Simple vs Extended Loop

loop *forms* are partitioned into two categories: simple **loop** *forms* and extended **loop** *forms*.

6.1.1.1.1 Simple Loop

A simple **loop** *form* is one that has a body containing only *compound forms*. 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. A simple **loop** *form* establishes an *implicit block* named **nil**. The execution of a simple **loop** can be terminated by explicitly transfering control to the *implicit block* (using **return** or **return-from**) or to some *exit point* outside of the *block* (e.g., using **throw**, **go**, or **return-from**).

6.1.1.1.2 Extended Loop

An extended **loop** *form* is one that has a body containing *atomic expressions*. When the **loop** *macro* processes such a *form*, it invokes a facility that is commonly called "the Loop Facility."

The Loop Facility provides standardized access to mechanisms commonly used in iterations through Loop schemas, which are introduced by *loop keywords*.

The body of an extended **loop** *form* is divided into **loop** clauses, each which is in turn made up of *loop keywords* and *forms*.

6.1.1.2 Loop Keywords

Loop keywords are not true *keywords*[1]; they are special *symbols*, recognized by *name* rather than *object* identity, that are meaningful only to the **loop** facility. A *loop keyword* is a *symbol* but is recognized by its *name* (not its identity), regardless of the *packages* in which it is *accessible*.

In general, *loop keywords* are not *external symbols* of the `COMMON-LISP` package, except in the coincidental situation that a *symbol* with the same name as a *loop keyword* was needed for some other purpose in Common Lisp. For example, there is a *symbol* in the `COMMON-LISP` package whose *name* is `"UNLESS"` but not one whose *name* is `"UNTIL"`.

If no *loop keywords* are supplied in a **loop** *form*, the Loop Facility executes the loop body repeatedly; see Section 6.1.1.1.1 (Simple Loop).

6.1.1.3 Parsing Loop Clauses

The syntactic parts of an extended **loop** *form* are called clauses; the rules for parsing are determined by that clause's keyword. The following example shows a **loop** *form* with six clauses:

```
(loop for i from 1 to (compute-top-value)       ; first clause
      while (not (unacceptable i))              ; second clause
      collect (square i)                        ; third clause
      do (format t "Working on ~D now" i)       ; fourth clause
      when (evenp i)                            ; fifth clause
        do (format t "~D is a non-odd number" i)
      finally (format t "About to exit!"))      ; sixth clause
```

Each *loop keyword* introduces either a compound loop clause or a simple loop clause that can consist of a *loop keyword* followed by a single *form*. The number of *forms* in a clause is determined by the *loop keyword* that begins the clause and by the auxiliary keywords in the clause. The keywords `do`, `doing`, `initially`, and `finally` are the only loop keywords that can take any number of *forms* and group them as an *implicit progn*.

Loop clauses can contain auxiliary keywords, which are sometimes called prepositions. For example, the first clause in the code above includes the prepositions `from` and `to`, which mark the value from which stepping begins and the value at which stepping ends.

For detailed information about **loop** syntax, see the *macro* **loop**.

6.1.1.4 Expanding Loop Forms

A **loop** *macro form* expands into a *form* containing one or more binding forms (that *establish bindings* of loop variables) and a **block** and a **tagbody** (that express a looping control structure). The variables established in **loop** are bound as if by **let** or **lambda**.

Implementations can interleave the setting of initial values with the *bindings*. However, the assignment of the initial values is always calculated in the order specified by the user. A variable is thus sometimes bound to a meaningless value of the correct *type*, and then later in the prologue it is set to the true initial value by using **setq**. One implication of this interleaving is that it is *implementation-dependent* whether the *lexical environment* in which the initial value *forms* (variously called the *form1*, *form2*, *form3*, *step-fun*, *vector*, *hash-table*, and *package*) in any *for-as-subclause*, except *for-as-equals-then*, are *evaluated* includes only the loop variables preceding that *form* or includes more or all of the loop variables; the *form1* and *form2* in a *for-as-equals-then* form includes the *lexical environment* of all the loop variables.

After the *form* is expanded, it consists of three basic parts in the **tagbody**: the loop prologue, the loop body, and the loop epilogue.

Loop prologue
: The loop prologue contains *forms* that are executed before iteration begins, such as any automatic variable initializations prescribed by the *variable* clauses, along with any `initially` clauses in the order they appear in the source.

Loop body
: The loop body contains those *forms* that are executed during iteration, including application-specific calculations, termination tests, and variable *stepping*[1].

Loop epilogue
: The loop epilogue contains *forms* that are executed after iteration terminates, such as `finally` clauses, if any, along with any implicit return value from an *accumulation* clause or an *termination-test* clause.

Some clauses from the source *form* contribute code only to the loop prologue; these clauses must come before other clauses that are in the main body of the **loop** form. Others contribute code only to the loop epilogue. All other clauses contribute to the final translated *form* in the same order given in the original source *form* of the **loop**.

Expansion of the **loop** macro produces an *implicit block* named **nil** unless `named` is supplied. Thus, **return-from** (and sometimes **return**) can be used to return values from **loop** or to exit **loop**.

6.1.1.5 Summary of Loop Clauses

Loop clauses fall into one of the following categories:

6.1.1.5.1 Summary of Variable Initialization and Stepping Clauses

The `for` and `as` constructs provide iteration control clauses that establish a variable to be initialized. `for` and `as` clauses can be combined with the loop keyword `and` to get *parallel* initialization and *stepping*[1]. Otherwise, the initialization and *stepping*[1] are *sequential*.

The `with` construct is similar to a single **let** clause. `with` clauses can be combined using the *loop keyword* `and` to get *parallel* initialization.

For more information, see Section 6.1.2 (Variable Initialization and Stepping Clauses).

6.1.1.5.2 Summary of Value Accumulation Clauses

The `collect` (or `collecting`) construct takes one *form* in its clause and adds the value of that *form* to the end of a *list* of values. By default, the *list* of values is returned when the **loop** finishes.

The `append` (or `appending`) construct takes one *form* in its clause and appends the value of that *form* to the end of a *list* of values. By default, the *list* of values is returned when the **loop** finishes.

The `nconc` (or `nconcing`) construct is similar to the `append` construct, but its *list* values are concatenated as if by the function `nconc`. By default, the *list* of values is returned when the **loop** finishes.

The `sum` (or `summing`) construct takes one *form* in its clause that must evaluate to a *number* and accumulates the sum of all these *numbers*. By default, the cumulative sum is returned when the **loop** finishes.

The `count` (or `counting`) construct takes one *form* in its clause and counts the number of times that the *form* evaluates to *true*. By default, the count is returned when the **loop** finishes.

The `minimize` (or `minimizing`) construct takes one *form* in its clause and determines the minimum value obtained by evaluating that *form*. By default, the minimum value is returned when the **loop** finishes.

The `maximize` (or `maximizing`) construct takes one *form* in its clause and determines the maximum value obtained by evaluating that *form*. By default, the maximum value is returned when the **loop** finishes.

For more information, see Section 6.1.3 (Value Accumulation Clauses).

6.1.1.5.3 Summary of Termination Test Clauses

The `for` and `as` constructs provide a termination test that is determined by the iteration control clause.

The `repeat` construct causes termination after a specified number of iterations. (It uses an internal variable to keep track of the number of iterations.)

The `while` construct takes one *form*, a *test*, and terminates the iteration if the *test* evaluates to *false*. A `while` clause is equivalent to the expression `(if (not` *test*`) (loop-finish))`.

The `until` construct is the inverse of `while`; it terminates the iteration if the *test* evaluates to any *non-nil* value. An `until` clause is equivalent to the expression `(if` *test* `(loop-finish))`.

The `always` construct takes one *form* and terminates the **loop** if the *form* ever evaluates to *false*; in this case, the **loop** *form* returns **nil**. Otherwise, it provides a default return value of **t**.

The `never` construct takes one *form* and terminates the **loop** if the *form* ever evaluates to *true*; in this case, the **loop** *form* returns **nil**. Otherwise, it provides a default return value of **t**.

The `thereis` construct takes one *form* and terminates the **loop** if the *form* ever evaluates to a *non-nil object*; in this case, the **loop** *form* returns that *object*. Otherwise, it provides a default return value of **nil**.

If multiple termination test clauses are specified, the **loop** *form* terminates if any are satisfied.

For more information, see Section 6.1.4 (Termination Test Clauses).

6.1.1.5.4 Summary of Unconditional Execution Clauses

The `do` (or `doing`) construct evaluates all *forms* in its clause.

The `return` construct takes one *form*. Any *values* returned by the *form* are immediately returned by the **loop** form. It is equivalent to the clause `do (return-from` *block-name value*`)`, where *block-name* is the name specified in a `named` clause, or **nil** if there is no `named` clause.

For more information, see Section 6.1.5 (Unconditional Execution Clauses).

6.1.1.5.5 Summary of Conditional Execution Clauses

The `if` and `when` constructs take one *form* as a test and a clause that is executed when the test *yields true*. The clause can be a value accumulation, unconditional, or another conditional clause; it can also be any combination of such clauses connected by the **loop** `and` keyword.

The **loop** `unless` construct is similar to the **loop** `when` construct except that it complements the test result.

The **loop** `else` construct provides an optional component of `if`, `when`, and `unless` clauses that is executed when an `if` or `when` test *yields false* or when an `unless` test *yields true*. The component is one of the clauses described under `if`.

The **loop** `end` construct provides an optional component to mark the end of a conditional clause.

For more information, see Section 6.1.6 (Conditional Execution Clauses).

6.1.1.5.6 Summary of Miscellaneous Clauses

The **loop** `named` construct gives a name for the *block* of the loop.

The **loop** `initially` construct causes its *forms* to be evaluated in the loop prologue, which precedes all **loop** code except for initial settings supplied by the constructs `with`, `for`, or `as`.

The **loop** `finally` construct causes its *forms* to be evaluated in the loop epilogue after normal iteration terminates.

For more information, see Section 6.1.7 (Miscellaneous Clauses).

6.1.1.6 Order of Execution

With the exceptions listed below, clauses are executed in the loop body in the order in which they appear in the source. Execution is repeated until a clause terminates the **loop** or until a **return**, **go**, or **throw** form is encountered which transfers control to a point outside of the loop. The following actions are exceptions to the linear order of execution:

* All variables are initialized first, regardless of where the establishing clauses appear in the source. The order of initialization follows the order of these clauses.
* The code for any `initially` clauses is collected into one **progn** in the order in which the clauses appear in the source. The collected code is executed once in the loop prologue after any implicit variable initializations.
* The code for any `finally` clauses is collected into one **progn** in the order in which the clauses appear in the source. The collected code is executed once in the loop epilogue before any implicit values from the accumulation clauses are returned. Explicit returns anywhere in the source, however, will exit the **loop** without executing the

epilogue code.
* A `with` clause introduces a variable *binding* and an optional initial value. The initial values are calculated in the order in which the `with` clauses occur.
* Iteration control clauses implicitly perform the following actions:
    -- initialize variables;
    -- *step* variables, generally between each execution of the loop body;
    -- perform termination tests, generally just before the execution of the loop body.

6.1.1.7 Destructuring

The *d-type-spec* argument is used for destructuring. If the *d-type-spec* argument consists solely of the *type* **fixnum**, **float**, **t**, or **nil**, the `of-type` keyword is optional. The `of-type` construct is optional in these cases to provide backwards compatibility; thus, the following two expressions are the same:

```
;;; This expression uses the old syntax for type specifiers.
 (loop for i fixnum upfrom 3 ...)

;;; This expression uses the new syntax for type specifiers.
 (loop for i of-type fixnum upfrom 3 ...)

;; Declare X and Y to be of type VECTOR and FIXNUM respectively.
 (loop for (x y) of-type (vector fixnum)
       in l do ...)
```

A *type specifier* for a destructuring pattern is a *tree* of *type specifiers* with the same shape as the *tree* of *variable names*, with the following exceptions:

* When aligning the *trees*, an *atom* in the *tree* of *type specifiers* that matches a *cons* in the variable tree declares the same *type* for each variable in the subtree rooted at the *cons*.
* A *cons* in the *tree* of *type specifiers* that matches an *atom* in the *tree* of *variable names* is a *compound type specifer*.

Destructuring allows *binding* of a set of variables to a corresponding set of values anywhere that a value can normally be bound to a single variable. During **loop** expansion, each variable in the variable list is matched with the values in the values list. If there are more variables in the variable list than there are values in the values list, the remaining variables are given a value of **nil**. If there are more values than variables listed, the extra values are discarded.

To assign values from a list to the variables `a`, `b`, and `c`, the `for` clause could be used to bind the variable `numlist` to the *car* of the supplied *form*, and then another `for` clause could be used to bind the variables `a`, `b`, and `c` *sequentially*.

```
;; Collect values by using FOR constructs.
 (loop for numlist in '((1 2 4.0) (5 6 8.3) (8 9 10.4))
       for a of-type integer = (first numlist)
       and b of-type integer = (second numlist)
       and c of-type float = (third numlist)
       collect (list c b a))
=>  ((4.0 2 1) (8.3 6 5) (10.4 9 8))
```

Destructuring makes this process easier by allowing the variables to be bound in each loop iteration. *Types* can be declared by using a list of *type-spec* arguments. If all the *types* are the same, a shorthand destructuring syntax can be used, as the second example illustrates.

```
;; Destructuring simplifies the process.
 (loop for (a b c) of-type (integer integer float) in
       '((1 2 4.0) (5 6 8.3) (8 9 10.4))
       collect (list c b a))
=>  ((4.0 2 1) (8.3 6 5) (10.4 9 8))


;; If all the types are the same, this way is even simpler.
 (loop for (a b c) of-type float in
       '((1.0 2.0 4.0) (5.0 6.0 8.3) (8.0 9.0 10.4))
       collect (list c b a))
=>  ((4.0 2.0 1.0) (8.3 6.0 5.0) (10.4 9.0 8.0))
```

If destructuring is used to declare or initialize a number of groups of variables into *types*, the *loop keyword* `and` can be used to simplify the process further.

```
;; Initialize and declare variables in parallel by using the AND construct.
 (loop with (a b) of-type float = '(1.0 2.0)
       and (c d) of-type integer = '(3 4)
       and (e f)
       return (list a b c d e f))
=>  (1.0 2.0 3 4 NIL NIL)
```

If **nil** is used in a destructuring list, no variable is provided for its place.

```
 (loop for (a nil b) = '(1 2 3)
       do (return (list a b)))
=>  (1 3)
```

Note that *dotted lists* can specify destructuring.

```
 (loop for (x . y) = '(1 . 2)
       do (return y))
=>  2
 (loop for ((a . b) (c . d)) of-type ((float . float) (integer . integer)) in
       '(((1.2 . 2.4) (3 . 4)) ((3.4 . 4.6) (5 . 6)))
       collect (list a b c d))
=>  ((1.2 2.4 3 4) (3.4 4.6 5 6))
```

An error of *type* **program-error** is signaled (at macro expansion time) if the same variable is bound twice in any variable-binding clause of a single **loop** expression. Such variables include local variables, iteration control variables, and variables found by destructuring.

6.1.1.8 Restrictions on Side-Effects

See Section 3.6 (Traversal Rules and Side Effects).

6.1.2 Variable Initialization and Stepping Clauses

6.1.2.1 Iteration Control

Iteration control clauses allow direction of **loop** iteration. The *loop keywords* `for` and `as` designate iteration control clauses. Iteration control clauses differ with respect to the specification of termination tests and to the initialization and *stepping*[1] of loop variables. Iteration clauses by themselves do not cause the Loop Facility to return values, but they can be used in conjunction with value-accumulation clauses to return values.

All variables are initialized in the loop prologue. A *variable binding* has *lexical scope* unless it is proclaimed **special**; thus, by default, the variable can be *accessed* only by *forms* that lie textually within the **loop**. Stepping assignments are made in the loop body before any other *forms* are evaluated in the body.

The variable argument in iteration control clauses can be a destructuring list. A destructuring list is a *tree* whose *non-nil atoms* are *variable names*. See Section 6.1.1.7 (Destructuring).

The iteration control clauses `for`, `as`, and `repeat` must precede any other loop clauses, except `initially`, `with`, and `named`, since they establish variable *bindings*. When iteration control clauses are used in a **loop**, the corresponding termination tests in the loop body are evaluated before any other loop body code is executed.

If multiple iteration clauses are used to control iteration, variable initialization and *stepping*[1] occur *sequentially* by default. The `and` construct can be used to connect two or more iteration clauses when *sequential binding* and *stepping*[1] are not necessary. The iteration behavior of clauses joined by `and` is analogous to the behavior of the macro **do** with respect to **do***.

The `for` and `as` clauses iterate by using one or more local loop variables that are initialized to some value and that can be modified or *stepped*[1] after each iteration. For these clauses, iteration terminates when a local variable reaches some supplied value or when some other loop clause terminates iteration. At each iteration, variables can be *stepped*[1] by an increment or a decrement or can be assigned a new value by the evaluation of a *form*). Destructuring can be used to assign values to variables during iteration.

The `for` and `as` keywords are synonyms; they can be used interchangeably. There are seven syntactic formats for these constructs. In each syntactic format, the *type* of *var* can be supplied by the optional *type-spec* argument. If *var* is a destructuring list, the *type* supplied by the *type-spec* argument must appropriately match the elements of the list. By convention, `for` introduces new iterations and `as` introduces iterations that depend on a previous iteration specification.

6.1.2.1.1 The for-as-arithmetic subclause

In the *for-as-arithmetic* subclause, the `for` or `as` construct iterates from the value supplied by *form1* to the value supplied by *form2* in increments or decrements denoted by *form3*. Each expression is evaluated only once and must evaluate to a *number*. The variable *var* is bound to the value of *form1* in the first iteration and is *stepped*[1] by the value of *form3* in each succeeding iteration, or by 1 if *form3* is not provided. The following *loop keywords* serve as valid prepositions within this syntax. At least one of the prepositions must be used; and at most one from each line may be used in a single subclause.

`from` | `downfrom` | `upfrom`
`to` | `downto` | `upto` | `below` | `above`
`by`

The prepositional phrases in each subclause may appear in any order. For example, either "`from x by y`" or "`by y from x`" is permitted. However, because left-to-right order of evaluation is preserved, the effects will be different in the case of side effects. Consider:

```
(let ((x 1)) (loop for i from x by (incf x) to 10 collect i))
=>  (1 3 5 7 9)
(let ((x 1)) (loop for i by (incf x) from x to 10 collect i))
=>  (2 4 6 8 10)
```

The descriptions of the prepositions follow:

`from`
: The *loop keyword* `from` specifies the value from which *stepping*[1] begins, as supplied by *form1*. *Stepping*[1] is incremental by default. If decremental *stepping*[1] is desired, the preposition `downto` or `above` must be used with *form2*. For incremental *stepping*[1], the default `from` value is 0.

`downfrom, upfrom`

The *loop keyword* `downfrom` indicates that the variable *var* is decreased in decrements supplied by *form3*; the *loop keyword* `upfrom` indicates that *var* is increased in increments supplied by *form3*.

`to`

The *loop keyword* `to` marks the end value for *stepping*[1] supplied in *form2*. *Stepping*[1] is incremental by default. If decremental *stepping*[1] is desired, the preposition `downfrom` must be used with *form1*, or else the preposition `downto` or `above` should be used instead of `to` with *form2*.

`downto, upto`

The *loop keyword* `downto` specifies decremental *stepping*; the *loop keyword* `upto` specifies incremental *stepping*. In both cases, the amount of change on each step is specified by *form3*, and the **loop** terminates when the variable *var* passes the value of *form2*. Since there is no default for *form1* in decremental *stepping*[1], a *form1* value must be supplied (using `from` or `downfrom`) when `downto` is supplied.

`below, above`

The *loop keywords* `below` and `above` are analogous to `upto` and `downto` respectively. These keywords stop iteration just before the value of the variable *var* reaches the value supplied by *form2*; the end value of *form2* is not included. Since there is no default for *form1* in decremental *stepping*[1], a *form1* value must be supplied (using `from` or `downfrom`) when `above` is supplied.

`by`

The *loop keyword* `by` marks the increment or decrement supplied by *form3*. The value of *form3* can be any positive *number*. The default value is 1.

In an iteration control clause, the `for` or `as` construct causes termination when the supplied limit is reached. That is, iteration continues until the value *var* is stepped to the exclusive or inclusive limit supplied by *form2*. The range is exclusive if *form3* increases or decreases *var* to the value of *form2* without reaching that value; the loop keywords `below` and `above` provide exclusive limits. An inclusive limit allows *var* to attain the value of *form2*; `to`, `downto`, and `upto` provide inclusive limits.

6.1.2.1.1.1 Examples of for-as-arithmetic subclause

```
;; Print some numbers.
 (loop for i from 1 to 3
       do (print i))
>>  1
>>  2
>>  3
=>  NIL

;; Print every third number.
 (loop for i from 10 downto 1 by 3
       do (print i))
>>  10
>>  7
>>  4
>>  1
=>  NIL

;; Step incrementally from the default starting value.
 (loop for i below 3
       do (print i))
>>  0
>>  1
>>  2
=>  NIL
```

6.1.2.1.2 The for-as-in-list subclause

In the *for-as-in-list* subclause, the `for` or `as` construct iterates over the contents of a *list*. It checks for the end of the *list* as if by using **endp**. The variable *var* is bound to the successive elements of the *list* in *form1* before each iteration. At the end of each iteration, the function *step-fun* is applied to the *list*; the default value for *step-fun* is **cdr**. The *loop keywords* `in` and `by` serve as valid prepositions in this syntax. The `for` or `as` construct causes termination when the end of the *list* is reached.

6.1.2.1.2.1 Examples of for-as-in-list subclause

```
;; Print every item in a list.
 (loop for item in '(1 2 3) do (print item))
>>  1
>>  2
>>  3
=>  NIL

;; Print every other item in a list.
 (loop for item in '(1 2 3 4 5) by #'cddr
       do (print item))
>>  1
>>  3
>>  5
=>  NIL

;; Destructure a list, and sum the x values using fixnum arithmetic.
 (loop for (item . x) of-type (t . fixnum) in '((A . 1) (B . 2) (C . 3))
       unless (eq item 'B) sum x)
=>  4
```

6.1.2.1.3 The for-as-on-list subclause

In the *for-as-on-list* subclause, the `for` or `as` construct iterates over a *list*. It checks for the end of the *list* as if by using **atom**. The variable *var* is bound to the successive tails of the *list* in *form1*. At the end of each iteration, the function *step-fun* is applied to the *list*; the default value for *step-fun* is **cdr**. The *loop keywords* `on` and `by` serve as valid prepositions in this syntax. The `for` or `as` construct causes termination when the end of the *list* is reached.

6.1.2.1.3.1 Examples of for-as-on-list subclause

```
;; Collect successive tails of a list.
 (loop for sublist on '(a b c d)
       collect sublist)
=>  ((A B C D) (B C D) (C D) (D))

;; Print a list by using destructuring with the loop keyword ON.
 (loop for (item) on '(1 2 3)
       do (print item))
>>  1
>>  2
>>  3
=>  NIL
```

6.1.2.1.4 The for-as-equals-then subclause

In the *for-as-equals-then* subclause the `for` or `as` construct initializes the variable *var* by setting it to the result of evaluating *form1* on the first iteration, then setting it to the result of evaluating *form2* on the second and subsequent iterations. If *form2* is omitted, the construct uses *form1* on the second and subsequent iterations. The *loop keywords* = and `then` serve as valid prepositions in this syntax. This construct does not provide any termination tests.

6.1.2.1.4.1 Examples of for-as-equals-then subclause

```
;; Collect some numbers.
 (loop for item = 1 then (+ item 10)
       for iteration from 1 to 5
       collect item)
=>  (1 11 21 31 41)
```

6.1.2.1.5 The for-as-across subclause

In the *for-as-across* subclause the `for` or `as` construct binds the variable *var* to the value of each element in the array *vector*. The *loop keyword* `across` marks the array *vector*; `across` is used as a preposition in this syntax. Iteration stops when there are no more elements in the supplied *array* that can be referenced. Some implementations might recognize a **the** special form in the *vector* form to produce more efficient code.

6.1.2.1.5.1 Examples of for-as-across subclause

```
 (loop for char across (the simple-string (find-message channel))
       do (write-char char stream))
```

6.1.2.1.6 The for-as-hash subclause

In the *for-as-hash* subclause the `for` or `as` construct iterates over the elements, keys, and values of a *hash-table*. In this syntax, a compound preposition is used to designate access to a *hash table*. The variable *var* takes on the value of each hash key or hash value in the supplied *hash-table*. The following *loop keywords* serve as valid prepositions within this syntax:

`being`
: The keyword `being` introduces either the Loop schema `hash-key` or `hash-value`.

`each`, `the`
: The *loop keyword* `each` follows the *loop keyword* `being` when `hash-key` or `hash-value` is used. The *loop keyword* `the` is used with `hash-keys` and `hash-values` only for ease of reading. This agreement isn't required.

`hash-key`, `hash-keys`
: These *loop keywords* access each key entry of the *hash table*. If the name `hash-value` is supplied in a `using` construct with one of these Loop schemas, the iteration can optionally access the keyed value. The order in which the keys are accessed is undefined; empty slots in the *hash table* are ignored.

`hash-value`, `hash-values`
: These *loop keywords* access each value entry of a *hash table*. If the name `hash-key` is supplied in a `using` construct with one of these Loop schemas, the iteration can optionally access the key that corresponds to the value. The order in which the keys are accessed is undefined; empty slots in the *hash table* are ignored.

`using`
: The *loop keyword* `using` introduces the optional key or the keyed value to be accessed. It allows access to the hash key if iteration is over the hash values, and the hash value if iteration is over the hash keys.

`in`, `of`
: These loop prepositions introduce *hash-table*.

In effect

`being` {`each` | `the`} {`hash-value` | `hash-values` | `hash-key` | `hash-keys`} {`in` | `of`}

is a compound preposition.

Iteration stops when there are no more hash keys or hash values to be referenced in the supplied *hash-table*.

6.1.2.1.7 The for-as-package subclause

In the *for-as-package* subclause the `for` or `as` construct iterates over the *symbols* in a *package*. In this syntax, a compound preposition is used to designate access to a *package*. The variable *var* takes on the value of each *symbol* in the supplied *package*. The following *loop keywords* serve as valid prepositions within this syntax:

`being`
: The keyword `being` introduces either the Loop schema `symbol`, `present-symbol`, or `external-symbol`.

`each`, `the`
: The *loop keyword* `each` follows the *loop keyword* `being` when `symbol`, `present-symbol`, or `external-symbol` is used. The *loop keyword* `the` is used with `symbols`, `present-symbols`, and `external-symbols` only for ease of reading. This agreement isn't required.

`present-symbol`, `present-symbols`
: These Loop schemas iterate over the *symbols* that are *present* in a *package*. The *package* to be iterated over is supplied in the same way that *package* arguments to **find-package** are supplied. If the *package* for the iteration is not supplied, the *current package* is used. If a *package* that does not exist is supplied, an error of *type* **package-error** is signaled.

`symbol`, `symbols`
: These Loop schemas iterate over *symbols* that are *accessible* in a given *package*. The *package* to be iterated over is supplied in the same way that *package* arguments to **find-package** are supplied. If the *package* for the iteration is not supplied, the *current package* is used. If a *package* that does not exist is supplied, an error of *type* **package-error** is signaled.

`external-symbol`, `external-symbols`
: These Loop schemas iterate over the *external symbols* of a *package*. The *package* to be iterated over is supplied in the same way that *package* arguments to **find-package** are supplied. If the *package* for the iteration is not supplied, the *current package* is used. If a *package* that does not exist is supplied, an error of *type* **package-error** is signaled.

`in`, `of`
: These loop prepositions introduce *package*.

In effect

`being {each | the} {symbol | symbols | present-symbol | present-symbols | external-symbol|external-symbols} {in|of}`

is a compound preposition.

Iteration stops when there are no more *symbols* to be referenced in the supplied *package*.

6.1.2.1.7.1 Examples of for-as-package subclause

```
 (let ((*package* (make-package "TEST-PACKAGE-1")))
   ;; For effect, intern some symbols
   (read-from-string "(THIS IS A TEST)")
   (export (intern "THIS"))
   (loop for x being each present-symbol of *package*
         do (print x)))
>>  A
>>  TEST
>>  THIS
>>  IS
=>  NIL
```

6.1.2.2 Local Variable Initializations

When a **loop** *form* is executed, the local variables are bound and are initialized to some value. These local variables exist until **loop** iteration terminates, at which point they cease to exist. Implicit variables are also established by iteration control clauses and the `into` preposition of accumulation clauses.

The `with` construct initializes variables that are local to a loop. The variables are initialized one time only. If the optional *type-spec* argument is supplied for the variable *var*, but there is no related expression to be evaluated, *var* is initialized to an appropriate default value for its *type*. For example, for the types **t**, **number**, and **float**, the default values are **nil**, `0`, and `0.0` respectively. The consequences are undefined if a *type-spec* argument is supplied for *var* if the related expression returns a value that is not of the supplied *type*. By default, the `with` construct initializes variables *sequentially*; that is, one variable is assigned a value before the next expression is evaluated. However, by using the *loop keyword* `and` to join several `with` clauses, initializations can be forced to occur in *parallel*; that is, all of the supplied *forms* are evaluated, and the results are bound to the respective variables simultaneously.

Sequential binding is used when it is desireable for the initialization of some variables to depend on the values of previously bound variables. For example, suppose the variables `a`, `b`, and `c` are to be bound in sequence:

```
 (loop with a = 1
       with b = (+ a 2)
       with c = (+ b 3)
       return (list a b c))
=>  (1 3 6)
```

The execution of the above **loop** is equivalent to the execution of the following code:

```
 (block nil
   (let* ((a 1)
          (b (+ a 2))
          (c (+ b 3)))
     (tagbody
         (next-loop (return (list a b c))
                    (go next-loop)
                    end-loop))))
```

If the values of previously bound variables are not needed for the initialization of other local variables, an `and` clause can be used to specify that the bindings are to occur in *parallel*:

```
 (loop with a = 1
       and b = 2
       and c = 3
       return (list a b c))
=>  (1 2 3)
```

The execution of the above loop is equivalent to the execution of the following code:

```
 (block nil
   (let ((a 1)
         (b 2)
         (c 3))
     (tagbody
         (next-loop (return (list a b c))
                    (go next-loop)
                    end-loop))))
```

6.1.2.2.1 Examples of WITH clause

```
;; These bindings occur in sequence.
 (loop with a = 1
       with b = (+ a 2)
       with c = (+ b 3)
       return (list a b c))
=>  (1 3 6)

;; These bindings occur in parallel.
 (setq a 5 b 10)
=>  10
 (loop with a = 1
       and b = (+ a 2)
       and c = (+ b 3)
       return (list a b c))
=>  (1 7 13)

;; This example shows a shorthand way to declare local variables
;; that are of different types.
 (loop with (a b c) of-type (float integer float)
       return (format nil "~A ~A ~A" a b c))
=>  "0.0 0 0.0"

;; This example shows a shorthand way to declare local variables
;; that are the same type.
 (loop with (a b c) of-type float
       return (format nil "~A ~A ~A" a b c))
=>  "0.0 0.0 0.0"
```

6.1.3 Value Accumulation Clauses

The constructs `collect`, `collecting`, `append`, `appending`, `nconc`, `nconcing`, `count`, `counting`, `maximize`, `maximizing`, `minimize`, `minimizing`, `sum`, and `summing`, allow values to be accumulated in a **loop**.

The constructs `collect`, `collecting`, `append`, `appending`, `nconc`, and `nconcing`, designate clauses that accumulate values in *lists* and return them. The constructs `count`, `counting`, `maximize`, `maximizing`, `minimize`, `minimizing`, `sum`, and `summing` designate clauses that accumulate and return numerical values.

During each iteration, the constructs `collect` and `collecting` collect the value of the supplied *form* into a *list*. When iteration terminates, the *list* is returned. The argument *var* is set to the *list* of collected values; if *var* is supplied, the **loop** does not return the final *list* automatically. If *var* is not supplied, it is equivalent to supplying an internal name for *var* and returning its value in a `finally` clause. The *var* argument is bound as if by the construct `with`. No mechanism is provided for declaring the *type* of *var*; it must be of *type* **list**.

The constructs `append`, `appending`, `nconc`, and `nconcing` are similar to `collect` except that the values of the supplied *form* must be *lists*.

* The `append` keyword causes its *list* values to be concatenated into a single *list*, as if they were arguments to the *function* **append**.
* The `nconc` keyword causes its *list* values to be concatenated into a single *list*, as if they were arguments to the *function* **nconc**.

The argument *var* is set to the *list* of concatenated values; if *var* is supplied, **loop** does not return the final *list* automatically. The *var* argument is bound as if by the construct `with`. A *type* cannot be supplied for *var*; it must be of *type* **list**. The construct `nconc` destructively modifies its argument *lists*.

The `count` construct counts the number of times that the supplied *form* returns *true*. The argument *var* accumulates the number of occurrences; if *var* is supplied, **loop** does not return the final count automatically. The *var* argument is bound as if by the construct `with` to a zero of the appropriate type. Subsequent values (including any necessary coercions) are computed as if by the function **1+**. If `into` *var* is used, a *type* can be supplied for *var* with the *type-spec* argument; the consequences are unspecified if a nonnumeric *type* is supplied. If there is no `into` variable, the optional *type-spec* argument applies to the internal variable that is keeping the count. The default *type* is *implementation-dependent*; but it must be a *supertype* of *type* **fixnum**.

The `maximize` and `minimize` constructs compare the value of the supplied *form* obtained during the first iteration with values obtained in successive iterations. The maximum (for `maximize`) or minimum (for `minimize`) value encountered is determined (as if by the *function* **max** for `maximize` and as if by the *function* **min** for `minimize`) and returned. If the `maximize` or `minimize` clause is never executed, the accumulated value is unspecified. The argument *var* accumulates the maximum or minimum value; if *var* is supplied, **loop** does not return the maximum or minimum automatically. The *var* argument is bound as if by the construct `with`. If `into` *var* is used, a *type* can be supplied for *var* with the *type-spec* argument; the consequences are unspecified if a nonnumeric *type* is supplied. If there is no `into` variable, the optional *type-spec* argument applies to the internal variable that is keeping the maximum or minimum value. The default *type* is *implementation-dependent*; but it must be a *supertype* of *type* **real**.

The `sum` construct forms a cumulative sum of the successive *primary values* of the supplied *form* at each iteration. The argument *var* is used to accumulate the sum; if *var* is supplied, **loop** does not return the final sum automatically. The *var* argument is bound as if by the construct `with` to a zero of the appropriate type. Subsequent values (including any necessary coercions) are computed as if by the *function* +. If `into` *var* is used, a *type* can be supplied for *var* with the *type-spec* argument; the consequences are unspecified if a nonnumeric *type* is supplied. If there is no `into` variable, the optional *type-spec* argument applies to the internal variable that is keeping the sum. The default *type* is *implementation-dependent*; but it must be a *supertype* of *type* **number**.

If `into` is used, the construct does not provide a default return value; however, the variable is available for use in any `finally` clause.

Certain kinds of accumulation clauses can be combined in a **loop** if their destination is the same (the result of **loop** or an `into` *var*) because they are considered to accumulate conceptually compatible quantities. In particular, any elements of following sets of accumulation clauses can be mixed with other elements of the same set for the same destination in a **loop** *form*:

* `collect`, `append`, `nconc`
* `sum`, `count`
* `maximize`, `minimize`

```
;; Collect every name and the kids in one list by using
;; COLLECT and APPEND.
 (loop for name in '(fred sue alice joe june)
       for kids in '((bob ken) () () (kris sunshine) ())
       collect name
       append kids)
=>  (FRED BOB KEN SUE ALICE JOE KRIS SUNSHINE JUNE)
```

Any two clauses that do not accumulate the same *type* of *object* can coexist in a **loop** only if each clause accumulates its values into a different *variable*.

6.1.3.1 Examples of COLLECT clause

```
;; Collect all the symbols in a list.
 (loop for i in '(bird 3 4 turtle (1 . 4) horse cat)
       when (symbolp i) collect i)
=>  (BIRD TURTLE HORSE CAT)
```

```
;; Collect and return odd numbers.
 (loop for i from 1 to 10
       if (oddp i) collect i)
=>  (1 3 5 7 9)

;; Collect items into local variable, but don't return them.
 (loop for i in '(a b c d) by #'cddr
       collect i into my-list
       finally (print my-list))
>>  (A C)
=>  NIL
```

6.1.3.2 Examples of APPEND and NCONC clauses

```
;; Use APPEND to concatenate some sublists.
  (loop for x in '((a) (b) ((c)))
        append x)
=>  (A B (C))

;; NCONC some sublists together.  Note that only lists made by the
;; call to LIST are modified.
  (loop for i upfrom 0
        as x in '(a b (c))
        nconc (if (evenp i) (list x) nil))
=>  (A (C))
```

6.1.3.3 Examples of COUNT clause

```
 (loop for i in '(a b nil c nil d e)
       count i)
=>  5
```

6.1.3.4 Examples of MAXIMIZE and MINIMIZE clauses

```
 (loop for i in '(2 1 5 3 4)
       maximize i)
=>  5
 (loop for i in '(2 1 5 3 4)
       minimize i)
=>  1

;; In this example, FIXNUM applies to the internal variable that holds
;; the maximum value.
 (setq series '(1.2 4.3 5.7))
=>  (1.2 4.3 5.7)
 (loop for v in series
       maximize (round v) of-type fixnum)
=>  6

;; In this example, FIXNUM applies to the variable RESULT.
 (loop for v of-type float in series
       minimize (round v) into result of-type fixnum
       finally (return result))
=>  1
```

6.1.3.5 Examples of SUM clause

```
 (loop for i of-type fixnum in '(1 2 3 4 5)
       sum i)
=>  15
 (setq series '(1.2 4.3 5.7))
```

```
=>  (1.2 4.3 5.7)
 (loop for v in series
       sum (* 2.0 v))
=>  22.4
```

6.1.4 Termination Test Clauses

The `repeat` construct causes iteration to terminate after a specified number of times. The loop body executes *n* times, where *n* is the value of the expression *form*. The *form* argument is evaluated one time in the loop prologue. If the expression evaluates to 0 or to a negative *number*, the loop body is not evaluated.

The constructs `always`, `never`, `thereis`, `while`, `until`, and the macro **loop-finish** allow conditional termination of iteration within a **loop**.

The constructs `always`, `never`, and `thereis` provide specific values to be returned when a **loop** terminates. Using `always`, `never`, or `thereis` in a loop with value accumulation clauses that are not `into` causes an error of *type* **program-error** to be signaled (at macro expansion time). Since `always`, `never`, and `thereis` use the **return-from** *special operator* to terminate iteration, any `finally` clause that is supplied is not evaluated when exit occurs due to any of these constructs. In all other respects these constructs behave like the `while` and `until` constructs.

The `always` construct takes one *form* and terminates the **loop** if the *form* ever evaluates to **nil**; in this case, it returns **nil**. Otherwise, it provides a default return value of **t**. If the value of the supplied *form* is never **nil**, some other construct can terminate the iteration.

The `never` construct terminates iteration the first time that the value of the supplied *form* is *non-nil*; the **loop** returns **nil**. If the value of the supplied *form* is always **nil**, some other construct can terminate the iteration. Unless some other clause contributes a return value, the default value returned is **t**.

The `thereis` construct terminates iteration the first time that the value of the supplied *form* is *non-nil*; the **loop** returns the value of the supplied *form*. If the value of the supplied *form* is always **nil**, some other construct can terminate the iteration. Unless some other clause contributes a return value, the default value returned is **nil**.

There are two differences between the `thereis` and `until` constructs:

* The `until` construct does not return a value or **nil** based on the value of the supplied *form*.
* The `until` construct executes any `finally` clause. Since `thereis` uses the **return-from** *special operator* to terminate iteration, any `finally` clause that is supplied is not evaluated when exit occurs due to `thereis`.

The `while` construct allows iteration to continue until the supplied *form* evaluates to *false*. The supplied *form* is reevaluated at the location of the `while` clause.

The `until` construct is equivalent to `while (not` *form*`)...`. If the value of the supplied *form* is *non-nil*, iteration terminates.

Termination-test control constructs can be used anywhere within the loop body. The termination tests are used in the order in which they appear. If an `until` or `while` clause causes termination, any clauses that precede it in the source are still evaluated. If the `until` and `while` constructs cause termination, control is passed to the loop epilogue, where any `finally` clauses will be executed.

There are two differences between the `never` and `until` constructs:

* The `until` construct does not return **t** or **nil** based on the value of the supplied *form*.
* The `until` construct does not bypass any `finally` clauses. Since `never` uses the **return-from** *special operator* to terminate iteration, any `finally` clause that is supplied is not evaluated when exit occurs due to `never`.

In most cases it is not necessary to use **loop-finish** because other loop control clauses terminate the **loop**. The macro **loop-finish** is used to provide a normal exit from a nested conditional inside a **loop**. Since **loop-finish** transfers control to the loop epilogue, using **loop-finish** within a `finally` expression can cause infinite looping.

6.1.4.1 Examples of REPEAT clause

```
 (loop repeat 3
       do (format t "~&What I say three times is true.~%"))
>>  What I say three times is true.
>>  What I say three times is true.
>>  What I say three times is true.
=>  NIL
 (loop repeat -15
   do (format t "What you see is what you expect~%"))
=>  NIL
```

6.1.4.2 Examples of ALWAYS, NEVER, and THEREIS clauses

```
;; Make sure I is always less than 11 (two ways).
;; The FOR construct terminates these loops.
 (loop for i from 0 to 10
       always (< i 11))
=>  T
 (loop for i from 0 to 10
       never (> i 11))
=>  T

;; If I exceeds 10 return I; otherwise, return NIL.
;; The THEREIS construct terminates this loop.
 (loop for i from 0
       thereis (when (> i 10) i) )
=>  11

;;; The FINALLY clause is not evaluated in these examples.
 (loop for i from 0 to 10
       always (< i 9)
       finally (print "you won't see this"))
=>  NIL
 (loop never t
       finally (print "you won't see this"))
=>  NIL
 (loop thereis "Here is my value"
       finally (print "you won't see this"))
=>  "Here is my value"

;; The FOR construct terminates this loop, so the FINALLY clause
;; is evaluated.
 (loop for i from 1 to 10
       thereis (> i 11)
       finally (prin1 'got-here))
>>  GOT-HERE
=>  NIL

;; If this code could be used to find a counterexample to Fermat's
;; last theorem, it would still not return the value of the
;; counterexample because all of the THEREIS clauses in this example
;; only return T.  But if Fermat is right, that won't matter
;; because this won't terminate.

 (loop for z upfrom 2
       thereis
         (loop for n upfrom 3 below (log z 2)
               thereis
```

```
                (loop for x below z
                      thereis
                        (loop for y below z
                              thereis (= (+ (expt x n) (expt y n))
                                         (expt z n))))))
```

6.1.4.3 Examples of WHILE and UNTIL clauses

```
 (loop while (hungry-p) do (eat))

;; UNTIL NOT is equivalent to WHILE.
 (loop until (not (hungry-p)) do (eat))

;; Collect the length and the items of STACK.
 (let ((stack '(a b c d e f)))
   (loop for item = (length stack) then (pop stack)
         collect item
         while stack))
=>  (6 A B C D E F)

;; Use WHILE to terminate a loop that otherwise wouldn't terminate.
;; Note that WHILE occurs after the WHEN.
 (loop for i fixnum from 3
       when (oddp i) collect i
       while (< i 5))
=>  (3 5)
```

6.1.5 Unconditional Execution Clauses

The `do` and `doing` constructs evaluate the supplied *forms* wherever they occur in the expanded form of **loop**. The *form* argument can be any *compound form*. Each *form* is evaluated in every iteration. Because every loop clause must begin with a *loop keyword*, the keyword `do` is used when no control action other than execution is required.

The `return` construct takes one *form*. Any *values* returned by the *form* are immediately returned by the **loop** form. It is equivalent to the clause `do (return-from` *block-name value*`)`, where *block-name* is the name specified in a `named` clause, or **nil** if there is no `named` clause.

6.1.5.1 Examples of unconditional execution

```
;; Print numbers and their squares.
;; The DO construct applies to multiple forms.
 (loop for i from 1 to 3
       do (print i)
          (print (* i i)))
>>  1
>>  1
>>  2
>>  4
>>  3
>>  9
=>  NIL
```

6.1.6 Conditional Execution Clauses

The `if`, `when`, and `unless` constructs establish conditional control in a **loop**. If the test passes, the succeeding loop clause is executed. If the test does not pass, the succeeding clause is skipped, and program control moves to the clause that follows the *loop keyword* `else`. If the test does not pass and no `else` clause is supplied, control is transferred to the clause or construct following the entire conditional clause.

If conditional clauses are nested, each `else` is paired with the closest preceding conditional clause that has no associated `else` or `end`.

In the `if` and `when` clauses, which are synonymous, the test passes if the value of *form* is *true*.

In the `unless` clause, the test passes if the value of *form* is *false*.

Clauses that follow the test expression can be grouped by using the *loop keyword* `and` to produce a conditional block consisting of a compound clause.

The *loop keyword* `it` can be used to refer to the result of the test expression in a clause. Use the *loop keyword* `it` in place of the form in a `return` clause or an *accumulation* clause that is inside a conditional execution clause. If multiple clauses are connected with `and`, the `it` construct must be in the first clause in the block.

The optional *loop keyword* `end` marks the end of the clause. If this keyword is not supplied, the next *loop keyword* marks the end. The construct `end` can be used to distinguish the scoping of compound clauses.

6.1.6.1 Examples of WHEN clause

```
;; Signal an exceptional condition.
 (loop for item in '(1 2 3 a 4 5)
       when (not (numberp item))
        return (cerror "enter new value" "non-numeric value: ~s" item))
Error: non-numeric value: A

;; The previous example is equivalent to the following one.
 (loop for item in '(1 2 3 a 4 5)
       when (not (numberp item))
        do (return
            (cerror "Enter new value" "non-numeric value: ~s" item)))
Error: non-numeric value: A

;; This example parses a simple printed string representation from
;; BUFFER (which is itself a string) and returns the index of the
;; closing double-quote character.
 (let ((buffer "\"a\" \"b\""))
   (loop initially (unless (char= (char buffer 0) #\")
                     (loop-finish))
         for i of-type fixnum from 1 below (length (the string buffer))
         when (char= (char buffer i) #\")
          return i))
=>  2

;; The collected value is returned.
 (loop for i from 1 to 10
       when (> i 5)
         collect i
       finally (prin1 'got-here))
>>  GOT-HERE
=>  (6 7 8 9 10)

;; Return both the count of collected numbers and the numbers.
 (loop for i from 1 to 10
       when (> i 5)
         collect i into number-list
         and count i into number-count
       finally (return (values number-count number-list)))
=>  5, (6 7 8 9 10)
```

6.1.7 Miscellaneous Clauses

6.1.7.1 Control Transfer Clauses

The `named` construct establishes a name for an *implicit block* surrounding the entire **loop** so that the **return-from** *special operator* can be used to return values from or to exit **loop**. Only one name per **loop** *form* can be assigned. If used, the `named` construct must be the first clause in the loop expression.

The `return` construct takes one *form*. Any *values* returned by the *form* are immediately returned by the **loop** form. This construct is similar to the **return-from** *special operator* and the **return** *macro*. The `return` construct does not execute any `finally` clause that the **loop** *form* is given.

6.1.7.1.1 Examples of NAMED clause

```
;; Just name and return.
 (loop named max
       for i from 1 to 10
       do (print i)
       do (return-from max 'done))
>>  1
=>  DONE
```

6.1.7.2 Initial and Final Execution

The `initially` and `finally` constructs evaluate forms that occur before and after the loop body.

The `initially` construct causes the supplied *compound-forms* to be evaluated in the loop prologue, which precedes all loop code except for initial settings supplied by constructs `with`, `for`, or `as`. The code for any `initially` clauses is executed in the order in which the clauses appeared in the **loop**.

The `finally` construct causes the supplied *compound-forms* to be evaluated in the loop epilogue after normal iteration terminates. The code for any `finally` clauses is executed in the order in which the clauses appeared in the **loop**. The collected code is executed once in the loop epilogue before any implicit values are returned from the accumulation clauses. An explicit transfer of control (e.g., by **return**, **go**, or **throw**) from the loop body, however, will exit the **loop** without executing the epilogue code.

Clauses such as `return`, `always`, `never`, and `thereis` can bypass the `finally` clause. **return** (or **return-from**, if the `named` option was supplied) can be used after `finally` to return values from a **loop**. Such an *explicit return* inside the `finally` clause takes precedence over returning the accumulation from clauses supplied by such keywords as `collect`, `nconc`, `append`, `sum`, `count`, `maximize`, and `minimize`; the accumulation values for these preempted clauses are not returned by **loop** if **return** or **return-from** is used.

6.1.8 Examples of Miscellaneous Loop Features

```
(let ((i 0))                      ; no loop keywords are used
   (loop (incf i) (if (= i 3) (return i)))) =>  3
(let ((i 0)(j 0))
   (tagbody
     (loop (incf j 3) (incf i) (if (= i 3) (go exit)))
     exit)
   j) =>  9
```

In the following example, the variable `x` is stepped before `y` is stepped; thus, the value of `y` reflects the updated value of `x`:

```
 (loop for x from 1 to 10
       for y = nil then x
       collect (list x y))
=>  ((1 NIL) (2 2) (3 3) (4 4) (5 5) (6 6) (7 7) (8 8) (9 9) (10 10))
```

In this example, x and y are stepped in *parallel*:

```
 (loop for x from 1 to 10
       and y = nil then x
       collect (list x y))
=>  ((1 NIL) (2 1) (3 2) (4 3) (5 4) (6 5) (7 6) (8 7) (9 8) (10 9))
```

6.1.8.1 Examples of clause grouping

```
;; Group conditional clauses.
 (loop for i in '(1 324 2345 323 2 4 235 252)
       when (oddp i)
         do (print i)
         and collect i into odd-numbers
         and do (terpri)
       else                             ; I is even.
         collect i into even-numbers
       finally
         (return (values odd-numbers even-numbers)))
>>  1
>>
>>  2345
>>
>>  323
>>
>>  235
=>  (1 2345 323 235), (324 2 4 252)

;; Collect numbers larger than 3.
 (loop for i in '(1 2 3 4 5 6)
       when (and (> i 3) i)
       collect it)                     ; IT refers to (and (> i 3) i).
=>  (4 5 6)

;; Find a number in a list.
 (loop for i in '(1 2 3 4 5 6)
       when (and (> i 3) i)
       return it)
=>  4

;; The above example is similar to the following one.
 (loop for i in '(1 2 3 4 5 6)
       thereis (and (> i 3) i))
=>  4


;; Nest conditional clauses.
 (let ((list '(0 3.0 apple 4 5 9.8 orange banana)))
   (loop for i in list
         when (numberp i)
           when (floatp i)
             collect i into float-numbers
           else                                 ; Not (floatp i)
             collect i into other-numbers
         else                                   ; Not (numberp i)
           when (symbolp i)
             collect i into symbol-list
           else                                 ; Not (symbolp i)
             do (error "found a funny value in list ~S, value ~S~%" list i)
```

```
         finally (return (values float-numbers other-numbers symbol-list))))
=>  (3.0 9.8), (0 4 5), (APPLE ORANGE BANANA)

;; Without the END preposition, the last AND would apply to the
;; inner IF rather than the outer one.
 (loop for x from 0 to 3
       do (print x)
       if (zerop (mod x 2))
         do (princ " a")
          and if (zerop (floor x 2))
                 do (princ " b")
                 end
          and do (princ " c"))
>>  0  a b c
>>  1
>>  2  a c
>>  3
=>  NIL
```

6.1.9 Notes about Loop

Types can be supplied for loop variables. It is not necessary to supply a *type* for any variable, but supplying the *type* can ensure that the variable has a correctly typed initial value, and it can also enable compiler optimizations (depending on the *implementation*).

The clause `repeat` *n* ... is roughly equivalent to a clause such as

```
(loop for internal-variable downfrom (- n 1) to 0 ...)
```

but in some *implementations*, the `repeat` construct might be more efficient.

Within the executable parts of the loop clauses and around the entire **loop** form, variables can be bound by using **let**.

Use caution when using a variable named `IT` (in any *package*) in connection with **loop**, since `it` is a *loop keyword* that can be used in place of a *form* in certain contexts.

There is no *standardized* mechanism for users to add extensions to **loop**.

7. Objects

7.1 Object Creation and Initialization

The *generic function* **make-instance** creates and returns a new *instance* of a *class*. The first argument is a *class* or the *name* of a *class*, and the remaining arguments form an *initialization argument list*.

The initialization of a new *instance* consists of several distinct steps, including the following: combining the explicitly supplied initialization arguments with default values for the unsupplied initialization arguments, checking the validity of the initialization arguments, allocating storage for the *instance*, filling *slots* with values, and executing user-supplied *methods* that perform additional initialization. Each step of **make-instance** is implemented by a *generic function* to provide a mechanism for customizing that step. In addition, **make-instance** is itself a *generic function* and thus also can be customized.

The object system specifies system-supplied primary *methods* for each step and thus specifies a well-defined standard behavior for the entire initialization process. The standard behavior provides four simple mechanisms for controlling initialization:

* Declaring a *symbol* to be an initialization argument for a *slot*. An initialization argument is declared by using the `:initarg` slot option to **defclass**. This provides a mechanism for supplying a value for a *slot* in a call to **make-instance**.
* Supplying a default value form for an initialization argument. Default value forms for initialization arguments are defined by using the `:default-initargs` class option to **defclass**. If an initialization argument is not explicitly provided as an argument to **make-instance**, the default value form is evaluated in the lexical environment of the **defclass** form that defined it, and the resulting value is used as the value of the initialization argument.
* Supplying a default initial value form for a *slot*. A default initial value form for a *slot* is defined by using the `:initform` slot option to **defclass**. If no initialization argument associated with that *slot* is given as an argument to **make-instance** or is defaulted by `:default-initargs`, this default initial value form is evaluated in the lexical environment of the **defclass** form that defined it, and the resulting value is stored in the *slot*. The `:initform` form for a *local slot* may be used when creating an *instance*, when updating an *instance* to conform to a redefined *class*, or when updating an *instance* to conform to the definition of a different *class*. The `:initform` form for a *shared slot* may be used when defining or re-defining the *class*.
* Defining *methods* for **initialize-instance** and **shared-initialize**. The slot-filling behavior described above is implemented by a system-supplied primary *method* for **initialize-instance** which invokes **shared-initialize**. The *generic function* **shared-initialize** implements the parts of initialization shared by these four situations: when making an *instance*, when re-initializing an *instance*, when updating an *instance* to conform to a redefined *class*, and when updating an *instance* to conform to the definition of a different *class*. The system-supplied primary *method* for **shared-initialize** directly implements the slot-filling behavior described above, and **initialize-instance** simply invokes **shared-initialize**.

7.1.1 Initialization Arguments

An initialization argument controls *object* creation and initialization. It is often convenient to use keyword *symbols* to name initialization arguments, but the *name* of an initialization argument can be any *symbol*, including **nil**. An initialization argument can be used in two ways: to fill a *slot* with a value or to provide an argument for an initialization *method*. A single initialization argument can be used for both purposes.

An *initialization argument list* is a *property list* of initialization argument names and values. Its structure is identical to a *property list* and also to the portion of an argument list processed for `&key` parameters. As in those lists, if an initialization argument name appears more than once in an initialization argument list, the leftmost occurrence supplies the value and the remaining occurrences are ignored. The arguments to **make-instance** (after the first argument) form an *initialization argument list*.

An initialization argument can be associated with a *slot*. If the initialization argument has a value in the *initialization argument list*, the value is stored into the *slot* of the newly created *object*, overriding any `:initform` form associated with the *slot*. A single initialization argument can initialize more than one *slot*. An initialization argument that initializes a *shared slot* stores its value into the *shared slot*, replacing any previous value.

An initialization argument can be associated with a *method*. When an *object* is created and a particular initialization argument is supplied, the *generic functions* **initialize-instance**, **shared-initialize**, and **allocate-instance** are called with that initialization argument's name and value as a keyword argument pair. If a value for the initialization argument is not supplied in the *initialization argument list*, the *method*'s *lambda list* supplies a default value.

Initialization arguments are used in four situations: when making an *instance*, when re-initializing an *instance*, when updating an *instance* to conform to a redefined *class*, and when updating an *instance* to conform to the definition of a different *class*.

Because initialization arguments are used to control the creation and initialization of an *instance* of some particular *class*, we say that an initialization argument is "an initialization argument for" that *class*.

7.1.2 Declaring the Validity of Initialization Arguments

Initialization arguments are checked for validity in each of the four situations that use them. An initialization argument may be valid in one situation and not another. For example, the system-supplied primary *method* for **make-instance** defined for the *class* **standard-class** checks the validity of its initialization arguments and signals an error if an initialization argument is supplied that is not declared as valid in that situation.

There are two means for declaring initialization arguments valid.

* Initialization arguments that fill *slots* are declared as valid by the `:initarg` slot option to **defclass**. The `:initarg` slot option is inherited from *superclasses*. Thus the set of valid initialization arguments that fill *slots* for a *class* is the union of the initialization arguments that fill *slots* declared as valid by that *class* and its *superclasses*. Initialization arguments that fill *slots* are valid in all four contexts.
* Initialization arguments that supply arguments to *methods* are declared as valid by defining those *methods*. The keyword name of each keyword parameter specified in the *method*'s *lambda list* becomes an initialization argument for all *classes* for which the *method* is applicable. The presence of `&allow-other-keys` in the *lambda list* of an applicable method disables validity checking of initialization arguments. Thus *method* inheritance controls the set of valid initialization arguments that supply arguments to *methods*. The *generic functions* for which *method* definitions serve to declare initialization arguments valid are as follows:

    -- Making an *instance* of a *class*: **allocate-instance**, **initialize-instance**, and **shared-initialize**. Initialization arguments declared as valid by these *methods* are valid when making an *instance* of a *class*.
    -- Re-initializing an *instance*: **reinitialize-instance** and **shared-initialize**. Initialization arguments declared as valid by these *methods* are valid when re-initializing an *instance*.
    -- Updating an *instance* to conform to a redefined *class*: **update-instance-for-redefined-class** and **shared-initialize**. Initialization arguments declared as valid by these *methods* are valid when updating an *instance* to conform to a redefined *class*.
    -- Updating an *instance* to conform to the definition of a different *class*: **update-instance-for-different-class** and **shared-initialize**. Initialization arguments declared as valid by these *methods* are valid when updating an *instance* to conform to the definition of a different *class*.

The set of valid initialization arguments for a *class* is the set of valid initialization arguments that either fill *slots* or supply arguments to *methods*, along with the predefined initialization argument `:allow-other-keys`. The default value for `:allow-other-keys` is **nil**. Validity checking of initialization arguments is disabled if the value of the initialization argument `:allow-other-keys` is *true*.

7.1.3 Defaulting of Initialization Arguments

A default value *form* can be supplied for an initialization argument by using the `:default-initargs` *class* option. If an initialization argument is declared valid by some particular *class*, its default value form might be specified by a different *class*. In this case `:default-initargs` is used to supply a default value for an inherited initialization argument.

The `:default-initargs` option is used only to provide default values for initialization arguments; it does not declare a *symbol* as a valid initialization argument name. Furthermore, the `:default-initargs` option is used only to provide default values for initialization arguments when making an *instance*.

The argument to the `:default-initargs` class option is a list of alternating initialization argument names and *forms*. Each *form* is the default value form for the corresponding initialization argument. The default value *form* of an initialization argument is used and evaluated only if that initialization argument does not appear in the arguments to **make-instance** and is not defaulted by a more specific *class*. The default value *form* is evaluated in the lexical environment of the **defclass** form that supplied it; the resulting value is used as the initialization argument's value.

The initialization arguments supplied to **make-instance** are combined with defaulted initialization arguments to produce a *defaulted initialization argument list*. A *defaulted initialization argument list* is a list of alternating initialization argument names and values in which unsupplied initialization arguments are defaulted and in which the explicitly supplied initialization arguments appear earlier in the list than the defaulted initialization arguments. Defaulted initialization arguments are ordered according to the order in the *class precedence list* of the *classes* that supplied the default values.

There is a distinction between the purposes of the `:default-initargs` and the `:initform` options with respect to the initialization of *slots*. The `:default-initargs` class option provides a mechanism for the user to give a default value *form* for an initialization argument without knowing whether the initialization argument initializes a *slot* or is passed to a *method*. If that initialization argument is not explicitly supplied in a call to **make-instance**, the default value *form* is used, just as if it had been supplied in the call. In contrast, the `:initform` slot option provides a mechanism for the user to give a default initial value form for a *slot*. An `:initform` form is used to initialize a *slot* only if no initialization argument associated with that *slot* is given as an argument to **make-instance** or is defaulted by `:default-initargs`.

The order of evaluation of default value *forms* for initialization arguments and the order of evaluation of `:initform` forms are undefined. If the order of evaluation is important, **initialize-instance** or **shared-initialize** *methods* should be used instead.

7.1.4 Rules for Initialization Arguments

The `:initarg` slot option may be specified more than once for a given *slot*.

The following rules specify when initialization arguments may be multiply defined:

* A given initialization argument can be used to initialize more than one *slot* if the same initialization argument name appears in more than one `:initarg` slot option.
* A given initialization argument name can appear in the *lambda list* of more than one initialization *method*.
* A given initialization argument name can appear both in an `:initarg` slot option and in the *lambda list* of an initialization *method*.

If two or more initialization arguments that initialize the same *slot* are given in the arguments to **make-instance**, the leftmost of these initialization arguments in the *initialization argument list* supplies the value, even if the initialization arguments have different names.

If two or more different initialization arguments that initialize the same *slot* have default values and none is given explicitly in the arguments to **make-instance**, the initialization argument that appears in a `:default-initargs` class option in the most specific of the *classes* supplies the value. If a single `:default-initargs` class option specifies two or more initialization arguments that initialize the same *slot* and none is given explicitly in the arguments to **make-instance**, the leftmost in the `:default-initargs` class option supplies the value, and the values of the remaining default value *forms* are ignored.

Initialization arguments given explicitly in the arguments to **make-instance** appear to the left of defaulted initialization arguments. Suppose that the classes C1 and C2 supply the values of defaulted initialization arguments for different *slots*, and suppose that C1 is more specific than C2; then the defaulted initialization argument whose value is supplied by C1 is to the left of the defaulted initialization argument whose value is supplied by C2 in the *defaulted initialization argument list*. If a single `:default-initargs` class option supplies the values of initialization arguments for two different *slots*, the initialization argument whose value is specified farther to the left in the `:default-initargs` class option appears farther to the left in the *defaulted initialization argument list*.

If a *slot* has both an `:initform` form and an `:initarg` slot option, and the initialization argument is defaulted using `:default-initargs` or is supplied to **make-instance**, the captured `:initform` form is neither used nor evaluated.

The following is an example of the above rules:

```
 (defclass q () ((x :initarg a)))
 (defclass r (q) ((x :initarg b))
   (:default-initargs a 1 b 2))

                              Defaulted
Form                          Initialization Argument List  Contents of Slot X
----------

(make-instance 'r)            (a 1 b 2)                     1
(make-instance 'r 'a 3)       (a 3 b 2)                     3
(make-instance 'r 'b 4)       (b 4 a 1)                     4
(make-instance 'r 'a 1 'a 2)  (a 1 a 2 b 2)                 1
```

7.1.5 Shared-Initialize

The *generic function* **shared-initialize** is used to fill the *slots* of an *instance* using initialization arguments and `:initform` forms when an *instance* is created, when an *instance* is re-initialized, when an *instance* is updated to conform to a redefined *class*, and when an *instance* is updated to conform to a different *class*. It uses standard *method* combination. It takes the following arguments: the *instance* to be initialized, a specification of a set of *names* of *slots accessible* in that *instance*, and any number of initialization arguments. The arguments after the first two must form an *initialization argument list*.

The second argument to **shared-initialize** may be one of the following:

* It can be a (possibly empty) *list* of *slot* names, which specifies the set of those *slot* names.
* It can be the symbol **t**, which specifies the set of all of the *slots*.

There is a system-supplied primary *method* for **shared-initialize** whose first *parameter specializer* is the *class* **standard-object**. This *method* behaves as follows on each *slot*, whether shared or local:

* If an initialization argument in the *initialization argument list* specifies a value for that *slot*, that value is stored into the *slot*, even if a value has already been stored in the *slot* before the *method* is run. The affected *slots* are independent of which *slots* are indicated by the second argument to **shared-initialize**.
* Any *slots* indicated by the second argument that are still unbound at this point are initialized according to their `:initform` forms. For any such *slot* that has an `:initform` form, that *form* is evaluated in the lexical environment of its defining **defclass** form and the result is stored into the *slot*. For example, if a *before method* stores a value in the *slot*, the `:initform` form will not be used to supply a value for the *slot*. If the second argument specifies a *name* that does not correspond to any *slots accessible* in the *instance*, the results are unspecified.
* The rules mentioned in Section 7.1.4 (Rules for Initialization Arguments) are obeyed.

The generic function **shared-initialize** is called by the system-supplied primary *methods* for **reinitialize-instance**, **update-instance-for-different-class**, **update-instance-for-redefined-class**, and **initialize-instance**. Thus, *methods* can be written for **shared-initialize** to specify actions that should be taken in all of these contexts.

7.1.6 Initialize-Instance

The *generic function* **initialize-instance** is called by **make-instance** to initialize a newly created *instance*. It uses *standard method combination*. *Methods* for **initialize-instance** can be defined in order to perform any initialization that cannot be achieved simply by supplying initial values for *slots*.

During initialization, **initialize-instance** is invoked after the following actions have been taken:

* The *defaulted initialization argument list* has been computed by combining the supplied *initialization argument list* with any default initialization arguments for the *class*.
* The validity of the *defaulted initialization argument list* has been checked. If any of the initialization arguments has not been declared as valid, an error is signaled.
* A new *instance* whose *slots* are unbound has been created.

The generic function **initialize-instance** is called with the new *instance* and the defaulted initialization arguments. There is a system-supplied primary *method* for **initialize-instance** whose *parameter specializer* is the *class* **standard-object**. This *method* calls the generic function **shared-initialize** to fill in the *slots* according to the initialization arguments and the `:initform` forms for the *slots*; the generic function **shared-initialize** is called with the following arguments: the *instance*, **t**, and the defaulted initialization arguments.

Note that **initialize-instance** provides the *defaulted initialization argument list* in its call to **shared-initialize**, so the first step performed by the system-supplied primary *method* for **shared-initialize** takes into account both the initialization arguments provided in the call to **make-instance** and the *defaulted initialization argument list*.

Methods for **initialize-instance** can be defined to specify actions to be taken when an *instance* is initialized. If only *after methods* for **initialize-instance** are defined, they will be run after the system-supplied primary *method* for initialization and therefore will not interfere with the default behavior of **initialize-instance**.

The object system provides two *functions* that are useful in the bodies of **initialize-instance** methods. The *function* **slot-boundp** returns a *generic boolean* value that indicates whether a specified *slot* has a value; this provides a mechanism for writing *after methods* for **initialize-instance** that initialize *slots* only if they have not already been initialized. The *function* **slot-makunbound** causes the *slot* to have no value.

7.1.7 Definitions of Make-Instance and Initialize-Instance

The generic function **make-instance** behaves as if it were defined as follows, except that certain optimizations are permitted:

```
(defmethod make-instance ((class standard-class) &rest initargs)
  ...
  (let ((instance (apply #'allocate-instance class initargs)))
    (apply #'initialize-instance instance initargs)
    instance))

(defmethod make-instance ((class-name symbol) &rest initargs)
  (apply #'make-instance (find-class class-name) initargs))
```

The elided code in the definition of **make-instance** augments the `initargs` with any *defaulted initialization arguments* and checks the resulting initialization arguments to determine whether an initialization argument was supplied that neither filled a *slot* nor supplied an argument to an applicable *method*.

The generic function **initialize-instance** behaves as if it were defined as follows, except that certain optimizations are permitted:

```
(defmethod initialize-instance ((instance standard-object) &rest initargs)
  (apply #'shared-initialize instance t initargs)))
```

These procedures can be customized.

Customizing at the Programmer Interface level includes using the `:initform`, `:initarg`, and `:default-initargs` options to **defclass**, as well as defining *methods* for **make-instance**, **allocate-instance**, and **initialize-instance**. It is also possible to define *methods* for **shared-initialize**, which would be invoked by the generic functions **reinitialize-instance**, **update-instance-for-redefined-class**,

update-instance-for-different-class, and **initialize-instance**. The meta-object level supports additional customization.

Implementations are permitted to make certain optimizations to **initialize-instance** and **shared-initialize**. The description of **shared-initialize** in Chapter 7 mentions the possible optimizations.

7.2 Changing the Class of an Instance

The *function* **change-class** can be used to change the *class* of an *instance* from its current class, Cfrom, to a different class, Cto; it changes the structure of the *instance* to conform to the definition of the class Cto.

Note that changing the *class* of an *instance* may cause *slots* to be added or deleted. Changing the *class* of an *instance* does not change its identity as defined by the **eq** function.

When **change-class** is invoked on an *instance*, a two-step updating process takes place. The first step modifies the structure of the *instance* by adding new *local slots* and discarding *local slots* that are not specified in the new version of the *instance*. The second step initializes the newly added *local slots* and performs any other user-defined actions. These two steps are further described in the two following sections.

7.2.1 Modifying the Structure of the Instance

In order to make the *instance* conform to the class Cto, *local slots* specified by the class Cto that are not specified by the class Cfrom are added, and *local slots* not specified by the class Cto that are specified by the class Cfrom are discarded.

The values of *local slots* specified by both the class Cto and the class Cfrom are retained. If such a *local slot* was unbound, it remains unbound.

The values of *slots* specified as shared in the class Cfrom and as local in the class Cto are retained.

This first step of the update does not affect the values of any *shared slots*.

7.2.2 Initializing Newly Added Local Slots

The second step of the update initializes the newly added *slots* and performs any other user-defined actions. This step is implemented by the generic function **update-instance-for-different-class**. The generic function **update-instance-for-different-class** is invoked by **change-class** after the first step of the update has been completed.

The generic function **update-instance-for-different-class** is invoked on arguments computed by **change-class**. The first argument passed is a copy of the *instance* being updated and is an *instance* of the class Cfrom; this copy has *dynamic extent* within the generic function **change-class**. The second argument is the *instance* as updated so far by **change-class** and is an *instance* of the class Cto. The remaining arguments are an *initialization argument list*.

There is a system-supplied primary *method* for **update-instance-for-different-class** that has two parameter specializers, each of which is the *class* **standard-object**. First this *method* checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared as valid. (For more information, see Section 7.1.2 (Declaring the Validity of Initialization Arguments).) Then it calls the generic function **shared-initialize** with the following arguments: the new *instance*, a list of *names* of the newly added *slots*, and the initialization arguments it received.

7.2.3 Customizing the Change of Class of an Instance

Methods for **update-instance-for-different-class** may be defined to specify actions to be taken when an *instance* is updated. If only *after methods* for **update-instance-for-different-class** are defined, they will be run after the system-supplied primary *method* for initialization and will not interfere with the default behavior of **update-instance-for-different-class**.

Methods for **shared-initialize** may be defined to customize *class* redefinition. For more information, see Section 7.1.5 (Shared-Initialize).

7.3 Reinitializing an Instance

The generic function **reinitialize-instance** may be used to change the values of *slots* according to initialization arguments.

The process of reinitialization changes the values of some *slots* and performs any user-defined actions. It does not modify the structure of an *instance* to add or delete *slots*, and it does not use any `:initform` forms to initialize *slots*.

The generic function **reinitialize-instance** may be called directly. It takes one required argument, the *instance*. It also takes any number of initialization arguments to be used by *methods* for **reinitialize-instance** or for **shared-initialize**. The arguments after the required *instance* must form an *initialization argument list*.

There is a system-supplied primary *method* for **reinitialize-instance** whose *parameter specializer* is the *class* **standard-object**. First this *method* checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared as valid. (For more information, see Section 7.1.2 (Declaring the Validity of Initialization Arguments).) Then it calls the generic function **shared-initialize** with the following arguments: the *instance*, **nil**, and the initialization arguments it received.

7.3.1 Customizing Reinitialization

Methods for **reinitialize-instance** may be defined to specify actions to be taken when an *instance* is updated. If only *after methods* for **reinitialize-instance** are defined, they will be run after the system-supplied primary *method* for initialization and therefore will not interfere with the default behavior of **reinitialize-instance**.

Methods for **shared-initialize** may be defined to customize *class* redefinition. For more information, see Section 7.1.5 (Shared-Initialize).

7.4 Meta-Objects

The implementation of the object system manipulates *classes*, *methods*, and *generic functions*. The object system contains a set of *generic functions* defined by *methods* on *classes*; the behavior of those *generic functions* defines the behavior of the object system. The *instances* of the *classes* on which those *methods* are defined are called meta-objects.

7.4.1 Standard Meta-objects

The object system supplies a set of meta-objects, called standard meta-objects. These include the *class* **standard-object** and *instances* of the classes **standard-method**, **standard-generic-function**, and **method-combination**.

* The *class* **standard-method** is the default *class* of *methods* defined by the **defmethod** and **defgeneric** *forms*.
* The *class* **standard-generic-function** is the default *class* of *generic functions* defined by the forms **defmethod**, **defgeneric**, and **defclass**.
* The *class* named **standard-object** is an *instance* of the *class* **standard-class** and is a *superclass* of every *class* that is an *instance* of **standard-class** except itself and **structure-class**.
* Every *method* combination object is an *instance* of a *subclass* of *class* **method-combination**.

7.5 Slots

7.5.1 Introduction to Slots

An *object* of *metaclass* **standard-class** has zero or more named *slots*. The *slots* of an *object* are determined by the *class* of the *object*. Each *slot* can hold one value. The *name* of a *slot* is a *symbol* that is syntactically valid for use as a variable name.

When a *slot* does not have a value, the *slot* is said to be *unbound*. When an unbound *slot* is read, the *generic function* **slot-unbound** is invoked. The system-supplied primary *method* for **slot-unbound** on *class* **t** signals an error. If **slot-unbound** returns, its *primary value* is used that time as the *value* of the *slot*.

The default initial value form for a *slot* is defined by the `:initform` slot option. When the `:initform` form is used to supply a value, it is evaluated in the lexical environment in which the **defclass** form was evaluated. The `:initform` along with the lexical environment in which the **defclass** form was evaluated is called a *captured initialization form*. For more details, see Section 7.1 (Object Creation and Initialization).

A *local slot* is defined to be a *slot* that is *accessible* to exactly one *instance*, namely the one in which the *slot* is allocated. A *shared slot* is defined to be a *slot* that is visible to more than one *instance* of a given *class* and its *subclasses*.

A *class* is said to define a *slot* with a given *name* when the **defclass** form for that *class* contains a *slot specifier* with that *name*. Defining a *local slot* does not immediately create a *slot*; it causes a *slot* to be created each time an *instance* of the *class* is created. Defining a *shared slot* immediately creates a *slot*.

The `:allocation` slot option to **defclass** controls the kind of *slot* that is defined. If the value of the `:allocation` slot option is `:instance`, a *local slot* is created. If the value of `:allocation` is `:class`, a *shared slot* is created.

A *slot* is said to be *accessible* in an *instance* of a *class* if the *slot* is defined by the *class* of the *instance* or is inherited from a *superclass* of that *class*. At most one *slot* of a given *name* can be *accessible* in an *instance*. A *shared slot* defined by a *class* is *accessible* in all *instances* of that *class*. A detailed explanation of the inheritance of *slots* is given in Section 7.5.3 (Inheritance of Slots and Slot Options).

7.5.2 Accessing Slots

Slots can be *accessed* in two ways: by use of the primitive function **slot-value** and by use of *generic functions* generated by the **defclass** form.

The *function* **slot-value** can be used with any of the *slot* names specified in the **defclass** form to *access* a specific *slot accessible* in an *instance* of the given *class*.

The macro **defclass** provides syntax for generating *methods* to read and write *slots*. If a reader *method* is requested, a *method* is automatically generated for reading the value of the *slot*, but no *method* for storing a value into it is generated. If a writer *method* is requested, a *method* is automatically generated for storing a value into the *slot*, but no *method* for reading its value is generated. If an accessor *method* is requested, a *method* for reading the value of the *slot* and a *method* for storing a value into the *slot* are automatically generated. Reader and writer *methods* are

implemented using **slot-value**.

When a reader or writer *method* is specified for a *slot*, the name of the *generic function* to which the generated *method* belongs is directly specified. If the *name* specified for the writer *method* is the symbol `name`, the *name* of the *generic function* for writing the *slot* is the symbol `name`, and the *generic function* takes two arguments: the new value and the *instance*, in that order. If the *name* specified for the accessor *method* is the symbol `name`, the *name* of the *generic function* for reading the *slot* is the symbol `name`, and the *name* of the *generic function* for writing the *slot* is the list `(setf name)`.

A *generic function* created or modified by supplying `:reader`, `:writer`, or `:accessor` *slot* options can be treated exactly as an ordinary *generic function*.

Note that **slot-value** can be used to read or write the value of a *slot* whether or not reader or writer *methods* exist for that *slot*. When **slot-value** is used, no reader or writer *methods* are invoked.

The macro **with-slots** can be used to establish a *lexical environment* in which specified *slots* are lexically available as if they were variables. The macro **with-slots** invokes the *function* **slot-value** to *access* the specified *slots*.

The macro **with-accessors** can be used to establish a lexical environment in which specified *slots* are lexically available through their accessors as if they were variables. The macro **with-accessors** invokes the appropriate accessors to *access* the specified *slots*.

7.5.3 Inheritance of Slots and Slot Options

The set of the *names* of all *slots accessible* in an *instance* of a *class* C is the union of the sets of *names* of *slots* defined by C and its *superclasses*. The structure of an *instance* is the set of *names* of *local slots* in that *instance*.

In the simplest case, only one *class* among C and its *superclasses* defines a *slot* with a given *slot* name. If a *slot* is defined by a *superclass* of C, the *slot* is said to be inherited. The characteristics of the *slot* are determined by the *slot specifier* of the defining *class*. Consider the defining *class* for a slot S. If the value of the `:allocation` slot option is `:instance`, then S is a *local slot* and each *instance* of C has its own *slot* named S that stores its own value. If the value of the `:allocation` slot option is `:class`, then S is a *shared slot*, the *class* that defined S stores the value, and all *instances* of C can *access* that single *slot*. If the `:allocation` slot option is omitted, `:instance` is used.

In general, more than one *class* among C and its *superclasses* can define a *slot* with a given *name*. In such cases, only one *slot* with the given name is *accessible* in an *instance* of C, and the characteristics of that *slot* are a combination of the several *slot* specifiers, computed as follows:

* All the *slot specifiers* for a given *slot* name are ordered from most specific to least specific, according to the order in C's *class precedence list* of the *classes* that define them. All references to the specificity of *slot specifiers* immediately below refers to this ordering.
* The allocation of a *slot* is controlled by the most specific *slot specifier*. If the most specific *slot specifier* does not contain an `:allocation` slot option, `:instance` is used. Less specific *slot specifiers* do not affect the allocation.
* The default initial value form for a *slot* is the value of the `:initform` slot option in the most specific *slot specifier* that contains one. If no *slot specifier* contains an `:initform` slot option, the *slot* has no default initial value form.
* The contents of a *slot* will always be of type `(and T1 ... Tn)` where T1 ...Tn are the values of the `:type` slot options contained in all of the *slot specifiers*. If no *slot specifier* contains the `:type` slot option, the contents of the *slot* will always be of *type* **t**. The consequences of attempting to store in a *slot* a value that does not satisfy the *type* of the *slot* are undefined.
* The set of initialization arguments that initialize a given *slot* is the union of the initialization arguments declared in the `:initarg` slot options in all the *slot specifiers*.

* The *documentation string* for a *slot* is the value of the `:documentation` slot option in the most specific *slot* specifier that contains one. If no *slot specifier* contains a `:documentation` slot option, the *slot* has no *documentation string*.

A consequence of the allocation rule is that a *shared slot* can be *shadowed*. For example, if a class C1 defines a *slot* named S whose value for the `:allocation` slot option is `:class`, that *slot* is *accessible* in *instances* of C1 and all of its *subclasses*. However, if C2 is a *subclass* of C1 and also defines a *slot* named S, C1's *slot* is not shared by *instances* of C2 and its *subclasses*. When a class C1 defines a *shared slot*, any subclass C2 of C1 will share this single *slot* unless the **defclass** form for C2 specifies a *slot* of the same *name* or there is a *superclass* of C2 that precedes C1 in the *class precedence list* of C2 that defines a *slot* of the same name.

A consequence of the type rule is that the value of a *slot* satisfies the type constraint of each *slot specifier* that contributes to that *slot*. Because the result of attempting to store in a *slot* a value that does not satisfy the type constraint for the *slot* is undefined, the value in a *slot* might fail to satisfy its type constraint.

The `:reader`, `:writer`, and `:accessor` slot options create *methods* rather than define the characteristics of a *slot*. Reader and writer *methods* are inherited in the sense described in Section 7.6.7 (Inheritance of Methods).

Methods that *access slots* use only the name of the *slot* and the *type* of the *slot*'s value. Suppose a *superclass* provides a *method* that expects to *access* a *shared slot* of a given *name*, and a *subclass* defines a *local slot* with the same *name*. If the *method* provided by the *superclass* is used on an *instance* of the *subclass*, the *method accesses* the *local slot*.

7.6 Generic Functions and Methods

7.6.1 Introduction to Generic Functions

A *generic function* is a function whose behavior depends on the *classes* or identities of the *arguments* supplied to it. A *generic function object* is associated with a set of *methods*, a *lambda list*, a *method combination*[2], and other information.

Like an *ordinary function*, a *generic function* takes *arguments*, performs a series of operations, and perhaps returns useful *values*. An *ordinary function* has a single body of *code* that is always *executed* when the *function* is called. A *generic function* has a set of bodies of *code* of which a subset is selected for *execution*. The selected bodies of *code* and the manner of their combination are determined by the *classes* or identities of one or more of the *arguments* to the *generic function* and by its *method combination*.

Ordinary functions and *generic functions* are called with identical syntax.

Generic functions are true *functions* that can be passed as *arguments* and used as the first *argument* to **funcall** and **apply**.

A *binding* of a *function name* to a *generic function* can be *established* in one of several ways. It can be *established* in the *global environment* by **ensure-generic-function**, **defmethod** (implicitly, due to **ensure-generic-function**) or **defgeneric** (also implicitly, due to **ensure-generic-function**). No *standardized* mechanism is provided for *establishing* a *binding* of a *function name* to a *generic function* in the *lexical environment*.

When a **defgeneric** form is evaluated, one of three actions is taken (due to **ensure-generic-function**):

* If a generic function of the given name already exists, the existing generic function object is modified. Methods specified by the current **defgeneric** form are added, and any methods in the existing generic function that were defined by a previous **defgeneric** form are removed. Methods added by the current **defgeneric** form might replace methods defined by **defmethod**, **defclass**, **define-condition**, or **defstruct**. No other methods in the generic function are affected or replaced.

* If the given name names an *ordinary function*, a *macro*, or a *special operator*, an error is signaled.
* Otherwise a generic function is created with the methods specified by the method definitions in the **defgeneric** form.

Some *operators* permit specification of the options of a *generic function*, such as the *type* of *method combination* it uses or its *argument precedence order*. These *operators* will be referred to as "operators that specify generic function options." The only *standardized operator* in this category is **defgeneric**.

Some *operators* define *methods* for a *generic function*. These *operators* will be referred to as *method-defining operators*; their associated *forms* are called *method-defining forms*. The *standardized method-defining operators* are listed in the next figure.

```
defgeneric        defmethod  defclass
define-condition  defstruct
```

Figure 7-1. Standardized Method-Defining Operators Note that of the *standardized method-defining operators* only **defgeneric** can specify *generic function* options. **defgeneric** and any *implementation-defined operators* that can specify *generic function* options are also referred to as "operators that specify generic function options."

7.6.2 Introduction to Methods

Methods define the class-specific or identity-specific behavior and operations of a *generic function*.

A *method object* is associated with *code* that implements the method's behavior, a sequence of *parameter specializers* that specify when the given *method* is applicable, a *lambda list*, and a sequence of *qualifiers* that are used by the method combination facility to distinguish among *methods*.

A method object is not a function and cannot be invoked as a function. Various mechanisms in the object system take a method object and invoke its method function, as is the case when a generic function is invoked. When this occurs it is said that the method is invoked or called.

A method-defining form contains the *code* that is to be run when the arguments to the generic function cause the method that it defines to be invoked. When a method-defining form is evaluated, a method object is created and one of four actions is taken:

* If a *generic function* of the given name already exists and if a *method object* already exists that agrees with the new one on *parameter specializers* and *qualifiers*, the new *method object* replaces the old one. For a definition of one method agreeing with another on *parameter specializers* and *qualifiers*, see Section 7.6.3 (Agreement on Parameter Specializers and Qualifiers).
* If a *generic function* of the given name already exists and if there is no *method object* that agrees with the new one on *parameter specializers* and *qualifiers*, the existing *generic function object* is modified to contain the new *method object*.
* If the given *name* names an *ordinary function*, a *macro*, or a *special operator*, an error is signaled.
* Otherwise a *generic function* is created with the *method* specified by the *method-defining form*.

If the *lambda list* of a new *method* is not *congruent* with the *lambda list* of the *generic function*, an error is signaled. If a *method-defining operator* that cannot specify *generic function* options creates a new *generic function*, a *lambda list* for that *generic function* is derived from the *lambda list* of the *method* in the *method-defining form* in such a way as to be *congruent* with it. For a discussion of *congruence*, see Section 7.6.4 (Congruent Lambda-lists for all Methods of a Generic Function).

Each method has a *specialized lambda list*, which determines when that method can be applied. A *specialized lambda list* is like an *ordinary lambda list* except that a specialized parameter may occur instead of the name of a required parameter. A specialized parameter is a list (*variable-name parameter-specializer-name*), where *parameter-specializer-name* is one of the following:

a *symbol*
: denotes a *parameter specializer* which is the *class* named by that *symbol*.

a *class*
: denotes a *parameter specializer* which is the *class* itself.

`(eql` *form*`)`
: denotes a *parameter specializer* which satisfies the *type specifier* `(eql` *object*`)`, where *object* is the result of evaluating *form*. The form *form* is evaluated in the lexical environment in which the method-defining form is evaluated. Note that *form* is evaluated only once, at the time the method is defined, not each time the generic function is called.

Parameter specializer names are used in macros intended as the user-level interface (**defmethod**), while *parameter specializers* are used in the functional interface.

Only required parameters may be specialized, and there must be a *parameter specializer* for each required parameter. For notational simplicity, if some required parameter in a *specialized lambda list* in a method-defining form is simply a variable name, its *parameter specializer* defaults to the *class* **t**.

Given a generic function and a set of arguments, an applicable method is a method for that generic function whose parameter specializers are satisfied by their corresponding arguments. The following definition specifies what it means for a method to be applicable and for an argument to satisfy a *parameter specializer*.

Let <A1, ..., An> be the required arguments to a generic function in order. Let <P1, ..., Pn> be the *parameter specializers* corresponding to the required parameters of the method M in order. The method M is applicable when each Ai is of the *type* specified by the *type specifier* Pi. Because every valid *parameter specializer* is also a valid *type specifier*, the *function* **typep** can be used during method selection to determine whether an argument satisfies a *parameter specializer*.

A method all of whose *parameter specializers* are the *class* **t** is called a *default method*; it is always applicable but may be shadowed by a more specific method.

Methods can have *qualifiers*, which give the method combination procedure a way to distinguish among methods. A method that has one or more *qualifiers* is called a *qualified method*. A method with no *qualifiers* is called an *unqualified method*. A *qualifier* is any *non-list*. The *qualifiers* defined by the *standardized* method combination types are *symbols*.

In this specification, the terms "*primary method*" and "*auxiliary method*" are used to partition *methods* within a method combination type according to their intended use. In standard method combination, *primary methods* are *unqualified methods* and *auxiliary methods* are methods with a single *qualifier* that is one of `:around`, `:before`, or `:after`. *Methods* with these *qualifiers* are called *around methods*, *before methods*, and *after methods*, respectively. When a method combination type is defined using the short form of **define-method-combination**, *primary methods* are methods qualified with the name of the type of method combination, and auxiliary methods have the *qualifier* `:around`. Thus the terms "*primary method*" and "*auxiliary method*" have only a relative definition within a given method combination type.

7.6.3 Agreement on Parameter Specializers and Qualifiers

Two *methods* are said to agree with each other on *parameter specializers* and *qualifiers* if the following conditions hold:

1. Both methods have the same number of required parameters. Suppose the *parameter specializers* of the two methods are P1,1...P1,n and P2,1...P2,n.
2. For each 1<=i<=n, P1,i agrees with P2,i. The *parameter specializer* P1,i agrees with P2,i if P1,i and P2,i are the same class or if P1,i=`(`**eql** *object*1`)`, P2,i=`(`**eql** *object*2`)`, and `(`**eql** *object*1 *object*2`)`. Otherwise P1,i and P2,i do not agree.

3. The two *lists* of *qualifiers* are the *same* under **equal**.

7.6.4 Congruent Lambda-lists for all Methods of a Generic Function

These rules define the congruence of a set of *lambda lists*, including the *lambda list* of each method for a given generic function and the *lambda list* specified for the generic function itself, if given.

1. Each *lambda list* must have the same number of required parameters.
2. Each *lambda list* must have the same number of optional parameters. Each method can supply its own default for an optional parameter.
3. If any *lambda list* mentions `&rest` or `&key`, each *lambda list* must mention one or both of them.
4. If the *generic function lambda list* mentions `&key`, each method must accept all of the keyword names mentioned after `&key`, either by accepting them explicitly, by specifying `&allow-other-keys`, or by specifying `&rest` but not `&key`. Each method can accept additional keyword arguments of its own. The checking of the validity of keyword names is done in the generic function, not in each method. A method is invoked as if the keyword argument pair whose name is `:allow-other-keys` and whose value is *true* were supplied, though no such argument pair will be passed.
5. The use of `&allow-other-keys` need not be consistent across *lambda lists*. If `&allow-other-keys` is mentioned in the *lambda list* of any applicable *method* or of the *generic function*, any keyword arguments may be mentioned in the call to the *generic function*.
6. The use of `&aux` need not be consistent across methods.

If a *method-defining operator* that cannot specify *generic function* options creates a *generic function*, and if the *lambda list* for the method mentions keyword arguments, the *lambda list* of the generic function will mention `&key` (but no keyword arguments).

7.6.5 Keyword Arguments in Generic Functions and Methods

When a generic function or any of its methods mentions `&key` in a *lambda list*, the specific set of keyword arguments accepted by the generic function varies according to the applicable methods. The set of keyword arguments accepted by the generic function for a particular call is the union of the keyword arguments accepted by all applicable methods and the keyword arguments mentioned after `&key` in the generic function definition, if any. A method that has `&rest` but not `&key` does not affect the set of acceptable keyword arguments. If the *lambda list* of any applicable method or of the generic function definition contains `&allow-other-keys`, all keyword arguments are accepted by the generic function.

The *lambda list* congruence rules require that each method accept all of the keyword arguments mentioned after `&key` in the generic function definition, by accepting them explicitly, by specifying `&allow-other-keys`, or by specifying `&rest` but not `&key`. Each method can accept additional keyword arguments of its own, in addition to the keyword arguments mentioned in the generic function definition.

If a *generic function* is passed a keyword argument that no applicable method accepts, an error should be signaled; see Section 3.5 (Error Checking in Function Calls).

7.6.5.1 Examples of Keyword Arguments in Generic Functions and Methods

For example, suppose there are two methods defined for `width` as follows:

```
(defmethod width ((c character-class) &key font) ...)

(defmethod width ((p picture-class) &key pixel-size) ...)
```

Assume that there are no other methods and no generic function definition for `width`. The evaluation of the following form should signal an error because the keyword argument `:pixel-size` is not accepted by the applicable method.

```
(width (make-instance 'character-class :char #\Q)
       :font 'baskerville :pixel-size 10)
```

The evaluation of the following form should signal an error.

```
(width (make-instance 'picture-class :glyph (glyph #\Q))
       :font 'baskerville :pixel-size 10)
```

The evaluation of the following form will not signal an error if the class named `character-picture-class` is a subclass of both `picture-class` and `character-class`.

```
(width (make-instance 'character-picture-class :char #\Q)
       :font 'baskerville :pixel-size 10)
```

7.6.6 Method Selection and Combination

When a *generic function* is called with particular arguments, it must determine the code to execute. This code is called the *effective method* for those *arguments*. The *effective method* is a combination of the *applicable methods* in the *generic function* that *calls* some or all of the *methods*.

If a *generic function* is called and no *methods* are *applicable*, the *generic function* **no-applicable-method** is invoked, with the *results* from that call being used as the *results* of the call to the original *generic function*. Calling **no-applicable-method** takes precedence over checking for acceptable keyword arguments; see Section 7.6.5 (Keyword Arguments in Generic Functions and Methods).

When the *effective method* has been determined, it is invoked with the same *arguments* as were passed to the *generic function*. Whatever *values* it returns are returned as the *values* of the *generic function*.

7.6.6.1 Determining the Effective Method

The effective method is determined by the following three-step procedure:

1. Select the applicable methods.
2. Sort the applicable methods by precedence order, putting the most specific method first.
3. Apply method combination to the sorted list of applicable methods, producing the effective method.

7.6.6.1.1 Selecting the Applicable Methods

This step is described in Section 7.6.2 (Introduction to Methods).

7.6.6.1.2 Sorting the Applicable Methods by Precedence Order

To compare the precedence of two methods, their *parameter specializers* are examined in order. The default examination order is from left to right, but an alternative order may be specified by the `:argument-precedence-order` option to **defgeneric** or to any of the other operators that specify generic function options.

The corresponding *parameter specializers* from each method are compared. When a pair of *parameter specializers* agree, the next pair are compared for agreement. If all corresponding parameter specializers agree, the two methods must have different *qualifiers*; in this case, either method can be selected to precede the other. For information about agreement, see Section 7.6.3 (Agreement on Parameter Specializers and Qualifiers).

If some corresponding *parameter specializers* do not agree, the first pair of *parameter specializers* that do not agree determines the precedence. If both *parameter specializers* are classes, the more specific of the two methods is the method whose *parameter specializer* appears earlier in the *class precedence list* of the corresponding argument. Because of the way in which the set of applicable methods is chosen, the *parameter specializers* are guaranteed to be present in the class precedence list of the class of the argument.

If just one of a pair of corresponding *parameter specializers* is `(eql` *object*`)`, the *method* with that *parameter specializer* precedes the other *method*. If both *parameter specializers* are **eql** *expressions*, the specializers must agree (otherwise the two *methods* would not both have been applicable to this argument).

The resulting list of *applicable methods* has the most specific *method* first and the least specific *method* last.

7.6.6.1.3 Applying method combination to the sorted list of applicable methods

In the simple case---if standard method combination is used and all applicable methods are primary methods---the effective method is the most specific method. That method can call the next most specific method by using the *function* **call-next-method**. The method that **call-next-method** will call is referred to as the *next method*. The predicate **next-method-p** tests whether a next method exists. If **call-next-method** is called and there is no next most specific method, the generic function **no-next-method** is invoked.

In general, the effective method is some combination of the applicable methods. It is described by a *form* that contains calls to some or all of the applicable methods, returns the value or values that will be returned as the value or values of the generic function, and optionally makes some of the methods accessible by means of **call-next-method**.

The role of each method in the effective method is determined by its *qualifiers* and the specificity of the method. A *qualifier* serves to mark a method, and the meaning of a *qualifier* is determined by the way that these marks are used by this step of the procedure. If an applicable method has an unrecognized *qualifier*, this step signals an error and does not include that method in the effective method.

When standard method combination is used together with qualified methods, the effective method is produced as described in Section 7.6.6.2 (Standard Method Combination).

Another type of method combination can be specified by using the `:method-combination` option of **defgeneric** or of any of the other operators that specify generic function options. In this way this step of the procedure can be customized.

New types of method combination can be defined by using the **define-method-combination** *macro*.

7.6.6.2 Standard Method Combination

Standard method combination is supported by the *class* **standard-generic-function**. It is used if no other type of method combination is specified or if the built-in method combination type **standard** is specified.

Primary methods define the main action of the effective method, while auxiliary methods modify that action in one of three ways. A primary method has no method *qualifiers*.

An auxiliary method is a method whose *qualifier* is `:before`, `:after`, or `:around`. Standard method combination allows no more than one *qualifier* per method; if a method definition specifies more than one *qualifier* per method, an error is signaled.

* A *before method* has the keyword `:before` as its only *qualifier*. A *before method* specifies *code* that is to be run before any *primary methods*.
* An *after method* has the keyword `:after` as its only *qualifier*. An *after method* specifies *code* that is to be run after *primary methods*.
* An *around method* has the keyword `:around` as its only *qualifier*. An *around method* specifies *code* that is to be run instead of other *applicable methods*, but which might contain explicit *code* which calls some of those *shadowed methods* (via **call-next-method**).

The semantics of standard method combination is as follows:

* If there are any *around methods*, the most specific *around method* is called. It supplies the value or values of the generic function.
* Inside the body of an *around method*, **call-next-method** can be used to call the *next method*. When the next method returns, the *around method* can execute more code, perhaps based on the returned value or values. The *generic function* **no-next-method** is invoked if **call-next-method** is used and there is no *applicable method* to call. The *function* **next-method-p** may be used to determine whether a *next method* exists.
* If an *around method* invokes **call-next-method**, the next most specific *around method* is called, if one is applicable. If there are no *around methods* or if **call-next-method** is called by the least specific *around method*, the other methods are called as follows:
    -- All the *before methods* are called, in most-specific-first order. Their values are ignored. An error is signaled if **call-next-method** is used in a *before method*.
    -- The most specific primary method is called. Inside the body of a primary method, **call-next-method** may be used to call the next most specific primary method. When that method returns, the previous primary method can execute more code, perhaps based on the returned value or values. The generic function **no-next-method** is invoked if **call-next-method** is used and there are no more applicable primary methods. The *function* **next-method-p** may be used to determine whether a *next method* exists. If **call-next-method** is not used, only the most specific *primary method* is called.
    -- All the *after methods* are called in most-specific-last order. Their values are ignored. An error is signaled if **call-next-method** is used in an *after method*.
* If no *around methods* were invoked, the most specific primary method supplies the value or values returned by the generic function. The value or values returned by the invocation of **call-next-method** in the least specific *around method* are those returned by the most specific primary method.

In standard method combination, if there is an applicable method but no applicable primary method, an error is signaled.

The *before methods* are run in most-specific-first order while the *after methods* are run in least-specific-first order. The design rationale for this difference can be illustrated with an example. Suppose class C1 modifies the behavior of its superclass, C2, by adding *before methods* and *after methods*. Whether the behavior of the class C2 is defined directly by methods on C2 or is inherited from its superclasses does not affect the relative order of invocation of methods on instances of the class C1. Class C1's *before method* runs before all of class C2's methods. Class C1's *after method* runs after all of class C2's methods.

By contrast, all *around methods* run before any other methods run. Thus a less specific *around method* runs before a more specific primary method.

If only primary methods are used and if **call-next-method** is not used, only the most specific method is invoked; that is, more specific methods shadow more general ones.

7.6.6.3 Declarative Method Combination

The macro **define-method-combination** defines new forms of method combination. It provides a mechanism for customizing the production of the effective method. The default procedure for producing an effective method is described in Section 7.6.6.1 (Determining the Effective Method). There are two forms of

define-method-combination. The short form is a simple facility while the long form is more powerful and more verbose. The long form resembles **defmacro** in that the body is an expression that computes a Lisp form; it provides mechanisms for implementing arbitrary control structures within method combination and for arbitrary processing of method *qualifiers*.

7.6.6.4 Built-in Method Combination Types

The object system provides a set of built-in method combination types. To specify that a generic function is to use one of these method combination types, the name of the method combination type is given as the argument to the `:method-combination` option to **defgeneric** or to the `:method-combination` option to any of the other operators that specify generic function options.

The names of the built-in method combination types are listed in the next figure.

```
+    append  max  nconc  progn
and  list    min  or     standard
```

Figure 7-2. Built-in Method Combination Types

The semantics of the **standard** built-in method combination type is described in Section 7.6.6.2 (Standard Method Combination). The other built-in method combination types are called simple built-in method combination types.

The simple built-in method combination types act as though they were defined by the short form of **define-method-combination**. They recognize two roles for *methods*:

* An *around method* has the keyword symbol `:around` as its sole *qualifier*. The meaning of `:around` *methods* is the same as in standard method combination. Use of the functions **call-next-method** and **next-method-p** is supported in *around methods*.
* A primary method has the name of the method combination type as its sole *qualifier*. For example, the built-in method combination type `and` recognizes methods whose sole *qualifier* is `and`; these are primary methods. Use of the functions **call-next-method** and **next-method-p** is not supported in *primary methods*.

The semantics of the simple built-in method combination types is as follows:

* If there are any *around methods*, the most specific *around method* is called. It supplies the value or values of the *generic function*.
* Inside the body of an *around method*, the function **call-next-method** can be used to call the *next method*. The *generic function* **no-next-method** is invoked if **call-next-method** is used and there is no applicable method to call. The *function* **next-method-p** may be used to determine whether a *next method* exists. When the *next method* returns, the *around method* can execute more code, perhaps based on the returned value or values.
* If an *around method* invokes **call-next-method**, the next most specific *around method* is called, if one is applicable. If there are no *around methods* or if **call-next-method** is called by the least specific *around method*, a Lisp form derived from the name of the built-in method combination type and from the list of applicable primary methods is evaluated to produce the value of the generic function. Suppose the name of the method combination type is *operator* and the call to the generic function is of the form

    (*generic-function* a1...an)

Let M1,...,Mk be the applicable primary methods in order; then the derived Lisp form is

    (*operator* <M1 a1...an>...<Mk a1...an>)

If the expression <Mi a1...an> is evaluated, the method Mi will be applied to the arguments a1...an. For example, if *operator* is `or`, the expression <Mi a1...an> is evaluated only if <Mj a1...an>, 1<=j<i, returned `nil`.

The default order for the primary methods is `:most-specific-first`. However, the order can be reversed by supplying `:most-specific-last` as the second argument to the `:method-combination` option.

The simple built-in method combination types require exactly one *qualifier* per method. An error is signaled if there are applicable methods with no *qualifiers* or with *qualifiers* that are not supported by the method combination type. An error is signaled if there are applicable *around methods* and no applicable primary methods.

7.6.7 Inheritance of Methods

A subclass inherits methods in the sense that any method applicable to all instances of a class is also applicable to all instances of any subclass of that class.

The inheritance of methods acts the same way regardless of which of the *method-defining operators* created the methods.

The inheritance of methods is described in detail in Section 7.6.6 (Method Selection and Combination).

8. Structures

9. Conditions

9.1 Condition System Concepts

Common Lisp constructs are described not only in terms of their behavior in situations during which they are intended to be used (see the "Description" part of each *operator* specification), but in all other situations (see the "Exceptional Situations" part of each *operator* specification).

A situation is the evaluation of an expression in a specific context. A *condition* is an *object* that represents a specific situation that has been detected. *Conditions* are *generalized instances* of the *class* **condition**. A hierarchy of *condition* classes is defined in Common Lisp. A *condition* has *slots* that contain data relevant to the situation that the *condition* represents.

An error is a situation in which normal program execution cannot continue correctly without some form of intervention (either interactively by the user or under program control). Not all errors are detected. When an error goes undetected, the effects can be *implementation-dependent*, *implementation-defined*, unspecified, or undefined. See Section 1.4 (Definitions). All detected errors can be represented by *conditions*, but not all *conditions* represent errors.

Signaling is the process by which a *condition* can alter the flow of control in a program by raising the *condition* which can then be *handled*. The functions **error**, **cerror**, **signal**, and **warn** are used to signal *conditions*.

The process of signaling involves the selection and invocation of a *handler* from a set of *active handlers*. A *handler* is a *function* of one argument (the *condition*) that is invoked to handle a *condition*. Each *handler* is associated with a *condition type*, and a *handler* will be invoked only on a *condition* of the *handler*'s associated *type*.

Active handlers are *established* dynamically (see **handler-bind** or **handler-case**). *Handlers* are invoked in a *dynamic environment* equivalent to that of the signaler, except that the set of *active handlers* is bound in such a way as to include only those that were *active* at the time the *handler* being invoked was *established*. Signaling a *condition* has no side-effect on the *condition*, and there is no dynamic state contained in a *condition*.

If a *handler* is invoked, it can address the *situation* in one of three ways:

Decline

It can decline to *handle* the *condition*. It does this by simply returning rather than transferring control. When this happens, any values returned by the handler are ignored and the next most recently established handler is invoked. If there is no such handler and the signaling function is **error** or **cerror**, the debugger is entered in the *dynamic environment* of the signaler. If there is no such handler and the signaling function is either **signal** or **warn**, the signaling function simply returns **nil**.

Handle

It can *handle* the *condition* by performing a non-local transfer of control. This can be done either primitively by using **go**, **return**, **throw** or more abstractly by using a function such as **abort** or **invoke-restart**.

Defer

It can put off a decision about whether to *handle* or *decline*, by any of a number of actions, but most commonly by signaling another condition, resignaling the same condition, or forcing entry into the debugger.

9.1.1 Condition Types

The next figure lists the *standardized condition types*. Additional *condition types* can be defined by using **define-condition**.

```
arithmetic-error                floating-point-overflow   simple-type-error
cell-error                      floating-point-underflow  simple-warning
condition                       package-error             storage-condition
control-error                   parse-error               stream-error
division-by-zero                print-not-readable        style-warning
end-of-file                     program-error             type-error
error                           reader-error              unbound-slot
file-error                      serious-condition         unbound-variable
floating-point-inexact          simple-condition          undefined-function
floating-point-invalid-operation simple-error             warning
```

Figure 9-1. Standardized Condition Types

All *condition* types are *subtypes* of *type* **condition**. That is,

```
(typep c 'condition) =>  true
```

if and only if *c* is a *condition*.

Implementations must define all specified *subtype* relationships. Except where noted, all *subtype* relationships indicated in this document are not mutually exclusive. A *condition* inherits the structure of its *supertypes*.

The metaclass of the *class* **condition** is not specified. *Names* of *condition types* may be used to specify *supertype* relationships in **define-condition**, but the consequences are not specified if an attempt is made to use a *condition type* as a *superclass* in a **defclass** *form*.

The next figure shows *operators* that define *condition types* and creating *conditions*.

```
define-condition  make-condition
```

Figure 9-2. Operators that define and create conditions.

The next figure shows *operators* that *read* the *value* of *condition slots*.

```
arithmetic-error-operands   simple-condition-format-arguments
arithmetic-error-operation  simple-condition-format-control
cell-error-name             stream-error-stream
file-error-pathname         type-error-datum
package-error-package       type-error-expected-type
print-not-readable-object   unbound-slot-instance
```

Figure 9-3. Operators that read condition slots.

9.1.1.1 Serious Conditions

A *serious condition* is a *condition* serious enough to require interactive intervention if not handled. *Serious conditions* are typically signaled with **error** or **cerror**; non-serious *conditions* are typically signaled with **signal** or **warn**.

9.1.2 Creating Conditions

The function **make-condition** can be used to construct a *condition object* explicitly. Functions such as **error**, **cerror**, **signal**, and **warn** operate on *conditions* and might create *condition objects* implicitly. Macros such as **ccase**, **ctypecase**, **ecase**, **etypecase**, **check-type**, and **assert** might also implicitly create (and *signal*) *conditions*.

9.1.2.1 Condition Designators

A number of the functions in the condition system take arguments which are identified as *condition designators*. By convention, those arguments are notated as

datum `&rest` *arguments*

Taken together, the *datum* and the *arguments* are "*designators* for a *condition* of default type *default-type*." How the denoted *condition* is computed depends on the type of the *datum*:

* If the *datum* is a *symbol* naming a *condition type* ...
  The denoted *condition* is the result of

```
(apply #'make-condition datum arguments)
```

* If the *datum* is a *format control* ...
  The denoted *condition* is the result of

```
(make-condition defaulted-type
                :format-control datum
                :format-arguments arguments)
```

  where the *defaulted-type* is a *subtype* of *default-type*.

* If the *datum* is a *condition* ...
  The denoted *condition* is the *datum* itself. In this case, unless otherwise specified by the description of the *operator* in question, the *arguments* must be *null*; that is, the consequences are undefined if any *arguments* were supplied.

Note that the *default-type* gets used only in the case where the *datum string* is supplied. In the other situations, the resulting condition is not necessarily of *type default-type*.

Here are some illustrations of how different *condition designators* can denote equivalent *condition objects*:

```
(let ((c (make-condition 'arithmetic-error :operator '/ :operands '(7 0))))
  (error c))
==  (error 'arithmetic-error :operator '/ :operands '(7 0))

(error "Bad luck.")
==  (error 'simple-error :format-control "Bad luck." :format-arguments '())
```

9.1.3 Printing Conditions

If the `:report` argument to **define-condition** is used, a print function is defined that is called whenever the defined *condition* is printed while the *value* of ***print-escape*** is *false*. This function is called the *condition reporter*; the text which it outputs is called a *report message*.

When a *condition* is printed and ***print-escape*** is *false*, the *condition reporter* for the *condition* is invoked. *Conditions* are printed automatically by functions such as **invoke-debugger**, **break**, and **warn**.

When ***print-escape*** is *true*, the *object* should print in an abbreviated fashion according to the style of the implementation (e.g., by **print-unreadable-object**). It is not required that a *condition* can be recreated by reading its printed representation.

No *function* is provided for directly *accessing* or invoking *condition reporters*.

9.1.3.1 Recommended Style in Condition Reporting

In order to ensure a properly aesthetic result when presenting *report messages* to the user, certain stylistic conventions are recommended.

There are stylistic recommendations for the content of the messages output by *condition reporters*, but there are no formal requirements on those *programs*. If a *program* violates the recommendations for some message, the display of that message might be less aesthetic than if the guideline had been observed, but the *program* is still considered a *conforming program*.

The requirements on a *program* or *implementation* which invokes a *condition reporter* are somewhat stronger. A *conforming program* must be permitted to assume that if these style guidelines are followed, proper aesthetics will be maintained. Where appropriate, any specific requirements on such routines are explicitly mentioned below.

9.1.3.1.1 Capitalization and Punctuation in Condition Reports

It is recommended that a *report message* be a complete sentences, in the proper case and correctly punctuated. In English, for example, this means the first letter should be uppercase, and there should be a trailing period.

```
(error "This is a message")  ; Not recommended
(error "this is a message.") ; Not recommended

(error "This is a message.") ; Recommended instead
```

9.1.3.1.2 Leading and Trailing Newlines in Condition Reports

It is recommended that a *report message* not begin with any introductory text, such as "`Error: `" or "`Warning: `" or even just *freshline* or *newline*. Such text is added, if appropriate to the context, by the routine invoking the *condition reporter*.

It is recommended that a *report message* not be followed by a trailing *freshline* or *newline*. Such text is added, if appropriate to the context, by the routine invoking the *condition reporter*.

```
(error "This is a message.~%")   ; Not recommended
(error "~&This is a message.")   ; Not recommended
(error "~&This is a message.~%") ; Not recommended

(error "This is a message.")     ; Recommended instead
```

9.1.3.1.3 Embedded Newlines in Condition Reports

Especially if it is long, it is permissible and appropriate for a *report message* to contain one or more embedded *newlines*.

If the calling routine conventionally inserts some additional prefix (such as "`Error: `" or "`;; Error: `") on the first line of the message, it must also assure that an appropriate prefix will be added to each subsequent line of the output, so that the left edge of the message output by the *condition reporter* will still be properly aligned.

```
(defun test ()
  (error "This is an error message.~%It has two lines."))

;; Implementation A
(test)
This is an error message.
It has two lines.

;; Implementation B
(test)
;; Error: This is an error message.
;;        It has two lines.

;; Implementation C
(test)
>> Error: This is an error message.
          It has two lines.
```

9.1.3.1.4 Note about Tabs in Condition Reports

Because the indentation of a *report message* might be shifted to the right or left by an arbitrary amount, special care should be taken with the semi-standard *character* <Tab> (in those *implementations* that support such a *character*). Unless the *implementation* specifically defines its behavior in this context, its use should be avoided.

9.1.3.1.5 Mentioning Containing Function in Condition Reports

The name of the containing function should generally not be mentioned in *report messages*. It is assumed that the *debugger* will make this information accessible in situations where it is necessary and appropriate.

9.1.4 Signaling and Handling Conditions

The operation of the condition system depends on the ordering of active *applicable handlers* from most recent to least recent.

Each *handler* is associated with a *type specifier* that must designate a *subtype* of *type* **condition**. A *handler* is said to be *applicable* to a *condition* if that *condition* is of the *type* designated by the associated *type specifier*.

Active handlers are *established* by using **handler-bind** (or an abstraction based on **handler-bind**, such as **handler-case** or **ignore-errors**).

Active handlers can be *established* within the dynamic scope of other *active handlers*. At any point during program execution, there is a set of *active handlers*. When a *condition* is signaled, the *most recent* active *applicable handler* for that *condition* is selected from this set. Given a *condition*, the order of recentness of active *applicable handlers* is defined by the following two rules:

1. Each handler in a set of active handlers H1 is more recent than every handler in a set H2 if the handlers in H2 were active when the handlers in H1 were established.
2. Let h1 and h2 be two applicable active handlers established by the same *form*. Then h1 is more recent than h2 if h1 was defined to the left of h2 in the *form* that established them.

Once a handler in a handler binding *form* (such as **handler-bind** or **handler-case**) has been selected, all handlers in that *form* become inactive for the remainder of the signaling process. While the selected *handler* runs, no other *handler* established by that *form* is active. That is, if the *handler* declines, no other handler established by that *form* will be considered for possible invocation.

The next figure shows *operators* relating to the *handling* of *conditions*.

```
handler-bind  handler-case  ignore-errors
```

Figure 9-4. Operators relating to handling conditions.

9.1.4.1 Signaling

When a *condition* is signaled, the most recent applicable *active handler* is invoked. Sometimes a handler will decline by simply returning without a transfer of control. In such cases, the next most recent applicable active handler is invoked.

If there are no applicable handlers for a *condition* that has been signaled, or if all applicable handlers decline, the *condition* is unhandled.

The functions **cerror** and **error** invoke the interactive *condition* handler (the debugger) rather than return if the *condition* being signaled, regardless of its *type*, is unhandled. In contrast, **signal** returns **nil** if the *condition* being signaled, regardless of its *type*, is unhandled.

The *variable* ***break-on-signals*** can be used to cause the debugger to be entered before the signaling process begins.

The next figure shows *defined names* relating to the *signaling* of *conditions*.

```
*break-on-signals*  error   warn
cerror              signal
```

Figure 9-5. Defined names relating to signaling conditions.

9.1.4.1.1 Resignaling a Condition

During the *dynamic extent* of the *signaling* process for a particular *condition object*, **signaling** the same *condition object* again is permitted if and only if the *situation* represented in both cases are the same.

For example, a *handler* might legitimately *signal* the *condition object* that is its *argument* in order to allow outer *handlers* first opportunity to *handle* the condition. (Such a *handlers* is sometimes called a "default handler.") This action is permitted because the *situation* which the second *signaling* process is addressing is really the same *situation*.

On the other hand, in an *implementation* that implemented asynchronous keyboard events by interrupting the user process with a call to **signal**, it would not be permissible for two distinct asynchronous keyboard events to *signal identical condition objects* at the same time for different situations.

9.1.4.2 Restarts

The interactive condition handler returns only through non-local transfer of control to specially defined *restarts* that can be set up either by the system or by user code. Transferring control to a restart is called "invoking" the restart. Like handlers, active *restarts* are *established* dynamically, and only active *restarts* can be invoked. An active *restart* can be invoked by the user from the debugger or by a program by using **invoke-restart**.

A *restart* contains a *function* to be *called* when the *restart* is invoked, an optional name that can be used to find or invoke the *restart*, and an optional set of interaction information for the debugger to use to enable the user to manually invoke a *restart*.

The name of a *restart* is used by **invoke-restart**. *Restarts* that can be invoked only within the debugger do not need names.

Restarts can be established by using **restart-bind**, **restart-case**, and **with-simple-restart**. A *restart* function can itself invoke any other *restart* that was active at the time of establishment of the *restart* of which the *function* is part.

The *restarts established* by a **restart-bind** *form*, a **restart-case** *form*, or a **with-simple-restart** *form* have *dynamic extent* which extends for the duration of that *form*'s execution.

Restarts of the same name can be ordered from least recent to most recent according to the following two rules:

1. Each *restart* in a set of active restarts R1 is more recent than every *restart* in a set R2 if the *restarts* in R2 were active when the *restarts* in R1 were established.
2. Let r1 and r2 be two active *restarts* with the same name established by the same *form*. Then r1 is more recent than r2 if r1 was defined to the left of r2 in the *form* that established them.

If a *restart* is invoked but does not transfer control, the values resulting from the *restart* function are returned by the function that invoked the restart, either **invoke-restart** or **invoke-restart-interactively**.

9.1.4.2.1 Interactive Use of Restarts

For interactive handling, two pieces of information are needed from a *restart*: a report function and an interactive function.

The report function is used by a program such as the debugger to present a description of the action the *restart* will take. The report function is specified and established by the `:report-function` keyword to **restart-bind** or the `:report` keyword to **restart-case**.

The interactive function, which can be specified using the `:interactive-function` keyword to **restart-bind** or `:interactive` keyword to **restart-case**, is used when the *restart* is invoked interactively, such as from the debugger, to produce a suitable list of arguments.

invoke-restart invokes the most recently *established restart* whose name is the same as the first argument to **invoke-restart**. If a *restart* is invoked interactively by the debugger and does not transfer control but rather returns values, the precise action of the debugger on those values is *implementation-defined*.

9.1.4.2.2 Interfaces to Restarts

Some *restarts* have functional interfaces, such as **abort**, **continue**, **muffle-warning**, **store-value**, and **use-value**. They are ordinary functions that use **find-restart** and **invoke-restart** internally, that have the same name as the *restarts* they manipulate, and that are provided simply for notational convenience.

The next figure shows *defined names* relating to *restarts*.

```
abort             invoke-restart-interactively  store-value
compute-restarts  muffle-warning                use-value
continue          restart-bind                  with-simple-restart
find-restart      restart-case
invoke-restart    restart-name
```

Figure 9-6. Defined names relating to restarts.

9.1.4.2.3 Restart Tests

Each *restart* has an associated test, which is a function of one argument (a *condition* or **nil**) which returns *true* if the *restart* should be visible in the current *situation*. This test is created by the `:test-function` option to **restart-bind** or the `:test` option to **restart-case**.

9.1.4.2.4 Associating a Restart with a Condition

A *restart* can be "associated with" a *condition* explicitly by **with-condition-restarts**, or implicitly by **restart-case**. Such an assocation has *dynamic extent*.

A single *restart* may be associated with several *conditions* at the same time. A single *condition* may have several associated *restarts* at the same time.

Active restarts associated with a particular *condition* can be detected by *calling* a *function* such as **find-restart**, supplying that *condition* as the *condition argument*. Active restarts can also be detected without regard to any associated *condition* by calling such a function without a *condition argument*, or by supplying a value of **nil** for such an *argument*.

9.1.5 Assertions

Conditional signaling of *conditions* based on such things as key match, form evaluation, and *type* are handled by assertion *operators*. The next figure shows *operators* relating to assertions.

```
assert  check-type  ecase
ccase   ctypecase   etypecase
```

Figure 9-7. Operators relating to assertions.

9.1.6 Notes about the Condition System’s Background

For a background reference to the abstract concepts detailed in this section, see *Exceptional Situations in Lisp*. The details of that paper are not binding on this document, but may be helpful in establishing a conceptual basis for understanding this material.

10. Symbols

10.1 Symbol Concepts

The next figure lists some *defined names* that are applicable to the *property lists* of *symbols*.

```
get  remprop  symbol-plist
```

Figure 10-1. Property list defined names

The next figure lists some *defined names* that are applicable to the creation of and inquiry about *symbols*.

```
copy-symbol  keywordp     symbol-package
gensym       make-symbol  symbol-value
gentemp      symbol-name
```

11. Packages

11.1 Package Concepts

11.1.1 Introduction to Packages

A *package* establishes a mapping from names to *symbols*. At any given time, one *package* is current. The *current package* is the one that is the *value* of ***package***. When using the *Lisp reader*, it is possible to refer to *symbols* in *packages* other than the current one through the use of *package prefixes* in the printed representation of the *symbol*.

The next figure lists some *defined names* that are applicable to *packages*. Where an *operator* 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 *package* argument may be either a *string*, a *symbol*, or a *package*. If a *symbol* is supplied, its name will be used as the *package* name.

```
*modules*            import                    provide
*package*            in-package                rename-package
defpackage           intern                    require
do-all-symbols       list-all-packages         shadow
do-external-symbols  make-package              shadowing-import
do-symbols           package-name              unexport
export               package-nicknames         unintern
find-all-symbols     package-shadowing-symbols unuse-package
find-package         package-use-list          use-package
find-symbol          package-used-by-list
```

Figure 11-1. Some Defined Names related to Packages

11.1.1.1 Package Names and Nicknames

Each *package* has a *name* (a *string*) and perhaps some *nicknames* (also *strings*). These are assigned when the *package* is created and can be changed later.

There is a single namespace 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*. **rename-package** removes a *package*'s current *name* and *nicknames* and replaces them with new ones specified by the caller.

11.1.1.2 Symbols in a Package

11.1.1.2.1 Internal and External Symbols

The mappings in a *package* are divided into two classes, external and internal. The *symbols* targeted by these different mappings are called *external symbols* and *internal symbols* of the *package*. Within a *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*. *Symbols* become *external symbols* of a given *package* if they have been *exported* from that *package*.

A *symbol* has the same *name* no matter what *package* it is *present* in, but it might be an *external symbol* of some *packages* and an *internal symbol* of others.

11.1.1.2.2 Package Inheritance

Packages can be built up in layers. From one point of view, a *package* is a single collection of mappings from *strings* into *internal symbols* and *external symbols*. However, some of these mappings might be established within the *package* itself, while other mappings are inherited from other *packages* via **use-package**. A *symbol* is said to be *present* in a *package* if the mapping is in the *package* itself and is not inherited from somewhere else.

There is no way to inherit the *internal symbols* of another *package*; to refer to an *internal symbol* using the *Lisp reader*, a *package* containing the *symbol* must be made to be the *current package*, a *package prefix* must be used, or the *symbol* must be *imported* into the *current package*.

11.1.1.2.3 Accessibility of Symbols in a Package

A *symbol* becomes *accessible* in a *package* if that is its *home package* when it is created, or if it is *imported* into that *package*, or by inheritance via **use-package**.

If a *symbol* is *accessible* in a *package*, it can be referred to when using the *Lisp reader* without a *package prefix* when that *package* is the *current package*, regardless of whether it is *present* or inherited.

Symbols from one *package* can be made *accessible* in another *package* in two ways.

-- Any individual *symbol* can be added to a *package* by use of **import**. After the call to **import** the *symbol* is *present* in the importing *package*. The status of the *symbol* in the *package* it came from (if any) is unchanged, and the *home package* for this *symbol* is unchanged. Once *imported*, a *symbol* is *present* in the importing *package* and can be removed only by calling **unintern**.

A *symbol* is *shadowed*[3] by another *symbol* in some *package* if the first *symbol* would be *accessible* by inheritance if not for the presence of the second *symbol*. See **shadowing-import**.

-- The second mechanism for making *symbols* from one *package accessible* in another is provided by **use-package**. All of the *external symbols* of the used *package* are inherited by the using *package*. The *function* **unuse-package** undoes the effects of a previous **use-package**.

11.1.1.2.4 Locating a Symbol in a Package

When a *symbol* is to be located in a given *package* the following occurs:

-- The *external symbols* and *internal symbols* of the *package* are searched for the *symbol*.
-- The *external symbols* of the used *packages* are searched in some unspecified order. The order does not matter; see the rules for handling name conflicts listed below.

11.1.1.2.5 Prevention of Name Conflicts in Packages

Within one *package*, any particular name can refer to at most one *symbol*. A name conflict is said to occur when there would be more than one candidate *symbol*. Any time a name conflict is about to occur, a *correctable error* is signaled.

The following rules apply to name conflicts:

-- Name conflicts are detected when they become possible, that is, when the package structure is altered. Name conflicts are not checked during every name lookup.
-- If the *same symbol* is *accessible* to a *package* through more than one path, there is no name conflict. A *symbol* cannot conflict with itself. Name conflicts occur only between *distinct symbols* with the same name (under **string=**).
-- 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* in the *package*. A name conflict involving a shadowing symbol is always resolved in favor of the shadowing *symbol*, without signaling an error (except for one exception involving **import**). See **shadow** and **shadowing-import**.
-- The functions **use-package**, **import**, and **export** check for name conflicts.
-- **shadow** and **shadowing-import** never signal a name-conflict error.
-- **unuse-package** and **unexport** do not need to do any name-conflict checking. **unintern** does name-conflict checking only when a *symbol* being *uninterned* is a *shadowing symbol*.
-- Giving a shadowing symbol to **unintern** can uncover a name conflict that had previously been resolved by the shadowing.
-- Package functions signal name-conflict errors of *type* **package-error** before making any change to the package structure. When multiple changes are to be made, it is permissible for the implementation to process each change separately. For example, when **export** is given a *list* of *symbols*, aborting from a name conflict caused by the second *symbol* in the *list* might still export the first *symbol* in the *list*. However, a name-conflict error caused by **export** of a single *symbol* will be signaled before that *symbol*'s *accessibility* in any *package* is changed.
-- Continuing from a name-conflict error must 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 present* in the using *package* and an *external symbol* of the used *package* is 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*.
-- A name conflict in **export** or **unintern** due to a *package*'s inheriting two *distinct symbols* with the *same name* (under **string=**) from two other *packages* can be resolved in favor of either *symbol* by importing it into the using *package* and making it a *shadowing symbol*, just as with **use-package**.

11.1.2 Standardized Packages

This section describes the *packages* that are available in every *conforming implementation*. A summary of the *names* and *nicknames* of those *standardized packages* is given in the next figure.

```
Name              Nicknames
COMMON-LISP       CL
COMMON-LISP-USER  CL-USER
KEYWORD           none
```

Figure 11-2. Standardized Package Names

11.1.2.1 The COMMON-LISP Package

The COMMON-LISP package contains the primitives of the Common Lisp system as defined by this specification. Its *external symbols* include all of the *defined names* (except for *defined names* in the KEYWORD package) that are present in the Common Lisp system, such as **car**, **cdr**, ***package***, etc. The COMMON-LISP package has the *nickname* CL.

The COMMON-LISP package has as *external symbols* those symbols enumerated in the figures in Section 1.9 (Symbols in the COMMON-LISP Package), and no others. These *external symbols* are *present* in the COMMON-LISP package but their *home package* need not be the COMMON-LISP package.

For example, the symbol HELP cannot be an *external symbol* of the COMMON-LISP package because it is not mentioned in Section 1.9 (Symbols in the COMMON-LISP Package). In contrast, the *symbol* **variable** must be an *external symbol* of the COMMON-LISP package even though it has no definition because it is listed in that section (to support its use as a valid second *argument* to the *function* **documentation**).

The COMMON-LISP package can have additional *internal symbols*.

11.1.2.1.1 Constraints on the COMMON-LISP Package for Conforming Implementations

In a *conforming implementation*, an *external symbol* of the COMMON-LISP package can have a *function*, *macro*, or *special operator* definition, a *global variable* definition (or other status as a *dynamic variable* due to a **special** *proclamation*), or a *type* definition only if explicitly permitted in this standard. For example, **fboundp** *yields false* for any *external symbol* of the COMMON-LISP package that is not the *name* of a *standardized function*, *macro* or *special operator*, and **boundp** returns *false* for any *external symbol* of the COMMON-LISP package that is not the *name* of a *standardized global variable*. It also follows that *conforming programs* can use *external symbols* of the COMMON-LISP package as the *names* of local *lexical variables* with confidence that those *names* have not been *proclaimed* **special** by the *implementation* unless those *symbols* are *names* of *standardized global variables*.

A *conforming implementation* must not place any *property* on an *external symbol* of the COMMON-LISP package using a *property indicator* that is either an *external symbol* of any *standardized package* or a *symbol* that is otherwise *accessible* in the COMMON-LISP-USER package.

11.1.2.1.2 Constraints on the COMMON-LISP Package for Conforming Programs

Except where explicitly allowed, the consequences are undefined if any of the following actions are performed on an *external symbol* of the COMMON-LISP package:

1. *Binding* or altering its value (lexically or dynamically). (Some exceptions are noted below.)
2. Defining, undefining, or *binding* it as a *function*. (Some exceptions are noted below.)
3. Defining, undefining, or *binding* it as a *macro* or *compiler macro*. (Some exceptions are noted below.)
4. Defining it as a *type specifier* (via **defstruct**, **defclass**, **deftype**, **define-condition**).
5. Defining it as a structure (via **defstruct**).
6. Defining it as a *declaration* with a **declaration** *proclamation*.
7. Defining it as a *symbol macro*.
8. Altering its *home package*.
9. Tracing it (via **trace**).
10. Declaring or proclaiming it **special** (via **declare**, **declaim**, or **proclaim**).

11. Declaring or proclaiming its **type** or **ftype** (via **declare**, **declaim**, or **proclaim**). (Some exceptions are noted below.)
12. Removing it from the COMMON-LISP package.
13. Defining a *setf expander* for it (via **defsetf** or **define-setf-method**).
14. Defining, undefining, or binding its *setf function name*.
15. Defining it as a *method combination* type (via **define-method-combination**).
16. Using it as the class-name argument to **setf** of **find-class**.
17. Binding it as a *catch tag*.
18. Binding it as a *restart name*.
19. Defining a *method* for a *standardized generic function* which is *applicable* when all of the *arguments* are *direct instances* of *standardized classes*.

11.1.2.1.2.1 Some Exceptions to Constraints on the COMMON-LISP Package for Conforming Programs

If an *external symbol* of the COMMON-LISP package is not globally defined as a *standardized dynamic variable* or *constant variable*, it is allowed to lexically *bind* it and to declare the **type** of that *binding*, and it is allowed to locally *establish* it as a *symbol macro* (e.g., with **symbol-macrolet**).

Unless explicitly specified otherwise, if an *external symbol* of the COMMON-LISP package is globally defined as a *standardized dynamic variable*, it is permitted to *bind* or *assign* that *dynamic variable* provided that the "Value Type" constraints on the *dynamic variable* are maintained, and that the new *value* of the *variable* is consistent with the stated purpose of the *variable*.

If an *external symbol* of the COMMON-LISP package is not defined as a *standardized function*, *macro*, or *special operator*, it is allowed to lexically *bind* it as a *function* (e.g., with **flet**), to declare the **ftype** of that *binding*, and (in *implementations* which provide the ability to do so) to **trace** that *binding*.

If an *external symbol* of the COMMON-LISP package is not defined as a *standardized function*, *macro*, or *special operator*, it is allowed to lexically *bind* it as a *macro* (e.g., with **macrolet**).

If an *external symbol* of the COMMON-LISP package is not defined as a *standardized function*, *macro*, or *special operator*, it is allowed to lexically *bind* its *setf function name* as a *function*, and to declare the **ftype** of that *binding*.

11.1.2.2 The COMMON-LISP-USER Package

The COMMON-LISP-USER package is the *current package* when a Common Lisp system starts up. This *package uses* the COMMON-LISP package. The COMMON-LISP-USER package has the *nickname* CL-USER. The COMMON-LISP-USER package can have additional *symbols interned* within it; it can *use* other *implementation-defined packages*.

11.1.2.3 The KEYWORD Package

The KEYWORD package contains *symbols*, called *keywords*[1], that are typically used as special markers in *programs* and their associated data *expressions*[1].

Symbol tokens that start with a *package marker* are parsed by the *Lisp reader* as *symbols* in the KEYWORD package; see Section 2.3.4 (Symbols as Tokens). This makes it notationally convenient to use *keywords* when communicating between programs in different *packages*. For example, the mechanism for passing *keyword parameters* in a *call* uses *keywords*[1] to name the corresponding *arguments*; see Section 3.4.1 (Ordinary Lambda Lists).

Symbols in the KEYWORD package are, by definition, of *type* **keyword**.

11.1.2.3.1 Interning a Symbol in the KEYWORD Package

The KEYWORD package is treated differently than other *packages* in that special actions are taken when a *symbol* is *interned* in it. In particular, when a *symbol* is *interned* in the KEYWORD package, it is automatically made to be an *external symbol* and is automatically made to be a *constant variable* with itself as a *value*.

11.1.2.3.2 Notes about The KEYWORD Package

It is generally best to confine the use of *keywords* to situations in which there are a finitely enumerable set of names to be selected between. For example, if there were two states of a light switch, they might be called :on and :off.

In situations where the set of names is not finitely enumerable (i.e., where name conflicts might arise) it is frequently best to use *symbols* in some *package* other than KEYWORD so that conflicts will be naturally avoided. For example, it is generally not wise for a *program* to use a *keyword*[1] as a *property indicator,* since if there were ever another *program* that did the same thing, each would clobber the other's data.

11.1.2.4 Implementation-Defined Packages

Other, *implementation-defined packages* might be present in the initial Common Lisp environment.

It is recommended, but not required, that the documentation for a *conforming implementation* contain a full list of all *package* names initially present in that *implementation* but not specified in this specification. (See also the *function* **list-all-packages**.)

12. Numbers

12.1 Number Concepts

12.1.1 Numeric Operations

Common Lisp provides a large variety of operations related to *numbers*. This section provides an overview of those operations by grouping them into categories that emphasize some of the relationships among them.

The next figure shows *operators* relating to arithmetic operations.

```
*  1+         gcd
+  1-         incf
-  conjugate  lcm
/  decf
```

Figure 12-1. Operators relating to Arithmetic.

The next figure shows *defined names* relating to exponential, logarithmic, and trigonometric operations.

```
abs    cos    signum
acos   cosh   sin
acosh  exp    sinh
asin   expt   sqrt
asinh  isqrt  tan
atan   log    tanh
atanh  phase
cis    pi
```

Figure 12-2. Defined names relating to Exponentials, Logarithms, and Trigonometry.

The next figure shows *operators* relating to numeric comparison and predication.

```
/=  >=      oddp
<   evenp   plusp
<=  max     zerop
=   min
>   minusp
```

Figure 12-3. Operators for numeric comparison and predication.

The next figure shows *defined names* relating to numeric type manipulation and coercion.

```
ceiling          float-radix           rational
complex          float-sign            rationalize
decode-float     floor                 realpart
denominator      fround                rem
fceiling         ftruncate             round
ffloor           imagpart              scale-float
float            integer-decode-float  truncate
float-digits     mod
float-precision  numerator
```

Figure 12-4. Defined names relating to numeric type manipulation and coercion.

12.1.1.1 Associativity and Commutativity in Numeric Operations

For functions that are mathematically associative (and possibly commutative), a *conforming implementation* may process the *arguments* in any manner consistent with associative (and possibly commutative) rearrangement. This does not affect the order in which the *argument forms* are *evaluated*; for a discussion of evaluation order, see Section 3.1.2.1.2.3 (Function Forms). What is unspecified is only the order in which the *parameter values* are processed. This implies that *implementations* may differ in which automatic *coercions* are applied; see Section 12.1.1.2 (Contagion in Numeric Operations).

A *conforming program* can control the order of processing explicitly by separating the operations into separate (possibly nested) *function forms*, or by writing explicit calls to *functions* that perform coercions.

12.1.1.1.1 Examples of Associativity and Commutativity in Numeric Operations

Consider the following expression, in which we assume that `1.0` and `1.0e-15` both denote *single floats*:

```
(+ 1/3 2/3 1.0d0 1.0 1.0e-15)
```

One *conforming implementation* might process the *arguments* from left to right, first adding `1/3` and `2/3` to get `1`, then converting that to a *double float* for combination with `1.0d0`, then successively converting and adding `1.0` and `1.0e-15`.

Another *conforming implementation* might process the *arguments* from right to left, first performing a *single float* addition of `1.0` and `1.0e-15` (perhaps losing accuracy in the process), then converting the sum to a *double float* and adding `1.0d0`, then converting `2/3` to a *double float* and adding it, and then converting `1/3` and adding that.

A third *conforming implementation* might first scan all the *arguments*, process all the *rationals* first to keep that part of the computation exact, then find an *argument* of the largest floating-point format among all the *arguments* and add that, and then add in all other *arguments*, converting each in turn (all in a perhaps misguided attempt to make the computation as accurate as possible).

In any case, all three strategies are legitimate.

A *conforming program* could control the order by writing, for example,

```
(+ (+ 1/3 2/3) (+ 1.0d0 1.0e-15) 1.0)
```

12.1.1.2 Contagion in Numeric Operations

For information about the contagion rules for implicit coercions of *arguments* in numeric operations, see Section 12.1.4.4 (Rule of Float Precision Contagion), Section 12.1.4.1 (Rule of Float and Rational Contagion), and Section 12.1.5.2 (Rule of Complex Contagion).

12.1.1.3 Viewing Integers as Bits and Bytes

12.1.1.3.1 Logical Operations on Integers

Logical operations require *integers* as arguments; an error of *type* **type-error** should be signaled if an argument is supplied that is not an *integer*. *Integer* arguments to logical operations are treated as if they were represented in two's-complement notation.

The next figure shows *defined names* relating to logical operations on numbers.

```
ash          boole-ior       logbitp
boole        boole-nand      logcount
boole-1      boole-nor       logeqv
boole-2      boole-orc1      logior
boole-and    boole-orc2      lognand
boole-andc1  boole-set       lognor
boole-andc2  boole-xor       lognot
boole-c1     integer-length  logorc1
boole-c2     logand          logorc2
boole-clr    logandc1        logtest
boole-eqv    logandc2        logxor
```

Figure 12-5. Defined names relating to logical operations on numbers.

12.1.1.3.2 Byte Operations on Integers

The byte-manipulation *functions* use *objects* called *byte specifiers* to designate the size and position of a specific *byte* within an *integer*. The representation of a *byte specifier* is *implementation-dependent*; it might or might not be a *number*. The *function* **byte** will construct a *byte specifier*, which various other byte-manipulation *functions* will accept.

The next figure shows *defined names* relating to manipulating *bytes* of *numbers*.

```
byte           deposit-field  ldb-test
byte-position  dpb            mask-field
byte-size      ldb
```

Figure 12-6. Defined names relating to byte manipulation.

12.1.2 Implementation-Dependent Numeric Constants

The next figure shows *defined names* relating to *implementation-dependent* details about *numbers*.

```
double-float-epsilon           most-negative-fixnum
double-float-negative-epsilon  most-negative-long-float
least-negative-double-float    most-negative-short-float
least-negative-long-float      most-negative-single-float
least-negative-short-float     most-positive-double-float
least-negative-single-float    most-positive-fixnum
least-positive-double-float    most-positive-long-float
least-positive-long-float      most-positive-short-float
least-positive-short-float     most-positive-single-float
least-positive-single-float    short-float-epsilon
long-float-epsilon             short-float-negative-epsilon
long-float-negative-epsilon    single-float-epsilon
most-negative-double-float     single-float-negative-epsilon
```

Figure 12-7. Defined names relating to implementation-dependent details about numbers.

12.1.3 Rational Computations

The rules in this section apply to *rational* computations.

12.1.3.1 Rule of Unbounded Rational Precision

Rational computations cannot overflow in the usual sense (though there may not be enough storage to represent a result), since *integers* and *ratios* may in principle be of any magnitude.

12.1.3.2 Rule of Canonical Representation for Rationals

If any computation produces a result that is a mathematical ratio of two integers such that the denominator evenly divides the numerator, then the result is converted to the equivalent *integer*.

If the denominator does not evenly divide the numerator, the canonical representation of a *rational* number is as the *ratio* that numerator and that denominator, where the greatest common divisor of the numerator and denominator is one, and where the denominator is positive and greater than one.

When used as input (in the default syntax), the notation `-0` always denotes the *integer* `0`. A *conforming implementation* must not have a representation of "minus zero" for *integers* that is distinct from its representation of zero for *integers*. However, such a distinction is possible for *floats*; see the *type* **float**.

12.1.3.3 Rule of Float Substitutability

When the arguments to an irrational mathematical *function* are all *rational* and the true mathematical result is also (mathematically) rational, then unless otherwise noted an implementation is free to return either an accurate *rational* result or a *single float* approximation. If the arguments are all *rational* but the result cannot be expressed as a *rational* number, then a *single float* approximation is always returned.

If the arguments to an irrational mathematical *function* are all of type `(or rational (complex rational))` and the true mathematical result is (mathematically) a complex number with rational real and imaginary parts, then unless otherwise noted an implementation is free to return either an accurate result of type `(or rational (complex rational))` or a *single float* (permissible only if the imaginary part of the true mathematical result is zero) or `(complex single-float)`. If the arguments are all of type `(or rational (complex rational))` but the result cannot be expressed as a *rational* or *complex rational*, then the returned

value will be of *type* **single-float** (permissible only if the imaginary part of the true mathematical result is zero) or `(complex single-float)`.

Float substitutability applies neither to the rational *functions* **+**, **-**, *****, and / nor to the related *operators* **1+**, **1-**, **incf**, **decf**, and **conjugate**. For rational *functions*, if all arguments are *rational*, then the result is *rational*; if all arguments are of type `(or rational (complex rational))`, then the result is of type `(or rational (complex rational))`.

```
Function  Sample Results
abs       (abs #c(3 4)) =>  5 or 5.0
acos      (acos 1) =>  0 or 0.0
acosh     (acosh 1) =>  0 or 0.0
asin      (asin 0) =>  0 or 0.0
asinh     (asinh 0) =>  0 or 0.0
atan      (atan 0) =>  0 or 0.0
atanh     (atanh 0) =>  0 or 0.0
cis       (cis 0) =>  1 or #c(1.0 0.0)
cos       (cos 0) =>  1 or 1.0
cosh      (cosh 0) =>  1 or 1.0
exp       (exp 0) =>  1 or 1.0
expt      (expt 8 1/3) =>  2 or 2.0
log       (log 1) =>  0 or 0.0
          (log 8 2) =>  3 or 3.0
phase     (phase 7) =>  0 or 0.0
signum    (signum #c(3 4)) =>  #c(3/5 4/5) or #c(0.6 0.8)
sin       (sin 0) =>  0 or 0.0
sinh      (sinh 0) =>  0 or 0.0
sqrt      (sqrt 4) =>  2 or 2.0
          (sqrt 9/16) =>  3/4 or 0.75
tan       (tan 0) =>  0 or 0.0
tanh      (tanh 0) =>  0 or 0.0
```

Figure 12-8. Functions Affected by Rule of Float Substitutability

12.1.4 Floating-point Computations

The following rules apply to floating point computations.

12.1.4.1 Rule of Float and Rational Contagion

When *rationals* and *floats* are combined by a numerical function, the *rational* is first converted to a *float* of the same format. For *functions* such as **+** that take more than two arguments, it is permitted that part of the operation be carried out exactly using *rationals* and the rest be done using floating-point arithmetic.

When *rationals* and *floats* are compared by a numerical function, the *function* **rational** is effectively called to convert the *float* to a *rational* and then an exact comparison is performed. In the case of *complex* numbers, the real and imaginary parts are effectively handled individually.

12.1.4.1.1 Examples of Rule of Float and Rational Contagion

```
;;;; Combining rationals with floats.
;;; This example assumes an implementation in which
;;; (float-radix 0.5) is 2 (as in IEEE) or 16 (as in IBM/360),
;;; or else some other implementation in which 1/2 has an exact
;;;  representation in floating point.
(+ 1/2 0.5) =>  1.0
(- 1/2 0.5d0) =>  0.0d0
(+ 0.5 -0.5 1/2) =>  0.5
```

```
;;;; Comparing rationals with floats.
;;; This example assumes an implementation in which the default float
;;; format is IEEE single-float, IEEE double-float, or some other format
;;; in which 5/7 is rounded upwards by FLOAT.
(< 5/7 (float 5/7)) =>  true
(< 5/7 (rational (float 5/7))) =>  true
(< (float 5/7) (float 5/7)) =>  false
```

12.1.4.3 Rule of Float Underflow and Overflow

An error of *type* **floating-point-overflow** or **floating-point-underflow** should be signaled if a floating-point computation causes exponent overflow or underflow, respectively.

12.1.4.4 Rule of Float Precision Contagion

The result of a numerical function is a *float* of the largest format among all the floating-point arguments to the *function*.

12.1.5 Complex Computations

The following rules apply to *complex* computations:

12.1.5.1 Rule of Complex Substitutability

Except during the execution of irrational and transcendental *functions*, no numerical *function* ever *yields* a *complex* unless one or more of its *arguments* is a *complex*.

12.1.5.2 Rule of Complex Contagion

12.1.5.3 Rule of Canonical Representation for Complex Rationals

If the result of any computation would be a *complex* number whose real part is of *type* **rational** and whose imaginary part is zero, the result is converted to the *rational* which is the real part. This rule does not apply to *complex* numbers whose parts are *floats*. For example, `#C(5 0)` and `5` are not *different objects* in Common Lisp(they are always the *same* under **eql**); `#C(5.0 0.0)` and `5.0` are always *different objects* in Common Lisp they are never the *same* under **eql**, although they are the *same* under **equalp** and `=`).

12.1.5.3.1 Examples of Rule of Canonical Representation for Complex Rationals

```
#c(1.0 1.0) =>  #C(1.0 1.0)
#c(0.0 0.0) =>  #C(0.0 0.0)
#c(1.0 1) =>  #C(1.0 1.0)
#c(0.0 0) =>  #C(0.0 0.0)
#c(1 1) =>  #C(1 1)
#c(0 0) =>  0
(typep #c(1 1) '(complex (eql 1))) =>  true
(typep #c(0 0) '(complex (eql 0))) =>  false
```

12.1.5.4 Principal Values and Branch Cuts

Many of the irrational and transcendental functions are multiply defined in the complex domain; for example, there are in general an infinite number of complex values for the logarithm function. In each such case, a *principal value* must be chosen for the function to return. In general, such values cannot be chosen so as to make the range continuous; lines in the domain called branch cuts must be defined, which in turn define the discontinuities in the range. Common Lisp defines the branch cuts, *principal values*, and boundary conditions for the complex functions following "Principal Values and Branch Cuts in Complex APL." The branch cut rules that apply to each function are located with the description of that function.

The next figure lists the identities that are obeyed throughout the applicable portion of the complex domain, even on the branch cuts:

```
sin i z = i sinh z  sinh i z = i sin z        arctan i z = i arctanh z
cos i z = cosh z    cosh i z = cos z          arcsinh i z = i arcsin z
tan i z = i tanh z  arcsin i z = i arcsinh z  arctanh i z = i arctan z
```

Figure 12-9. Trigonometric Identities for Complex Domain

The quadrant numbers referred to in the discussions of branch cuts are as illustrated in the next figure.

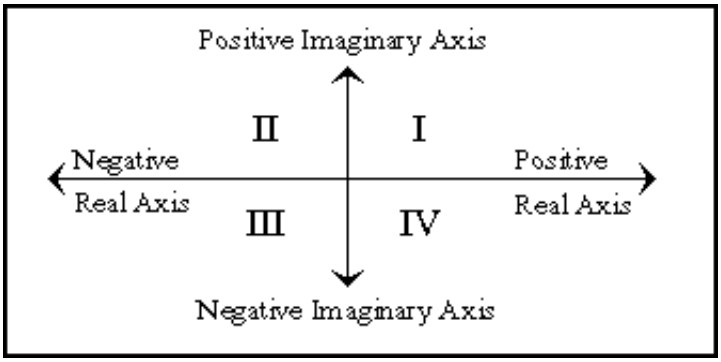

Figure 12-10. Quadrant Numbering for Branch Cuts

12.1.6 Interval Designators

The *compound type specifier* form of the numeric *type specifiers* permit the user to specify an interval on the real number line which describe a *subtype* of the *type* which would be described by the corresponding *atomic type specifier*. A *subtype* of some *type T* is specified using an ordered pair of *objects* called *interval designators* for *type T*.

The first of the two *interval designators* for *type T* can be any of the following:

a number *N* of *type T*
: This denotes a lower inclusive bound of *N*. That is, *elements* of the *subtype* of *T* will be greater than or equal to *N*.

a *singleton list* whose *element* is a number *M* of *type T*
: This denotes a lower exclusive bound of *M*. That is, *elements* of the *subtype* of *T* will be greater than *M*.

the symbol *****
: This denotes the absence of a lower bound on the interval.

The second of the two *interval designators* for *type T* can be any of the following:

a number *N* of *type T*

This denotes an upper inclusive bound of *N*. That is, *elements* of the *subtype* of *T* will be less than or equal to *N*.

a *singleton list* whose *element* is a number *M* of *type T*

This denotes an upper exclusive bound of *M*. That is, *elements* of the *subtype* of *T* will be less than *M*.

the symbol *****

This denotes the absence of an upper bound on the interval.

12.1.7 Random-State Operations

The next figure lists some *defined names* that are applicable to *random states*.

```
*random-state*     random
make-random-state  random-state-p
```

Figure 12-11. Random-state defined names

13. Characters

13.1 Character Concepts

13.1.1 Introduction to Characters

A *character* is an *object* that represents a unitary token (e.g., a letter, a special symbol, or a "control character") in an aggregate quantity of text (e.g., a *string* or a text *stream*).

Common Lisp allows an implementation to provide support for international language *characters* as well as *characters* used in specialized arenas (e.g., mathematics).

The following figures contain lists of *defined names* applicable to *characters*.

The next figure lists some *defined names* relating to *character attributes* and *character predicates*.

```
alpha-char-p     char-not-equal     char>
alphanumericp    char-not-greaterp  char>=
both-case-p      char-not-lessp     digit-char-p
char-code-limit  char/=             graphic-char-p
char-equal       char<              lower-case-p
char-greaterp    char<=             standard-char-p
char-lessp       char=              upper-case-p
```

Figure 13-1. Character defined names -- 1

The next figure lists some *character* construction and conversion *defined names*.

```
char-code      char-name    code-char
char-downcase  char-upcase  digit-char
char-int       character    name-char
```

Figure 13-2. Character defined names -- 2

13.1.2 Introduction to Scripts and Repertoires

13.1.2.1 Character Scripts

A *script* is one of possibly several sets that form an *exhaustive partition* of the type **character**.

The number of such sets and boundaries between them is *implementation-defined*. Common Lisp does not require these sets to be *types*, but an *implementation* is permitted to define such *types* as an extension. Since no *character* from one *script* can ever be a member of another *script*, it is generally more useful to speak about *character repertoires*.

Although the term "*script*" is chosen for definitional compatibility with ISO terminology, no *conforming implementation* is required to use any particular *scripts* standardized by ISO or by any other standards organization.

Whether and how the *script* or *scripts* used by any given *implementation* are named is *implementation-dependent*.

13.1.2.2 Character Repertoires

A *repertoire* is a *type specifier* for a *subtype* of *type* **character**. This term is generally used when describing a collection of *characters* independent of their coding. *Characters* in *repertoires* are only identified by name, by *glyph*, or by character description.

A *repertoire* can contain *characters* from several *scripts*, and a *character* can appear in more than one *repertoire*.

For some examples of *repertoires*, see the coded character standards ISO 8859/1, ISO 8859/2, and ISO 6937/2. Note, however, that although the term "*repertoire*" is chosen for definitional compatibility with ISO terminology, no *conforming implementation* is required to use *repertoires* standardized by ISO or any other standards organization.

13.1.3 Character Attributes

Characters have only one *standardized attribute*: a *code*. A *character*'s *code* is a non-negative *integer*. This *code* is composed from a character *script* and a character label in an *implementation-dependent* way. See the *functions* **char-code** and **code-char**.

Additional, *implementation-defined attributes* of *characters* are also permitted so that, for example, two *characters* with the same *code* may differ in some other, *implementation-defined* way.

For any *implementation-defined attribute* there is a distinguished value called the *null* value for that *attribute*. A *character* for which each *implementation-defined attribute* has the null value for that *attribute* is called a *simple character*. If the *implementation* has no *implementation-defined attributes*, then all *characters* are *simple characters*.

13.1.4 Character Categories

There are several (overlapping) categories of *characters* that have no formally associated *type* but that are nevertheless useful to name. They include *graphic characters*, *alphabetic*[1] *characters*, *characters* with *case* (*uppercase* and *lowercase characters*), *numeric characters*, *alphanumeric characters*, and *digits* (in a given *radix*).

For each *implementation-defined attribute* of a *character*, the documentation for that *implementation* must specify whether *characters* that differ only in that *attribute* are permitted to differ in whether are not they are members of one of the aforementioned categories.

Note that these terms are defined independently of any special syntax which might have been enabled in the *current readtable*.

13.1.4.1 Graphic Characters

Characters that are classified as *graphic*, or displayable, are each associated with a glyph, a visual representation of the *character*.

A *graphic character* is one that has a standard textual representation as a single *glyph*, such as `A` or `*` or `=`. *Space*, which effectively has a blank *glyph*, is defined to be a *graphic*.

Of the *standard characters*, *newline* is *non-graphic* and all others are *graphic*; see Section 2.1.3 (Standard Characters).

Characters that are not *graphic* are called *non-graphic*. *Non-graphic characters* are sometimes informally called "formatting characters" or "control characters."

`#\Backspace`, `#\Tab`, `#\Rubout`, `#\Linefeed`, `#\Return`, and `#\Page`, if they are supported by the *implementation*, are *non-graphic*.

13.1.4.2 Alphabetic Characters

The *alphabetic*[1] *characters* are a subset of the *graphic characters*. Of the *standard characters*, only these are the *alphabetic*[1] *characters*:

```
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
```

Any *implementation-defined character* that has *case* must be *alphabetic*[1]. For each *implementation-defined graphic character* that has no *case*, it is *implementation-defined* whether that *character* is *alphabetic*[1].

13.1.4.3 Characters With Case

The *characters* with *case* are a subset of the *alphabetic*[1] *characters*. A *character* with *case* has the property of being either *uppercase* or *lowercase*. Every *character* with *case* is in one-to-one correspondence with some other *character* with the opposite *case*.

13.1.4.3.1 Uppercase Characters

An uppercase *character* is one that has a corresponding *lowercase character* that is *different* (and can be obtained using **char-downcase**).

Of the *standard characters*, only these are *uppercase characters*:

```
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
```

13.1.4.3.2 Lowercase Characters

A lowercase *character* is one that has a corresponding *uppercase character* that is *different* (and can be obtained using **char-upcase**).

Of the *standard characters*, only these are *lowercase characters*:

```
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
```

13.1.4.3.3 Corresponding Characters in the Other Case

The *uppercase standard characters* `A` through `Z` mentioned above respectively correspond to the *lowercase standard characters* `a` through `z` mentioned above. For example, the *uppercase character* `E` corresponds to the *lowercase character* `e`, and vice versa.

13.1.4.3.4 Case of Implementation-Defined Characters

An *implementation* may define that other *implementation-defined graphic characters* have *case*. Such definitions must always be done in pairs---one *uppercase character* in one-to-one *correspondence* with one *lowercase character*.

13.1.4.4 Numeric Characters

The *numeric characters* are a subset of the *graphic characters*. Of the *standard characters*, only these are *numeric characters*:

```
0 1 2 3 4 5 6 7 8 9
```

For each *implementation-defined graphic character* that has no *case*, the *implementation* must define whether or not it is a *numeric character*.

13.1.4.5 Alphanumeric Characters

The set of *alphanumeric characters* is the union of the set of *alphabetic*[1] *characters* and the set of *numeric characters*.

13.1.4.6 Digits in a Radix

What qualifies as a *digit* depends on the *radix* (an *integer* between `2` and `36`, inclusive). The potential *digits* are:

```
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
```

Their respective weights are `0`, `1`, `2`, ... `35`. In any given radix n, only the first n potential *digits* are considered to be *digits*. For example, the digits in radix `2` are `0` and `1`, the digits in radix `10` are `0` through `9`, and the digits in radix `16` are `0` through `F`.

Case is not significant in *digits*; for example, in radix `16`, both `F` and `f` are *digits* with weight `15`.

13.1.5 Identity of Characters

Two *characters* that are **eql**, **char=**, or **char-equal** are not necessarily **eq**.

13.1.6 Ordering of Characters

The total ordering on *characters* is guaranteed to have the following properties:

* If two *characters* have the same *implementation-defined attributes*, then their ordering by **char<** is consistent with the numerical ordering by the predicate < on their code *attributes*.
* If two *characters* differ in any *attribute*, then they are not **char=**.
* The total ordering is not necessarily the same as the total ordering on the *integers* produced by applying **char-int** to the *characters*.
* While *alphabetic*[1] *standard characters* of a given *case* must obey a partial ordering, they need not be contiguous; it is permissible for *uppercase* and *lowercase characters* to be interleaved. Thus `(char<= #\a x #\z)` is not a valid way of determining whether or not `x` is a *lowercase character*.

Of the *standard characters*, those which are *alphanumeric* obey the following partial ordering:

```
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
0<1<2<3<4<5<6<7<8<9
either 9<A or Z<0
either 9<a or z<0
```

This implies that, for *standard characters*, *alphabetic*[1] ordering holds within each *case* (*uppercase* and *lowercase*), and that the *numeric characters* as a group are not interleaved with *alphabetic characters*. However, the ordering or possible interleaving of *uppercase characters* and *lowercase characters* is *implementation-defined*.

13.1.7 Character Names

The following *character names* must be present in all *conforming implementations*:

`Newline`
: The character that represents the division between lines. An implementation must translate between `#\Newline`, a single-character representation, and whatever external representation(s) may be used.

`Space`
: The space or blank character.

The following names are *semi-standard*; if an *implementation* supports them, they should be used for the described *characters* and no others.

`Rubout`
: The rubout or delete character.

`Page`
: The form-feed or page-separator character.

`Tab`
: The tabulate character.

`Backspace`
: The backspace character.

`Return`
: The carriage return character.

`Linefeed`
: The line-feed character.

In some *implementations*, one or more of these *character names* might denote a *standard character*; for example, `#\Linefeed` and `#\Newline` might be the *same character* in some *implementations*.

13.1.8 Treatment of Newline during Input and Output

When the character `#\Newline` is written to an output file, the 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. When reading, a corresponding reverse transformation must take place.

13.1.9 Character Encodings

A *character* is sometimes represented merely by its *code*, and sometimes by another *integer* value which is composed from the *code* and all *implementation-defined attributes* (in an *implementation-defined* way that might vary between *Lisp images* even in the same *implementation*). This *integer*, returned by the function **char-int**, is called the character's "encoding." There is no corresponding function from a character's encoding back to the *character*, since its primary intended uses include things like hashing where an inverse operation is not really called for.

13.1.10 Documentation of Implementation-Defined Scripts

An *implementation* must document the *character scripts* it supports. For each *character script* supported, the documentation must describe at least the following:

* Character labels, glyphs, and descriptions. Character labels must be uniquely named using only Latin capital letters A--Z, hyphen (-), and digits 0--9.
* Reader canonicalization. Any mechanisms by which **read** treats *different* characters as equivalent must be documented.
* The impact on **char-upcase**, **char-downcase**, and the case-sensitive *format directives*. In particular, for each *character* with *case*, whether it is *uppercase* or *lowercase*, and which *character* is its equivalent in the opposite case.
* The behavior of the case-insensitive *functions* **char-equal**, **char-not-equal**, **char-lessp**, **char-greaterp**, **char-not-greaterp**, and **char-not-lessp**.
* The behavior of any *character predicates*; in particular, the effects of **alpha-char-p**, **lower-case-p**, **upper-case-p**, **both-case-p**, **graphic-char-p**, and **alphanumericp**.
* The interaction with file I/O, in particular, the supported coded character sets (for example, ISO8859/1-1987) and external encoding schemes supported are documented.

14. Conses

14.1 Cons Concepts

A *cons* is a compound data *object* having two components called the *car* and the *cdr*.

```
car  cons    rplacd
cdr  rplaca
```

Figure 14-1. Some defined names relating to conses.

Depending on context, a group of connected *conses* can be viewed in a variety of different ways. A variety of operations is provided to support each of these various views.

14.1.1 Conses as Trees

A *tree* is a binary recursive data structure made up of *conses* and *atoms*: the *conses* are themselves also *trees* (sometimes called "subtrees" or "branches"), and the *atoms* are terminal nodes (sometimes called *leaves*). Typically, the *leaves* represent data while the branches establish some relationship among that data.

```
caaaar  caddar  cdar       nsubst
caaadr  cadddr  cddaar     nsubst-if
caaar   caddr   cddadr     nsubst-if-not
caadar  cadr    cddar      nthcdr
caaddr  cdaaar  cdddar     sublis
caadr   cdaadr  cddddr     subst
caar    cdaar   cdddr      subst-if
cadaar  cdadar  cddr       subst-if-not
cadadr  cdaddr  copy-tree  tree-equal
cadar   cdadr   nsublis
```

Figure 14-2. Some defined names relating to trees.

14.1.1.1 General Restrictions on Parameters that must be Trees

Except as explicitly stated otherwise, for any *standardized function* that takes a *parameter* that is required to be a *tree*, the consequences are undefined if that *tree* is circular.

14.1.2 Conses as Lists

A *list* is a chain of *conses* in which the *car* of each *cons* is an *element* of the *list*, and the *cdr* of each *cons* is either the next link in the chain or a terminating *atom*.

A *proper list* is a *list* terminated by the *empty list*. The *empty list* is a *proper list*, but is not a *cons*.

An *improper list* is a *list* that is not a *proper list*; that is, it is a *circular list* or a *dotted list*.

A *dotted list* is a *list* that has a terminating *atom* that is not the *empty list*. A *non-nil atom* by itself is not considered to be a *list* of any kind---not even a *dotted list*.

A *circular list* is a chain of *conses* that has no termination because some *cons* in the chain is the *cdr* of a later *cons*.

```
append      last          nbutlast  rest
butlast     ldiff         nconc     revappend
copy-alist  list          ninth     second
copy-list   list*         nreconc   seventh
eighth      list-length   nth       sixth
endp        make-list     nthcdr    tailp
fifth       member        pop       tenth
first       member-if     push      third
fourth      member-if-not pushnew
```

Figure 14-3. Some defined names relating to lists.

14.1.2.1 Lists as Association Lists

An *association list* is a *list* of *conses* representing an association of *keys* with *values*, where the *car* of each *cons* is the *key* and the *cdr* is the *value* associated with that *key*.

```
acons  assoc-if      pairlis  rassoc-if
assoc  assoc-if-not  rassoc   rassoc-if-not
```

Figure 14-4. Some defined names related to assocation lists.

14.1.2.2 Lists as Sets

Lists are sometimes viewed as sets by considering their elements unordered and by assuming there is no duplication of elements.

```
adjoin         nset-difference    set-difference    union
intersection   nset-exclusive-or  set-exclusive-or
nintersection  nunion             subsetp
```

Figure 14-5. Some defined names related to sets.

14.1.2.3 General Restrictions on Parameters that must be Lists

Except as explicitly specified otherwise, any *standardized function* that takes a *parameter* that is required to be a *list* should be prepared to signal an error of *type* **type-error** if the *value* received is a *dotted list*.

Except as explicitly specified otherwise, for any *standardized function* that takes a *parameter* that is required to be a *list*, the consequences are undefined if that *list* is *circular*.

15. Arrays

15.1 Array Concepts

15.1.1 Array Elements

An *array* contains a set of *objects* called *elements* that can be referenced individually according to a rectilinear coordinate system.

15.1.1.1 Array Indices

An *array element* is referred to by a (possibly empty) series of indices. The length of the series must equal the *rank* of the *array*. Each index must be a non-negative *fixnum* less than the corresponding *array dimension*. *Array* indexing is zero-origin.

15.1.1.2 Array Dimensions

An axis of an *array* is called a *dimension*.

Each *dimension* is a non-negative *fixnum*; if any dimension of an *array* is zero, the *array* has no elements. It is permissible for a *dimension* to be zero, in which case the *array* has no elements, and any attempt to *access* an *element* is an error. However, other properties of the *array*, such as the *dimensions* themselves, may be used.

15.1.1.2.1 Implementation Limits on Individual Array Dimensions

An *implementation* may impose a limit on *dimensions* of an *array*, but there is a minimum requirement on that limit. See the *variable* **array-dimension-limit**.

15.1.1.3 Array Rank

An *array* can have any number of *dimensions* (including zero). The number of *dimensions* is called the *rank*.

If the rank of an *array* is zero then the *array* is said to have no *dimensions*, and the product of the dimensions (see **array-total-size**) is then 1; a zero-rank *array* therefore has a single element.

15.1.1.3.1 Vectors

An *array* of *rank* one (i.e., a one-dimensional *array*) is called a *vector*.

15.1.1.3.1.1 Fill Pointers

A *fill pointer* is a non-negative *integer* no larger than the total number of *elements* in a *vector*. Not all *vectors* have *fill pointers*. See the *functions* **make-array** and **adjust-array**.

An *element* of a *vector* is said to be *active* if it has an index that is greater than or equal to zero, but less than the *fill pointer* (if any). For an *array* that has no *fill pointer*, all *elements* are considered *active*.

Only *vectors* may have *fill pointers*; multidimensional *arrays* may not. A multidimensional *array* that is displaced to a *vector* that has a *fill pointer* can be created.

15.1.1.3.2 Multidimensional Arrays

15.1.1.3.2.1 Storage Layout for Multidimensional Arrays

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.

15.1.1.3.2.2 Implementation Limits on Array Rank

An *implementation* may impose a limit on the *rank* of an *array*, but there is a minimum requirement on that limit. See the *variable* **array-rank-limit**.

15.1.2 Specialized Arrays

An *array* can be a *general array*, meaning each *element* may be any *object*, or it may be a *specialized array*, meaning that each *element* must be of a restricted *type*.

The phrasing "an *array specialized* to *type* <<*type*>>" is sometimes used to emphasize the *element type* of an *array*. This phrasing is tolerated even when the <<*type*>> is **t**, even though an *array specialized* to *type t* is a *general array*, not a *specialized array*.

The next figure lists some *defined names* that are applicable to *array* creation, *access*, and information operations.

```
adjust-array           array-has-fill-pointer-p  make-array
adjustable-array-p     array-in-bounds-p         svref
aref                   array-rank                upgraded-array-element-type
array-dimension        array-rank-limit          upgraded-complex-part-type
array-dimension-limit  array-row-major-index     vector
array-dimensions       array-total-size          vector-pop
array-displacement     array-total-size-limit    vector-push
array-element-type     fill-pointer              vector-push-extend
```

Figure 15-1. General Purpose Array-Related Defined Names

15.1.2.1 Array Upgrading

The *upgraded array element type* of a *type* T1 is a *type* T2 that is a *supertype* of T1 and that is used instead of T1 whenever T1 is used as an *array element type* for object creation or type discrimination.

During creation of an *array*, the *element type* that was requested is called the *expressed array element type*. The *upgraded array element type* of the *expressed array element type* becomes the *actual array element type* of the *array* that is created.

Type upgrading implies a movement upwards in the type hierarchy lattice. A *type* is always a *subtype* of its *upgraded array element type*. Also, if a *type* Tx is a *subtype* of another *type* Ty, then the *upgraded array element type* of Tx must be a *subtype* of the *upgraded array element type* of Ty. Two *disjoint types* can be *upgraded* to the same *type*.

The *upgraded array element type* T2 of a *type* T1 is a function only of T1 itself; that is, it is independent of any other property of the *array* for which T2 will be used, such as *rank*, *adjustability*, *fill pointers*, or displacement. The *function* **upgraded-array-element-type** can be used by *conforming programs* to predict how the *implementation* will *upgrade* a given *type*.

15.1.2.2 Required Kinds of Specialized Arrays

Vectors whose *elements* are restricted to *type* **character** or a *subtype* of **character** are called *strings*. *Strings* are of *type* **string**. The next figure lists some *defined names* related to *strings*.

Strings are *specialized arrays* and might logically have been included in this chapter. However, for purposes of readability most information about *strings* does not appear in this chapter; see instead Section 16 (Strings).

```
char                string-equal         string-upcase
make-string         string-greaterp      string/=
nstring-capitalize  string-left-trim     string<
nstring-downcase    string-lessp         string<=
nstring-upcase      string-not-equal     string=
schar               string-not-greaterp  string>
string              string-not-lessp     string>=
string-capitalize   string-right-trim
string-downcase     string-trim
```

Figure 15-2. Operators that Manipulate Strings

Vectors whose *elements* are restricted to *type* **bit** are called *bit vectors*. *Bit vectors* are of *type* **bit-vector**. The next figure lists some *defined names* for operations on *bit arrays*.

```
bit         bit-ior   bit-orc2
bit-and     bit-nand  bit-xor
bit-andc1   bit-nor   sbit
bit-andc2   bit-not
bit-eqv     bit-orc1
```

Figure 15-3. Operators that Manipulate Bit Arrays

16. Strings

16.1 String Concepts

16.1.1 Implications of Strings Being Arrays

Since all *strings* are *arrays*, all rules which apply generally to *arrays* also apply to *strings*. See Section 15.1 (Array Concepts).

For example, *strings* can have *fill pointers*, and *strings* are also subject to the rules of *element type upgrading* that apply to *arrays*.

16.1.2 Subtypes of STRING

All functions that operate on *strings* will operate on *subtypes* of *string* as well.

However, the consequences are undefined if a *character* is inserted into a *string* for which the *element type* of the *string* does not include that *character*.

17. Sequences

17.1 Sequence Concepts

A *sequence* is an ordered collection of *elements*, implemented as either a *vector* or a *list*.

Sequences can be created by the *function* **make-sequence**, as well as other *functions* that create *objects* of *types* that are *subtypes* of **sequence** (e.g., **list**, **make-list**, **mapcar**, and **vector**).

A *sequence function* is a *function* defined by this specification or added as an extension by the *implementation* that operates on one or more *sequences*. Whenever a *sequence function* must construct and return a new *vector*, it always returns a *simple vector*. Similarly, any *strings* constructed will be *simple strings*.

```
concatenate         length              remove
copy-seq            map                 remove-duplicates
count               map-into            remove-if
count-if            merge               remove-if-not
count-if-not        mismatch            replace
delete              notany              reverse
delete-duplicates   notevery            search
delete-if           nreverse            some
delete-if-not       nsubstitute         sort
elt                 nsubstitute-if      stable-sort
every               nsubstitute-if-not  subseq
fill                position            substitute
find                position-if         substitute-if
find-if             position-if-not     substitute-if-not
find-if-not         reduce
```

Figure 17-1. Standardized Sequence Functions

17.1.1 General Restrictions on Parameters that must be Sequences

In general, *lists* (including *association lists* and *property lists*) that are treated as *sequences* must be *proper lists*.

17.2 Rules about Test Functions

17.2.1 Satisfying a Two-Argument Test

When an *object* O is being considered iteratively against each *element* Ei of a *sequence* S by an *operator* F listed in the next figure, it is sometimes useful to control the way in which the presence of O is tested in S is tested by F. This control is offered on the basis of a *function* designated with either a `:test` or `:test-not` *argument*.

```
adjoin           nset-exclusive-or  search
assoc            nsublis            set-difference
count            nsubst             set-exclusive-or
delete           nsubstitute        sublis
find             nunion             subsetp
intersection     position           subst
member           pushnew            substitute
mismatch         rassoc             tree-equal
nintersection    remove             union
nset-difference  remove-duplicates
```

Figure 17-2. Operators that have Two-Argument Tests to be Satisfied

The object O might not be compared directly to Ei. If a `:key` *argument* is provided, it is a *designator* for a *function* of one *argument* to be called with each Ei as an *argument*, and *yielding* an *object* Zi to be used for comparison. (If there is no `:key` *argument*, Zi is Ei.)

The *function* designated by the `:key` *argument* is never called on O itself. However, if the function operates on multiple sequences (e.g., as happens in **set-difference**), O will be the result of calling the `:key` function on an *element* of the other sequence.

A `:test` *argument*, if supplied to F, is a *designator* for a *function* of two *arguments*, O and Zi. An Ei is said (or, sometimes, an O and an Ei are said) to *satisfy the test* if this `:test` *function* returns a *generalized boolean* representing *true*.

A `:test-not` *argument*, if supplied to F, is *designator* for a *function* of two *arguments*, O and Zi. An Ei is said (or, sometimes, an O and an Ei are said) to *satisfy the test* if this `:test-not` *function* returns a *generalized boolean* representing *false*.

If neither a `:test` nor a `:test-not` *argument* is supplied, it is as if a `:test` argument of `#'eql` was supplied.

The consequences are unspecified if both a `:test` and a `:test-not` *argument* are supplied in the same *call* to F.

17.2.1.1 Examples of Satisfying a Two-Argument Test

```
 (remove "FOO" '(foo bar "FOO" "BAR" "foo" "bar") :test #'equal)
=>  (foo bar "BAR" "foo" "bar")
 (remove "FOO" '(foo bar "FOO" "BAR" "foo" "bar") :test #'equalp)
=>  (foo bar "BAR" "bar")
 (remove "FOO" '(foo bar "FOO" "BAR" "foo" "bar") :test #'string-equal)
=>  (bar "BAR" "bar")
 (remove "FOO" '(foo bar "FOO" "BAR" "foo" "bar") :test #'string=)
=>  (BAR "BAR" "foo" "bar")
```

```
 (remove 1 '(1 1.0 #C(1.0 0.0) 2 2.0 #C(2.0 0.0)) :test-not #'eql)
=>  (1)
 (remove 1 '(1 1.0 #C(1.0 0.0) 2 2.0 #C(2.0 0.0)) :test-not #'=)
=>  (1 1.0 #C(1.0 0.0))
 (remove 1 '(1 1.0 #C(1.0 0.0) 2 2.0 #C(2.0 0.0)) :test (complement #'=))
=>  (1 1.0 #C(1.0 0.0))

 (count 1 '((one 1) (uno 1) (two 2) (dos 2)) :key #'cadr) =>  2

 (count 2.0 '(1 2 3) :test #'eql :key #'float) =>  1

 (count "FOO" (list (make-pathname :name "FOO" :type "X")
                    (make-pathname :name "FOO" :type "Y"))
        :key #'pathname-name
        :test #'equal)
=>  2
```

17.2.2 Satisfying a One-Argument Test

When using one of the *functions* in the next figure, the elements E of a *sequence* S are filtered not on the basis of the presence or absence of an object O under a two *argument predicate*, as with the *functions* described in Section 17.2.1 (Satisfying a Two-Argument Test), but rather on the basis of a one *argument predicate*.

```
assoc-if       member-if           rassoc-if
assoc-if-not   member-if-not       rassoc-if-not
count-if       nsubst-if           remove-if
count-if-not   nsubst-if-not       remove-if-not
delete-if      nsubstitute-if      subst-if
delete-if-not  nsubstitute-if-not  subst-if-not
find-if        position-if         substitute-if
find-if-not    position-if-not     substitute-if-not
```

Figure 17-3. Operators that have One-Argument Tests to be Satisfied

The element Ei might not be considered directly. If a `:key` *argument* is provided, it is a *designator* for a *function* of one *argument* to be called with each Ei as an *argument*, and *yielding* an *object* Zi to be used for comparison. (If there is no `:key` *argument*, Zi is Ei.)

Functions defined in this specification and having a name that ends in "`-if`" accept a first *argument* that is a *designator* for a *function* of one *argument*, Zi. An Ei is said to *satisfy the test* if this `:test` *function* returns a *generalized boolean* representing *true*.

Functions defined in this specification and having a name that ends in "`-if-not`" accept a first *argument* that is a *designator* for a *function* of one *argument*, Zi. An Ei is said to *satisfy the test* if this `:test` *function* returns a *generalized boolean* representing *false*.

17.2.2.1 Examples of Satisfying a One-Argument Test

```
 (count-if #'zerop '(1 #C(0.0 0.0) 0 0.0d0 0.0s0 3)) =>  4

 (remove-if-not #'symbolp '(0 1 2 3 4 5 6 7 8 9 A B C D E F))
=>  (A B C D E F)
 (remove-if (complement #'symbolp) '(0 1 2 3 4 5 6 7 8 9 A B C D E F))
=>  (A B C D E F)

 (count-if #'zerop '("foo" "" "bar" "" "" "baz" "quux") :key #'length)
=>  3
```

18. Hash Tables

18.1 Hash Table Concepts

18.1.1 Hash-Table Operations

The next figure lists some *defined names* that are applicable to *hash tables*. The following rules apply to *hash tables*.

-- A *hash table* can only associate one value with a given key. If an attempt is made to add a second value for a given key, the second value will replace the first. Thus, adding a value to a *hash table* is a destructive operation; the *hash table* is modified.
-- There are four kinds of *hash tables*: those whose keys are compared with **eq**, those whose keys are compared with **eql**, those whose keys are compared with **equal**, and those whose keys are compared with **equalp**.
-- *Hash tables* are created by **make-hash-table**. **gethash** is used to look up a key and find the associated value. New entries are added to *hash tables* using **setf** with **gethash**. **remhash** is used to remove an entry. For example:

```
(setq a (make-hash-table)) =>  #<HASH-TABLE EQL 0/120 32536573>
(setf (gethash 'color a) 'brown) =>  BROWN
(setf (gethash 'name a) 'fred) =>  FRED
(gethash 'color a) =>  BROWN, true
(gethash 'name a) =>  FRED, true
(gethash 'pointy a) =>  NIL, false
```

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

-- A key or a value may be any *object*.
-- The existence of an entry in the *hash table* can be determined from the *secondary value* returned by **gethash**.

```
clrhash           hash-table-p     remhash
gethash           make-hash-table  sxhash
hash-table-count  maphash
```

Figure 18-1. Hash-table defined names

18.1.2 Modifying Hash Table Keys

The function supplied as the `:test` argument to **make-hash-table** specifies the 'equivalence test' for the *hash table* it creates.

An *object* is 'visibly modified' with regard to an equivalence test if there exists some set of *objects* (or potential *objects*) which are equivalent to the *object* before the modification but are no longer equivalent afterwards.

If an *object* O1 is used as a key in a *hash table* H and is then visibly modified with regard to the equivalence test of H, then the consequences are unspecified if O1, or any *object* O2 equivalent to O1 under the equivalence test (either before or after the modification), is used as a key in further operations on H. The consequences of using O1 as a key are unspecified even if O1 is visibly modified and then later modified again in such a way as to undo the visible modification.

Following are specifications of the modifications which are visible to the equivalence tests which must be supported by *hash tables*. The modifications are described in terms of modification of components, and are defined recursively. Visible modifications of components of the *object* are visible modifications of the *object*.

18.1.2.1 Visible Modification of Objects with respect to EQ and EQL

No *standardized function* is provided that is capable of visibly modifying an *object* with regard to **eq** or **eql**.

18.1.2.2 Visible Modification of Objects with respect to EQUAL

As a consequence of the behavior for **equal**, the rules for visible modification of *objects* not explicitly mentioned in this section are inherited from those in Section 18.1.2.1 (Visible Modification of Objects with respect to EQ and EQL).

18.1.2.2.1 Visible Modification of Conses with respect to EQUAL

Any visible change to the *car* or the *cdr* of a *cons* is considered a visible modification with regard to **equal**.

18.1.2.2.2 Visible Modification of Bit Vectors and Strings with respect to EQUAL

For a *vector* of *type* **bit-vector** or of *type* **string**, any visible change to an *active element* of the *vector*, or to the *length* of the *vector* (if it is *actually adjustable* or has a *fill pointer*) is considered a visible modification with regard to **equal**.

18.1.2.3 Visible Modification of Objects with respect to EQUALP

As a consequence of the behavior for **equalp**, the rules for visible modification of *objects* not explicitly mentioned in this section are inherited from those in Section 18.1.2.2 (Visible Modification of Objects with respect to EQUAL).

18.1.2.3.1 Visible Modification of Structures with respect to EQUALP

Any visible change to a *slot* of a *structure* is considered a visible modification with regard to **equalp**.

18.1.2.3.2 Visible Modification of Arrays with respect to EQUALP

In an *array*, any visible change to an *active element*, to the *fill pointer* (if the *array* can and does have one), or to the *dimensions* (if the *array* is *actually adjustable*) is considered a visible modification with regard to **equalp**.

18.1.2.3.3 Visible Modification of Hash Tables with respect to EQUALP

In a *hash table*, any visible change to the count of entries in the *hash table*, to the keys, or to the values associated with the keys is considered a visible modification with regard to **equalp**.

Note that the visibility of modifications to the keys depends on the equivalence test of the *hash table*, not on the specification of **equalp**.

18.1.2.4 Visible Modifications by Language Extensions

Implementations that extend the language by providing additional mutator functions (or additional behavior for existing mutator functions) must document how the use of these extensions interacts with equivalence tests and *hash table* searches.

Implementations that extend the language by defining additional acceptable equivalence tests for *hash tables* (allowing additional values for the `:test` argument to **make-hash-table**) must document the visible components of these tests.

19. Filenames

19.1 Overview of Filenames

There are many kinds of *file systems*, varying widely both in their superficial syntactic details, and in their underlying power and structure. The facilities provided by Common Lisp for referring to and manipulating *files* has been chosen to be compatible with many kinds of *file systems*, while at the same time minimizing the program-visible differences between kinds of *file systems*.

Since *file systems* vary in their conventions for naming *files*, there are two distinct ways to represent *filenames*: as *namestrings* and as *pathnames*.

19.1.1 Namestrings as Filenames

A *namestring* is a *string* that represents a *filename*.

In general, the syntax of *namestrings* involves the use of *implementation-defined* conventions, usually those customary for the *file system* in which the named *file* resides. The only exception is the syntax of a *logical pathname namestring*, which is defined in this specification; see Section 19.3.1 (Syntax of Logical Pathname Namestrings).

A *conforming program* must never unconditionally use a *literal namestring* other than a *logical pathname namestring* because Common Lisp does not define any *namestring* syntax other than that for *logical pathnames* that would be guaranteed to be portable. However, a *conforming program* can, if it is careful, successfully manipulate user-supplied data which contains or refers to non-portable *namestrings*.

A *namestring* can be *coerced* to a *pathname* by the *functions* **pathname** or **parse-namestring**.

19.1.2 Pathnames as Filenames

Pathnames are structured *objects* that can represent, in an *implementation-independent* way, the *filenames* that are used natively by an underlying *file system*.

In addition, *pathnames* can also represent certain partially composed *filenames* for which an underlying *file system* might not have a specific *namestring* representation.

A *pathname* need not correspond to any file that actually exists, and more than one *pathname* can refer to the same file. For example, the *pathname* with a version of `:newest` might refer to the same file as a *pathname* with the same components except a certain number as the version. Indeed, a *pathname* with version `:newest` might refer to different files as time passes, because the meaning of such a *pathname* depends on the state of the file system.

Some *file systems* naturally use a structural model for their *filenames*, while others do not. Within the Common Lisp *pathname* model, all *filenames* are seen as having a particular structure, even if that structure is not reflected in the underlying *file system*. The nature of the mapping between structure imposed by *pathnames* and the structure, if any, that is used by the underlying *file system* is *implementation-defined*.

Every *pathname* has six components: a host, a device, a directory, a name, a type, and a version. By naming *files* with *pathnames*, Common Lisp programs can work in essentially the same way even in *file systems* that seem superficially quite different. For a detailed description of these components, see Section 19.2.1 (Pathname Components).

The mapping of the *pathname* components into the concepts peculiar to each *file system* is *implementation-defined*. There exist conceivable *pathnames* for which there is no mapping to a syntactically valid *filename* in a particular *implementation*. An *implementation* may use various strategies in an attempt to find a mapping; for example, an *implementation* may quietly truncate *filenames* that exceed length limitations imposed by the underlying *file system*, or ignore certain *pathname* components for which the *file system* provides no support. If such a mapping cannot be found, an error of *type* **file-error** is signaled.

The time at which this mapping and associated error signaling occurs is *implementation-dependent*. Specifically, it may occur at the time the *pathname* is constructed, when coercing a *pathname* to a *namestring*, or when an attempt is made to *open* or otherwise access the *file* designated by the *pathname*.

The next figure lists some *defined names* that are applicable to *pathnames*.

```
*default-pathname-defaults*  namestring          pathname-name
directory-namestring         open               pathname-type
enough-namestring            parse-namestring   pathname-version
file-namestring              pathname           pathnamep
file-string-length           pathname-device    translate-pathname
host-namestring              pathname-directory truename
make-pathname                pathname-host      user-homedir-pathname
merge-pathnames              pathname-match-p   wild-pathname-p
```

19.1.3 Parsing Namestrings Into Pathnames

Parsing is the operation used to convert a *namestring* into a *pathname*. Except in the case of parsing *logical pathname namestrings*, this operation is *implementation-dependent*, because the format of *namestrings* is *implementation-dependent*.

A *conforming implementation* is free to accommodate other *file system* features in its *pathname* representation and provides a parser that can process such specifications in *namestrings*. *Conforming programs* must not depend on any such features, since those features will not be portable.

19.2 Pathnames

19.2.1 Pathname Components

A *pathname* has six components: a host, a device, a directory, a name, a type, and a version.

19.2.1.1 The Pathname Host Component

The name of the file system on which the file resides, or the name of a *logical host*.

19.2.1.2 The Pathname Device Component

Corresponds to the "device" or "file structure" concept in many host file systems: the name of a logical or physical device containing files.

19.2.1.3 The Pathname Directory Component

Corresponds to the "directory" concept in many host file systems: the name of a group of related files.

19.2.1.4 The Pathname Name Component

The "name" part of a group of *files* that can be thought of as conceptually related.

19.2.1.5 The Pathname Type Component

Corresponds to the "filetype" or "extension" concept in many host file systems. This says what kind of file this is. This component is always a *string*, **nil**, `:wild`, or `:unspecific`.

19.2.1.6 The Pathname Version Component

Corresponds to the "version number" concept in many host file systems.

The version is either a positive *integer* or a *symbol* from the following list: **nil**, `:wild`, `:unspecific`, or `:newest` (refers to the largest version number that already exists in the file system when reading a file, or to a version number greater than any already existing in the file system when writing a new file). Implementations can define other special version *symbols*.

19.2.2 Interpreting Pathname Component Values

19.2.2.1 Strings in Component Values

19.2.2.1.1 Special Characters in Pathname Components

Strings in *pathname* component values never contain special *characters* that represent separation between *pathname* fields, such as *slash* in Unix *filenames*. Whether separator *characters* are permitted as part of a *string* in a *pathname* component is *implementation-defined*; however, if the *implementation* does permit it, it must arrange to properly "quote" the character for the *file system* when constructing a *namestring*. For example,

```
 ;; In a TOPS-20 implementation, which uses ^V to quote
 (NAMESTRING (MAKE-PATHNAME :HOST "OZ" :NAME "<TEST>"))
=>  #P"OZ:PS:^V<TEST^V>"
NOT=>  #P"OZ:PS:<TEST>"
```

19.2.2.1.2 Case in Pathname Components

Namestrings always use local file system *case* conventions, but Common Lisp *functions* that manipulate *pathname* components allow the caller to select either of two conventions for representing *case* in component values by supplying a value for the `:case` keyword argument. The next figure lists the functions relating to *pathnames* that permit a `:case` argument:

```
make-pathname    pathname-directory  pathname-name
pathname-device  pathname-host       pathname-type
```

Figure 19-2. Pathname functions using a :CASE argument

19.2.2.1.2.1 Local Case in Pathname Components

For the functions in Figure 19-2, a value of `:local` for the `:case` argument (the default for these functions) indicates that the functions should receive and yield *strings* in component values as if they were already represented according to the host *file system*'s convention for *case*.

If the *file system* supports both *cases*, *strings* given or received as *pathname* component values under this protocol are to be used exactly as written. If the file system only supports one *case*, the *strings* will be translated to that *case*.

19.2.2.1.2.2 Common Case in Pathname Components

For the functions in Figure 19-2, a value of `:common` for the `:case` argument that these *functions* should receive and yield *strings* in component values according to the following conventions:

* All *uppercase* means to use a file system's customary *case*.
* All *lowercase* means to use the opposite of the customary *case*.
* Mixed *case* represents itself.

Note that these conventions have been chosen in such a way that translation from `:local` to `:common` and back to `:local` is information-preserving.

19.2.2.2 Special Pathname Component Values

19.2.2.2.1 NIL as a Component Value

As a *pathname* component value, **nil**represents that the component is "unfilled"; see Section 19.2.3 (Merging Pathnames).

The value of any *pathname* component can be **nil**.

When constructing a *pathname*, **nil** in the host component might mean a default host rather than an actual **nil** in some *implementations*.

19.2.2.2.2 :WILD as a Component Value

If `:wild` is the value of a *pathname* component, that component is considered to be a wildcard, which matches anything.

A *conforming program* must be prepared to encounter a value of `:wild` as the value of any *pathname* component, or as an *element* of a *list* that is the value of the directory component.

When constructing a *pathname*, a *conforming program* may use `:wild` as the value of any or all of the directory, name, type, or version component, but must not use `:wild` as the value of the host, or device component.

If `:wild` is used as the value of the directory component in the construction of a *pathname*, the effect is equivalent to specifying the list `(:absolute :wild-inferiors)`, or the same as `(:absolute :wild)` in a *file system* that does not support `:wild-inferiors`.

19.2.2.2.3 :UNSPECIFIC as a Component Value

If `:unspecific` is the value of a *pathname* component, the component is considered to be "absent" or to "have no meaning" in the *filename* being represented by the *pathname*.

Whether a value of `:unspecific` is permitted for any component on any given *file system* accessible to the *implementation* is *implementation-defined*. A *conforming program* must never unconditionally use a `:unspecific` as the value of a *pathname* component because such a value is not guaranteed to be permissible in all implementations. However, a *conforming program* can, if it is careful, successfully manipulate user-supplied data which contains or refers to non-portable *pathname* components. And certainly a *conforming program* should be prepared for the possibility that any components of a *pathname* could be `:unspecific`.

When *reading*[1] the value of any *pathname* component, *conforming programs* should be prepared for the value to be `:unspecific`.

When *writing*[1] the value of any *pathname* component, the consequences are undefined if `:unspecific` is given for a *pathname* in a *file system* for which it does not make sense.

19.2.2.2.3.1 Relation between component values NIL and :UNSPECIFIC

If a *pathname* is converted to a *namestring*, the *symbols* **nil** and `:unspecific` cause the field to be treated as if it were empty. That is, both **nil** and `:unspecific` cause the component not to appear in the *namestring*.

However, when merging a *pathname* with a set of defaults, only a **nil** value for a component will be replaced with the default for that component, while a value of `:unspecific` will be left alone as if the field were "filled"; see the *function* **merge-pathnames** and Section 19.2.3 (Merging Pathnames).

19.2.2.3 Restrictions on Wildcard Pathnames

Wildcard *pathnames* can be used with **directory** but not with **open**, and return true from **wild-pathname-p**. When examining wildcard components of a wildcard *pathname*, conforming programs must be prepared to encounter any of the following additional values in any component or any element of a *list* that is the directory component:

* The *symbol* `:wild`, which matches anything.
* A *string* containing *implementation-dependent* special wildcard *characters*.
* Any *object*, representing an *implementation-dependent* wildcard pattern.

19.2.2.4 Restrictions on Examining Pathname Components

The space of possible *objects* that a *conforming program* must be prepared to *read*[1] as the value of a *pathname* component is substantially larger than the space of possible *objects* that a *conforming program* is permitted to *write*[1] into such a component.

While the values discussed in the subsections of this section, in Section 19.2.2.2 (Special Pathname Component Values), and in Section 19.2.2.3 (Restrictions on Wildcard Pathnames) apply to values that might be seen when reading the component values, substantially more restrictive rules apply to constructing pathnames; see Section 19.2.2.5 (Restrictions on Constructing Pathnames).

When examining *pathname* components, *conforming programs* should be aware of the following restrictions.

19.2.2.4.1 Restrictions on Examining a Pathname Host Component

It is *implementation-dependent* what *object* is used to represent the host.

19.2.2.4.2 Restrictions on Examining a Pathname Device Component

The device might be a *string*, `:wild`, `:unspecific`, or **nil**.

Note that `:wild` might result from an attempt to *read*[1] the *pathname* component, even though portable programs are restricted from *writing*[1] such a component value; see Section 19.2.2.3 (Restrictions on Wildcard Pathnames) and Section 19.2.2.5 (Restrictions on Constructing Pathnames).

19.2.2.4.3 Restrictions on Examining a Pathname Directory Component

The directory might be a *string*, `:wild`, `:unspecific`, or **nil**.

The directory can be a *list* of *strings* and *symbols*. The *car* of the *list* is one of the symbols `:absolute` or `:relative`, meaning:

`:absolute`
: A *list* whose *car* is the symbol `:absolute` represents a directory path starting from the root directory. The list `(:absolute)` represents the root directory. The list `(:absolute "foo" "bar" "baz")` represents the directory called `"/foo/bar/baz"` in Unix (except possibly for *case*).

`:relative`
: A *list* whose *car* is the symbol `:relative` represents a directory path starting from a default directory. The list `(:relative)` has the same meaning as **nil** and hence is not used. The list `(:relative "foo" "bar")` represents the directory named `"bar"` in the directory named `"foo"` in the default directory.

Each remaining element of the *list* is a *string* or a *symbol*.

Each *string* names a single level of directory structure. The *strings* should contain only the directory names themselves---no punctuation characters.

In place of a *string*, at any point in the *list*, *symbols* can occur to indicate special file notations. The next figure lists the *symbols* that have standard meanings. Implementations are permitted to add additional *objects* of any *type* that is disjoint from **string** if necessary to represent features of their file systems that cannot be represented with the standard *strings* and *symbols*.

Supplying any non-*string*, including any of the *symbols* listed below, to a file system for which it does not make sense signals an error of *type* **file-error**. For example, Unix does not support `:wild-inferiors` in most implementations.

```
Symbol           Meaning
:wild            Wildcard match of one level of directory structure
:wild-inferiors  Wildcard match of any number of directory levels
:up              Go upward in directory structure (semantic)
:back            Go upward in directory structure (syntactic)
```

Figure 19-3. Special Markers In Directory Component

The following notes apply to the previous figure:

Invalid Combinations

Using `:absolute` or `:wild-inferiors` immediately followed by `:up` or `:back` signals an error of *type* **file-error**.

Syntactic vs Semantic

"Syntactic" means that the action of `:back` depends only on the *pathname* and not on the contents of the file system.

"Semantic" means that the action of `:up` depends on the contents of the file system; to resolve a *pathname* containing `:up` to a *pathname* whose directory component contains only `:absolute` and *strings* requires probing the file system.

`:up` differs from `:back` only in file systems that support multiple names for directories, perhaps via symbolic links. For example, suppose that there is a directory `(:absolute "X" "Y" "Z")` linked to `(:absolute "A" "B" "C")` and there also exist directories `(:absolute "A" "B" "Q")` and `(:absolute "X" "Y" "Q")`. Then `(:absolute "X" "Y" "Z" :up "Q")` designates `(:absolute "A" "B" "Q")` while `(:absolute "X" "Y" "Z" :back "Q")` designates `(:absolute "X" "Y" "Q")`

19.2.2.4.3.1 Directory Components in Non-Hierarchical File Systems

In non-hierarchical *file systems*, the only valid *list* values for the directory component of a *pathname* are `(:absolute` *`string`*`)` and `(:absolute :wild)`. `:relative` directories and the keywords `:wild-inferiors`, `:up`, and `:back` are not used in non-hierarchical *file systems*.

19.2.2.4.4 Restrictions on Examining a Pathname Name Component

The name might be a *string*, `:wild`, `:unspecific`, or **nil**.

19.2.2.4.5 Restrictions on Examining a Pathname Type Component

The type might be a *string*, `:wild`, `:unspecific`, or **nil**.

19.2.2.4.6 Restrictions on Examining a Pathname Version Component

The version can be any *symbol* or any *integer*.

The symbol `:newest` refers to the largest version number that already exists in the *file system* when reading, overwriting, appending, superseding, or directory listing an existing *file*. The symbol `:newest` refers to the smallest version number greater than any existing version number when creating a new file.

The symbols **nil**, `:unspecific`, and `:wild` have special meanings and restrictions; see Section 19.2.2.2 (Special Pathname Component Values) and Section 19.2.2.5 (Restrictions on Constructing Pathnames).

Other *symbols* and *integers* have *implementation-defined* meaning.

19.2.2.4.7 Notes about the Pathname Version Component

It is suggested, but not required, that implementations do the following:

* Use positive *integers* starting at 1 as version numbers.
* Recognize the symbol `:oldest` to designate the smallest existing version number.

* Use *keywords* for other special versions.

19.2.2.5 Restrictions on Constructing Pathnames

When constructing a *pathname* from components, conforming programs must follow these rules:

* Any component can be **nil**. **nil** in the host might mean a default host rather than an actual **nil** in some implementations.
* The host, device, directory, name, and type can be *strings*. There are *implementation-dependent* limits on the number and type of *characters* in these *strings*.
* The directory can be a *list* of *strings* and *symbols*. There are *implementation-dependent* limits on the *list*'s length and contents.
* The version can be `:newest`.
* Any component can be taken from the corresponding component of another *pathname*. When the two *pathnames* are for different file systems (in implementations that support multiple file systems), an appropriate translation occurs. If no meaningful translation is possible, an error is signaled. The definitions of "appropriate" and "meaningful" are *implementation-dependent*.
* An implementation might support other values for some components, but a portable program cannot use those values. A conforming program can use *implementation-dependent* values but this can make it non-portable; for example, it might work only with Unix file systems.

19.2.3 Merging Pathnames

Merging takes a *pathname* with unfilled components and supplies values for those components from a source of defaults.

If a component's value is **nil**, that component is considered to be unfilled. If a component's value is any *non-nil object*, including `:unspecific`, that component is considered to be filled.

Except as explicitly specified otherwise, for functions that manipulate or inquire about *files* in the *file system*, the pathname argument to such a function is merged with ***default-pathname-defaults*** before accessing the *file system* (as if by **merge-pathnames**).

19.2.3.1 Examples of Merging Pathnames

Although the following examples are possible to execute only in *implementations* which permit `:unspecific` in the indicated position andwhich permit four-letter type components, they serve to illustrate the basic concept of *pathname* merging.

```
 (pathname-type
   (merge-pathnames (make-pathname :type "LISP")
                    (make-pathname :type "TEXT")))
=>  "LISP"

 (pathname-type
   (merge-pathnames (make-pathname :type nil)
                    (make-pathname :type "LISP")))
=>  "LISP"

 (pathname-type
   (merge-pathnames (make-pathname :type :unspecific)
                    (make-pathname :type "LISP")))
=>  :UNSPECIFIC
```

19.3 Logical Pathnames

19.3.1 Syntax of Logical Pathname Namestrings

The syntax of a *logical pathname namestring* is as follows. (Note that unlike many notational descriptions in this document, this is a syntactic description of character sequences, not a structural description of *objects*.)

```
logical-pathname::= [host host-marker]
                    [relative-directory-marker] {directory directory-marker}*
                    [name] [type-marker type [version-marker version]]

host::= word

directory::= word | wildcard-word | wild-inferiors-word

name::= word | wildcard-word

type::= word | wildcard-word

version::= pos-int | newest-word | wildcard-version
```

host-marker---a *colon*.

relative-directory-marker---a *semicolon*.

directory-marker---a *semicolon*.

type-marker---a *dot*.

version-marker---a *dot*.

wild-inferiors-word---The two character sequence "`**`" (two *asterisks*).

newest-word---The six character sequence "`newest`" or the six character sequence "`NEWEST`".

wildcard-version---an *asterisk*.

wildcard-word---one or more *asterisks*, uppercase letters, digits, and hyphens, including at least one *asterisk*, with no two *asterisks* adjacent.

word---one or more uppercase letters, digits, and hyphens.

pos-int---a positive *integer*.

19.3.1.1 Additional Information about Parsing Logical Pathname Namestrings

19.3.1.1.1 The Host part of a Logical Pathname Namestring

The *host* must have been defined as a *logical pathname* host; this can be done by using **setf** of **logical-pathname-translations**.

The *logical pathname* host name `"SYS"` is reserved for the implementation. The existence and meaning of `SYS:` *logical pathnames* is *implementation-defined*.

19.3.1.1.2 The Device part of a Logical Pathname Namestring

There is no syntax for a *logical pathname* device since the device component of a *logical pathname* is always `:unspecific`; see Section 19.3.2.1 (Unspecific Components of a Logical Pathname).

19.3.1.1.3 The Directory part of a Logical Pathname Namestring

If a *relative-directory-marker* precedes the *directories*, the directory component parsed is as *relative*; otherwise, the directory component is parsed as *absolute*.

If a *wild-inferiors-marker* is specified, it parses into `:wild-inferiors`.

19.3.1.1.4 The Type part of a Logical Pathname Namestring

The *type* of a *logical pathname* for a *source file* is `"LISP"`. This should be translated into whatever type is appropriate in a physical pathname.

19.3.1.1.5 The Version part of a Logical Pathname Namestring

Some *file systems* do not have *versions*. *Logical pathname* translation to such a *file system* ignores the *version*. This implies that a program cannot rely on being able to store more than one version of a file named by a *logical pathname*.

If a *wildcard-version* is specified, it parses into `:wild`.

19.3.1.1.6 Wildcard Words in a Logical Pathname Namestring

Each *asterisk* in a *wildcard-word* matches a sequence of zero or more characters. The *wildcard-word* `"*"` parses into `:wild`; other *wildcard-words* parse into *strings*.

19.3.1.1.7 Lowercase Letters in a Logical Pathname Namestring

When parsing *words* and *wildcard-words*, lowercase letters are translated to uppercase.

19.3.1.1.8 Other Syntax in a Logical Pathname Namestring

The consequences of using characters other than those specified here in a *logical pathname namestring* are unspecified.

The consequences of using any value not specified here as a *logical pathname* component are unspecified.

19.3.2 Logical Pathname Components

19.3.2.1 Unspecific Components of a Logical Pathname

The device component of a *logical pathname* is always `:unspecific`; no other component of a *logical pathname* can be `:unspecific`.

19.3.2.2 Null Strings as Components of a Logical Pathname

The null string, "", is not a valid value for any component of a *logical pathname*.

20. Files

20.1 File System Concepts

This section describes the Common Lisp interface to file systems. The model used by this interface assumes that *files* are named by *filenames*, that a *filename* can be represented by a *pathname object*, and that given a *pathname* a *stream* can be constructed that connects to a *file* whose *filename* it represents.

For information about opening and closing *files*, and manipulating their contents, see Section 21 (Streams).

The next figure lists some *operators* that are applicable to *files* and directories.

```
compile-file  file-length      open
delete-file   file-position    probe-file
directory     file-write-date  rename-file
file-author   load             with-open-file
```

Figure 20-1. File and Directory Operations

20.1.1 Coercion of Streams to Pathnames

A *stream associated with a file* is either a *file stream* or a *synonym stream* whose target is a *stream associated with a file*. Such streams can be used as *pathname designators*.

Normally, when a *stream associated with a file* is used as a *pathname designator*, it denotes the *pathname* used to open the *file*; this may be, but is not required to be, the actual name of the *file*.

Some functions, such as **truename** and **delete-file**, coerce *streams* to *pathnames* in a different way that involves referring to the actual *file* that is open, which might or might not be the file whose name was opened originally. Such special situations are always notated specifically and are not the default.

20.1.2 File Operations on Open and Closed Streams

Many *functions* that perform *file* operations accept either *open* or *closed streams* as *arguments*; see Section 21.1.3 (Stream Arguments to Standardized Functions).

Of these, the *functions* in the next figure treat *open* and *closed streams* differently.

```
delete-file  file-author      probe-file
directory    file-write-date  truename
```

Figure 20-2. File Functions that Treat Open and Closed Streams Differently

Since treatment of *open streams* by the *file system* may vary considerably between *implementations*, however, a *closed stream* might be the most reliable kind of *argument* for some of these functions---in particular, those in the next figure. For example, in some *file systems*, *open files* are written under temporary names and not renamed until *closed* and/or are held invisible until *closed*. In general, any code that is intended to be portable should use such *functions* carefully.

```
directory  probe-file  truename
```

Figure 20-3. File Functions where Closed Streams Might Work Best

20.1.3 Truenames

Many *file systems* permit more than one *filename* to designate a particular *file*.

Even where multiple names are possible, most *file systems* have a convention for generating a canonical *filename* in such situations. Such a canonical *filename* (or the *pathname* representing such a *filename*) is called a *truename*.

The *truename* of a *file* may differ from other *filenames* for the file because of symbolic links, version numbers, logical device translations in the *file system*, *logical pathname* translations within Common Lisp, or other artifacts of the *file system*.

The *truename* for a *file* is often, but not necessarily, unique for each *file*. For instance, a Unix *file* with multiple hard links could have several *truenames*.

20.1.3.1 Examples of Truenames

For example, a DEC TOPS-20 system with *files* `PS:<JOE>FOO.TXT.1` and `PS:<JOE>FOO.TXT.2` might permit the second *file* to be referred to as `PS:<JOE>FOO.TXT.0`, since the "`.0`" notation denotes "newest" version of several *files*. In the same *file system*, a "logical device" "`JOE:`" might be taken to refer to `PS:<JOE>`" and so the names `JOE:FOO.TXT.2` or `JOE:FOO.TXT.0` might refer to `PS:<JOE>FOO.TXT.2`. In all of these cases, the *truename* of the file would probably be `PS:<JOE>FOO.TXT.2`.

If a *file* is a symbolic link to another *file* (in a *file system* permitting such a thing), it is conventional for the *truename* to be the canonical name of the *file* after any symbolic links have been followed; that is, it is the canonical name of the *file* whose contents would become available if an *input stream* to that *file* were opened.

In the case of a *file* still being created (that is, of an *output stream* open to such a *file*), the exact *truename* of the file might not be known until the *stream* is closed. In this case, the *function* **truename** might return different values for such a *stream* before and after it was closed. In fact, before it is closed, the name returned might not even be a valid name in the *file system*---for example, while a file is being written, it might have version `:newest` and might only take on a specific numeric value later when the file is closed even in a *file system* where all files have numeric versions.

21. Streams

21.1 Stream Concepts

21.1.1 Introduction to Streams

A *stream* is an *object* that can be used with an input or output function to identify an appropriate source or sink of *characters* or *bytes* for that operation. A *character stream* is a source or sink of *characters*. A *binary stream* is a source or sink of *bytes*.

Some operations may be performed on any kind of *stream*; the next figure provides a list of *standardized* operations that are potentially useful with any kind of *stream*.

```
close                 stream-element-type
input-stream-p        streamp
interactive-stream-p  with-open-stream
output-stream-p
```

Figure 21-1. Some General-Purpose Stream Operations

Other operations are only meaningful on certain *stream types*. For example, **read-char** is only defined for *character streams* and **read-byte** is only defined for *binary streams*.

21.1.1.1 Abstract Classifications of Streams

21.1.1.1.1 Input, Output, and Bidirectional Streams

A *stream*, whether a *character stream* or a *binary stream*, can be an *input stream* (source of data), an *output stream* (sink for data), both, or (e.g., when "`:direction :probe`" is given to **open**) neither.

The next figure shows *operators* relating to *input streams*.

```
clear-input  read-byte            read-from-string
listen       read-char            read-line
peek-char    read-char-no-hang    read-preserving-whitespace
read         read-delimited-list  unread-char
```

Figure 21-2. Operators relating to Input Streams.

The next figure shows *operators* relating to *output streams*.

```
clear-output   prin1            write
finish-output  prin1-to-string  write-byte
force-output   princ            write-char
format         princ-to-string  write-line
fresh-line     print            write-string
pprint         terpri           write-to-string
```

Figure 21-3. Operators relating to Output Streams.

A *stream* that is both an *input stream* and an *output stream* is called a *bidirectional stream*. See the *functions* **input-stream-p** and **output-stream-p**.

Any of the *operators* listed in Figure 21-2 or Figure 21-3 an be used with *bidirectional streams*. In addition, the next figure hows a list of *operators* that relate specificaly to *bidirectional streams*.

```
y-or-n-p  yes-or-no-p
```

Figure 21-4. Operators relating to Bidirectional Streams.

21.1.1.1.2 Open and Closed Streams

Streams are either *open* or *closed*.

Except as explicitly specified otherwise, operations that create and return *streams* return *open streams*.

The action of *closing* a *stream* marks the end of its use as a source or sink of data, permitting the *implementation* to reclaim its internal data structures, and to free any external resources which might have been locked by the *stream* when it was opened.

Except as explicitly specified otherwise, the consequences are undefined when a *closed stream* is used where a *stream* is called for.

Coercion of *streams* to *pathnames* is permissible for *closed streams*; in some situations, such as for a *truename* computation, the result might be different for an *open stream* and for that same *stream* once it has been *closed*.

21.1.1.1.3 Interactive Streams

An *interactive stream* is one on which it makes sense to perform interactive querying.

The precise meaning of an *interactive stream* is *implementation-defined*, and may depend on the underlying operating system. Some examples of the things that an *implementation* might choose to use as identifying characteristics of an *interactive stream* include:

* The *stream* is connected to a person (or equivalent) in such a way that the program can prompt for information and expect to receive different input depending on the prompt.
* The program is expected to prompt for input and support "normal input editing".
* **read-char** might wait for the user to type something before returning instead of immediately returning a character or end-of-file.

The general intent of having some *streams* be classified as *interactive streams* is to allow them to be distinguished from streams containing batch (or background or command-file) input. Output to batch streams is typically discarded or saved for later viewing, so interactive queries to such streams might not have the expected effect.

Terminal I/O might or might not be an *interactive stream*.

21.1.1.2 Abstract Classifications of Streams

21.1.1.2.1 File Streams

Some *streams*, called *file streams*, provide access to *files*. An *object* of *class* **file-stream** is used to represent a *file stream*.

The basic operation for opening a *file* is **open**, which typically returns a *file stream* (see its dictionary entry for details). The basic operation for closing a *stream* is **close**. The macro **with-open-file** is useful to express the common idiom of opening a *file* for the duration of a given body of *code*, and assuring that the resulting *stream* is closed upon exit from that body.

21.1.1.3 Other Subclasses of Stream

The *class* **stream** has a number of *subclasses* defined by this specification. The next figure shows some information about these subclasses.

```
Class                Related Operators
broadcast-stream     make-broadcast-stream
                     broadcast-stream-streams
concatenated-stream  make-concatenated-stream
                     concatenated-stream-streams
echo-stream          make-echo-stream
                     echo-stream-input-stream
                     echo-stream-output-stream
string-stream        make-string-input-stream
                     with-input-from-string
                     make-string-output-stream
                     with-output-to-string
```

```
                    get-output-stream-string
synonym-stream      make-synonym-stream
                    synonym-stream-symbol
two-way-stream      make-two-way-stream
                    two-way-stream-input-stream
                    two-way-stream-output-stream
```

Figure 21-5. Defined Names related to Specialized Streams

21.1.2 Stream Variables

Variables whose *values* must be *streams* are sometimes called *stream variables*.

Certain *stream variables* are defined by this specification to be the proper source of input or output in various *situations* where no specific *stream* has been specified instead. A complete list of such *standardized stream variables* appears in the next figure. The consequences are undefined if at any time the *value* of any of these *variables* is not an *open stream*.

```
Glossary Term    Variable Name
debug I/O        *debug-io*
error output     *error-output*
query I/O        *query-io*
standard input   *standard-input*
standard output  *standard-output*
terminal I/O     *terminal-io*
trace output     *trace-output*
```

Figure 21-6. Standardized Stream Variables

Note that, by convention, *standardized stream variables* have names ending in "`-input*`" if they must be *input streams*, ending in "`-output*`" if they must be *output streams*, or ending in "`-io*`" if they must be *bidirectional streams*.

User programs may *assign* or *bind* any *standardized stream variable* except ***terminal-io***.

21.1.3 Stream Arguments to Standardized Functions

The *operators* in the next figure accept *stream arguments* that might be either *open* or *closed streams*.

```
broadcast-stream-streams     file-author       pathnamep
close                        file-namestring   probe-file
compile-file                 file-write-date   rename-file
compile-file-pathname        host-namestring   streamp
concatenated-stream-streams  load              synonym-stream-symbol
delete-file                  logical-pathname  translate-logical-pathname
directory                    merge-pathnames   translate-pathname
directory-namestring         namestring        truename
dribble                      open              two-way-stream-input-stream
echo-stream-input-stream     open-stream-p     two-way-stream-output-stream
echo-stream-ouput-stream     parse-namestring  wild-pathname-p
ed                           pathname          with-open-file
enough-namestring            pathname-match-p
```

Figure 21-7. Operators that accept either Open or Closed Streams

The *operators* in the next figure accept *stream arguments* that must be *open streams*.

```
clear-input                output-stream-p          read-char-no-hang
clear-output               peek-char                read-delimited-list
file-length                pprint                   read-line
file-position              pprint-fill              read-preserving-whitespace
file-string-length         pprint-indent            stream-element-type
finish-output              pprint-linear            stream-external-format
force-output               pprint-logical-block     terpri
format                     pprint-newline           unread-char
fresh-line                 pprint-tab               with-open-stream
get-output-stream-string   pprint-tabular           write
input-stream-p             prin1                    write-byte
interactive-stream-p       princ                    write-char
listen                     print                    write-line
make-broadcast-stream      print-object             write-string
make-concatenated-stream   print-unreadable-object  y-or-n-p
make-echo-stream           read                     yes-or-no-p
make-synonym-stream        read-byte
make-two-way-stream        read-char
```

Figure 21-8. Operators that accept Open Streams only

21.1.4 Restrictions on Composite Streams

The consequences are undefined if any *component* of a *composite stream* is *closed* before the *composite stream* is *closed*.

The consequences are undefined if the *synonym stream symbol* is not *bound* to an *open stream* from the time of the *synonym stream*'s creation until the time it is *closed*.

22. Printer

22.1 The Lisp Printer

22.1.1 Overview of The Lisp Printer

Common Lisp provides a representation of most *objects* in the form of printed text called the printed representation. Functions such as **print** take an *object* and send the characters of its printed representation to a *stream*. The collection of routines that does this is known as the (Common Lisp) printer.

Reading a printed representation typically produces an *object* that is **equal** to the originally printed *object*.

22.1.1.1 Multiple Possible Textual Representations

Most *objects* have more than one possible textual representation. For example, the positive *integer* with a magnitude of twenty-seven can be textually expressed in any of these ways:

```
 27    27.    #o33    #x1B    #b11011    #.(* 3 3 3)    81/3
```

A list containing the two symbols A and B can also be textually expressed in a variety of ways:

```
 (A B)    (a b)    (  a  b )    (\A |B|)
(|\A|
  B
)
```

In general, from the point of view of the *Lisp reader*, wherever *whitespace* is permissible in a textual representation, any number of *spaces* and *newlines* can appear in *standard syntax*.

When a function such as **print** produces a printed representation, it must choose from among many possible textual representations. In most cases, it chooses a program readable representation, but in certain cases it might use a more compact notation that is not program-readable.

A number of option variables, called *printer control variables*, are provided to permit control of individual aspects of the printed representation of *objects*. The next figure shows the *standardized printer control variables*; there might also be *implementation-defined printer control variables*.

```
*print-array*   *print-gensym*       *print-pprint-dispatch*
*print-base*    *print-length*       *print-pretty*
*print-case*    *print-level*        *print-radix*
*print-circle*  *print-lines*        *print-readably*
*print-escape*  *print-miser-width*  *print-right-margin*
```

Figure 22-1. Standardized Printer Control Variables

In addition to the *printer control variables*, the following additional *defined names* relate to or affect the behavior of the *Lisp printer*:

```
*package*                    *read-eval*  readtable-case
*read-default-float-format*  *readtable*
```

Figure 22-2. Additional Influences on the Lisp printer.

22.1.1.1.1 Printer Escaping

The *variable* ***print-escape*** controls whether the *Lisp printer* tries to produce notations such as escape characters and package prefixes.

The *variable* ***print-readably*** can be used to override many of the individual aspects controlled by the other *printer control variables* when program-readable output is especially important.

One of the many effects of making the *value* of ***print-readably*** be *true* is that the *Lisp printer* behaves as if ***print-escape*** were also *true*. For notational convenience, we say that if the value of either ***print-readably*** or ***print-escape*** is *true*, then *printer escaping* is "enabled"; and we say that if the values of both ***print-readably*** and ***print-escape*** are *false*, then *printer escaping* is "disabled".

22.1.2 Printer Dispatching

The *Lisp printer* makes its determination of how to print an *object* as follows:

If the *value* of ***print-pretty*** is *true*, printing is controlled by the *current pprint dispatch table*; see Section 22.2.1.4 (Pretty Print Dispatch Tables).

Otherwise (if the *value* of ***print-pretty*** is *false*), the object's **print-object** method is used; see Section 22.1.3 (Default Print-Object Methods).

22.1.3 Default Print-Object Methods

This section describes the default behavior of **print-object** methods for the *standardized types*.

22.1.3.1 Printing Numbers

22.1.3.1.1 Printing Integers

Integers are printed in the radix specified by the *current output base* in positional notation, most significant digit first. If appropriate, a radix specifier can be printed; see ***print-radix***. If an *integer* is negative, a minus sign is printed and then the absolute value of the *integer* is printed. The *integer* zero is represented by the single digit `0` and never has a sign. A decimal point might be printed, depending on the *value* of ***print-radix***.

For related information about the syntax of an *integer*, see Section 2.3.2.1.1 (Syntax of an Integer).

22.1.3.1.2 Printing Ratios

Ratios are printed as follows: the absolute value of the numerator is printed, as for an *integer*; then a `/`; then the denominator. The numerator and denominator are both printed in the radix specified by the *current output base*; they are obtained as if by **numerator** and **denominator**, and so *ratios* are printed in reduced form (lowest terms). If appropriate, a radix specifier can be printed; see ***print-radix***. If the ratio is negative, a minus sign is printed before the numerator.

For related information about the syntax of a *ratio*, see Section 2.3.2.1.2 (Syntax of a Ratio).

22.1.3.1.3 Printing Floats

If the magnitude of the *float* is either zero or between 10^-3 (inclusive) and 10^7 (exclusive), it is printed as the integer part of the number, then a decimal point, followed by the fractional part of the number; there is always at least one digit on each side of the decimal point. If the sign of the number (as determined by **float-sign**) is negative, then a minus sign is printed before the number. If the format of the number does not match that specified by ***read-default-float-format***, then the *exponent marker* for that format and the digit `0` are also printed. For example, the base of the natural logarithms as a *short float* might be printed as `2.71828S0`.

For non-zero magnitudes outside of the range 10^-3 to 10^7, a *float* is printed in computerized scientific notation. The representation of the number is scaled to be between 1 (inclusive) and 10 (exclusive) and then printed, with one digit before the decimal point and at least one digit after the decimal point. Next the *exponent marker* for the format is printed, except that if the format of the number matches that specified by ***read-default-float-format***, then the *exponent marker* `E` is used. Finally, the power of ten by which the fraction must be multiplied to equal the original number is printed as a decimal integer. For example, Avogadro's number as a *short float* is printed as `6.02S23`.

For related information about the syntax of a *float*, see Section 2.3.2.2 (Syntax of a Float).

22.1.3.1.4 Printing Complexes

A *complex* is printed as `#C`, an open parenthesis, the printed representation of its real part, a space, the printed representation of its imaginary part, and finally a close parenthesis.

For related information about the syntax of a *complex*, see Section 2.3.2.3 (Syntax of a Complex) and Section 2.4.8.11 (Sharpsign C).

22.1.3.1.5 Note about Printing Numbers

The printed representation of a number must not contain *escape characters*; see Section 2.3.1.1.1 (Escape Characters and Potential Numbers).

22.1.3.2 Printing Characters

When *printer escaping* is disabled, a *character* prints as itself; it is sent directly to the output *stream*. When *printer escaping* is enabled, then #\ syntax is used.

When the printer types out the name of a *character*, it uses the same table as the #\ *reader macro* would use; therefore any *character* name that is typed out is acceptable as input (in that *implementation*). If a *non-graphic character* has a *standardized name*[5], that *name* is preferred over non-standard *names* for printing in #\ notation. For the *graphic standard characters*, the *character* itself is always used for printing in #\ notation---even if the *character* also has a *name*[5].

For details about the #\ *reader macro*, see Section 2.4.8.1 (Sharpsign Backslash).

22.1.3.3 Printing Symbols

When *printer escaping* is disabled, only the characters of the *symbol*'s *name* are output (but the case in which to print characters in the *name* is controlled by ***print-case***; see Section 22.1.3.3.2 (Effect of Readtable Case on the Lisp Printer)).

The remainder of Section 22.1.3.3 applies only when *printer escaping* is enabled.

When printing a *symbol*, the printer inserts enough *single escape* and/or *multiple escape* characters (*backslashes* and/or *vertical-bars*) so that if **read** were called with the same ***readtable*** and with ***read-base*** bound to the *current output base*, it would return the same *symbol* (if it is not *apparently uninterned*) or an *uninterned symbol* with the same *print name* (otherwise).

For example, if the *value* of ***print-base*** were `16` when printing the symbol `face`, it would have to be printed as `\FACE` or `\face` or `|FACE|`, because the token `face` would be read as a hexadecimal number (decimal value 64206) if the *value* of ***read-base*** were `16`.

For additional restrictions concerning characters with nonstandard *syntax types* in the *current readtable*, see the *variable* ***print-readably***

For information about how the *Lisp reader* parses *symbols*, see Section 2.3.4 (Symbols as Tokens) and Section 2.4.8.5 (Sharpsign Colon).

nil might be printed as `()` when ***print-pretty*** is *true* and *printer escaping* is enabled.

22.1.3.3.1 Package Prefixes for Symbols

Package prefixes are printed if necessary. The rules for *package prefixes* are as follows. When the *symbol* is printed, if it is in the `KEYWORD` package, then it is printed with a preceding *colon*; otherwise, if it is *accessible* in the *current package*, it is printed without any *package prefix*; otherwise, it is printed with a *package prefix*.

A *symbol* that is *apparently uninterned* is printed preceded by "`#:`" if ***print-gensym*** is *true* and *printer escaping* is enabled; if ***print-gensym*** is *false* or *printer escaping* is disabled, then the *symbol* is printed without a prefix, as if it were in the *current package*.

Because the `#:` syntax does not intern the following symbol, it is necessary to use circular-list syntax if ***print-circle*** is *true* and the same uninterned symbol appears several times in an expression to be printed. For example, the result of

```
(let ((x (make-symbol "FOO"))) (list x x))
```

would be printed as `(#:foo #:foo)` if ***print-circle*** were *false*, but as `(#1=#:foo #1#)` if ***print-circle*** were *true*.

A summary of the preceding package prefix rules follows:

`foo:bar`
: `foo:bar` is printed when *symbol* `bar` is external in its *home package* `foo` and is not *accessible* in the *current package*.

`foo::bar`
: `foo::bar` is printed when `bar` is internal in its *home package* `foo` and is not *accessible* in the *current package*.

`:bar`
: `:bar` is printed when the home package of `bar` is the `KEYWORD` package.

`#:bar`
: `#:bar` is printed when `bar` is *apparently uninterned*, even in the pathological case that `bar` has no *home package* but is nevertheless somehow *accessible* in the *current package*.

22.1.3.3.2 Effect of Readtable Case on the Lisp Printer

When *printer escaping* is disabled, or the characters under consideration are not already quoted specifically by *single escape* or *multiple escape* syntax, the *readtable case* of the *current readtable* affects the way the *Lisp printer* writes *symbols* in the following ways:

`:upcase`
: When the *readtable case* is `:upcase`, *uppercase characters* are printed in the case specified by ***print-case***, and *lowercase characters* are printed in their own case.

`:downcase`
: When the *readtable case* is `:downcase`, *uppercase characters* are printed in their own case, and *lowercase characters* are printed in the case specified by ***print-case***.

`:preserve`
: When the *readtable case* is `:preserve`, all *alphabetic characters* are printed in their own case.

`:invert`
: When the *readtable case* is `:invert`, the case of all *alphabetic characters* in single case symbol names is inverted. Mixed-case symbol names are printed as is.

The rules for escaping *alphabetic characters* in symbol names are affected by the **readtable-case** if *printer escaping* is enabled. *Alphabetic characters* are escaped as follows:

`:upcase`
: When the *readtable case* is `:upcase`, all *lowercase characters* must be escaped.

`:downcase`
: When the *readtable case* is `:downcase`, all *uppercase characters* must be escaped.

`:preserve`
: When the *readtable case* is `:preserve`, no *alphabetic characters* need be escaped.

`:invert`
: When the *readtable case* is `:invert`, no *alphabetic characters* need be escaped.

22.1.3.3.2.1 Examples of Effect of Readtable Case on the Lisp Printer

```
(defun test-readtable-case-printing ()
  (let ((*readtable* (copy-readtable nil))
        (*print-case* *print-case*))
    (format t "READTABLE-CASE *PRINT-CASE*  Symbol-name  Output~
             ~%--------------------------------------------------~
             ~%")
    (dolist (readtable-case '(:upcase :downcase :preserve :invert))
      (setf (readtable-case *readtable*) readtable-case)
      (dolist (print-case '(:upcase :downcase :capitalize))
        (dolist (symbol '(|ZEBRA| |Zebra| |zebra|))
          (setq *print-case* print-case)
          (format t "~&:~A~15T:~A~29T~A~42T~A"
                  (string-upcase readtable-case)
                  (string-upcase print-case)
                  (symbol-name symbol)
                  (prin1-to-string symbol)))))))
```

The output from `(test-readtable-case-printing)` should be as follows:

```
READTABLE-CASE *PRINT-CASE*  Symbol-name  Output
--------------------------------------------------
:UPCASE        :UPCASE       ZEBRA        ZEBRA
:UPCASE        :UPCASE       Zebra        |Zebra|
:UPCASE        :UPCASE       zebra        |zebra|
:UPCASE        :DOWNCASE     ZEBRA        zebra
:UPCASE        :DOWNCASE     Zebra        |Zebra|
:UPCASE        :DOWNCASE     zebra        |zebra|
:UPCASE        :CAPITALIZE   ZEBRA        Zebra
:UPCASE        :CAPITALIZE   Zebra        |Zebra|
:UPCASE        :CAPITALIZE   zebra        |zebra|
:DOWNCASE      :UPCASE       ZEBRA        |ZEBRA|
:DOWNCASE      :UPCASE       Zebra        |Zebra|
:DOWNCASE      :UPCASE       zebra        ZEBRA
:DOWNCASE      :DOWNCASE     ZEBRA        |ZEBRA|
:DOWNCASE      :DOWNCASE     Zebra        |Zebra|
:DOWNCASE      :DOWNCASE     zebra        zebra
:DOWNCASE      :CAPITALIZE   ZEBRA        |ZEBRA|
:DOWNCASE      :CAPITALIZE   Zebra        |Zebra|
:DOWNCASE      :CAPITALIZE   zebra        Zebra
:PRESERVE      :UPCASE       ZEBRA        ZEBRA
:PRESERVE      :UPCASE       Zebra        Zebra
:PRESERVE      :UPCASE       zebra        zebra
:PRESERVE      :DOWNCASE     ZEBRA        ZEBRA
:PRESERVE      :DOWNCASE     Zebra        Zebra
:PRESERVE      :DOWNCASE     zebra        zebra
:PRESERVE      :CAPITALIZE   ZEBRA        ZEBRA
:PRESERVE      :CAPITALIZE   Zebra        Zebra
:PRESERVE      :CAPITALIZE   zebra        zebra
:INVERT        :UPCASE       ZEBRA        zebra
:INVERT        :UPCASE       Zebra        Zebra
:INVERT        :UPCASE       zebra        ZEBRA
:INVERT        :DOWNCASE     ZEBRA        zebra
:INVERT        :DOWNCASE     Zebra        Zebra
:INVERT        :DOWNCASE     zebra        ZEBRA
:INVERT        :CAPITALIZE   ZEBRA        zebra
:INVERT        :CAPITALIZE   Zebra        Zebra
:INVERT        :CAPITALIZE   zebra        ZEBRA
```

22.1.3.4 Printing Strings

The characters of the *string* are output in order. If *printer escaping* is enabled, a *double-quote* is output before and after, and all *double-quotes* and *single escapes* are preceded by *backslash*. The printing of *strings* is not affected by ***print-array***. Only the *active elements* of the *string* are printed.

For information on how the *Lisp reader* parses *strings*, see Section 2.4.5 (Double-Quote).

22.1.3.5 Printing Lists and Conses

Wherever possible, list notation is preferred over dot notation. Therefore the following algorithm is used to print a *cons* x:

1. A *left-parenthesis* is printed.
2. The *car* of x is printed.
3. If the *cdr* of x is itself a *cons*, it is made to be the current *cons* (i.e., x becomes that *cons*), a *space* is printed, and step 2 is re-entered.
4. If the *cdr* of x is not *null*, a *space*, a *dot*, a *space*, and the *cdr* of x are printed.
5. A *right-parenthesis* is printed.

Actually, the above algorithm is only used when ***print-pretty*** is *false*. When ***print-pretty*** is *true* (or when **pprint** is used), additional *whitespace*[1] may replace the use of a single *space*, and a more elaborate algorithm with similar goals but more presentational flexibility is used; see Section 22.1.2 (Printer Dispatching).

Although the two expressions below are equivalent, and the reader accepts either one and produces the same *cons*, the printer always prints such a *cons* in the second form.

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

The printing of *conses* is affected by ***print-level***, ***print-length***, and ***print-circle***.

Following are examples of printed representations of *lists*:

```
(a . b)     ;A dotted pair of a and b
(a.b)       ;A list of one element, the symbol named a.b
(a. b)      ;A list of two elements a. and b
(a .b)      ;A list of two elements a and .b
(a b . c)   ;A dotted list of a and b with c at the end; two conses
.iot        ;The symbol whose name is .iot
(. b)       ;Invalid -- an error is signaled if an attempt is made to read
            ;this syntax.
(a .)       ;Invalid -- an error is signaled.
(a .. b)    ;Invalid -- an error is signaled.
(a . . b)   ;Invalid -- an error is signaled.
(a b c ...) ;Invalid -- an error is signaled.
(a \. b)    ;A list of three elements a, ., and b
(a |.| b)   ;A list of three elements a, ., and b
(a \... b)  ;A list of three elements a, ..., and b
(a |...| b) ;A list of three elements a, ..., and b
```

For information on how the *Lisp reader* parses *lists* and *conses*, see Section 2.4.1 (Left-Parenthesis).

22.1.3.6 Printing Bit Vectors

A *bit vector* is printed as `#*` followed by the bits of the *bit vector* in order. If ***print-array*** is *false*, then the *bit vector* is printed in a format (using #<) that is concise but not readable. Only the *active elements* of the *bit vector* are printed.

For information on *Lisp reader* parsing of *bit vectors*, see Section 2.4.8.4 (Sharpsign Asterisk).

22.1.3.7 Printing Other Vectors

If ***print-array*** is *true* and ***print-readably*** is *false*, any *vector* other than a *string* or *bit vector* is printed using general-vector syntax; this means that information about specialized vector representations does not appear. The printed representation of a zero-length *vector* is `#()`. The printed representation of a non-zero-length *vector* begins with `#(`. Following that, the first element of the *vector* is printed. If there are any other elements, they are printed in turn, with each such additional element preceded by a *space* if ***print-pretty*** is *false*, or *whitespace*[1] if ***print-pretty*** is *true*. A *right-parenthesis* after the last element terminates the printed representation of the *vector*. The printing of *vectors* is affected by ***print-level*** and ***print-length***. If the *vector* has a *fill pointer*, then only those elements below the *fill pointer* are printed.

If both ***print-array*** and ***print-readably*** are *false*, the *vector* is not printed as described above, but in a format (using #<) that is concise but not readable.

If ***print-readably*** is *true*, the *vector* prints in an *implementation-defined* manner; see the *variable* ***print-readably***.

For information on how the *Lisp reader* parses these "other *vectors*," see Section 2.4.8.3 (Sharpsign Left-Parenthesis).

22.1.3.8 Printing Other Arrays

If ***print-array*** is *true* and ***print-readably*** is *false*, any *array* other than a *vector* is printed using `#nA` format. Let `n` be the *rank* of the *array*. Then # is printed, then `n` as a decimal integer, then `A`, then `n` open parentheses. Next the *elements* are scanned in row-major order, using **write** on each *element*, and separating *elements* from each other with *whitespace*[1]. The array's dimensions are numbered 0 to `n`-1 from left to right, and are enumerated with the rightmost index changing fastest. Every time the index for dimension `j` is incremented, the following actions are taken:

* If `j` < `n`-1, then a close parenthesis is printed.
* If incrementing the index for dimension `j` caused it to equal dimension `j`, that index is reset to zero and the index for dimension `j`-1 is incremented (thereby performing these three steps recursively), unless `j`=0, in which case the entire algorithm is terminated. If incrementing the index for dimension `j` did not cause it to equal dimension `j`, then a space is printed.
* If `j` < `n`-1, then an open parenthesis is printed.

This causes the contents to be printed in a format suitable for `:initial-contents` to **make-array**. The lists effectively printed by this procedure are subject to truncation by ***print-level*** and ***print-length***.

If the *array* is of a specialized *type*, containing bits or characters, then the innermost lists generated by the algorithm given above can instead be printed using bit-vector or string syntax, provided that these innermost lists would not be subject to truncation by ***print-length***.

If both ***print-array*** and ***print-readably*** are *false*, then the *array* is printed in a format (using #<) that is concise but not readable.

If ***print-readably*** is *true*, the *array* prints in an *implementation-defined* manner; see the *variable* ***print-readably***. In particular, this may be important for arrays having some dimension `0`.

For information on how the *Lisp reader* parses these "other *arrays*," see Section 2.4.8.12 (Sharpsign A).

22.1.3.9 Examples of Printing Arrays

```
 (let ((a (make-array '(3 3)))
       (*print-pretty* t)
       (*print-array* t))
   (dotimes (i 3) (dotimes (j 3) (setf (aref a i j) (format nil "<~D,~D>" i j))))
   (print a)
   (print (make-array 9 :displaced-to a)))
>>  #2A(("<0,0>" "<0,1>" "<0,2>")
>>      ("<1,0>" "<1,1>" "<1,2>")
>>      ("<2,0>" "<2,1>" "<2,2>"))
>>  #("<0,0>" "<0,1>" "<0,2>" "<1,0>" "<1,1>" "<1,2>" "<2,0>" "<2,1>" "<2,2>")
=>  #<ARRAY 9 indirect 36363476>
```

22.1.3.10 Printing Random States

A specific syntax for printing *objects* of *type* **random-state** is not specified. However, every *implementation* must arrange to print a *random state object* in such a way that, within the same implementation, **read** can construct from the printed representation a copy of the *random state* object as if the copy had been made by **make-random-state**.

If the type *random state* is effectively implemented by using the machinery for **defstruct**, the usual structure syntax can then be used for printing *random state* objects; one might look something like

```
 #S(RANDOM-STATE :DATA #(14 49 98436589 786345 8734658324 ... ))
```

where the components are *implementation-dependent*.

22.1.3.11 Printing Pathnames

When *printer escaping* is enabled, the syntax `#P"..."` is how a *pathname* is printed by **write** and the other functions herein described. The `"..."` is the namestring representation of the pathname.

When *printer escaping* is disabled, **write** writes a *pathname P* by writing `(namestring` *P*`)` instead.

For information on how the *Lisp reader* parses *pathnames*, see Section 2.4.8.14 (Sharpsign P).

22.1.3.12 Printing Structures

By default, a *structure* of type S is printed using `#S` syntax. This behavior can be customized by specifying a `:print-function` or `:print-object` option to the **defstruct** *form* that defines S, or by writing a **print-object** *method* that is *specialized* for *objects* of type S.

Different structures might print out in different ways; the default notation for structures is:

```
 #S(structure-name {slot-key slot-value}*)
```

where `#S` indicates structure syntax, *structure-name* is a *structure name*, each *slot-key* is an initialization argument *name* for a *slot* in the *structure*, and each corresponding *slot-value* is a representation of the *object* in that *slot*.

For information on how the *Lisp reader* parses *structures*, see Section 2.4.8.13 (Sharpsign S).

22.1.3.13 Printing Other Objects

Other *objects* are printed in an *implementation-dependent* manner. It is not required that an *implementation* print those *objects readably*.

For example, *hash tables*, *readtables*, *packages*, *streams*, and *functions* might not print *readably*.

A common notation to use in this circumstance is `#<...>`. Since `#<` is not readable by the *Lisp reader*, the precise format of the text which follows is not important, but a common format to use is that provided by the **print-unreadable-object** *macro*.

For information on how the *Lisp reader* treats this notation, see Section 2.4.8.20 (Sharpsign Less-Than-Sign). For information on how to notate *objects* that cannot be printed *readably*, see Section 2.4.8.6 (Sharpsign Dot).

22.1.4 Examples of Printer Behavior

```
 (let ((*print-escape* t)) (fresh-line) (write #\a))
>>  #\a
=>  #\a
 (let ((*print-escape* nil) (*print-readably* nil))
   (fresh-line)
   (write #\a))
>>  a
=>  #\a
 (progn (fresh-line) (prin1 #\a))
>>  #\a
=>  #\a
 (progn (fresh-line) (print #\a))
>>
>>  #\a
=>  #\a
 (progn (fresh-line) (princ #\a))
>>  a
=>  #\a

 (dolist (val '(t nil))
   (let ((*print-escape* val) (*print-readably* val))
     (print '#\a)
     (prin1 #\a) (write-char #\Space)
     (princ #\a) (write-char #\Space)
     (write #\a)))
>>  #\a #\a a #\a
>>  #\a #\a a a
=>  NIL

 (progn (fresh-line) (write '(let ((a 1) (b 2)) (+ a b))))
>>  (LET ((A 1) (B 2)) (+ A B))
=>  (LET ((A 1) (B 2)) (+ A B))

 (progn (fresh-line) (pprint '(let ((a 1) (b 2)) (+ a b))))
>>  (LET ((A 1)
>>        (B 2))
>>    (+ A B))
=>  (LET ((A 1) (B 2)) (+ A B))

 (progn (fresh-line)
        (write '(let ((a 1) (b 2)) (+ a b)) :pretty t))
>>  (LET ((A 1)
>>        (B 2))
```

```
>>    (+ A B))
=>  (LET ((A 1) (B 2)) (+ A B))

 (with-output-to-string (s)
    (write 'write :stream s)
    (prin1 'prin1 s))
=>  "WRITEPRIN1"
```

22.2 The Lisp Pretty Printer

22.2.1 Pretty Printer Concepts

The facilities provided by the *pretty printer* permit *programs* to redefine the way in which *code* is displayed, and allow the full power of *pretty printing* to be applied to complex combinations of data structures.

Whether any given style of output is in fact "pretty" is inherently a somewhat subjective issue. However, since the effect of the *pretty printer* can be customized by *conforming programs*, the necessary flexibility is provided for individual *programs* to achieve an arbitrary degree of aesthetic control.

By providing direct access to the mechanisms within the pretty printer that make dynamic decisions about layout, the macros and functions **pprint-logical-block**, **pprint-newline**, and **pprint-indent** make it possible to specify pretty printing layout rules as a part of any function that produces output. They also make it very easy for the detection of circularity and sharing, and abbreviation based on length and nesting depth to be supported by the function.

The *pretty printer* is driven entirely by dispatch based on the *value* of ***print-pprint-dispatch***. The *function* **set-pprint-dispatch** makes it possible for *conforming programs* to associate new pretty printing functions with a *type*.

22.2.1.1 Dynamic Control of the Arrangement of Output

The actions of the *pretty printer* when a piece of output is too large to fit in the space available can be precisely controlled. Three concepts underlie the way these operations work---*logical blocks*, *conditional newlines*, and *sections*. Before proceeding further, it is important to define these terms.

The first line of the next figure shows a schematic piece of output. Each of the characters in the output is represented by "`-`". The positions of conditional newlines are indicated by digits. The beginnings and ends of logical blocks are indicated by "`<`" and "`>`" respectively.

The output as a whole is a logical block and the outermost section. This section is indicated by the `0`'s on the second line of Figure 1. Logical blocks nested within the output are specified by the macro **pprint-logical-block**. Conditional newline positions are specified by calls to **pprint-newline**. Each conditional newline defines two sections (one before it and one after it) and is associated with a third (the section immediately containing it).

The section after a conditional newline consists of: all the output up to, but not including, (a) the next conditional newline immediately contained in the same logical block; or if (a) is not applicable, (b) the next newline that is at a lesser level of nesting in logical blocks; or if (b) is not applicable, (c) the end of the output.

The section before a conditional newline consists of: all the output back to, but not including, (a) the previous conditional newline that is immediately contained in the same logical block; or if (a) is not applicable, (b) the beginning of the immediately containing logical block. The last four lines in Figure 1 indicate the sections before and after the four conditional newlines.

The section immediately containing a conditional newline is the shortest section that contains the conditional newline in question. In the next figure, the first conditional newline is immediately contained in the section marked with `0`'s, the second and third conditional newlines are immediately contained in the section before the fourth conditional newline, and the fourth conditional newline is immediately contained in the section after the first conditional newline.

```
<-1---<--<--2---3->--4-->->
00000000000000000000000000
11 1111111111111111111111
          22 222
             333 3333
       44444444444444 44444
```

Figure 22-3. Example of Logical Blocks, Conditional Newlines, and Sections

Whenever possible, the pretty printer displays the entire contents of a section on a single line. However, if the section is too long to fit in the space available, line breaks are inserted at conditional newline positions within the section.

22.2.1.2 Format Directive Interface

The primary interface to operations for dynamically determining the arrangement of output is provided through the functions and macros of the pretty printer. The next figure shows the defined names related to *pretty printing*.

```
*print-lines*            pprint-dispatch               pprint-pop
*print-miser-width*      pprint-exit-if-list-exhausted pprint-tab
*print-pprint-dispatch*  pprint-fill                   pprint-tabular
*print-right-margin*     pprint-indent                 set-pprint-dispatch
copy-pprint-dispatch     pprint-linear                 write
format                   pprint-logical-block
formatter                pprint-newline
```

Figure 22-4. Defined names related to pretty printing.

The next figure identifies a set of *format directives* which serve as an alternate interface to the same pretty printing operations in a more textually compact form.

```
~I   ~W      ~<...~:>
~:T  ~/.../  ~_
```

Figure 22-5. Format directives related to Pretty Printing

22.2.1.3 Compiling Format Strings

A *format string* is essentially a program in a special-purpose language that performs printing, and that is interpreted by the *function* **format**. The **formatter** *macro* provides the efficiency of using a *compiled function* to do that same printing but without losing the textual compactness of *format strings*.

A *format control* is either a *format string* or a *function* that was returned by the the **formatter** *macro*.

22.2.1.4 Pretty Print Dispatch Tables

A *pprint dispatch table* is a mapping from keys to pairs of values. Each key is a *type specifier*. The values associated with a key are a "function" (specifically, a *function designator* or **nil**) and a "numerical priority" (specifically, a *real*). Basic insertion and retrieval is done based on the keys with the equality of keys being tested by **equal**.

When ***print-pretty*** is *true*, the *current pprint dispatch table* (in ***print-pprint-dispatch***) controls how *objects* are printed. The information in this table takes precedence over all other mechanisms for specifying how to print *objects*. In particular, it has priority over user-defined **print-object** *methods* because the *current pprint dispatch table* is consulted first.

The function is chosen from the *current pprint dispatch table* by finding the highest priority function that is associated with a *type specifier* that matches the *object*; if there is more than one such function, it is *implementation-dependent* which is used.

However, if there is no information in the table about how to *pretty print* a particular kind of *object*, a *function* is invoked which uses **print-object** to print the *object*. The value of ***print-pretty*** is still *true* when this function is *called*, and individual methods for **print-object** might still elect to produce output in a special format conditional on the *value* of ***print-pretty***.

22.2.1.5 Pretty Printer Margins

A primary goal of pretty printing is to keep the output between a pair of margins. The column where the output begins is taken as the left margin. If the current column cannot be determined at the time output begins, the left margin is assumed to be zero. The right margin is controlled by ***print-right-margin***.

22.2.2 Examples of using the Pretty Printer

As an example of the interaction of logical blocks, conditional newlines, and indentation, consider the function `simple-pprint-defun` below. This function prints out lists whose *cars* are **defun** in the standard way assuming that the list has exactly length `4`.

```
(defun simple-pprint-defun (*standard-output* list)
  (pprint-logical-block (*standard-output* list :prefix "(" :suffix ")")
    (write (first list))
    (write-char #\Space)
    (pprint-newline :miser)
    (pprint-indent :current 0)
    (write (second list))
    (write-char #\Space)
    (pprint-newline :fill)
    (write (third list))
    (pprint-indent :block 1)
    (write-char #\Space)
    (pprint-newline :linear)
    (write (fourth list))))
```

Suppose that one evaluates the following:

```
(simple-pprint-defun *standard-output* '(defun prod (x y) (* x y)))
```

If the line width available is greater than or equal to `26`, then all of the output appears on one line. If the line width available is reduced to `25`, a line break is inserted at the linear-style conditional newline before the *expression* `(* x y)`, producing the output shown. The `(pprint-indent :block 1)` causes `(* x y)` to be printed at a relative indentation of `1` in the logical block.

```
(DEFUN PROD (X Y)
  (* X Y))
```

If the line width available is `15`, a line break is also inserted at the fill style conditional newline before the argument list. The call on `(pprint-indent :current 0)` causes the argument list to line up under the function name.

```
(DEFUN PROD
       (X Y)
  (* X Y))
```

If ***print-miser-width*** were greater than or equal to 14, the example output above would have been as follows, because all indentation changes are ignored in miser mode and line breaks are inserted at miser-style conditional newlines.

```
(DEFUN
 PROD
 (X Y)
 (* X Y))
```

As an example of a per-line prefix, consider that evaluating the following produces the output shown with a line width of `20` and ***print-miser-width*** of **nil**.

```
(pprint-logical-block (*standard-output* nil :per-line-prefix ";;; ")
  (simple-pprint-defun *standard-output* '(defun prod (x y) (* x y))))

;;; (DEFUN PROD
;;;        (X Y)
;;;   (* X Y))
```

As a more complex (and realistic) example, consider the function `pprint-let` below. This specifies how to print a **let** *form* in the traditional style. It is more complex than the example above, because it has to deal with nested structure. Also, unlike the example above it contains complete code to readably print any possible list that begins with the *symbol* **let**. The outermost **pprint-logical-block** *form* handles the printing of the input list as a whole and specifies that parentheses should be printed in the output. The second **pprint-logical-block** *form* handles the list of binding pairs. Each pair in the list is itself printed by the innermost **pprint-logical-block**. (A **loop** *form* is used instead of merely decomposing the pair into two *objects* so that readable output will be produced no matter whether the list corresponding to the pair has one element, two elements, or (being malformed) has more than two elements.) A space and a fill-style conditional newline are placed after each pair except the last. The loop at the end of the topmost **pprint-logical-block** *form* prints out the forms in the body of the **let** *form* separated by spaces and linear-style conditional newlines.

```
(defun pprint-let (*standard-output* list)
  (pprint-logical-block (nil list :prefix "(" :suffix ")")
    (write (pprint-pop))
    (pprint-exit-if-list-exhausted)
    (write-char #\Space)
    (pprint-logical-block (nil (pprint-pop) :prefix "(" :suffix ")")
      (pprint-exit-if-list-exhausted)
      (loop (pprint-logical-block (nil (pprint-pop) :prefix "(" :suffix ")")
              (pprint-exit-if-list-exhausted)
              (loop (write (pprint-pop))
                    (pprint-exit-if-list-exhausted)
                    (write-char #\Space)
                    (pprint-newline :linear)))
            (pprint-exit-if-list-exhausted)
            (write-char #\Space)
            (pprint-newline :fill)))
    (pprint-indent :block 1)
    (loop (pprint-exit-if-list-exhausted)
          (write-char #\Space)
          (pprint-newline :linear)
          (write (pprint-pop)))))
```

Suppose that one evaluates the following with ***print-level*** being 4, and ***print-circle*** being *true*.

```
(pprint-let *standard-output*
            '#1=(let (x (*print-length* (f (g 3)))
                      (z . 2) (k (car y)))
                  (setq x (sqrt z)) #1#))
```

If the line length is greater than or equal to 77, the output produced appears on one line. However, if the line length is 76, line breaks are inserted at the linear-style conditional newlines separating the forms in the body and the output below is produced. Note that, the degenerate binding pair x is printed readably even though it fails to be a list; a depth abbreviation marker is printed in place of (g 3); the binding pair (z . 2) is printed readably even though it is not a proper list; and appropriate circularity markers are printed.

```
#1=(LET (X (*PRINT-LENGTH* (F #)) (Z . 2) (K (CAR Y)))
     (SETQ X (SQRT Z))
     #1#)
```

If the line length is reduced to 35, a line break is inserted at one of the fill-style conditional newlines separating the binding pairs.

```
#1=(LET (X (*PRINT-PRETTY* (F #))
         (Z . 2) (K (CAR Y)))
     (SETQ X (SQRT Z))
     #1#)
```

Suppose that the line length is further reduced to 22 and ***print-length*** is set to 3. In this situation, line breaks are inserted after both the first and second binding pairs. In addition, the second binding pair is itself broken across two lines. Clause (b) of the description of fill-style conditional newlines (see the *function* **pprint-newline**) prevents the binding pair (z . 2) from being printed at the end of the third line. Note that the length abbreviation hides the circularity from view and therefore the printing of circularity markers disappears.

```
(LET (X
      (*PRINT-LENGTH*
       (F #))
      (Z . 2) ...)
  (SETQ X (SQRT Z))
  ...)
```

The next function prints a vector using "#(...)" notation.

```
(defun pprint-vector (*standard-output* v)
  (pprint-logical-block (nil nil :prefix "#(" :suffix ")")
    (let ((end (length v)) (i 0))
      (when (plusp end)
        (loop (pprint-pop)
              (write (aref v i))
              (if (= (incf i) end) (return nil))
              (write-char #\Space)
              (pprint-newline :fill))))))
```

Evaluating the following with a line length of 15 produces the output shown.

```
(pprint-vector *standard-output* '#(12 34 567 8 9012 34 567 89 0 1 23))

#(12 34 567 8
  9012 34 567
  89 0 1 23)
```

As examples of the convenience of specifying pretty printing with *format strings*, consider that the functions simple-pprint-defun and pprint-let used as examples above can be compactly defined as follows. (The function pprint-vector cannot be defined using **format** because the data structure it traverses is not a list.)

```
(defun simple-pprint-defun (*standard-output* list)
  (format T "~:<~W ~@_~:I~W ~:_~W~1I ~_~W~:>" list))

(defun pprint-let (*standard-output* list)
  (format T "~:<~W~^~:<~@{~:<~@{~W~^~_~}~:>~^~:_~}~:>~1I~@{~^~_~W~}~:>" list))
```

In the following example, the first *form* restores ***print-pprint-dispatch*** to the equivalent of its initial value. The next two forms then set up a special way to pretty print ratios. Note that the more specific *type specifier* has to be associated with a higher priority.

```
(setq *print-pprint-dispatch* (copy-pprint-dispatch nil))

(set-pprint-dispatch 'ratio
  #'(lambda (s obj)
      (format s "#.(/ ~W ~W)"
                (numerator obj) (denominator obj))))

(set-pprint-dispatch '(and ratio (satisfies minusp))
  #'(lambda (s obj)
      (format s "#.(- (/ ~W ~W))"
              (- (numerator obj)) (denominator obj)))
  5)

(pprint '(1/3 -2/3))
(#.(/ 1 3) #.(- (/ 2 3)))
```

The following two *forms* illustrate the definition of pretty printing functions for types of *code*. The first *form* illustrates how to specify the traditional method for printing quoted objects using *single-quote*. Note the care taken to ensure that data lists that happen to begin with **quote** will be printed readably. The second form specifies that lists beginning with the symbol `my-let` should print the same way that lists beginning with **let** print when the initial *pprint dispatch table* is in effect.

```
(set-pprint-dispatch '(cons (member quote)) ()
  #'(lambda (s list)
      (if (and (consp (cdr list)) (null (cddr list)))
         (funcall (formatter "'~W") s (cadr list))
         (pprint-fill s list))))

(set-pprint-dispatch '(cons (member my-let))
                     (pprint-dispatch '(let) nil))
```

The next example specifies a default method for printing lists that do not correspond to function calls. Note that the functions **pprint-linear**, **pprint-fill**, and **pprint-tabular** are all defined with optional *colon-p* and *at-sign-p* arguments so that they can be used as **pprint dispatch functions** as well as `~/.../` functions.

```
(set-pprint-dispatch '(cons (not (and symbol (satisfies fboundp))))
                     #'pprint-fill -5)

;; Assume a line length of 9
(pprint '(0 b c d e f g h i j k))
(0 b c d
 e f g h
 i j k)
```

This final example shows how to define a pretty printing function for a user defined data structure.

```
(defstruct family mom kids)

(set-pprint-dispatch 'family
  #'(lambda (s f)
      (funcall (formatter "~@<#<~;~W and ~2I~_~/pprint-fill/~;>~:>")
               s (family-mom f) (family-kids f))))
```

The pretty printing function for the structure `family` specifies how to adjust the layout of the output so that it can fit aesthetically into a variety of line widths. In addition, it obeys the printer control variables ***print-level***, ***print-length***, ***print-lines***, ***print-circle*** and ***print-escape***, and can tolerate several different kinds of malformity in the data structure. The output below shows what is printed out with a right margin of `25`, ***print-pretty*** being *true*, ***print-escape*** being *false*, and a malformed `kids` list.

```
(write (list 'principal-family
             (make-family :mom "Lucy"
                          :kids '("Mark" "Bob" . "Dan")))
       :right-margin 25 :pretty T :escape nil :miser-width nil)
(PRINCIPAL-FAMILY
 #<Lucy and
     Mark Bob . Dan>)
```

Note that a pretty printing function for a structure is different from the structure's **print-object** *method*. While **print-object** *methods* are permanently associated with a structure, pretty printing functions are stored in *pprint dispatch tables* and can be rapidly changed to reflect different printing needs. If there is no pretty printing function for a structure in the current *pprint dispatch table*, its **print-object** *method* is used instead.

22.2.3 Notes about the Pretty Printer's Background

For a background reference to the abstract concepts detailed in this section, see *XP: A Common Lisp Pretty Printing System*. The details of that paper are not binding on this document, but may be helpful in establishing a conceptual basis for understanding this material.

22.3 Formatted Output

format is useful for producing nicely formatted text, producing good-looking messages, and so on. **format** can generate and return a *string* or output to *destination*.

The *control-string* argument to **format** is actually a *format control*. That is, it can be either a *format string* or a *function*, for example a *function* returned by the **formatter** *macro*.

If it is a *function*, the *function* is called with the appropriate output stream as its first argument and the data arguments to **format** as its remaining arguments. The function should perform whatever output is necessary and return the unused tail of the arguments (if any).

The compilation process performed by **formatter** produces a *function* that would do with its *arguments* as the **format** interpreter would do with those *arguments*.

The remainder of this section describes what happens if the *control-string* is a *format string*.

Control-string is composed of simple text (*characters*) and embedded directives.

format writes the simple text as is; each embedded directive specifies further text output that is to appear at the corresponding point within the simple text. Most directives use one or more elements of *args* to create their output.

A directive consists of a *tilde*, optional prefix parameters separated by commas, optional *colon* and *at-sign* modifiers, and a single character indicating what kind of directive this is. There is no required ordering between the *at-sign* and *colon* modifier. The *case* of the directive character is ignored. Prefix parameters are notated as signed (sign is optional) decimal numbers, or as a *single-quote* followed by a character. For example, `~5,'0d` can be used to print an *integer* in decimal radix in five columns with leading zeros, or `~5,'*d` to get leading asterisks.

In place of a prefix parameter to a directive, V (or v) can be used. In this case, **format** takes an argument from *args* as a parameter to the directive. The argument should be an *integer* or *character*. If the *arg* used by a V parameter is **nil**, the effect is as if the parameter had been omitted. # can be used in place of a prefix parameter; it represents the number of *args* remaining to be processed. When used within a recursive format, in the context of ~? or ~{, the # prefix parameter represents the number of *format arguments* remaining within the recursive call.

Examples of *format strings*:

```
"~S"        ;This is an S directive with no parameters or modifiers.
"~3,-4:@s"  ;This is an S directive with two parameters, 3 and -4,
            ; and both the colon and at-sign flags.
"~,+4S"     ;Here the first prefix parameter is omitted and takes
            ; on its default value, while the second parameter is 4.
```

Figure 22-6. Examples of format control strings

format sends the output to *destination*. If *destination* is **nil**, **format** creates and returns a *string* containing the output from *control-string*. If *destination* is *non-nil*, it must be a *string* with a *fill pointer*, a *stream*, or the symbol **t**. If *destination* is a *string* with a *fill pointer*, the output is added to the end of the *string*. If *destination* is a *stream*, the output is sent to that *stream*. If *destination* is **t**, the output is sent to *standard output*.

In the description of the directives that follows, the term *arg* in general refers to the next item of the set of *args* to be processed. The word or phrase at the beginning of each description is a mnemonic for the directive. **format** directives do not bind any of the printer control variables (***print-...***) except as specified in the following descriptions. Implementations may specify the binding of new, implementation-specific printer control variables for each **format** directive, but they may neither bind any standard printer control variables not specified in description of a **format** directive nor fail to bind any standard printer control variables as specified in the description.

22.3.1 FORMAT Basic Output

22.3.1.1 Tilde C: Character

The next *arg* should be a *character*; it is printed according to the modifier flags.

~C prints the *character* as if by using **write-char** if it is a *simple character*. *Characters* that are not *simple* are not necessarily printed as if by **write-char**, but are displayed in an *implementation-defined*, abbreviated format. For example,

```
 (format nil "~C" #\A) =>  "A"
 (format nil "~C" #\Space) =>  " "
```

~:C is the same as ~C for *printing characters*, but other *characters* are "spelled out." The intent is that this is a "pretty" format for printing characters. For *simple characters* that are not *printing*, what is spelled out is the *name* of the *character* (see **char-name**). For *characters* that are not *simple* and not *printing*, what is spelled out is *implementation-defined*. For example,

```
 (format nil "~:C" #\A) =>  "A"
 (format nil "~:C" #\Space) =>  "Space"
;; This next example assumes an implementation-defined "Control" attribute.
 (format nil "~:C" #\Control-Space)
=>  "Control-Space"
OR=>  "c-Space"
```

~:@C prints what ~:C would, and then if the *character* requires unusual shift keys on the keyboard to type it, this fact is mentioned. For example,

```
(format nil "~:@C" #\Control-Partial) =>  "Control-<PARTIAL> (Top-F)"
```

This is the format used for telling the user about a key he is expected to type, in prompts, for instance. The precise output may depend not only on the implementation, but on the particular I/O devices in use.

~@C prints the *character* in a way that the *Lisp reader* can understand, using #\ syntax.

~@C binds ***print-escape*** to **t**.

22.3.1.2 Tilde Percent: Newline

This outputs a #\Newline character, thereby terminating the current output line and beginning a new one. ~*n*% outputs *n* newlines. No *arg* is used.

22.3.1.3 Tilde Ampersand: Fresh-Line

Unless it can be determined that the output stream is already at the beginning of a line, this outputs a newline. ~*n*& calls **fresh-line** and then outputs *n*-1 newlines. ~0& does nothing.

22.3.1.4 Tilde Vertical-Bar: Page

This outputs a page separator character, if possible. ~*n*| does this *n* times.

22.3.1.5 Tilde Tilde: Tilde

This outputs a *tilde*. ~*n*~ outputs *n* tildes.

22.3.2 FORMAT Radix Control

22.3.2.1 Tilde R: Radix

~*n*R prints *arg* in radix *n*. The modifier flags and any remaining parameters are used as for the ~D directive. ~D is the same as ~10R. The full form is ~*radix*,*mincol*,*padchar*,*commachar*,*comma-interval*R.

If no prefix parameters are given to ~R, then a different interpretation is given. The argument should be an *integer*. For example, if *arg* is 4:

* ~R prints *arg* as a cardinal English number: four.
* ~:R prints *arg* as an ordinal English number: fourth.
* ~@R prints *arg* as a Roman numeral: IV.
* ~:@R prints *arg* as an old Roman numeral: IIII.

For example:

```
(format nil "~,,' ,4:B" 13) =>  "1101"
(format nil "~,,' ,4:B" 17) =>  "1 0001"
(format nil "~19,0,' ,4:B" 3333) =>  "0000 1101 0000 0101"
(format nil "~3,,,' ,2:R" 17) =>  "1 22"
(format nil "~,,'|,2:D" #xFFFF) =>   "6|55|35"
```

If and only if the first parameter, *n*, is supplied, ~R binds ***print-escape*** to *false*, ***print-radix*** to *false*, ***print-base*** to *n*, and ***print-readably*** to *false*.

If and only if no parameters are supplied, ~R binds ***print-base*** to 10.

22.3.2.2 Tilde D: Decimal

An *arg*, which should be an *integer*, is printed in decimal radix. ~D will never put a decimal point after the number.

~*mincol*D uses a column width of *mincol*; spaces are inserted on the left if the number requires fewer than *mincol* columns for its digits and sign. If the number doesn't fit in *mincol* columns, additional columns are used as needed.

~*mincol*,*padchar*D uses *padchar* as the pad character instead of space.

If *arg* is not an *integer*, it is printed in ~A format and decimal base.

The @ modifier causes the number's sign to be printed always; the default is to print it only if the number is negative. The : modifier causes commas to be printed between groups of digits; *commachar* may be used to change the character used as the comma. *comma-interval* must be an *integer* and defaults to 3. When the : modifier is given to any of these directives, the *commachar* is printed between groups of *comma-interval* digits.

Thus the most general form of ~D is ~*mincol*,*padchar*,*commachar*,*comma-interval*D.

~D binds ***print-escape*** to *false*, ***print-radix*** to *false*, ***print-base*** to 10, and ***print-readably*** to *false*.

22.3.2.3 Tilde B: Binary

This is just like ~D but prints in binary radix (radix 2) instead of decimal. The full form is therefore ~*mincol*,*padchar*,*commachar*,*comma-interval*B.

~B binds ***print-escape*** to *false*, ***print-radix*** to *false*, ***print-base*** to 2, and ***print-readably*** to *false*.

22.3.2.4 Tilde O: Octal

This is just like ~D but prints in octal radix (radix 8) instead of decimal. The full form is therefore ~*mincol*,*padchar*,*commachar*,*comma-interval*O.

~O binds ***print-escape*** to *false*, ***print-radix*** to *false*, ***print-base*** to 8, and ***print-readably*** to *false*.

22.3.2.5 Tilde X: Hexadecimal

This is just like ~D but prints in hexadecimal radix (radix 16) instead of decimal. The full form is therefore ~*mincol*,*padchar*,*commachar*,*comma-interval*X.

~X binds ***print-escape*** to *false*, ***print-radix*** to *false*, ***print-base*** to 16, and ***print-readably*** to *false*.

22.3.3 FORMAT Floating-Point Printers

22.3.3.1 Tilde F: Fixed-Format Floating-Point

The next *arg* is printed as a *float*.

The full form is `~`*w*`,`*d*`,`*k*`,`*overflowchar*`,`*padchar*`F`. The parameter *w* is the width of the field to be printed; *d* is the number of digits to print after the decimal point; *k* is a scale factor that defaults to zero.

Exactly *w* characters will be output. First, leading copies of the character *padchar* (which defaults to a space) are printed, if necessary, to pad the field on the left. If the *arg* is negative, then a minus sign is printed; if the *arg* is not negative, then a plus sign is printed if and only if the `@` modifier was supplied. Then a sequence of digits, containing a single embedded decimal point, is printed; this represents the magnitude of the value of *arg* times 10^*k*, rounded to *d* fractional digits. When rounding up and rounding down would produce printed values equidistant from the scaled value of *arg*, then the implementation is free to use either one. For example, printing the argument `6.375` using the format `~4,2F` may correctly produce either `6.37` or `6.38`. Leading zeros are not permitted, except that a single zero digit is output before the decimal point if the printed value is less than one, and this single zero digit is not output at all if *w*=*d*+1.

If it is impossible to print the value in the required format in a field of width *w*, then one of two actions is taken. If the parameter *overflowchar* is supplied, then *w* copies of that parameter are printed instead of the scaled value of *arg*. If the *overflowchar* parameter is omitted, then the scaled value is printed using more than *w* characters, as many more as may be needed.

If the *w* parameter is omitted, then the field is of variable width. In effect, a value is chosen for *w* in such a way that no leading pad characters need to be printed and exactly *d* characters will follow the decimal point. For example, the directive `~,2F` will print exactly two digits after the decimal point and as many as necessary before the decimal point.

If the parameter *d* is omitted, then there is no constraint on the number of digits to appear after the decimal point. A value is chosen for *d* in such a way that as many digits as possible may be printed subject to the width constraint imposed by the parameter *w* and the constraint that no trailing zero digits may appear in the fraction, except that if the fraction to be printed is zero, then a single zero digit should appear after the decimal point if permitted by the width constraint.

If both *w* and *d* are omitted, then the effect is to print the value using ordinary free-format output; **prin1** uses this format for any number whose magnitude is either zero or between 10^-3 (inclusive) and 10^7 (exclusive).

If *w* is omitted, then if the magnitude of *arg* is so large (or, if *d* is also omitted, so small) that more than 100 digits would have to be printed, then an implementation is free, at its discretion, to print the number using exponential notation instead, as if by the directive `~E` (with all parameters to `~E` defaulted, not taking their values from the `~F` directive).

If *arg* is a *rational* number, then it is coerced to be a *single float* and then printed. Alternatively, an implementation is permitted to process a *rational* number by any other method that has essentially the same behavior but avoids loss of precision or overflow because of the coercion. If *w* and *d* are not supplied and the number has no exact decimal representation, for example `1/3`, some precision cutoff must be chosen by the implementation since only a finite number of digits may be printed.

If *arg* is a *complex* number or some non-numeric *object*, then it is printed using the format directive `~`*w*`D`, thereby printing it in decimal radix and a minimum field width of *w*.

`~F` binds ***print-escape*** to *false* and ***print-readably*** to *false*.

22.3.3.2 Tilde E: Exponential Floating-Point

The next *arg* is printed as a *float* in exponential notation.

The full form is `~`*w*`,`*d*`,`*e*`,`*k*`,`*overflowchar*`,`*padchar*`,`*exponentchar*`E`. The parameter *w* is the width of the field to be printed; *d* is the number of digits to print after the decimal point; *e* is the number of digits to use when printing the exponent; *k* is a scale factor that defaults to one (not zero).

Exactly *w* characters will be output. First, leading copies of the character *padchar* (which defaults to a space) are printed, if necessary, to pad the field on the left. If the *arg* is negative, then a minus sign is printed; if the *arg* is not negative, then a plus sign is printed if and only if the `@` modifier was supplied. Then a sequence of digits containing a single embedded decimal point is printed. The form of this sequence of digits depends on the scale factor *k*. If *k* is zero, then *d* digits are printed after the decimal point, and a single zero digit appears before the decimal point if the total field width will permit it. If *k* is positive, then it must be strictly less than *d*+2; *k* significant digits are printed before the decimal point, and *d*-*k*+1 digits are printed after the decimal point. If *k* is negative, then it must be strictly greater than -*d*; a single zero digit appears before the decimal point if the total field width will permit it, and after the decimal point are printed first -*k* zeros and then *d*+*k* significant digits. The printed fraction must be properly rounded. When rounding up and rounding down would produce printed values equidistant from the scaled value of *arg*, then the implementation is free to use either one. For example, printing the argument `637.5` using the format `~8,2E` may correctly produce either `6.37E+2` or `6.38E+2`.

Following the digit sequence, the exponent is printed. First the character parameter *exponentchar* is printed; if this parameter is omitted, then the *exponent marker* that **prin1** would use is printed, as determined from the type of the *float* and the current value of ***read-default-float-format***. Next, either a plus sign or a minus sign is printed, followed by *e* digits representing the power of ten by which the printed fraction must be multiplied to properly represent the rounded value of *arg*.

If it is impossible to print the value in the required format in a field of width *w*, possibly because *k* is too large or too small or because the exponent cannot be printed in *e* character positions, then one of two actions is taken. If the parameter *overflowchar* is supplied, then *w* copies of that parameter are printed instead of the scaled value of *arg*. If the *overflowchar* parameter is omitted, then the scaled value is printed using more than *w* characters, as many more as may be needed; if the problem is that *d* is too small for the supplied *k* or that *e* is too small, then a larger value is used for *d* or *e* as may be needed.

If the *w* parameter is omitted, then the field is of variable width. In effect a value is chosen for *w* in such a way that no leading pad characters need to be printed.

If the parameter *d* is omitted, then there is no constraint on the number of digits to appear. A value is chosen for *d* in such a way that as many digits as possible may be printed subject to the width constraint imposed by the parameter *w*, the constraint of the scale factor *k*, and the constraint that no trailing zero digits may appear in the fraction, except that if the fraction to be printed is zero then a single zero digit should appear after the decimal point.

If the parameter *e* is omitted, then the exponent is printed using the smallest number of digits necessary to represent its value.

If all of *w*, *d*, and *e* are omitted, then the effect is to print the value using ordinary free-format exponential-notation output; **prin1** uses a similar format for any non-zero number whose magnitude is less than 10^-3 or greater than or equal to 10^7. The only difference is that the `~E` directive always prints a plus or minus sign in front of the exponent, while **prin1** omits the plus sign if the exponent is non-negative.

If *arg* is a *rational* number, then it is coerced to be a *single float* and then printed. Alternatively, an implementation is permitted to process a *rational* number by any other method that has essentially the same behavior but avoids loss of precision or overflow because of the coercion. If *w* and *d* are unsupplied and the number has no exact decimal representation, for example `1/3`, some precision cutoff must be chosen by the implementation since only a

finite number of digits may be printed.

If *arg* is a *complex* number or some non-numeric *object*, then it is printed using the format directive ~*w*D, thereby printing it in decimal radix and a minimum field width of *w*.

~E binds ***print-escape*** to *false* and ***print-readably*** to *false*.

22.3.3.3 Tilde G: General Floating-Point

The next *arg* is printed as a *float* in either fixed-format or exponential notation as appropriate.

The full form is ~*w*,*d*,*e*,*k*,*overflowchar*,*padchar*,*exponentchar*G. The format in which to print *arg* depends on the magnitude (absolute value) of the *arg*. Let *n* be an integer such that 10^*n*-1 <= |*arg*| < 10^*n*. Let *ee* equal *e*+2, or 4 if *e* is omitted. Let *ww* equal *w*-*ee*, or **nil** if *w* is omitted. If *d* is omitted, first let *q* be the number of digits needed to print *arg* with no loss of information and without leading or trailing zeros; then let *d* equal (max *q* (min *n* 7)). Let *dd* equal *d*-*n*.

If 0 <= *dd* <= *d*, then *arg* is printed as if by the format directives

~*ww*,*dd*,,*overflowchar*,*padchar*F~*ee*@T

Note that the scale factor *k* is not passed to the ~F directive. For all other values of *dd*, *arg* is printed as if by the format directive

~*w*,*d*,*e*,*k*,*overflowchar*,*padchar*,*exponentchar*E

In either case, an @ modifier is supplied to the ~F or ~E directive if and only if one was supplied to the ~G directive.

~G binds ***print-escape*** to *false* and ***print-readably*** to *false*.

22.3.3.4 Tilde Dollarsign: Monetary Floating-Point

The next *arg* is printed as a *float* in fixed-format notation.

The full form is ~*d*,*n*,*w*,*padchar*$. The parameter *d* is the number of digits to print after the decimal point (default value 2); *n* is the minimum number of digits to print before the decimal point (default value 1); *w* is the minimum total width of the field to be printed (default value 0).

First padding and the sign are output. If the *arg* is negative, then a minus sign is printed; if the *arg* is not negative, then a plus sign is printed if and only if the @ modifier was supplied. If the : modifier is used, the sign appears before any padding, and otherwise after the padding. If *w* is supplied and the number of other characters to be output is less than *w*, then copies of *padchar* (which defaults to a space) are output to make the total field width equal *w*. Then *n* digits are printed for the integer part of *arg*, with leading zeros if necessary; then a decimal point; then *d* digits of fraction, properly rounded.

If the magnitude of *arg* is so large that more than *m* digits would have to be printed, where *m* is the larger of *w* and 100, then an implementation is free, at its discretion, to print the number using exponential notation instead, as if by the directive ~*w*,*q*,,,,*padchar*E, where *w* and *padchar* are present or omitted according to whether they were present or omitted in the ~$ directive, and where *q*=*d*+*n*-1, where *d* and *n* are the (possibly default) values given to the ~$ directive.

If *arg* is a *rational* number, then it is coerced to be a *single float* and then printed. Alternatively, an implementation is permitted to process a *rational* number by any other method that has essentially the same behavior but avoids loss of precision or overflow because of the coercion.

If *arg* is a *complex* number or some non-numeric *object*, then it is printed using the format directive ~*w*D, thereby printing it in decimal radix and a minimum field width of *w*.

~$ binds ***print-escape*** to *false* and ***print-readably*** to *false*.

22.3.4 FORMAT Printer Operations

22.3.4.1 Tilde A: Aesthetic

An *arg*, any *object*, is printed without escape characters (as by **princ**). If *arg* is a *string*, its *characters* will be output verbatim. If *arg* is **nil** it will be printed as **nil**; the *colon* modifier (~:A) will cause an *arg* of **nil** to be printed as `()`, but if *arg* is a composite structure, such as a *list* or *vector*, any contained occurrences of **nil** will still be printed as **nil**.

~*mincol*A inserts spaces on the right, if necessary, to make the width at least *mincol* columns. The @ modifier causes the spaces to be inserted on the left rather than the right.

~*mincol*,*colinc*,*minpad*,*padchar*A is the full form of ~A, which allows control of the padding. The *string* is padded on the right (or on the left if the @ modifier is used) with at least *minpad* copies of *padchar*; padding characters are then inserted *colinc* characters at a time until the total width is at least *mincol*. The defaults are 0 for *mincol* and *minpad*, 1 for *colinc*, and the space character for *padchar*.

~A binds ***print-escape*** to *false*, and ***print-readably*** to *false*.

22.3.4.2 Tilde S: Standard

This is just like ~A, but *arg* is printed with escape characters (as by **prin1** rather than `princ`). The output is therefore suitable for input to **read**. ~S accepts all the arguments and modifiers that ~A does.

~S binds ***print-escape*** to **t**.

22.3.4.3 Tilde W: Write

An argument, any *object*, is printed obeying every printer control variable (as by **write**). In addition, ~W interacts correctly with depth abbreviation, by not resetting the depth counter to zero. ~W does not accept parameters. If given the *colon* modifier, ~W binds ***print-pretty*** to *true*. If given the *at-sign* modifier, ~W binds ***print-level*** and ***print-length*** to **nil**.

~W provides automatic support for the detection of circularity and sharing. If the *value* of ***print-circle*** is not **nil** and ~W is applied to an argument that is a circular (or shared) reference, an appropriate #*n*# marker is inserted in the output instead of printing the argument.

22.3.5 FORMAT Pretty Printer Operations

The following constructs provide access to the *pretty printer*:

22.3.5.1 Tilde Underscore: Conditional Newline

Without any modifiers, ~_ is the same as `(pprint-newline :linear)`. ~@_ is the same as `(pprint-newline :miser)`. ~:_ is the same as `(pprint-newline :fill)`. ~:@_ is the same as `(pprint-newline :mandatory)`.

22.3.5.2 Tilde Less-Than-Sign: Logical Block

`~<...~:>`

If `~:>` is used to terminate a `~<...~>`, the directive is equivalent to a call to **pprint-logical-block**. The argument corresponding to the `~<...~:>` directive is treated in the same way as the *list* argument to **pprint-logical-block**, thereby providing automatic support for non-*list* arguments and the detection of circularity, sharing, and depth abbreviation. The portion of the *control-string* nested within the `~<...~:>` specifies the `:prefix` (or `:per-line-prefix`), `:suffix`, and body of the **pprint-logical-block**.

The *control-string* portion enclosed by `~<...~:>` can be divided into segments `~<`*prefix*`~;`*body*`~;`*suffix*`~:>` by `~;` directives. If the first section is terminated by `~@;`, it specifies a per-line prefix rather than a simple prefix. The *prefix* and *suffix* cannot contain format directives. An error is signaled if either the prefix or suffix fails to be a constant string or if the enclosed portion is divided into more than three segments.

If the enclosed portion is divided into only two segments, the *suffix* defaults to the null string. If the enclosed portion consists of only a single segment, both the *prefix* and the *suffix* default to the null string. If the *colon* modifier is used (i.e., `~:<...~:>`), the *prefix* and *suffix* default to `"("` and `")"` (respectively) instead of the null string.

The body segment can be any arbitrary *format string*. This *format string* is applied to the elements of the list corresponding to the `~<...~:>` directive as a whole. Elements are extracted from this list using **pprint-pop**, thereby providing automatic support for malformed lists, and the detection of circularity, sharing, and length abbreviation. Within the body segment, `~^` acts like **pprint-exit-if-list-exhausted**.

`~<...~:>` supports a feature not supported by **pprint-logical-block**. If `~:@>` is used to terminate the directive (i.e., `~<...~:@>`), then a fill-style conditional newline is automatically inserted after each group of blanks immediately contained in the body (except for blanks after a <Newline> directive). This makes it easy to achieve the equivalent of paragraph filling.

If the *at-sign* modifier is used with `~<...~:>`, the entire remaining argument list is passed to the directive as its argument. All of the remaining arguments are always consumed by `~@<...~:>`, even if they are not all used by the *format string* nested in the directive. Other than the difference in its argument, `~@<...~:>` is exactly the same as `~<...~:>` except that circularity detection is not applied if `~@<...~:>` is encountered at top level in a *format string*. This ensures that circularity detection is applied only to data lists, not to *format argument lists*.

`" . #`*n*`#"` is printed if circularity or sharing has to be indicated for its argument as a whole.

To a considerable extent, the basic form of the directive `~<...~>` is incompatible with the dynamic control of the arrangement of output by `~W`, `~_`, `~<...~:>`, `~I`, and `~:T`. As a result, an error is signaled if any of these directives is nested within `~<...~>`. Beyond this, an error is also signaled if the `~<...~:;...~>` form of `~<...~>` is used in the same *format string* with `~W`, `~_`, `~<...~:>`, `~I`, or `~:T`.

See also Section 22.3.6.2 (Tilde Less-Than-Sign: Justification).

22.3.5.3 Tilde I: Indent

`~`*n*`I` is the same as `(pprint-indent :block n)`.

`~`*n*`:I` is the same as `(pprint-indent :current n)`. In both cases, *n* defaults to zero, if it is omitted.

22.3.5.4 Tilde Slash: Call Function

~/name/

User defined functions can be called from within a format string by using the directive *~/name/*. The *colon* modifier, the *at-sign* modifier, and arbitrarily many parameters can be specified with the *~/name/* directive. *name* can be any arbitrary string that does not contain a "/". All of the characters in *name* are treated as if they were upper case. If *name* contains a single *colon* (:) or double *colon* (::), then everything up to but not including the first ":" or "::" is taken to be a *string* that names a *package*. Everything after the first ":" or "::" (if any) is taken to be a *string* that names a symbol. The function corresponding to a ~/name/ directive is obtained by looking up the *symbol* that has the indicated name in the indicated *package*. If *name* does not contain a ":" or "::", then the whole *name* string is looked up in the COMMON-LISP-USER package.

When a ~/name/ directive is encountered, the indicated function is called with four or more arguments. The first four arguments are: the output stream, the *format argument* corresponding to the directive, a *generalized boolean* that is *true* if the *colon* modifier was used, and a *generalized boolean* that is *true* if the *at-sign* modifier was used. The remaining arguments consist of any parameters specified with the directive. The function should print the argument appropriately. Any values returned by the function are ignored.

The three *functions* **pprint-linear**, **pprint-fill**, and **pprint-tabular** are specifically designed so that they can be called by ~/.../ (i.e., ~/pprint-linear/, ~/pprint-fill/, and ~/pprint-tabular/). In particular they take *colon* and *at-sign* arguments.

22.3.6 FORMAT Layout Control

22.3.6.1 Tilde T: Tabulate

This spaces over to a given column. ~*colnum*,*colinc*T will output sufficient spaces to move the cursor to column *colnum*. If the cursor is already at or beyond column *colnum*, it will output spaces to move it to column *colnum*+*k***colinc* for the smallest positive integer *k* possible, unless *colinc* is zero, in which case no spaces are output if the cursor is already at or beyond column *colnum*. *colnum* and *colinc* default to 1.

If for some reason the current absolute column position cannot be determined by direct inquiry, **format** may be able to deduce the current column position by noting that certain directives (such as ~%, or ~&, or ~A with the argument being a string containing a newline) cause the column position to be reset to zero, and counting the number of characters emitted since that point. If that fails, **format** may attempt a similar deduction on the riskier assumption that the destination was at column zero when **format** was invoked. If even this heuristic fails or is implementationally inconvenient, at worst the ~T operation will simply output two spaces.

~@T performs relative tabulation. ~*colrel*,*colinc*@T outputs *colrel* spaces and then outputs the smallest non-negative number of additional spaces necessary to move the cursor to a column that is a multiple of *colinc*. For example, the directive ~3,8@T outputs three spaces and then moves the cursor to a "standard multiple-of-eight tab stop" if not at one already. If the current output column cannot be determined, however, then *colinc* is ignored, and exactly *colrel* spaces are output.

If the *colon* modifier is used with the ~T directive, the tabbing computation is done relative to the horizontal position where the section immediately containing the directive begins, rather than with respect to a horizontal position of zero. The numerical parameters are both interpreted as being in units of *ems* and both default to 1. ~*n*,*m*:T is the same as (pprint-tab :section *n m*). ~*n*,*m*:@T is the same as (pprint-tab :section-relative *n m*).

22.3.6.2 Tilde Less-Than-Sign: Justification

~mincol`,`*colinc*`,`*minpad*`,`*padchar*`<`*str*`~>`

This justifies the text produced by processing *str* within a field at least *mincol* columns wide. *str* may be divided up into segments with `~;`, in which case the spacing is evenly divided between the text segments.

With no modifiers, the leftmost text segment is left justified in the field, and the rightmost text segment is right justified. If there is only one text element, as a special case, it is right justified. The `:` modifier causes spacing to be introduced before the first text segment; the `@` modifier causes spacing to be added after the last. The *minpad* parameter (default `0`) is the minimum number of padding characters to be output between each segment. The padding character is supplied by *padchar*, which defaults to the space character. If the total width needed to satisfy these constraints is greater than *mincol*, then the width used is *mincol*+*k***colinc* for the smallest possible non-negative integer value *k*. *colinc* defaults to `1`, and *mincol* defaults to `0`.

Note that *str* may include **format** directives. All the clauses in *str* are processed in order; it is the resulting pieces of text that are justified.

The `~^` directive may be used to terminate processing of the clauses prematurely, in which case only the completely processed clauses are justified.

If the first clause of a `~<` is terminated with `~:;` instead of `~;`, then it is used in a special way. All of the clauses are processed (subject to `~^`, of course), but the first one is not used in performing the spacing and padding. When the padded result has been determined, then if it will fit on the current line of output, it is output, and the text for the first clause is discarded. If, however, the padded text will not fit on the current line, then the text segment for the first clause is output before the padded text. The first clause ought to contain a newline (such as a `~%` directive). The first clause is always processed, and so any arguments it refers to will be used; the decision is whether to use the resulting segment of text, not whether to process the first clause. If the `~:;` has a prefix parameter *n*, then the padded text must fit on the current line with *n* character positions to spare to avoid outputting the first clause's text. For example, the control string

```
"~%;; ~{ ~<~%;; ~1:; ~S~>~^ ,~} .~%"
```

can be used to print a list of items separated by commas without breaking items over line boundaries, beginning each line with `;;` . The prefix parameter `1` in `~1:;` accounts for the width of the comma that will follow the justified item if it is not the last element in the list, or the period if it is. If `~:;` has a second prefix parameter, then it is used as the width of the line, thus overriding the natural line width of the output stream. To make the preceding example use a line width of 50, one would write

```
"~%;; ~{ ~<~%;; ~1,50:; ~S~>~^ ,~}  .~%"
```

If the second argument is not supplied, then **format** uses the line width of the *destination* output stream. If this cannot be determined (for example, when producing a *string* result), then **format** uses `72` as the line length.

See also Section 22.3.5.2 (Tilde Less-Than-Sign: Logical Block).

22.3.6.3 Tilde Greater-Than-Sign: End of Justification

`~>` terminates a `~<`. The consequences of using it elsewhere are undefined.

22.3.7 FORMAT Control-Flow Operations

22.3.7.1 Tilde Asterisk: Go-To

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

~:* backs up in the list of arguments so that the argument last processed will be processed again. ~*n*:* backs up *n* arguments.

When within a ~{ construct (see below), the ignoring (in either direction) is relative to the list of arguments being processed by the iteration.

~*n*@* goes to the *n*th *arg*, where 0 means the first one; *n* defaults to 0, so ~@* goes back to the first *arg*. Directives after a ~*n*@* will take arguments in sequence beginning with the one gone to. When within a ~{ construct, the "goto" is relative to the list of arguments being processed by the iteration.

22.3.7.2 Tilde Left-Bracket: Conditional Expression

~[*str0*~;*str1*~;...~;*strn*~]

This is a set of control strings, called *clauses*, one of which is chosen and used. The clauses are separated by ~; and the construct is terminated by ~]. For example,

```
"~[Siamese~;Manx~;Persian~] Cat"
```

The *arg*th clause is selected, where the first clause is number 0. If a prefix parameter is given (as ~*n*[), then the parameter is used instead of an argument. If *arg* is out of range then no clause is selected and no error is signaled. After the selected alternative has been processed, the control string continues after the ~].

~[*str0*~;*str1*~;...~;*strn*~:;*default*~] has a default case. If the *last* ~; used to separate clauses is ~:; instead, then the last clause is an else clause that is performed if no other clause is selected. For example:

```
"~[Siamese~;Manx~;Persian~:;Alley~] Cat"
```

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

~@[*consequent*~] tests the argument. If it is *true*, then the argument is not used up by the ~[command but remains as the next one to be processed, and the one clause *consequent* is processed. If the *arg* is *false*, then the argument is used up, and the clause is not processed. The clause therefore should normally use exactly one argument, and may expect it to be *non-nil*. For example:

```
 (setq *print-level* nil *print-length* 5)
 (format nil
        "~@[ print level = ~D~]~@[ print length = ~D~]"
        *print-level* *print-length*)
=>   " print length = 5"
```

Note also that

```
 (format stream "...~@[str~]..." ...)
==  (format stream "...~:[~;~:*str~]..." ...)
```

The combination of ~[and # is useful, for example, for dealing with English conventions for printing lists:

```
 (setq foo "Items:~#[ none~; ~S~; ~S and ~S~
           ~:;~@{~#[~; and~] ~S~^ ,~}~].")
 (format nil foo) =>   "Items: none."
 (format nil foo 'foo) =>   "Items: FOO."
 (format nil foo 'foo 'bar) =>   "Items: FOO and BAR."
 (format nil foo 'foo 'bar 'baz) =>   "Items: FOO, BAR, and BAZ."
 (format nil foo 'foo 'bar 'baz 'quux) =>   "Items: FOO, BAR, BAZ, and QUUX."
```

22.3.7.3 Tilde Right-Bracket: End of Conditional Expression

~] terminates a ~[. The consequences of using it elsewhere are undefined.

22.3.7.4 Tilde Left-Brace: Iteration

~{*str*~}

This is an iteration construct. The argument should be a *list*, which is used as a set of arguments as if for a recursive call to **format**. The *string str* is used repeatedly as the control string. Each iteration can absorb as many elements of the *list* as it likes as arguments; if *str* uses up two arguments by itself, then two elements of the *list* will get used up each time around the loop. If before any iteration step the *list* is empty, then the iteration is terminated. Also, if a prefix parameter *n* is given, then there will be at most *n* repetitions of processing of *str*. Finally, the ~^ directive can be used to terminate the iteration prematurely.

For example:

```
 (format nil "The winners are:~{ ~S~}."
         '(fred harry jill))
=>  "The winners are: FRED HARRY JILL."
 (format nil "Pairs:~{ <~S,~S>~}."
         '(a 1 b 2 c 3))
=>  "Pairs: <A,1> <B,2> <C,3>."
```

~:{*str*~} is similar, but the argument should be a *list* of sublists. At each repetition step, one sublist is used as the set of arguments for processing *str*; on the next repetition, a new sublist is used, whether or not all of the last sublist had been processed. For example:

```
 (format nil "Pairs:~:{ <~S,~S>~} ."
                '((a 1) (b 2) (c 3)))
=>  "Pairs: <A,1> <B,2> <C,3>."
```

~@{*str*~} is similar to ~{*str*~}, but instead of using one argument that is a list, all the remaining arguments are used as the list of arguments for the iteration. Example:

```
 (format nil "Pairs:~@{ <~S,~S>~} ." 'a 1 'b 2 'c 3)
=>  "Pairs: <A,1> <B,2> <C,3>."
```

If the iteration is terminated before all the remaining arguments are consumed, then any arguments not processed by the iteration remain to be processed by any directives following the iteration construct.

~:@{*str*~} combines the features of ~:{*str*~} and ~@{*str*~}. All the remaining arguments are used, and each one must be a *list*. On each iteration, the next argument is used as a *list* of arguments to *str*. Example:

```
 (format nil "Pairs:~:@{ <~S,~S>~} ."
             '(a 1) '(b 2) '(c 3))
=>  "Pairs: <A,1> <B,2> <C,3>."
```

Terminating the repetition construct with ~:} instead of ~} forces *str* to be processed at least once, even if the initial list of arguments is null. However, this will not override an explicit prefix parameter of zero.

If *str* is empty, then an argument is used as *str*. It must be a *format control* and precede any arguments processed by the iteration. As an example, the following are equivalent:

```
    (apply #'format stream string arguments)
 ==  (format stream "~1{~:}" string arguments)
```

This will use `string` as a formatting string. The `~1{` says it will be processed at most once, and the `~:}` says it will be processed at least once. Therefore it is processed exactly once, using `arguments` as the arguments. This case may be handled more clearly by the `~?` directive, but this general feature of `~{` is more powerful than `~?`.

22.3.7.5 Tilde Right-Brace: End of Iteration

`~}` terminates a `~{`. The consequences of using it elsewhere are undefined.

22.3.7.6 Tilde Question-Mark: Recursive Processing

The next *arg* must be a *format control*, and the one after it a *list*; both are consumed by the `~?` directive. The two are processed as a *control-string*, with the elements of the *list* as the arguments. Once the recursive processing has been finished, the processing of the control string containing the `~?` directive is resumed. Example:

```
(format nil "~? ~D" "<~A ~D>" '("Foo" 5) 7) =>  "<Foo 5> 7"
(format nil "~? ~D" "<~A ~D>" '("Foo" 5 14) 7) =>  "<Foo 5> 7"
```

Note that in the second example three arguments are supplied to the *format string* `"<~A ~D>"`, but only two are processed and the third is therefore ignored.

With the `@` modifier, only one *arg* is directly consumed. The *arg* must be a *string*; it is processed as part of the control string as if it had appeared in place of the `~@?` construct, and any directives in the recursively processed control string may consume arguments of the control string containing the `~@?` directive. Example:

```
(format nil "~@? ~D" "<~A ~D>" "Foo" 5 7) =>  "<Foo 5> 7"
(format nil "~@? ~D" "<~A ~D>" "Foo" 5 14 7) =>  "<Foo 5> 14"
```

22.3.8 FORMAT Miscellaneous Operations

22.3.8.1 Tilde Left-Paren: Case Conversion

`~(`*str*`~)`

The contained control string *str* is processed, and what it produces is subject to case conversion.

With no flags, every *uppercase character* is converted to the corresponding *lowercase character*.

`~:(` capitalizes all words, as if by **string-capitalize**.

`~@(` capitalizes just the first word and forces the rest to lower case.

`~:@(` converts every lowercase character to the corresponding uppercase character.

In this example `~@(` is used to cause the first word produced by `~@R` to be capitalized:

```
 (format nil "~@R ~(~@R~)" 14 14)
=>  "XIV xiv"
 (defun f (n) (format nil "~@(~R~) error~:P detected." n)) =>  F
 (f 0) =>  "Zero errors detected."
 (f 1) =>  "One error detected."
 (f 23) =>  "Twenty-three errors detected."
```

When case conversions appear nested, the outer conversion dominates, as illustrated in the following example:

```
 (format nil "~@(how is ~:(BOB SMITH~)?~)")
 =>  "How is bob smith?"
 NOT=>  "How is Bob Smith?"
```

22.3.8.2 Tilde Right-Paren: End of Case Conversion

`~)` terminates a `~(`. The consequences of using it elsewhere are undefined.

22.3.8.3 Tilde P: Plural

If *arg* is not **eql** to the integer `1`, a lowercase `s` is printed; if *arg* is **eql** to `1`, nothing is printed. If *arg* is a floating-point `1.0`, the `s` is printed.

`~:P` does the same thing, after doing a `~:*` to back up one argument; that is, it prints a lowercase `s` if the previous argument was not `1`.

`~@P` prints `y` if the argument is `1`, or `ies` if it is not. `~:@P` does the same thing, but backs up first.

```
 (format nil "~D tr~:@P/~D win~:P" 7 1) =>  "7 tries/1 win"
 (format nil "~D tr~:@P/~D win~:P" 1 0) =>  "1 try/0 wins"
 (format nil "~D tr~:@P/~D win~:P" 1 3) =>  "1 try/3 wins"
```

22.3.9 FORMAT Miscellaneous Pseudo-Operations

22.3.9.1 Tilde Semicolon: Clause Separator

This separates clauses in `~[` and `~<` constructs. The consequences of using it elsewhere are undefined.

22.3.9.2 Tilde Circumflex: Escape Upward

`~^`

This is an escape construct. If there are no more arguments remaining to be processed, then the immediately enclosing `~{` or `~<` construct is terminated. If there is no such enclosing construct, then the entire formatting operation is terminated. In the `~<` case, the formatting is performed, but no more segments are processed before doing the justification. `~^` may appear anywhere in a `~{` construct.

```
 (setq donestr "Done.~^ ~D warning~:P.~^ ~D error~:P.")
=>  "Done.~^ ~D warning~:P.~^ ~D error~:P."
 (format nil donestr) =>  "Done."
 (format nil donestr 3) =>  "Done. 3 warnings."
 (format nil donestr 1 5) =>  "Done. 1 warning. 5 errors."
```

If a prefix parameter is given, then termination occurs if the parameter is zero. (Hence `~^` is equivalent to `~#^`.) If two parameters are given, termination occurs if they are equal. If three parameters are given, termination occurs if the first is less than or equal to the second and the second is less than or equal to the third. Of course, this is useless if all the prefix parameters are constants; at least one of them should be a `#` or a `V` parameter.

If ~^ is used within a ~:{ construct, then it terminates the current iteration step because in the standard case it tests for remaining arguments of the current step only; the next iteration step commences immediately. ~:^ is used to terminate the iteration process. ~:^ may be used only if the command it would terminate is ~:{ or ~:@{. The entire iteration process is terminated if and only if the sublist that is supplying the arguments for the current iteration step is the last sublist in the case of ~:{, or the last **format** argument in the case of ~:@{. ~:^ is not equivalent to ~#:^; the latter terminates the entire iteration if and only if no arguments remain for the current iteration step. For example:

```
 (format nil "~:{ ~@?~:^ ...~} " '(("a") ("b"))) =>  "a...b"
```

If ~^ appears within a control string being processed under the control of a ~? directive, but not within any ~{ or ~< construct within that string, then the string being processed will be terminated, thereby ending processing of the ~? directive. Processing then continues within the string containing the ~? directive at the point following that directive.

If ~^ appears within a ~[or ~(construct, then all the commands up to the ~^ are properly selected or case-converted, the ~[or ~(processing is terminated, and the outward search continues for a ~{ or ~< construct to be terminated. For example:

```
 (setq tellstr "~@(~@[~R~]~^ ~A!~)")
=>  "~@(~@[~R~]~^ ~A!~)"
 (format nil tellstr 23) =>  "Twenty-three!"
 (format nil tellstr nil "losers") =>  " Losers!"
 (format nil tellstr 23 "losers") =>  "Twenty-three losers!"
```

Following are examples of the use of ~^ within a ~< construct.

```
 (format nil "~15<~S~;~^~S~;~^~S~>" 'foo)
=>   "            FOO"
 (format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar)
=>   "FOO         BAR"
 (format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar 'baz)
=>   "FOO   BAR   BAZ"
```

22.3.9.3 Tilde Newline: Ignored Newline

Tilde immediately followed by a *newline* ignores the *newline* and any following non-newline *whitespace*[1] characters. With a :, the *newline* is ignored, but any following *whitespace*[1] is left in place. With an @, the *newline* is left in place, but any following *whitespace*[1] is ignored. For example:

```
 (defun type-clash-error (fn nargs argnum right-type wrong-type)
   (format *error-output*
           "~&~S requires its ~:[~:R~;~*~]~
           argument to be of type ~S,~%but it was called ~
           with an argument of type ~S.~%"
           fn (eql nargs 1) argnum right-type wrong-type))
 (type-clash-error 'aref nil 2 'integer 'vector)  prints:
AREF requires its second argument to be of type INTEGER,
but it was called with an argument of type VECTOR.
NIL
 (type-clash-error 'car 1 1 'list 'short-float)  prints:
CAR requires its argument to be of type LIST,
but it was called with an argument of type SHORT-FLOAT.
NIL
```

Note that in this example newlines appear in the output only as specified by the ~& and ~% directives; the actual newline characters in the control string are suppressed because each is preceded by a tilde.

22.3.10 Additional Information about FORMAT Operations

22.3.10.1 Nesting of FORMAT Operations

The case-conversion, conditional, iteration, and justification constructs can contain other formatting constructs by bracketing them. These constructs must nest properly with respect to each other. For example, it is not legitimate to put the start of a case-conversion construct in each arm of a conditional and the end of the case-conversion construct outside the conditional:

```
 (format nil "~:[abc~:@(def~;ghi~
:@(jkl~]mno~)" x) ;Invalid!
```

This notation is invalid because the `~[...~;...~]` and `~(...~)` constructs are not properly nested.

The processing indirection caused by the `~?` directive is also a kind of nesting for the purposes of this rule of proper nesting. It is not permitted to start a bracketing construct within a string processed under control of a `~?` directive and end the construct at some point after the `~?` construct in the string containing that construct, or vice versa. For example, this situation is invalid:

```
 (format nil "~@?ghi~)" "abc~@(def") ;Invalid!
```

This notation is invalid because the `~?` and `~(...~)` constructs are not properly nested.

22.3.10.2 Missing and Additional FORMAT Arguments

The consequences are undefined if no *arg* remains for a directive requiring an argument. However, it is permissible for one or more *args* to remain unprocessed by a directive; such *args* are ignored.

22.3.10.3 Additional FORMAT Parameters

The consequences are undefined if a format directive is given more parameters than it is described here as accepting.

22.3.10.4 Undefined FORMAT Modifier Combinations

The consequences are undefined if *colon* or *at-sign* modifiers are given to a directive in a combination not specifically described here as being meaningful.

22.3.11 Examples of FORMAT

```
 (format nil "foo") =>  "foo"
 (setq x 5) =>  5
 (format nil "The answer is ~D." x) =>  "The answer is 5."
 (format nil "The answer is ~3D." x) =>  "The answer is   5."
 (format nil "The answer is ~3,'0D." x) =>  "The answer is 005."
 (format nil "The answer is ~:D." (expt 47 x))
=>  "The answer is 229,345,007."
 (setq y "elephant") =>  "elephant"
 (format nil "Look at the ~A!" y) =>  "Look at the elephant!"
 (setq n 3) =>  3
 (format nil "~D item~:P found." n) =>  "3 items found."
 (format nil "~R dog~:[s are~; is~] here." n (= n 1))
=>  "three dogs are here."
```

```
 (format nil "~R dog~:*~[s are~; is~:;s are~] here." n)
=>  "three dogs are here."
 (format nil "Here ~[are~;is~:;are~] ~:*~R pupp~:@P." n)
=>  "Here are three puppies."

 (defun foo (x)
   (format nil "~6,2F|~6,2,1,'*F|~6,2,,'?F|~6F|~,2F|~F"
           x x x x x x)) =>  FOO
 (foo 3.14159)  =>  "  3.14| 31.42|  3.14|3.1416|3.14|3.14159"
 (foo -3.14159) =>  " -3.14|-31.42| -3.14|-3.142|-3.14|-3.14159"
 (foo 100.0)    =>  "100.00|******|100.00| 100.0|100.00|100.0"
 (foo 1234.0)   =>  "1234.00|******|??????|1234.0|1234.00|1234.0"
 (foo 0.006)    =>  "  0.01|  0.06|  0.01| 0.006|0.01|0.006"

 (defun foo (x)
    (format nil
           "~9,2,1,,'*E|~10,3,2,2,'?,,'$E|~
            ~9,3,2,-2,'%@E|~9,2E"
           x x x x))
 (foo 3.14159)  =>  "  3.14E+0| 31.42$-01|+.003E+03|  3.14E+0"
 (foo -3.14159) =>  " -3.14E+0|-31.42$-01|-.003E+03| -3.14E+0"
 (foo 1100.0)   =>  "  1.10E+3| 11.00$+02|+.001E+06|  1.10E+3"
 (foo 1100.0L0) =>  "  1.10L+3| 11.00$+02|+.001L+06|  1.10L+3"
 (foo 1.1E13)   =>  "*********| 11.00$+12|+.001E+16| 1.10E+13"
 (foo 1.1L120)  =>  "*********|??????????|%%%%%%%%%|1.10L+120"
 (foo 1.1L1200) =>  "*********|??????????|%%%%%%%%%|1.10L+1200"
```

As an example of the effects of varying the scale factor, the code

```
 (dotimes (k 13)
   (format t "~%Scale factor ~2D: |~13,6,2,VE|"
           (- k 5) (- k 5) 3.14159))
```

produces the following output:

```
Scale factor -5: | 0.000003E+06|
Scale factor -4: | 0.000031E+05|
Scale factor -3: | 0.000314E+04|
Scale factor -2: | 0.003142E+03|
Scale factor -1: | 0.031416E+02|
Scale factor  0: | 0.314159E+01|
Scale factor  1: | 3.141590E+00|
Scale factor  2: | 31.41590E-01|
Scale factor  3: | 314.1590E-02|
Scale factor  4: | 3141.590E-03|
Scale factor  5: | 31415.90E-04|
Scale factor  6: | 314159.0E-05|
Scale factor  7: | 3141590.E-06|

 (defun foo (x)
   (format nil "~9,2,1,,'*G|~9,3,2,3,'?,,'$G|~9,3,2,0,'%G|~9,2G"
           x x x x))
 (foo 0.0314159) =>  "  3.14E-2|314.2$-04|0.314E-01|  3.14E-2"
 (foo 0.314159)  =>  "  0.31   |0.314    |0.314    | 0.31    "
 (foo 3.14159)   =>  "   3.1   | 3.14    | 3.14    |  3.1    "
 (foo 31.4159)   =>  "   31.   | 31.4    | 31.4    |  31.    "
 (foo 314.159)   =>  "  3.14E+2| 314.    | 314.    |  3.14E+2"
 (foo 3141.59)   =>  "  3.14E+3|314.2$+01|0.314E+04|  3.14E+3"
 (foo 3141.59L0) =>  "  3.14L+3|314.2$+01|0.314L+04|  3.14L+3"
 (foo 3.14E12)   =>  "*********|314.0$+10|0.314E+13| 3.14E+12"
 (foo 3.14L120)  =>  "*********|?????????|%%%%%%%%%|3.14L+120"
 (foo 3.14L1200) =>  "*********|?????????|%%%%%%%%%|3.14L+1200"
```

```
(format nil "~10<foo~;bar~>")   =>  "foo    bar"
(format nil "~10:<foo~;bar~>")  =>  "  foo  bar"
(format nil "~10<foobar~>")     =>  "    foobar"
(format nil "~10:<foobar~>")    =>  "    foobar"
(format nil "~10:@<foo~;bar~>") =>  "  foo bar "
(format nil "~10@<foobar~>")    =>  "foobar    "
(format nil "~10:@<foobar~>")   =>  "  foobar  "

 (FORMAT NIL "Written to ~A." #P"foo.bin")
 =>  "Written to foo.bin."
```

22.3.12 Notes about FORMAT

Formatted output is performed not only by **format**, but by certain other functions that accept a *format control* the way **format** does. For example, error-signaling functions such as **cerror** accept *format controls*.

Note that the meaning of **nil** and **t** as destinations to **format** are different than those of **nil** and **t** as *stream designators*.

The `~^` should appear only at the beginning of a `~<` clause, because it aborts the entire clause in which it appears (as well as all following clauses).

23. Reader

23.1 Reader Concepts

23.1.1 Dynamic Control of the Lisp Reader

Various aspects of the *Lisp reader* can be controlled dynamically. See Section 2.1.1 (Readtables) and Section 2.1.2 (Variables that affect the Lisp Reader).

23.1.2 Effect of Readtable Case on the Lisp Reader

The *readtable case* of the *current readtable* affects the *Lisp reader* in the following ways:

`:upcase`
: When the *readtable case* is `:upcase`, unescaped constituent *characters* are converted to *uppercase*, as specified in Section 2.2 (Reader Algorithm).

`:downcase`
: When the *readtable case* is `:downcase`, unescaped constituent *characters* are converted to *lowercase*.

`:preserve`
: When the *readtable case* is `:preserve`, the case of all *characters* remains unchanged.

`:invert`
: When the *readtable case* is `:invert`, then if all of the unescaped letters in the extended token are of the same *case*, those (unescaped) letters are converted to the opposite *case*.

23.1.2.1 Examples of Effect of Readtable Case on the Lisp Reader

```
(defun test-readtable-case-reading ()
  (let ((*readtable* (copy-readtable nil)))
    (format t "READTABLE-CASE  Input   Symbol-name~
             ~%-----------------------------------~
             ~%")
    (dolist (readtable-case '(:upcase :downcase :preserve :invert))
      (setf (readtable-case *readtable*) readtable-case)
```

```
  (dolist (input '("ZEBRA" "Zebra" "zebra"))
    (format t "~&:~A~16T~A~24T~A"
            (string-upcase readtable-case)
            input
            (symbol-name (read-from-string input)))))))
```

The output from `(test-readtable-case-reading)` should be as follows:

```
READTABLE-CASE     Input Symbol-name
-----------------------------------
   :UPCASE         ZEBRA   ZEBRA
   :UPCASE         Zebra   ZEBRA
   :UPCASE         zebra   ZEBRA
   :DOWNCASE       ZEBRA   zebra
   :DOWNCASE       Zebra   zebra
   :DOWNCASE       zebra   zebra
   :PRESERVE       ZEBRA   ZEBRA
   :PRESERVE       Zebra   Zebra
   :PRESERVE       zebra   zebra
   :INVERT         ZEBRA   zebra
   :INVERT         Zebra   Zebra
   :INVERT         zebra   ZEBRA
```

23.1.3 Argument Conventions of Some Reader Functions

23.1.3.1 The EOF-ERROR-P argument

Eof-error-p in input function calls controls what happens if input is from a file (or any other input source that has a definite end) and the end of the file is reached. If *eof-error-p* is *true* (the default), an error of *type* **end-of-file** is signaled at end of file. If it is *false*, then no error is signaled, and instead the function returns *eof-value*.

Functions such as **read** that read the representation of an *object* rather than a single character always signals an error, regardless of *eof-error-p*, if the file ends in the middle of an object representation. For example, if a file does not contain enough right parentheses to balance the left parentheses in it, **read** signals an error. If a file ends in a *symbol* or a *number* immediately followed by end-of-file, **read** reads the *symbol* or *number* successfully and when called again will act according to *eof-error-p*. Similarly, the *function* **read-line** successfully reads the last line of a file even if that line is terminated by end-of-file rather than the newline character. Ignorable text, such as lines containing only *whitespace*[2] or comments, are not considered to begin an *object*; if **read** begins to read an *expression* but sees only such ignorable text, it does not consider the file to end in the middle of an *object*. Thus an *eof-error-p* argument controls what happens when the file ends between *objects*.

23.1.3.2 The RECURSIVE-P argument

If *recursive-p* is supplied and not **nil**, it specifies that this function call is not an outermost call to **read** but an embedded call, typically from a *reader macro function*. It is important to distinguish such recursive calls for three reasons.

1. An outermost call establishes the context within which the *#n=* and *#n#* syntax is scoped. Consider, for example, the expression

```
(cons '#3=(p q r) '(x y . #3#))
```

If the *single-quote reader macro* were defined in this way:

```
(set-macro-character #\'       ;incorrect
   #'(lambda (stream char)
        (declare (ignore char))
        (list 'quote (read stream))))
```

then each call to the *single-quote reader macro function* would establish independent contexts for the scope of **read** information, including the scope of identifications between markers like "#3=" and "#3#". However, for this expression, the scope was clearly intended to be determined by the outer set of parentheses, so such a definition would be incorrect. The correct way to define the *single-quote reader macro* uses *recursive-p*:

```
(set-macro-character #\'       ;correct
   #'(lambda (stream char)
        (declare (ignore char))
        (list 'quote (read stream t nil t))))
```

2. A recursive call does not alter whether the reading process is to preserve *whitespace*[2] or not (as determined by whether the outermost call was to **read** or **read-preserving-whitespace**). Suppose again that *single-quote* were to be defined as shown above in the incorrect definition. Then a call to **read-preserving-whitespace** that read the expression `'foo<Space>` would fail to preserve the space character following the symbol `foo` because the *single-quote reader macro function* calls **read**, not **read-preserving-whitespace**, to read the following expression (in this case `foo`). The correct definition, which passes the value *true* for *recursive-p* to **read**, allows the outermost call to determine whether *whitespace*[2] is preserved.
3. When end-of-file is encountered and the *eof-error-p* argument is not **nil**, the kind of error that is signaled may depend on the value of *recursive-p*. If *recursive-p* is *true*, then the end-of-file is deemed to have occurred within the middle of a printed representation; if *recursive-p* is *false*, then the end-of-file may be deemed to have occurred between *objects* rather than within the middle of one.

24. System Construction

24.1 System Construction Concepts

24.1.1 Loading

To **load** a *file* is to treat its contents as *code* and *execute* that *code*. The *file* may contain *source code* or *compiled code*.

A *file* containing *source code* is called a *source file*. *Loading* a *source file* is accomplished essentially by sequentially *reading*[2] the *forms* in the file, *evaluating* each immediately after it is *read*.

A *file* containing *compiled code* is called a *compiled file*. *Loading* a *compiled file* is similar to *loading* a *source file*, except that the *file* does not contain text but rather an *implementation-dependent* representation of pre-digested *expressions* created by the *compiler*. Often, a *compiled file* can be *loaded* more quickly than a *source file*. See Section 3.2 (Compilation).

The way in which a *source file* is distinguished from a *compiled file* is *implementation-dependent*.

24.1.2 Features

A *feature* is an aspect or attribute of Common Lisp, of the *implementation*, or of the *environment*. A *feature* is identified by a *symbol*.

A *feature* is said to be *present* in a *Lisp image* if and only if the *symbol* naming it is an *element* of the *list* held by the *variable* ***features***, which is called the *features list*.

24.1.2.1 Feature Expressions

Boolean combinations of *features*, called *feature expressions*, are used by the `#+` and `#-` *reader macros* in order to direct conditional *reading* of *expressions* by the *Lisp reader*.

The rules for interpreting a *feature expression* are as follows:

feature
: If a *symbol* naming a *feature* is used as a *feature expression*, the *feature expression* succeeds if that *feature* is *present*; otherwise it fails.

`(not` *feature-conditional*`)`
: A **not** *feature expression* succeeds if its argument *feature-conditional* fails; otherwise, it succeeds.

`(and` *feature-conditional*`*)`
: An **and** *feature expression* succeeds if all of its argument *feature-conditionals* succeed; otherwise, it fails.

`(or` *feature-conditional*`*)`
: An **or** *feature expression* succeeds if any of its argument *feature-conditionals* succeed; otherwise, it fails.

24.1.2.1.1 Examples of Feature Expressions

For example, suppose that in *implementation* A, the *features* `spice` and `perq` are *present*, but the *feature* `lispm` is not *present*; in *implementation* B, the feature `lispm` is *present*, but the *features* `spice` and `perq` are not *present*; and in *implementation* C, none of the features `spice`, *lispm*, or `perq` are *present*. The next figure shows some sample *expressions*, and how they would be *read*[2] in these *implementations*.

```
(cons #+spice "Spice" #-spice "Lispm" x)

in implementation A ...  (CONS "Spice" X)
in implementation B ...  (CONS "Lispm" X)
in implementation C ...  (CONS "Lispm" X)

(cons #+spice "Spice" #+LispM "Lispm" x)

in implementation A ...  (CONS "Spice" X)
in implementation B ...  (CONS "Lispm" X)
in implementation C ...  (CONS X)

(setq a '(1 2 #+perq 43 #+(not perq) 27))

in implementation A ...  (SETQ A '(1 2 43))
in implementation B ...  (SETQ A '(1 2 27))
in implementation C ...  (SETQ A '(1 2 27))

(let ((a 3) #+(or spice lispm) (b 3)) (foo a))

in implementation A ...  (LET ((A 3) (B 3)) (FOO A))
in implementation B ...  (LET ((A 3) (B 3)) (FOO A))
in implementation C ...  (LET ((A 3)) (FOO A))

(cons #+Lispm "#+Spice" #+Spice "foo" #-(or Lispm Spice) 7 x)

in implementation A ...  (CONS "foo" X)
in implementation B ...  (CONS "#+Spice" X)
in implementation C ...  (CONS 7 X)
```

25. Environment

25.1 The External Environment

25.1.1 Top level loop

The top level loop is the Common Lisp mechanism by which the user normally interacts with the Common Lisp system. This loop is sometimes referred to as the *Lisp read-eval-print loop* because it typically consists of an endless loop that reads an expression, evaluates it and prints the results.

The top level loop is not completely specified; thus the user interface is *implementation-defined*. The top level loop prints all values resulting from the evaluation of a *form*. The next figure lists variables that are maintained by the *Lisp read-eval-print loop*.

```
*    +    /    -
**   ++   //
***  +++  ///
```

Figure 25-1. Variables maintained by the Read-Eval-Print Loop

25.1.2 Debugging Utilities

The next figure shows *defined names* relating to debugging.

```
*debugger-hook*  documentation    step
apropos          dribble          time
apropos-list     ed               trace
break            inspect          untrace
describe         invoke-debugger
```

Figure 25-2. Defined names relating to debugging

25.1.3 Environment Inquiry

Environment inquiry *defined names* provide information about the hardware and software configuration on which a Common Lisp program is being executed.

The next figure shows *defined names* relating to environment inquiry.

```
*features*                   machine-instance  short-site-name
lisp-implementation-type     machine-type      software-type
lisp-implementation-version  machine-version   software-version
long-site-name               room
```

Figure 25-3. Defined names relating to environment inquiry.

25.1.4 Time

Time is represented in four different ways in Common Lisp: *decoded time*, *universal time*, *internal time*, and seconds. *Decoded time* and *universal time* are used primarily to represent calendar time, and are precise only to one second. *Internal time* is used primarily to represent measurements of computer time (such as run time) and is precise to some *implementation-dependent* fraction of a second called an *internal time unit*, as specified by **internal-time-units-per-second**. An *internal time* can be used for either *absolute* and *relative time* measurements. Both a *universal time* and a *decoded time* can be used only for *absolute time* measurements. In the case of one function, **sleep**, time intervals are represented as a non-negative *real* number of seconds.

The next figure shows *defined names* relating to *time*.

```
decode-universal-time   get-internal-run-time
encode-universal-time   get-universal-time
get-decoded-time        internal-time-units-per-second
get-internal-real-time  sleep
```

Figure 25-4. Defined names involving Time.

25.1.4.1 Decoded Time

A *decoded time* is an ordered series of nine values that, taken together, represent a point in calendar time (ignoring *leap seconds*):

Second
An *integer* between 0 and 59, inclusive.

Minute
An *integer* between 0 and 59, inclusive.

Hour
An *integer* between 0 and 23, inclusive.

Date
An *integer* between 1 and 31, inclusive (the upper limit actually depends on the month and year, of course).

Month
An *integer* between 1 and 12, inclusive; 1 means January, 2 means February, and so on; 12 means December.

Year
An *integer* indicating the year A.D. However, if this *integer* is between 0 and 99, the "obvious" year is used; more precisely, that year is assumed that is equal to the *integer* modulo 100 and within fifty years of the current year (inclusive backwards and exclusive forwards). Thus, in the year 1978, year 28 is 1928 but year 27 is 2027. (Functions that return time in this format always return a full year number.)

Day of week
An *integer* between 0 and 6, inclusive; 0 means Monday, 1 means Tuesday, and so on; 6 means Sunday.

Daylight saving time flag
A *generalized boolean* that, if *true*, indicates that daylight saving time is in effect.

Time zone
A *time zone*.

The next figure shows *defined names* relating to *decoded time*.

```
decode-universal-time  get-decoded-time
```

Figure 25-5. Defined names involving time in Decoded Time.

25.1.4.2 Universal Time

Universal time is an *absolute time* represented as a single non-negative *integer*---the number of seconds since midnight, January 1, 1900 GMT (ignoring *leap seconds*). Thus the time 1 is 00:00:01 (that is, 12:00:01 a.m.) on January 1, 1900 GMT. Similarly, the time 2398291201 corresponds to time 00:00:01 on January 1, 1976 GMT. Recall that the year 1900 was not a leap year; for the purposes of Common Lisp, a year is a leap year if and only if its number is divisible by 4, except that years divisible by 100 are not leap years, except that years divisible by 400 are leap years. Therefore the year 2000 will be a leap year. Because *universal time* must be a non-negative *integer*, times before the base time of midnight, January 1, 1900 GMT cannot be processed by Common Lisp.

```
decode-universal-time  get-universal-time
encode-universal-time
```

Figure 25-6. Defined names involving time in Universal Time.

25.1.4.3 Internal Time

Internal time represents time as a single *integer*, in terms of an *implementation-dependent* unit called an *internal time unit*. Relative time is measured as a number of these units. Absolute time is relative to an arbitrary time base.

The next figure shows *defined names* related to *internal time*.

```
get-internal-real-time  internal-time-units-per-second
get-internal-run-time
```

Figure 25-7. Defined names involving time in Internal Time.

25.1.4.4 Seconds

One function, **sleep**, takes its argument as a non-negative *real* number of seconds. Informally, it may be useful to think of this as a *relative universal time*, but it differs in one important way: *universal times* are always non-negative *integers*, whereas the argument to **sleep** can be any kind of non-negative *real*, in order to allow for the possibility of fractional seconds.

```
sleep
```

Figure 25-8. Defined names involving time in Seconds.

26. Glossary

26.1 Glossary

Non-alphabetic

() [’nil], *n.* an alternative notation for writing the symbol **nil**, used to emphasize the use of *nil* as an *empty list*.

A

absolute *adj.* 1. (of a *time*) representing a specific point in time. 2. (of a *pathname*) representing a specific position in a directory hierarchy. See *relative*.

access *n., v.t.* 1. *v.t.* (a *place*, or *array*) to *read*[1] or *write*[1] the *value* of the *place* or an *element* of the *array*. 2. *n.* (of a *place*) an attempt to *access*[1] the *value* of the *place*.

accessibility *n.* the state of being *accessible*.

accessible *adj.* 1. (of an *object*) capable of being *referenced*. 2. (of *shared slots* or *local slots* in an *instance* of a *class*) having been defined by the *class* of the *instance* or *inherited* from a *superclass* of that *class*. 3. (of a *symbol* in a *package*) capable of being *referenced* without a *package prefix* when that *package* is current, regardless of whether the *symbol* is *present* in that *package* or is *inherited*.

accessor *n.* an *operator* that performs an *access*. See *reader* and *writer*.

active *adj.* 1. (of a *handler*, a *restart*, or a *catch tag*) having been *established* but not yet *disestablished*. 2. (of an *element* of an *array*) having an index that is greater than or equal to zero, but less than the *fill pointer* (if any). For an *array* that has no *fill pointer*, all *elements* are considered *active*.

actual adjustability *n.* (of an *array*) a *generalized boolean* that is associated with the *array*, representing whether the *array* is *actually adjustable*. See also *expressed adjustability* and **adjustable-array-p**.

actual argument *n. Trad.* an *argument*.

actual array element type *n.* (of an *array*) the *type* for which the *array* is actually specialized, which is the *upgraded array element type* of the *expressed array element type* of the *array*. See the *function* **array-element-type**.

actual complex part type *n.* (of a *complex*) the *type* in which the real and imaginary parts of the *complex* are actually represented, which is the *upgraded complex part type* of the *expressed complex part type* of the *complex*.

actual parameter *n. Trad.* an *argument*.

actually adjustable *adj.* (of an *array*) such that **adjust-array** can adjust its characteristics by direct modification. A *conforming program* may depend on an *array* being *actually adjustable* only if either that *array* is known to have been *expressly adjustable* or if that *array* has been explicitly tested by **adjustable-array-p**.

adjustability *n.* (of an *array*) 1. *expressed adjustability*. 2. *actual adjustability*.

adjustable *adj.* (of an *array*) 1. *expressly adjustable*. 2. *actually adjustable*.

after method *n.* a *method* having the *qualifier* `:after`.

alist [’ay,list], *n.* an *association list*.

alphabetic *n., adj.* 1. *adj.* (of a *character*) being one of the *standard characters* `A` through `Z` or `a` through `z`, or being any *implementation-defined* character that has *case*, or being some other *graphic character* defined by the *implementation* to be *alphabetic*[1]. 2. a. *n.* one of several possible *constituent traits* of a *character*. For details, see Section 2.1.4.1 (Constituent Characters) and Section 2.2 (Reader Algorithm). b. *adj.* (of a *character*) being a *character* that has *syntax type constituent* in the *current readtable* and that has the *constituent trait alphabetic*[2a]. See Figure 2-8.

alphanumeric *adj.* (of a *character*) being either an *alphabetic*[1] *character* or a *numeric* character.

ampersand *n.* the *standard character* that is called "ampersand" (`&`). See Figure 2-5.

anonymous *adj.* 1. (of a *class* or *function*) having no *name* 2. (of a *restart*) having a *name* of **nil**.

apparently uninterned *adj.* having a *home package* of **nil**. (An *apparently uninterned symbol* might or might not be an *uninterned symbol*. *Uninterned symbols* have a *home package* of **nil**, but *symbols* which have been *uninterned* from their *home package* also have a *home package* of **nil**, even though they might still be *interned* in some other *package*.)

applicable *adj.* 1. (of a *handler*) being an *applicable handler*. 2. (of a *method*) being an *applicable method*. 3. (of a *restart*) being an *applicable restart*.

applicable handler *n.* (for a *condition* being *signaled*) an *active handler* for which the associated type contains the *condition*.

applicable method *n.* (of a *generic function* called with *arguments*) a *method* of the *generic function* for which the *arguments* satisfy the *parameter specializers* of that *method*. See Section 7.6.6.1.1 (Selecting the Applicable Methods).

applicable restart *n.* 1. (for a *condition*) an *active handler* for which the associated test returns *true* when given the *condition* as an argument. 2. (for no particular *condition*) an *active handler* for which the associated test returns *true* when given **nil** as an argument.

apply *v.t.* (a *function* to a *list*) to *call* the *function* with arguments that are the *elements* of the *list*. "Applying the function + to a list of integers returns the sum of the elements of that list."

argument *n.* 1. (of a *function*) an *object* which is offered as data to the *function* when it is *called*. 2. (of a *format control*) a *format argument*.

argument evaluation order *n.* the order in which *arguments* are evaluated in a function call. "The argument evaluation order for Common Lisp is left to right." See Section 3.1 (Evaluation).

argument precedence order *n.* the order in which the *arguments* to a *generic function* are considered when sorting the *applicable methods* into precedence order.

around method *n.* a *method* having the *qualifier* `:around`.

array *n.* an *object* of *type* **array**, which serves as a container for other *objects* arranged in a Cartesian coordinate system.

array element type *n.* (of an *array*) 1. a *type* associated with the *array*, and of which all *elements* of the *array* are constrained to be members. 2. the *actual array element type* of the *array*. 3. the *expressed array element type* of the *array*.

array total size *n.* the total number of *elements* in an *array*, computed by taking the product of the *dimensions* of the *array*. (The size of a zero-dimensional *array* is therefore one.)

assign *v.t.* (a *variable*) to change the *value* of the *variable* in a *binding* that has already been *established*. See the *special operator* **setq**.

association list *n.* a *list* of *conses* representing an association of *keys* with *values*, where the *car* of each *cons* is the *key* and the *cdr* is the *value* associated with that *key*.

asterisk *n.* the *standard character* that is variously called "asterisk" or "star" (`*`). See Figure 2-5.

at-sign *n.* the *standard character* that is variously called "commercial at" or "at sign" (@). See Figure 2-5.

atom *n.* any *object* that is not a *cons*. "A vector is an atom."

atomic *adj.* being an *atom*. "The number 3, the symbol `foo`, and **nil** are atomic."

atomic type specifier *n.* a *type specifier* that is *atomic*. For every *atomic type specifier*, *x*, there is an equivalent *compound type specifier* with no arguments supplied, (*x*).

attribute *n.* (of a *character*) a program-visible aspect of the *character*. The only *standardized attribute* of a *character* is its *code*[2], but *implementations* are permitted to have additional *implementation-defined attributes*. See Section 13.1.3 (Character Attributes). "An implementation that support fonts might make font information an attribute of a character, while others might represent font information separately from characters."

aux variable *n.* a *variable* that occurs in the part of a *lambda list* that was introduced by `&aux`. Unlike all other *variables* introduced by a *lambda-list*, *aux variables* are not *parameters*.

auxiliary method *n.* a member of one of two sets of *methods* (the set of *primary methods* is the other) that form an exhaustive partition of the set of *methods* on the *method*'s *generic function*. How these sets are determined is dependent on the *method combination* type; see Section 7.6.2 (Introduction to Methods).

B

backquote *n.* the *standard character* that is variously called "grave accent" or "backquote" (`‘`). See Figure 2-5.

backslash *n.* the *standard character* that is variously called "reverse solidus" or "backslash" (\). See Figure 2-5.

base character *n.* a *character* of *type* **base-char**.

base string *n.* a *string* of *type* **base-string**.

before method *n.* a *method* having the *qualifier* `:before`.

bidirectional *adj.* (of a *stream*) being both an *input stream* and an *output stream*.

binary *adj.* 1. (of a *stream*) being a *stream* that has an *element type* that is a *subtype* of *type* **integer**. The most fundamental operation on a *binary input stream* is **read-byte** and on a *binary output stream* is **write-byte**. See *character*. 2. (of a *file*) having been created by opening a *binary stream*. (It is *implementation-dependent* whether this is an detectable aspect of the *file*, or whether any given *character file* can be treated as a *binary file*.)

bind *v.t.* (a *variable*) to establish a *binding* for the *variable*.

binding *n.* an association between a *name* and that which the *name* denotes. "A lexical binding is a lexical association between a name and its value." When the term *binding* is qualified by the name of a *namespace*, such as "variable" or "function," it restricts the binding to the indicated namespace, as in: "**let** establishes variable bindings." or "**let** establishes bindings of variables."

bit *n.* an *object* of *type* **bit**; that is, the *integer* `0` or the *integer* `1`.

bit array *n.* a specialized *array* that is of *type* `(array bit)`, and whose elements are of *type* **bit**.

bit vector *n.* a specialized *vector* that is of *type* **bit-vector**, and whose elements are of *type* **bit**.

bit-wise logical operation specifier *n.* an *object* which names one of the sixteen possible bit-wise logical operations that can be performed by the **boole** function, and which is the *value* of exactly one of the *constant variables* **boole-clr**, **boole-set**, **boole-1**, **boole-2**, **boole-c1**, **boole-c2**, **boole-and**, **boole-ior**, **boole-xor**, **boole-eqv**, **boole-nand**, **boole-nor**, **boole-andc1**, **boole-andc2**, **boole-orc1**, or **boole-orc2**.

block *n.* a named lexical *exit point*, *established* explicitly by **block** or implicitly by *operators* such as **loop**, **do** and **prog**, to which control and values may be transfered by using a **return-from** *form* with the name of the *block*.

block tag *n.* the *symbol* that, within the *lexical scope* of a **block** *form*, names the *block established* by that **block** *form*. See **return** or **return-from**.

boa lambda list *n.* a *lambda list* that is syntactically like an *ordinary lambda list*, but that is processed in "**b**y **o**rder of **a**rgument" style. See Section 3.4.6 (Boa Lambda Lists).

body parameter *n.* a *parameter* available in certain *lambda lists* which from the point of view of *conforming programs* is like a *rest parameter* in every way except that it is introduced by `&body` instead of `&rest`. (*Implementations* are permitted to provide extensions which distinguish *body parameters* and *rest parameters*---e.g., the *forms* for *operators* which were defined using a *body parameter* might be pretty printed slightly differently than *forms* for *operators* which were defined using *rest parameters*.)

boolean *n.* an *object* of *type* **boolean**; that is, one of the following *objects*: the symbol **t** (representing *true*), or the symbol **nil** (representing *false*). See *generalized boolean*.

boolean equivalent *n.* (of an *object* O1) any *object* O2 that has the same truth value as O1 when both O1 and O2 are viewed as *generalized booleans*.

bound *adj.*, *v.t.* 1. *adj.* having an associated denotation in a *binding*. "The variables named by a **let** are bound within its body." See *unbound*. 2. *adj.* having a local *binding* which *shadows*[2] another. "The variable ***print-escape*** is bound while in the **princ** function." 3. *v.t.* the past tense of *bind*.

bound declaration *n.* a *declaration* that refers to or is associated with a *variable* or *function* and that appears within the *special form* that *establishes* the *variable* or *function*, but before the body of that *special form* (specifically, at the head of that *form*'s body). (If a *bound declaration* refers to a *function binding* or a *lexical variable binding*, the *scope* of the *declaration* is exactly the *scope* of that *binding*. If the *declaration* refers to a *dynamic variable binding*, the *scope* of the *declaration* is what the *scope* of the *binding* would have been if it were lexical rather than dynamic.)

bounded *adj.* (of a *sequence* S, by an ordered pair of *bounding indices* istart and iend) restricted to a subrange of the *elements* of S that includes each *element* beginning with (and including) the one indexed by istart and continuing up to (but not including) the one indexed by iend.

bounding index *n.* (of a *sequence* with *length* n) either of a conceptual pair of *integers*, istart and iend, respectively called the "lower bounding index" and "upper bounding index", such that 0 <=istart <=iend <=n, and which therefore delimit a subrange of the *sequence bounded* by istart and iend.

bounding index designator (for a *sequence*) one of two *objects* that, taken together as an ordered pair, behave as a *designator* for *bounding indices* of the *sequence*; that is, they denote *bounding indices* of the *sequence*, and are either: an *integer* (denoting itself) and **nil** (denoting the *length* of the *sequence*), or two *integers* (each denoting themselves).

break loop *n.* A variant of the normal *Lisp read-eval-print loop* that is recursively entered, usually because the ongoing *evaluation* of some other *form* has been suspended for the purpose of debugging. Often, a *break loop* provides the ability to exit in such a way as to continue the suspended computation. See the *function* **break**.

broadcast stream *n.* an *output stream* of *type* **broadcast-stream**.

built-in class *n.* a *class* that is a *generalized instance* of *class* **built-in-class**.

built-in type *n.* one of the *types* in Figure 4-2.

byte *n.* 1. adjacent bits within an *integer*. (The specific number of bits can vary from point to point in the program; see the *function* **byte**.) 2. an integer in a specified range. (The specific range can vary from point to point in the program; see the *functions* **open** and **write-byte**.)

byte specifier *n.* An *object* of *implementation-dependent* nature that is returned by the *function* **byte** and that specifies the range of bits in an *integer* to be used as a *byte* by *functions* such as **ldb**.

C

cadr ['ka,duhr], *n.* (of an *object*) the *car* of the *cdr* of that *object*.

call *v.t.*, *n.* 1. *v.t.* (a *function* with *arguments*) to cause the *code* represented by that *function* to be *executed* in an *environment* where *bindings* for the *values* of its *parameters* have been *established* based on the *arguments*. "Calling the function `+` with the arguments `5` and `1` yields a value of `6`." 2. *n.* a *situation* in which a *function* is called.

captured initialization form *n.* an *initialization form* along with the *lexical environment* in which the *form* that defined the *initialization form* was *evaluated*. "Each newly added shared slot is set to the result of evaluating the captured initialization form for the slot that was specified in the **defclass** form for the new class."

car *n.* 1. a. (of a *cons*) the component of a *cons* corresponding to the first *argument* to **cons**; the other component is the *cdr*. "The function **rplaca** modifies the car of a cons." b. (of a *list*) the first *element* of the *list*, or **nil** if the *list* is the *empty list*. 2. the *object* that is held in the *car*[1]. "The function **car** returns the car of a cons."

case *n.* (of a *character*) the property of being either *uppercase* or *lowercase*. Not all *characters* have *case*. "The characters `#\A` and `#\a` have case, but the character `#\$` has no case." See Section 13.1.4.3 (Characters With Case) and the *function* **both-case-p**.

case sensitivity mode *n.* one of the *symbols* `:upcase`, `:downcase`, `:preserve`, or `:invert`.

catch *n.* an *exit point* which is *established* by a **catch** *form* within the *dynamic scope* of its body, which is named by a *catch tag*, and to which control and *values* may be *thrown*.

catch tag *n.* an *object* which names an *active catch*. (If more than one *catch* is active with the same *catch tag*, it is only possible to *throw* to the innermost such *catch* because the outer one is *shadowed*[2].)

cddr [’kduh,duhr] or [’kuh,dduhr], *n.* (of an *object*) the *cdr* of the *cdr* of that *object*.

cdr [’k,duhr], *n.* 1. a. (of a *cons*) the component of a *cons* corresponding to the second *argument* to **cons**; the other component is the *car*. "The function **rplacd** modifies the cdr of a cons." b. (of a *list* L1) either the *list* L2 that contains the *elements* of L1 that follow after the first, or else **nil** if L1 is the *empty list*. 2. the *object* that is held in the *cdr*[1]. "The function **cdr** returns the cdr of a cons."

cell *n. Trad.* (of an *object*) a conceptual *slot* of that *object*. The *dynamic variable* and global *function bindings* of a *symbol* are sometimes referred to as its *value cell* and *function cell*, respectively.

character *n., adj.* 1. *n.* an *object* of *type* **character**; that is, an *object* that represents a unitary token in an aggregate quantity of text; see Section 13.1 (Character Concepts). 2. *adj.* a. (of a *stream*) having an *element type* that is a *subtype* of *type* **character**. The most fundamental operation on a *character input stream* is **read-char** and on a *character output stream* is **write-char**. See *binary*. b. (of a *file*) having been created by opening a *character stream*. (It is *implementation-dependent* whether this is an inspectable aspect of the *file*, or whether any given *binary file* can be treated as a *character file*.)

character code *n.* 1. one of possibly several *attributes* of a *character*. 2. a non-negative *integer* less than the *value* of **char-code-limit** that is suitable for use as a *character code*[1].

character designator *n.* a *designator* for a *character*; that is, an *object* that denotes a *character* and that is one of: a *designator* for a *string* of *length* one (denoting the *character* that is its only *element*), or a *character* (denoting itself).

circular *adj.* 1. (of a *list*) a *circular list*. 2. (of an arbitrary *object*) having a *component*, *element*, *constituent*[2], or *subexpression* (as appropriate to the context) that is the *object* itself.

circular list *n.* a chain of *conses* that has no termination because some *cons* in the chain is the *cdr* of a later *cons*.

class *n.* 1. an *object* that uniquely determines the structure and behavior of a set of other *objects* called its *direct instances*, that contributes structure and behavior to a set of other *objects* called its *indirect instances*, and that acts as a *type specifier* for a set of objects called its *generalized instances*. "The class **integer** is a subclass of the class **number**." (Note that the phrase "the *class* `foo`" is often substituted for the more precise phrase "the *class* named `foo`"---in both cases, a *class object* (not a *symbol*) is denoted.) 2. (of an *object*) the uniquely determined *class* of which the *object* is a *direct instance*. See the *function* **class-of**. "The class of the object returned by **gensym** is **symbol**." (Note that with this usage a phrase such as "its *class* is `foo`" is often substituted for the more precise phrase "its *class* is the *class* named `foo`"---in both cases, a *class object* (not a *symbol*) is denoted.)

class designator *n.* a *designator* for a *class*; that is, an *object* that denotes a *class* and that is one of: a *symbol* (denoting the *class* named by that *symbol*; see the *function* **find-class**) or a *class* (denoting itself).

class precedence list *n.* a unique total ordering on a *class* and its *superclasses* that is consistent with the *local precedence orders* for the *class* and its *superclasses*. For detailed information, see Section 4.3.5 (Determining the Class Precedence List).

close *v.t.* (a *stream*) to terminate usage of the *stream* as a source or sink of data, permitting the *implementation* to reclaim its internal data structures, and to free any external resources which might have been locked by the *stream* when it was opened.

closed *adj.* (of a *stream*) having been *closed* (see <I>}</I>close). Some (but not all) operations that are valid on *open streams* are not valid on *closed streams*. See Section 21.1.1.1.2 (Open and Closed Streams).

closure *n.* a *lexical closure*.

coalesce *v.t.* (*literal objects* that are *similar*) to consolidate the identity of those *objects*, such that they become the *same object*. See Section 3.2.1 (Compiler Terminology).

code *n.* 1. *Trad.* any representation of actions to be performed, whether conceptual or as an actual *object*, such as *forms*, *lambda expressions*, *objects* of *type function*, text in a *source file*, or instruction sequences in a *compiled file*. This is a generic term; the specific nature of the representation depends on its context. 2. (of a *character*) a *character code*.

coerce *v.t.* (an *object* to a *type*) to produce an *object* from the given *object*, without modifying that *object*, by following some set of coercion rules that must be specifically stated for any context in which this term is used. The resulting *object* is necessarily of the indicated *type*, except when that type is a *subtype* of *type* **complex**; in that case, if a *complex rational* with an imaginary part of zero would result, the result is a *rational* rather than a *complex*---see Section 12.1.5.3 (Rule of Canonical Representation for Complex Rationals).

colon *n.* the *standard character* that is called "colon" (`:`). See Figure 2-5.

comma *n.* the *standard character* that is called "comma" (`,`). See Figure 2-5.

compilation *n.* the process of *compiling code* by the *compiler*.

compilation environment *n.* 1. An *environment* that represents information known by the *compiler* about a *form* that is being *compiled*. See Section 3.2.1 (Compiler Terminology). 2. An *object* that represents the *compilation environment*[1] and that is used as a second argument to a *macro function* (which supplies a *value* for any `&environment` *parameter* in the *macro function*'s definition).

compilation unit *n.* an interval during which a single unit of compilation is occurring. See the *macro* **with-compilation-unit**.

compile *v.t.* 1. (*code*) to perform semantic preprocessing of the *code*, usually optimizing one or more qualities of the code, such as run-time speed of *execution* or run-time storage usage. The minimum semantic requirements of compilation are that it must remove all macro calls and arrange for all *load time values* to be resolved prior to run time. 2. (a *function*) to produce a new *object* of *type* **compiled-function** which represents the result of *compiling* the *code* represented by the *function*. See the *function* **compile**. 3. (a *source file*) to produce a *compiled file* from a *source file*. See the *function* **compile-file**.

compile time *n.* the duration of time that the *compiler* is processing *source code*.

compile-time definition *n.* a definition in the *compilation environment*.

compiled code *n.* 1. *compiled functions*. 2. *code* that represents *compiled functions*, such as the contents of a *compiled file*.

compiled file *n.* a *file* which represents the results of *compiling* the *forms* which appeared in a corresponding *source file*, and which can be *loaded*. See the *function* **compile-file**.

compiled function *n.* an *object* of *type* **compiled-function**, which is a *function* that has been *compiled*, which contains no references to *macros* that must be expanded at run time, and which contains no unresolved references to *load time values*.

compiler *n.* a facility that is part of Lisp and that translates *code* into an *implementation-dependent* form that might be represented or *executed* efficiently. The functions **compile** and **compile-file** permit programs to invoke the *compiler*.

compiler macro *n.* an auxiliary macro definition for a globally defined *function* or *macro* which might or might not be called by any given *conforming implementation* and which must preserve the semantics of the globally defined *function* or *macro* but which might perform some additional optimizations. (Unlike a *macro*, a *compiler macro* does not extend the syntax of Common Lisp; rather, it provides an alternate implementation strategy for some existing syntax or functionality.)

compiler macro expansion *n.* 1. the process of translating a *form* into another *form* by a *compiler macro*. 2. the *form* resulting from this process.

compiler macro form *n.* a *function form* or *macro form* whose *operator* has a definition as a *compiler macro*, or a **funcall** *form* whose first *argument* is a **function** *form* whose *argument* is the *name* of a *function* that has a definition as a *compiler macro*.

compiler macro function *n.* a *function* of two arguments, a *form* and an *environment*, that implements *compiler macro expansion* by producing either a *form* to be used in place of the original argument *form* or else **nil**, indicating that the original *form* should not be replaced. See Section 3.2.2.1 (Compiler Macros).

complex *n.* an *object* of *type* **complex**.

complex float *n.* an *object* of *type* **complex** which has a *complex part type* that is a *subtype* of **float**. A *complex float* is a *complex*, but it is not a *float*.

complex part type *n.* (of a *complex*) 1. the *type* which is used to represent both the real part and the imaginary part of the *complex*. 2. the *actual complex part type* of the *complex*. 3. the *expressed complex part type* of the *complex*.

complex rational *n.* an *object* of *type* **complex** which has a *complex part type* that is a *subtype* of **rational**. A *complex rational* is a *complex*, but it is not a *rational*. No *complex rational* has an imaginary part of zero because such a number is always represented by Common Lisp as an *object* of *type* **rational**; see Section 12.1.5.3 (Rule of Canonical Representation for Complex Rationals).

complex single float *n.* an *object* of *type* **complex** which has a *complex part type* that is a *subtype* of **single-float**. A *complex single float* is a *complex*, but it is not a *single float*.

composite stream *n.* a *stream* that is composed of one or more other *streams*. "**make-synonym-stream** creates a composite stream."

compound form *n.* a *non-empty list* which is a *form*: a *special form*, a *lambda form*, a *macro form*, or a *function form*.

compound type specifier *n.* a *type specifier* that is a *cons*; i.e., a *type specifier* that is not an *atomic type specifier*. "`(vector single-float)` is a compound type specifier."

concatenated stream *n.* an *input stream* of *type* **concatenated-stream**.

condition *n.* 1. an *object* which represents a *situation*---usually, but not necessarily, during *signaling*. 2. an *object* of *type* **condition**.

condition designator *n.* one or more *objects* that, taken together, denote either an existing *condition object* or a *condition object* to be implicitly created. For details, see Section 9.1.2.1 (Condition Designators).

condition handler *n.* a *function* that might be invoked by the act of *signaling*, that receives the *condition* being signaled as its only argument, and that is permitted to *handle* the *condition* or to *decline*. See Section 9.1.4.1 (Signaling).

condition reporter *n.* a *function* that describes how a *condition* is to be printed when the *Lisp printer* is invoked while ***print-escape*** is *false*. See Section 9.1.3 (Printing Conditions).

conditional newline *n.* a point in output where a *newline* might be inserted at the discretion of the *pretty printer*. There are four kinds of *conditional newlines*, called "linear-style," "fill-style," "miser-style," and "mandatory-style." See the *function* **pprint-newline** and Section 22.2.1.1 (Dynamic Control of the Arrangement of Output).

conformance *n.* a state achieved by proper and complete adherence to the requirements of this specification. See Section 1.5 (Conformance).

conforming code *n. code* that is all of part of a *conforming program*.

conforming implementation *n.* an *implementation*, used to emphasize complete and correct adherance to all conformance criteria. A *conforming implementation* is capable of accepting a *conforming program* as input, preparing that *program* for *execution*, and executing the prepared *program* in accordance with this specification. An *implementation* which has been extended may still be a *conforming implementation* provided that no extension interferes with the correct function of any *conforming program*.

conforming processor *n. ANSI* a *conforming implementation*.

conforming program *n.* a *program*, used to emphasize the fact that the *program* depends for its correctness only upon documented aspects of Common Lisp, and can therefore be expected to run correctly in any *conforming implementation*.

congruent *n.* conforming to the rules of *lambda list* congruency, as detailed in Section 7.6.4 (Congruent Lambda-lists for all Methods of a Generic Function).

cons *n.v.* 1. *n.* a compound data *object* having two components called the *car* and the *cdr*. 2. *v.* to create such an *object*. 3. *v. Idiom.* to create any *object*, or to allocate storage.

constant *n.* 1. a *constant form*. 2. a *constant variable*. 3. a *constant object*. 4. a *self-evaluating object*.

constant form *n.* any *form* for which *evaluation* always *yields* the same *value*, that neither affects nor is affected by the *environment* in which it is *evaluated* (except that it is permitted to refer to the names of *constant variables* defined in the *environment*), and that neither affects nor is affected by the state of any *object* except those *objects* that are *otherwise inaccessible parts* of *objects* created by the *form* itself. "A **car** form in which the argument is a **quote** form is a constant form."

constant object *n.* an *object* that is constrained (e.g., by its context in a *program* or by the source from which it was obtained) to be *immutable*. "A literal object that has been processed by **compile-file** is a constant object."

constant variable *n.* a *variable*, the *value* of which can never change; that is, a *keyword*[1] or a *named constant*. "The symbols **t**, **nil**, `:direction`, and **most-positive-fixnum** are constant variables."

constituent *n., adj.* 1. a. *n.* the *syntax type* of a *character* that is part of a *token*. For details, see Section 2.1.4.1 (Constituent Characters). b. *adj.* (of a *character*) having the *constituent*[1a] *syntax type*[2]. c. *n.* a *constituent*[1b] *character*. 2. *n.* (of a *composite stream*) one of possibly several *objects* that collectively comprise the source or sink of that *stream*.

constituent trait *n.* (of a *character*) one of several classifications of a *constituent character* in a *readtable*. See Section 2.1.4.1 (Constituent Characters).

constructed stream *n.* a *stream* whose source or sink is a Lisp *object*. Note that since a *stream* is another Lisp *object*, *composite streams* are considered *constructed streams*. "A string stream is a constructed stream."

contagion *n.* a process whereby operations on *objects* of differing *types* (e.g., arithmetic on mixed *types* of *numbers*) produce a result whose *type* is controlled by the dominance of one *argument*'s *type* over the *types* of the other *arguments*. See Section 12.1.1.2 (Contagion in Numeric Operations).

continuable *n.* (of an *error*) an *error* that is *correctable* by the `continue` restart.

control form *n.* 1. a *form* that establishes one or more places to which control can be transferred. 2. a *form* that transfers control.

copy *n.* 1. (of a *cons* C) a *fresh cons* with the *same car* and *cdr* as C. 2. (of a *list* L) a *fresh list* with the *same elements* as L. (Only the *list structure* is *fresh*; the *elements* are the *same*.) See the *function* **copy-list**. 3. (of an *association list* A with *elements* Ai) a *fresh list* B with *elements* Bi, each of which is **nil** if Ai is **nil**, or else a *copy* of the *cons* Ai. See the *function* **copy-alist**. 4. (of a *tree* T) a *fresh tree* with the *same leaves* as T. See the *function* **copy-tree**. 5. (of a *random state* R) a *fresh random state* that, if used as an argument to to the *function* **random** would produce the same series of "random" values as R would produce. 6. (of a *structure* S) a *fresh structure* that has the same *type* as S, and that has slot values, each of which is the *same* as the corresponding slot value of S. (Note that since the difference between a *cons*, a *list*, and a *tree* is a matter of "view" or "intention," there can be no general-purpose *function* which, based solely on the *type* of an *object*, can determine which of these distinct meanings is intended. The distinction rests solely on the basis of the text description within this document. For example, phrases like "a *copy* of the given *list*" or "copy of the *list x*" imply the second definition.)

correctable *adj.* (of an *error*) 1. (by a *restart* other than **abort** that has been associated with the *error*) capable of being corrected by invoking that *restart*. "The function **cerror** signals an error that is correctable by the **continue** *restart*." (Note that correctability is not a property of an *error object*, but rather a property of the *dynamic environment* that is in effect when the *error* is *signaled*. Specifically, the *restart* is "associated with" the *error condition object*. See Section 9.1.4.2.4 (Associating a Restart with a Condition).) 2. (when no specific *restart* is mentioned) *correctable*[1] by at least one *restart*. "**import** signals a correctable error of *type* **package-error** if any of the imported symbols has the same name as some distinct symbol already accessible in the package."

current input base *n.* (in a *dynamic environment*) the *radix* that is the *value* of ***read-base*** in that *environment*, and that is the default *radix* employed by the *Lisp reader* and its related *functions*.

current logical block *n.* the context of the innermost lexically enclosing use of **pprint-logical-block**.

current output base *n.* (in a *dynamic environment*) the *radix* that is the *value* of ***print-base*** in that *environment*, and that is the default *radix* employed by the *Lisp printer* and its related *functions*.

current package *n.* (in a *dynamic environment*) the *package* that is the *value* of ***package*** in that *environment*, and that is the default *package* employed by the *Lisp reader* and *Lisp printer*, and their related *functions*.

current pprint dispatch table *n.* (in a *dynamic environment*) the *pprint dispatch table* that is the *value* of ***print-pprint-dispatch**** in that *environment*, and that is the default *pprint dispatch table* employed by the *pretty printer*.

current random state *n.* (in a *dynamic environment*) the *random state* that is the *value* of ***random-state**** in that *environment*, and that is the default *random state* employed by **random**.

current readtable *n.* (in a *dynamic environment*) the *readtable* that is the *value* of ***readtable**** in that *environment*, and that affects the way in which *expressions*[2] are parsed into *objects* by the *Lisp reader*.

D

data type *n. Trad.* a *type*.

debug I/O *n.* the *bidirectional stream* that is the *value* of the *variable* ***debug-io****.

debugger *n.* a facility that allows the *user* to handle a *condition* interactively. For example, the *debugger* might permit interactive selection of a *restart* from among the *active restarts*, and it might perform additional *implementation-defined* services for the purposes of debugging.

declaration *n.* a *global declaration* or *local declaration*.

declaration identifier *n.* one of the *symbols* **declaration**, **dynamic-extent**, **ftype**, **function**, **ignore**, **inline**, **notinline**, **optimize**, **special**, or **type**; or a *symbol* which is the *name* of a *type*; or a *symbol* which has been *declared* to be a *declaration identifier* by using a **declaration** *declaration*.

declaration specifier *n.* an *expression* that can appear at top level of a **declare** expression or a **declaim** form, or as the argument to **proclaim**, and which has a *car* which is a *declaration identifier*, and which has a *cdr* that is data interpreted according to rules specific to the *declaration identifier*.

declare *v.* to *establish* a *declaration*. See **declare**, **declaim**, or **proclaim**.

decline *v.* (of a *handler*) to return normally without having *handled* the *condition* being *signaled*, permitting the signaling process to continue as if the *handler* had not been present.

decoded time *n. absolute time*, represented as an ordered series of nine *objects* which, taken together, form a description of a point in calendar time, accurate to the nearest second (except that *leap seconds* are ignored). See Section 25.1.4.1 (Decoded Time).

default method *n.* a *method* having no *parameter specializers* other than the *class* **t**. Such a *method* is always an *applicable method* but might be *shadowed*[2] by a more specific *method*.

defaulted initialization argument list *n.* a *list* of alternating initialization argument *names* and *values* in which unsupplied initialization arguments are defaulted, used in the protocol for initializing and reinitializing *instances* of *classes*.

define-method-combination arguments lambda list *n.* a *lambda list* used by the `:arguments` option to **define-method-combination**. See Section 3.4.10 (Define-method-combination Arguments Lambda Lists).

define-modify-macro lambda list *n.* a *lambda list* used by **define-modify-macro**. See Section 3.4.9 (Define-modify-macro Lambda Lists).

defined name *n.* a *symbol* the meaning of which is defined by Common Lisp.

defining form *n.* a *form* that has the side-effect of *establishing* a definition. "**defun** and **defparameter** are defining forms."

defsetf lambda list *n.* a *lambda list* that is like an *ordinary lambda list* except that it does not permit `&aux` and that it permits use of `&environment`. See Section 3.4.7 (Defsetf Lambda Lists).

deftype lambda list *n.* a *lambda list* that is like a *macro lambda list* except that the default *value* for unsupplied *optional parameters* and *keyword parameters* is the *symbol* ***** (rather than **nil**). See Section 3.4.8 (Deftype Lambda Lists).

denormalized *adj., ANSI, IEEE* (of a *float*) conforming to the description of "denormalized" as described by *IEEE Standard for Binary Floating-Point Arithmetic*. For example, in an *implementation* where the minimum possible exponent was `-7` but where `0.001` was a valid mantissa, the number `1.0e-10` might be representable as `0.001e-7` internally even if the *normalized* representation would call for it to be represented instead as `1.0e-10` or `0.1e-9`. By their nature, *denormalized floats* generally have less precision than *normalized floats*.

derived type *n.* a *type specifier* which is defined in terms of an expansion into another *type specifier*. **deftype** defines *derived types*, and there may be other *implementation-defined operators* which do so as well.

derived type specifier *n.* a *type specifier* for a *derived type*.

designator *n.* an *object* that denotes another *object*. In the dictionary entry for an *operator* if a *parameter* is described as a *designator* for a *type*, the description of the *operator* is written in a way that assumes that appropriate coercion to that *type* has already occurred; that is, that the *parameter* is already of the denoted *type*. For more detailed information, see Section 1.4.1.5 (Designators).

destructive *adj.* (of an *operator*) capable of modifying some program-visible aspect of one or more *objects* that are either explicit *arguments* to the *operator* or that can be obtained directly or indirectly from the *global environment* by the *operator*.

destructuring lambda list *n.* an *extended lambda list* used in **destructuring-bind** and nested within *macro lambda lists*. See Section 3.4.5 (Destructuring Lambda Lists).

different *adj.* not the *same* "The strings `"FOO"` and `"foo"` are different under **equal** but not under **equalp**."

digit *n.* (in a *radix*) a *character* that is among the possible digits (`0` to `9`, `A` to `Z`, and `a` to `z`) and that is defined to have an associated numeric weight as a digit in that *radix*. See Section 13.1.4.6 (Digits in a Radix).

dimension *n.* 1. a non-negative *integer* indicating the number of *objects* an *array* can hold along one axis. If the *array* is a *vector* with a *fill pointer*, the *fill pointer* is ignored. "The second dimension of that array is 7." 2. an axis of an array. "This array has six dimensions."

direct instance *n.* (of a *class* C) an *object* whose *class* is C itself, rather than some *subclass* of C. "The function **make-instance** always returns a direct instance of the class which is (or is named by) its first argument."

direct subclass *n.* (of a *class* C1) a *class* C2, such that C1 is a *direct superclass* of C2.

direct superclass *n.* (of a *class* C1) a *class* C2 which was explicitly designated as a *superclass* of C1 in the definition of C1.

disestablish *v.t.* to withdraw the *establishment* of an *object*, a *binding*, an *exit point*, a *tag*, a *handler*, a *restart*, or an *environment*.

disjoint *n.* (of *types*) having no *elements* in common.

dispatching macro character *n.* a *macro character* that has an associated table that specifies the *function* to be called for each *character* that is seen following the *dispatching macro character*. See the *function* **make-dispatch-macro-character**.

displaced array *n.* an *array* which has no storage of its own, but which is instead indirected to the storage of another *array*, called its *target*, at a specified offset, in such a way that any attempt to *access* the *displaced array* implicitly references the *target array*.

distinct *adj.* not *identical*.

documentation string *n.* (in a defining *form*) A *literal string* which because of the context in which it appears (rather than because of some intrinsically observable aspect of the *string*) is taken as documentation. In some cases, the *documentation string* is saved in such a way that it can later be obtained by supplying either an *object*, or by supplying a *name* and a "kind" to the *function* **documentation**. "The body of code in a **defmacro** form can be preceded by a documentation string of kind **function**."

dot *n.* the *standard character* that is variously called "full stop," "period," or "dot" (**.**). See Figure 2-5.

dotted list *n.* a *list* which has a terminating *atom* that is not **nil**. (An *atom* by itself is not a *dotted list*, however.)

dotted pair *n.* 1. a *cons* whose *cdr* is a *non-list*. 2. any *cons*, used to emphasize the use of the *cons* as a symmetric data pair.

double float *n.* an *object* of *type* **double-float**.

double-quote *n.* the *standard character* that is variously called "quotation mark" or "double quote" (**"**). See Figure 2-5.

dynamic binding *n.* a *binding* in a *dynamic environment*.

dynamic environment *n.* that part of an *environment* that contains *bindings* with *dynamic extent*. A *dynamic environment* contains, among other things: *exit points* established by **unwind-protect**, and *bindings* of *dynamic variables*, *exit points* established by **catch**, *condition handlers*, and *restarts*.

dynamic extent *n.* an *extent* whose duration is bounded by points of *establishment* and *disestablishment* within the execution of a particular *form*. See *indefinite extent*. "Dynamic variable bindings have dynamic extent."

dynamic scope *n. indefinite scope* along with *dynamic extent*.

dynamic variable *n.* a *variable* the *binding* for which is in the *dynamic environment*. See **special**.

E

echo stream *n.* a *stream* of *type* **echo-stream**.

effective method *n.* the combination of *applicable methods* that are executed when a *generic function* is invoked with a particular sequence of *arguments*.

element *n.* 1. (of a *list*) an *object* that is the *car* of one of the *conses* that comprise the *list*. 2. (of an *array*) an *object* that is stored in the *array*. 3. (of a *sequence*) an *object* that is an *element* of the *list* or *array* that is the *sequence*. 4. (of a *type*) an *object* that is a member of the set of *objects* designated by the *type*. 5. (of an *input stream*) a *character* or *number* (as appropriate to the *element type* of the *stream*) that is among the ordered series of *objects* that can be read from the *stream* (using **read-char** or **read-byte**, as appropriate to the *stream*). 6. (of an *output stream*) a *character* or *number* (as appropriate to the *element type* of the *stream*) that is among the ordered series of *objects* that has been or will be written to the *stream* (using **write-char** or **write-byte**, as appropriate to the *stream*). 7. (of a *class*) a *generalized instance* of the *class*.

element type *n.* 1. (of an *array*) the *array element type* of the *array*. 2. (of a *stream*) the *stream element type* of the *stream*.

em *n. Trad.* a context-dependent unit of measure commonly used in typesetting, equal to the displayed width of of a letter "M" in the current font. (The letter "M" is traditionally chosen because it is typically represented by the widest *glyph* in the font, and other characters' widths are typically fractions of an *em*. In implementations providing non-Roman characters with wider characters than "M," it is permissible for another character to be the *implementation-defined* reference character for this measure, and for "M" to be only a fraction of an *em* wide.) In a fixed width font, a line with *n* characters is *n ems* wide; in a variable width font, *n ems* is the expected upper bound on the width of such a line.

empty list *n.* the *list* containing no *elements*. See *()*.

empty type *n.* the *type* that contains no *elements*, and that is a *subtype* of all *types* (including itself). See *nil*.

end of file *n.* 1. the point in an *input stream* beyond which there is no further data. Whether or not there is such a point on an *interactive stream* is *implementation-defined*. 2. a *situation* that occurs upon an attempt to obtain data from an *input stream* that is at the *end of file*[1].

environment *n.* 1. a set of *bindings*. See Section 3.1.1 (Introduction to Environments). 2. an *environment object*. "**macroexpand** takes an optional environment argument."

environment object *n.* an *object* representing a set of *lexical bindings*, used in the processing of a *form* to provide meanings for *names* within that *form*. "**macroexpand** takes an optional environment argument." (The *object* **nil** when used as an *environment object* denotes the *null lexical environment*; the *values* of *environment parameters* to *macro functions* are *objects* of *implementation-dependent* nature which represent the *environment*[1] in which the corresponding *macro form* is to be expanded.) See Section 3.1.1.4 (Environment Objects).

environment parameter *n.* A *parameter* in a *defining form* f for which there is no corresponding *argument*; instead, this *parameter* receives as its value an *environment object* which corresponds to the *lexical environment* in which the *defining form* f appeared.

error *n.* 1. (only in the phrase "is an error") a *situation* in which the semantics of a program are not specified, and in which the consequences are undefined. 2. a *condition* which represents an *error situation*. See Section 1.4.2 (Error Terminology). 3. an *object* of *type* **error**.

error output *n.* the *output stream* which is the *value* of the *dynamic variable* ***error-output***.

escape *n., adj.* 1. *n.* a *single escape* or a *multiple escape*. 2. *adj. single escape* or *multiple escape*.

establish *v.t.* to build or bring into being a *binding*, a *declaration*, an *exit point*, a *tag*, a *handler*, a *restart*, or an *environment*. "**let** establishes lexical bindings."

evaluate *v.t.* (a *form* or an *implicit progn*) to *execute* the *code* represented by the *form* (or the series of *forms* making up the *implicit progn*) by applying the rules of *evaluation*, returning zero or more values.

evaluation *n.* a model whereby *forms* are *executed*, returning zero or more values. Such execution might be implemented directly in one step by an interpreter or in two steps by first *compiling* the *form* and then *executing* the *compiled code*; this choice is dependent both on context and the nature of the *implementation*, but in any case is not in general detectable by any program. The evaluation model is designed in such a way that a *conforming implementation* might legitimately have only a compiler and no interpreter, or vice versa. See Section 3.1.2 (The Evaluation Model).

evaluation environment *n.* a *run-time environment* in which macro expanders and code specified by **eval-when** to be evaluated are evaluated. All evaluations initiated by the *compiler* take place in the *evaluation environment*.

execute *v.t. Trad.* (*code*) to perform the imperative actions represented by the *code*.

execution time *n.* the duration of time that *compiled code* is being *executed*.

exhaustive partition *n.* (of a *type*) a set of *pairwise disjoint types* that form an *exhaustive union*.

exhaustive union *n.* (of a *type*) a set of *subtypes* of the *type*, whose union contains all *elements* of that *type*.

exit point *n.* a point in a *control form* from which (e.g., **block**), through which (e.g., **unwind-protect**), or to which (e.g., **tagbody**) control and possibly *values* can be transferred both actively by using another *control form* and passively through the normal control and data flow of *evaluation*. "**catch** and **block** establish bindings for exit points to which **throw** and **return-from**, respectively, can transfer control and values; **tagbody** establishes a binding for an exit point with lexical extent to which **go** can transfer control; and **unwind-protect** establishes an exit point through which control might be transferred by operators such as **throw**, **return-from**, and **go**."

explicit return *n.* the act of transferring control (and possibly *values*) to a *block* by using **return-from** (or **return**).

explicit use *n.* (of a *variable* V in a *form* F) a reference to V that is directly apparent in the normal semantics of F; i.e., that does not expose any undocumented details of the *macro expansion* of the *form* itself. References to V exposed by expanding *subforms* of F are, however, considered to be *explicit uses* of V.

exponent marker *n.* a character that is used in the textual notation for a *float* to separate the mantissa from the exponent. The characters defined as *exponent markers* in the *standard readtable* are shown in the next figure. For more information, see Section 2.1 (Character Syntax). "The exponent marker ‘d’ in ‘3.0d7’ indicates that this number is to be represented as a double float."

```
Marker  Meaning
D or d  double-float
E or e  float (see *read-default-float-format*)
F or f  single-float
L or l  long-float
S or s  short-float
```

Figure 26-1. Exponent Markers

export *v.t.* (a *symbol* in a *package*) to add the *symbol* to the list of *external symbols* of the *package*.

exported *adj.* (of a *symbol* in a *package*) being an *external symbol* of the *package*.

expressed adjustability *n.* (of an *array*) a *generalized boolean* that is conceptually (but not necessarily actually) associated with the *array*, representing whether the *array* is *expressly adjustable*. See also *actual adjustability*.

expressed array element type *n.* (of an *array*) the *type* which is the *array element type* implied by a *type declaration* for the *array*, or which is the requested *array element type* at its time of creation, prior to any selection of an *upgraded array element type*. (Common Lisp does not provide a way of detecting this *type* directly at run time, but an *implementation* is permitted to make assumptions about the *array*’s contents and the operations which may be performed on the *array* when this *type* is noted during code analysis, even if those assumptions would not be valid in general for the *upgraded array element type* of the *expressed array element type*.)

expressed complex part type *n.* (of a *complex*) the *type* which is implied as the *complex part type* by a *type declaration* for the *complex*, or which is the requested *complex part type* at its time of creation, prior to any selection of an *upgraded complex part type*. (Common Lisp does not provide a way of detecting this *type* directly at run time, but an *implementation* is permitted to make assumptions about the operations which may be performed on the *complex* when this *type* is noted during code analysis, even if those assumptions would not be valid in general for the *upgraded complex part type* of the *expressed complex part type*.)

expression *n.* 1. an *object*, often used to emphasize the use of the *object* to encode or represent information in a specialized format, such as program text. "The second expression in a **let** form is a list of bindings." 2. the textual notation used to notate an *object* in a source file. "The expression `'sample` is equivalent to `(quote sample)`."

expressly adjustable *adj.* (of an *array*) being *actually adjustable* by virtue of an explicit request for this characteristic having been made at the time of its creation. All *arrays* that are *expressly adjustable* are *actually adjustable*, but not necessarily vice versa.

extended character *n.* a *character* of *type* **extended-char**: a *character* that is not a *base character*.

extended function designator *n.* a *designator* for a *function*; that is, an *object* that denotes a *function* and that is one of: a *function name* (denoting the *function* it names in the *global environment*), or a *function* (denoting itself). The consequences are undefined if a *function name* is used as an *extended function designator* but it does not have a global definition as a *function*, or if it is a *symbol* that has a global definition as a *macro* or a *special form*. See also *function designator*.

extended lambda list *n.* a list resembling an *ordinary lambda list* in form and purpose, but offering additional syntax or functionality not available in an *ordinary lambda list*. "**defmacro** uses extended lambda lists."

extension *n.* a facility in an *implementation* of Common Lisp that is not specified by this standard.

extent *n.* the interval of time during which a *reference* to an *object*, a *binding*, an *exit point*, a *tag*, a *handler*, a *restart*, or an *environment* is defined.

external file format *n.* an *object* of *implementation-dependent* nature which determines one of possibly several *implementation-dependent* ways in which *characters* are encoded externally in a *character file*.

external file format designator *n.* a *designator* for an *external file format*; that is, an *object* that denotes an *external file format* and that is one of: the *symbol* `:default` (denoting an *implementation-dependent* default *external file format* that can accomodate at least the *base characters*), some other *object* defined by the *implementation* to be an *external file format designator* (denoting an *implementation-defined external file format*), or some other *object* defined by the *implementation* to be an *external file format* (denoting itself).

external symbol *n.* (of a *package*) a *symbol* that is part of the ‘external interface’ to the *package* and that are *inherited*[3] by any other *package* that *uses* the *package*. When using the *Lisp reader*, if a *package prefix* is used, the *name* of an *external symbol* is separated from the *package name* by a single *package marker* while the *name* of an *internal symbol* is separated from the *package name* by a double *package marker*; see Section 2.3.4 (Symbols as Tokens).

externalizable object *n.* an *object* that can be used as a *literal object* in *code* to be processed by the *file compiler*.

F

false *n.* the *symbol* **nil**, used to represent the failure of a *predicate* test.

fbound [’ef,band] *adj.* (of a *function name*) *bound* in the *function namespace*. (The *names* of *macros* and *special operators* are *fbound*, but the nature and *type* of the *object* which is their *value* is *implementation-dependent*. Further, defining a *setf expander F* does not cause the *setf function* `(setf` *F*`)` to become defined; as such, if there is a such a definition of a *setf expander F*, the *function* `(setf` *F*`)` can be *fbound* if and only if, by design or coincidence, a function binding for `(setf` *F*`)` has been independently established.) See the *functions* **fboundp** and **symbol-function**.

feature *n.* 1. an aspect or attribute of Common Lisp, of the *implementation*, or of the *environment*. 2. a *symbol* that names a *feature*[1]. See Section 24.1.2 (Features). "The `:ansi-cl` feature is present in all conforming implementations."

feature expression *n.* A boolean combination of *features* used by the `#+` and `#-` *reader macros* in order to direct conditional *reading* of *expressions* by the *Lisp reader*. See Section 24.1.2.1 (Feature Expressions).

features list *n.* the *list* that is the *value* of ***features***.

file *n.* a named entry in a *file system*, having an *implementation-defined* nature.

file compiler *n.* any *compiler* which *compiles source code* contained in a *file*, producing a *compiled file* as output. The **compile-file** function is the only interface to such a *compiler* provided by Common Lisp, but there might be other, *implementation-defined* mechanisms for invoking the *file compiler*.

file position *n.* (in a *stream*) a non-negative *integer* that represents a position in the *stream*. Not all *streams* are able to represent the notion of *file position*; in the description of any *operator* which manipulates *file positions*, the behavior for *streams* that don't have this notion must be explicitly stated. For *binary streams*, the *file position* represents the number of preceding *bytes* in the *stream*. For *character streams*, the constraint is more relaxed: *file positions* must increase monotonically, the amount of the increase between *file positions* corresponding to any two successive characters in the *stream* is *implementation-dependent*.

file position designator *n.* (in a *stream*) a *designator* for a *file position* in that *stream*; that is, the symbol `:start` (denoting `0`, the first *file position* in that *stream*), the symbol `:end` (denoting the last *file position* in that *stream*; i.e., the position following the last *element* of the *stream*), or a *file position* (denoting itself).

file stream *n.* an *object* of *type* **file-stream**.

file system *n.* a facility which permits aggregations of data to be stored in named *files* on some medium that is external to the *Lisp image* and that therefore persists from *session* to *session*.

filename *n.* a handle, not necessarily ever directly represented as an *object*, that can be used to refer to a *file* in a *file system*. *Pathnames* and *namestrings* are two kinds of *objects* that substitute for *filenames* in Common Lisp.

fill pointer *n.* (of a *vector*) an *integer* associated with a *vector* that represents the index above which no *elements* are *active*. (A *fill pointer* is a non-negative *integer* no larger than the total number of *elements* in the *vector*. Not all *vectors* have *fill pointers*.)

finite *adj.* (of a *type*) having a finite number of *elements*. "The type specifier `(integer 0 5)` denotes a finite type, but the type specifiers **integer** and `(integer 0)` do not."

fixnum *n.* an *integer* of *type* **fixnum**.

float *n.* an *object* of *type* **float**.

for-value *adj.* (of a *reference* to a *binding*) being a *reference* that *reads*[1] the *value* of the *binding*.

form *n.* 1. any *object* meant to be *evaluated*. 2. a *symbol*, a *compound form*, or a *self-evaluating object*. 3. (for an *operator*, as in "<<*operator*>> *form*") a *compound form* having that *operator* as its first element. "A **quote** form is a constant form."

formal argument *n. Trad.* a *parameter*.

formal parameter *n. Trad.* a *parameter*.

format *v.t.* (a *format control* and *format arguments*) to perform output as if by **format**, using the *format string* and *format arguments*.

format argument *n.* an *object* which is used as data by functions such as **format** which interpret *format controls*.

format control *n.* a *format string*, or a *function* that obeys the *argument* conventions for a *function* returned by the **formatter** *macro*. See Section 22.2.1.3 (Compiling Format Strings).

format directive *n.* 1. a sequence of *characters* in a *format string* which is introduced by a *tilde*, and which is specially interpreted by *code* which processes *format strings* to mean that some special operation should be performed, possibly involving data supplied by the *format arguments* that accompanied the *format string*. See the *function* **format**. "In `"~D base 10 = ~8R"`, the character sequences '`~D`' and '`~8R`' are format directives." 2. the conceptual category of all *format directives*[1] which use the same dispatch character. "Both `"~3d"` and `"~3,'0D"` are valid uses of the '`~D`' format directive."

format string *n.* a *string* which can contain both ordinary text and *format directives*, and which is used in conjunction with *format arguments* to describe how text output should be formatted by certain functions, such as **format**.

free declaration *n.* a declaration that is not a *bound declaration*. See **declare**.

fresh *adj.* 1. (of an *object yielded* by a *function*) having been newly-allocated by that *function*. (The caller of a *function* that returns a *fresh object* may freely modify the *object* without fear that such modification will compromise the future correct behavior of that *function*.) 2. (of a *binding* for a *name*) newly-allocated; not shared with other *bindings* for that *name*.

freshline *n.* a conceptual operation on a *stream*, implemented by the *function* **fresh-line** and by the *format directive* `~&`, which advances the display position to the beginning of the next line (as if a *newline* had been typed, or the *function* **terpri** had been called) unless the *stream* is already known to be positioned at the beginning of a line. Unlike *newline*, *freshline* is not a *character*.

funbound ['efunband] *n.* (of a *function name*) not *fbound*.

function *n.* 1. an *object* representing code, which can be *called* with zero or more *arguments*, and which produces zero or more *values*. 2. an *object* of *type* **function**.

function block name *n.* (of a *function name*) The *symbol* that would be used as the name of an *implicit block* which surrounds the body of a *function* having that *function name*. If the *function name* is a *symbol*, its *function block name* is the *function name* itself. If the *function name* is a *list* whose *car* is **setf** and whose *cadr* is a *symbol*, its *function block name* is the *symbol* that is the *cadr* of the *function name*. An *implementation* which supports additional kinds of *function names* must specify for each how the corresponding *function block name* is computed.

function cell *n. Trad.* (of a *symbol*) The *place* which holds the *definition* of the global *function binding*, if any, named by that *symbol*, and which is *accessed* by **symbol-function**. See *cell*.

function designator *n.* a *designator* for a *function*; that is, an *object* that denotes a *function* and that is one of: a *symbol* (denoting the *function* named by that *symbol* in the *global environment*), or a *function* (denoting itself). The consequences are undefined if a *symbol* is used as a *function designator* but it does not have a global definition as a *function*, or it has a global definition as a *macro* or a *special form*. See also *extended function designator*.

function form *n.* a *form* that is a *list* and that has a first element which is the *name* of a *function* to be called on *arguments* which are the result of *evaluating* subsequent elements of the *function form*.

function name *n.* 1. (in an *environment*) A *symbol* or a *list* `(setf` *symbol*`)` that is the *name* of a *function* in that *environment*. 2. A *symbol* or a *list* `(setf` *symbol*`)`.

functional evaluation *n.* the process of extracting a *functional value* from a *function name* or a *lambda expression*. The evaluator performs *functional evaluation* implicitly when it encounters a *function name* or a *lambda expression* in the *car* of a *compound form*, or explicitly when it encounters a **function** *special form*. Neither a use of a *symbol*

as a *function designator* nor a use of the *function* **symbol-function** to extract the *functional value* of a *symbol* is considered a *functional evaluation*.

functional value *n.* 1. (of a *function name* N in an *environment* E) The *value* of the *binding* named N in the *function namespace* for *environment* E; that is, the contents of the *function cell* named N in *environment* E. 2. (of an *fbound symbol* S) the contents of the *symbol*'s *function cell*; that is, the *value* of the *binding* named S in the *function namespace* of the *global environment*. (A *name* that is a *macro name* in the *global environment* or is a *special operator* might or might not be *fbound*. But if S is such a *name* and is *fbound*, the specific nature of its *functional value* is *implementation-dependent*; in particular, it might or might not be a *function*.)

further compilation *n. implementation-dependent* compilation beyond *minimal compilation*. Further compilation is permitted to take place at *run time*. "Block compilation and generation of machine-specific instructions are examples of further compilation."

G

general *adj*. (of an *array*) having *element type* **t**, and consequently able to have any *object* as an *element*.

generalized boolean *n.* an *object* used as a truth value, where the symbol **nil** represents *false* and all other *objects* represent *true*. See *boolean*.

generalized instance *n.* (of a *class*) an *object* the *class* of which is either that *class* itself, or some subclass of that *class*. (Because of the correspondence between types and classes, the term "generalized instance of X" implies "object of type X" and in cases where X is a *class* (or *class name*) the reverse is also true. The former terminology emphasizes the view of X as a *class* while the latter emphasizes the view of X as a *type specifier*.)

generalized reference *n.* a reference to a location storing an *object* as if to a *variable*. (Such a reference can be either to *read* or *write* the location.) See Section 5.1 (Generalized Reference). See also *place*.

generalized synonym stream *n.* (with a *synonym stream symbol*) 1. (to a *stream*) a *synonym stream* to the *stream*, or a *composite stream* which has as a target a *generalized synonym stream* to the *stream*. 2. (to a *symbol*) a *synonym stream* to the *symbol*, or a *composite stream* which has as a target a *generalized synonym stream* to the *symbol*.

generic function *n.* a *function* whose behavior depends on the *classes* or identities of the arguments supplied to it and whose parts include, among other things, a set of *methods*, a *lambda list*, and a *method combination* type.

generic function lambda list *n.* A *lambda list* that is used to describe data flow into a *generic function*. See Section 3.4.2 (Generic Function Lambda Lists).

gensym *n. Trad.* an *uninterned symbol*. See the *function* **gensym**.

global declaration *n.* a *form* that makes certain kinds of information about code globally available; that is, a **proclaim** *form* or a **declaim** *form*.

global environment *n.* that part of an *environment* that contains *bindings* with *indefinite scope* and *indefinite extent*.

global variable *n.* a *dynamic variable* or a *constant variable*.

glyph *n.* a visual representation. "Graphic characters have associated glyphs."

go *v.* to transfer control to a *go point*. See the *special operator* **go**.

go point one of possibly several *exit points* that are *established* by **tagbody** (or other abstractions, such as **prog**, which are built from **tagbody**).

go tag *n.* the *symbol* or *integer* that, within the *lexical scope* of a **tagbody** *form*, names an *exit point established* by that **tagbody** *form*.

graphic *adj.* (of a *character*) being a "printing" or "displayable" *character* that has a standard visual representation as a single *glyph*, such as A or * or =. *Space* is defined to be *graphic*. Of the *standard characters*, all but *newline* are *graphic*. See *non-graphic*.

H

handle *v.* (of a *condition* being *signaled*) to perform a non-local transfer of control, terminating the ongoing *signaling* of the *condition*.

handler *n.* a *condition handler*.

hash table *n.* an *object* of *type* **hash-table**, which provides a mapping from *keys* to *values*.

home package *n.* (of a *symbol*) the *package*, if any, which is contents of the *package cell* of the *symbol*, and which dictates how the *Lisp printer* prints the *symbol* when it is not *accessible* in the *current package*. (*Symbols* which have **nil** in their *package cell* are said to have no *home package*, and also to be *apparently uninterned*.)

I

I/O customization variable *n.* one of the *stream variables* in the next figure, or some other (*implementation-defined*) *stream variable* that is defined by the *implementation* to be an *I/O customization variable*.

```
*debug-io*        *error-io*         query-io*
*standard-input*  *standard-output*  *trace-output*
```

Figure 26-2. Standardized I/O Customization Variables

identical *adj.* the *same* under **eq**.

identifier *n.* 1. a *symbol* used to identify or to distinguish *names*. 2. a *string* used the same way.

immutable *adj.* not subject to change, either because no *operator* is provided which is capable of effecting such change or because some constraint exists which prohibits the use of an *operator* that might otherwise be capable of effecting such a change. Except as explicitly indicated otherwise, *implementations* are not required to detect attempts to modify *immutable objects* or *cells*; the consequences of attempting to make such modification are undefined. "Numbers are immutable."

implementation *n.* a system, mechanism, or body of *code* that implements the semantics of Common Lisp.

implementation limit *n.* a restriction imposed by an *implementation*.

implementation-defined *adj.* *implementation-dependent*, but required by this specification to be defined by each *conforming implementation* and to be documented by the corresponding implementor.

implementation-dependent *adj.* describing a behavior or aspect of Common Lisp which has been deliberately left unspecified, that might be defined in some *conforming implementations* but not in others, and whose details may differ between *implementations*. A *conforming implementation* is encouraged (but not required) to document its treatment of each item in this specification which is marked *implementation-dependent*, although in some cases such documentation might simply identify the item as "undefined."

implementation-independent *adj.* used to identify or emphasize a behavior or aspect of Common Lisp which does not vary between *conforming implementations*.

implicit block *n.* a *block* introduced by a *macro form* rather than by an explicit **block** *form*.

implicit compilation *n. compilation* performed during *evaluation*.

implicit progn *n.* an ordered set of adjacent *forms* appearing in another *form*, and defined by their context in that *form* to be executed as if within a **progn**.

implicit tagbody *n.* an ordered set of adjacent *forms* and/or *tags* appearing in another *form*, and defined by their context in that *form* to be executed as if within a **tagbody**.

import *v.t.* (a *symbol* into a *package*) to make the *symbol* be *present* in the *package*.

improper list *n.* a *list* which is not a *proper list*: a *circular list* or a *dotted list*.

inaccessible *adj.* not *accessible*.

indefinite extent *n.* an *extent* whose duration is unlimited. "Most Common Lisp objects have indefinite extent."

indefinite scope *n. scope* that is unlimited.

indicator *n.* a *property indicator*.

indirect instance *n.* (of a *class* C1) an *object* of *class* C2, where C2 is a *subclass* of C1. "An integer is an indirect instance of the class **number**."

inherit *v.t.* 1. to receive or acquire a quality, trait, or characteristic; to gain access to a feature defined elsewhere. 2. (a *class*) to acquire the structure and behavior defined by a *superclass*. 3. (a *package*) to make *symbols exported* by another *package accessible* by using **use-package**.

initial pprint dispatch table *n.* the *value* of ***print-pprint-dispatch*** at the time the *Lisp image* is started.

initial readtable *n.* the *value* of ***readtable*** at the time the *Lisp image* is started.

initialization argument list *n.* a *property list* of initialization argument *names* and *values* used in the protocol for initializing and reinitializing *instances* of *classes*. See Section 7.1 (Object Creation and Initialization).

initialization form *n.* a *form* used to supply the initial *value* for a *slot* or *variable*. "The initialization form for a slot in a **defclass** form is introduced by the keyword `:initform`."

input *adj.* (of a *stream*) supporting input operations (i.e., being a "data source"). An *input stream* might also be an *output stream*, in which case it is sometimes called a *bidirectional stream*. See the *function* **input-stream-p**.

instance *n.* 1. a *direct instance*. 2. a *generalized instance*. 3. an *indirect instance*.

integer *n.* an *object* of *type* **integer**, which represents a mathematical integer.

interactive stream *n.* a *stream* on which it makes sense to perform interactive querying. See Section 21.1.1.1.3 (Interactive Streams).

intern *v.t.* 1. (a *string* in a *package*) to look up the *string* in the *package*, returning either a *symbol* with that *name* which was already *accessible* in the *package* or a newly created *internal symbol* of the *package* with that *name*. 2. *Idiom.* generally, to observe a protocol whereby objects which are equivalent or have equivalent names under some predicate defined by the protocol are mapped to a single canonical object.

internal symbol *n.* (of a *package*) a symbol which is *accessible* in the *package*, but which is not an *external symbol* of the *package*.

internal time *n. time*, represented as an *integer* number of *internal time units*. *Absolute internal time* is measured as an offset from an arbitrarily chosen, *implementation-dependent* base. See Section 25.1.4.3 (Internal Time).

internal time unit *n.* a unit of time equal to 1/n of a second, for some *implementation-defined integer* value of n. See the *variable* **internal-time-units-per-second**.

interned *adj. Trad.* 1. (of a *symbol*) *accessible*[3] in any *package*. 2. (of a *symbol* in a specific *package*) *present* in that *package*.

interpreted function *n.* a *function* that is not a *compiled function*. (It is possible for there to be a *conforming implementation* which has no *interpreted functions*, but a *conforming program* must not assume that all *functions* are *compiled functions*.)

interpreted implementation *n.* an *implementation* that uses an execution strategy for *interpreted functions* that does not involve a one-time semantic analysis pre-pass, and instead uses "lazy" (and sometimes repetitious) semantic analysis of *forms* as they are encountered during execution.

interval designator *n.* (of *type* T) an ordered pair of *objects* that describe a *subtype* of T by delimiting an interval on the real number line. See Section 12.1.6 (Interval Designators).

invalid *n., adj.* 1. *n.* a possible *constituent trait* of a *character* which if present signifies that the *character* cannot ever appear in a *token* except under the control of a *single escape character*. For details, see Section 2.1.4.1 (Constituent Characters). 2. *adj.* (of a *character*) being a *character* that has *syntax type constituent* in the *current readtable* and that has the *constituent trait invalid*[1]. See Figure 2-8.

iteration form *n.* a *compound form* whose *operator* is named in the next figure, or a *compound form* that has an *implementation-defined operator* and that is defined by the *implementation* to be an *iteration form*.

```
do              do-external-symbols  dotimes
do*             do-symbols           loop
do-all-symbols  dolist
```

Figure 26-3. Standardized Iteration Forms

iteration variable *n.* a *variable* V, the *binding* for which was created by an *explicit use* of V in an *iteration form*.

K

key *n.* an *object* used for selection during retrieval. See *association list*, *property list*, and *hash table*. Also, see Section 17.1 (Sequence Concepts).

keyword *n.* 1. a *symbol* the *home package* of which is the KEYWORD package. 2. any *symbol*, usually but not necessarily in the KEYWORD package, that is used as an identifying marker in keyword-style argument passing. See **lambda**. 3. *Idiom.* a *lambda list keyword*.

keyword parameter *n.* A *parameter* for which a corresponding keyword *argument* is optional. (There is no such thing as a required keyword *argument*.) If the *argument* is not supplied, a default value is used. See also *supplied-p parameter*.

keyword/value pair *n.* two successive *elements* (a *keyword* and a *value*, respectively) of a *property list*.

L

lambda combination *n. Trad.* a *lambda form*.

lambda expression *n.* a *list* which can be used in place of a *function name* in certain contexts to denote a *function* by directly describing its behavior rather than indirectly by referring to the name of an *established function*; its name derives from the fact that its first element is the *symbol* `lambda`. See **lambda**.

lambda form *n.* a *form* that is a *list* and that has a first element which is a *lambda expression* representing a *function* to be called on *arguments* which are the result of *evaluating* subsequent elements of the *lambda form*.

lambda list *n.* a *list* that specifies a set of *parameters* (sometimes called *lambda variables*) and a protocol for receiving *values* for those *parameters*; that is, an *ordinary lambda list*, an *extended lambda list*, or a *modified lambda list*.

lambda list keyword *n.* a *symbol* whose *name* begins with *ampersand* and that is specially recognized in a *lambda list*. Note that no *standardized lambda list keyword* is in the `KEYWORD` package.

lambda variable *n.* a *formal parameter*, used to emphasize the *variable*'s relation to the *lambda list* that *established* it.

leaf *n.* 1. an *atom* in a *tree*[1]. 2. a terminal node of a *tree*[2].

leap seconds *n.* additional one-second intervals of time that are occasionally inserted into the true calendar by official timekeepers as a correction similar to "leap years." All Common Lisp *time* representations ignore *leap seconds*; every day is assumed to be exactly 86400 seconds long.

left-parenthesis *n.* the *standard character* "`(`", that is variously called "left parenthesis" or "open parenthesis" See Figure 2-5.

length *n.* (of a *sequence*) the number of *elements* in the *sequence*. (Note that if the *sequence* is a *vector* with a *fill pointer*, its *length* is the same as the *fill pointer* even though the total allocated size of the *vector* might be larger.)

lexical binding *n.* a *binding* in a *lexical environment*.

lexical closure *n.* a *function* that, when invoked on *arguments*, executes the body of a *lambda expression* in the *lexical environment* that was captured at the time of the creation of the *lexical closure*, augmented by *bindings* of the *function*'s *parameters* to the corresponding *arguments*.

lexical environment *n.* that part of the *environment* that contains *bindings* whose names have *lexical scope*. A *lexical environment* contains, among other things: ordinary *bindings* of *variable names* to *values*, lexically *established bindings* of *function names* to *functions*, *macros*, *symbol macros*, *blocks*, *tags*, and *local declarations* (see **declare**).

lexical scope *n. scope* that is limited to a spatial or textual region within the establishing *form*. "The names of parameters to a function normally are lexically scoped."

lexical variable *n.* a *variable* the *binding* for which is in the *lexical environment*.

Lisp image *n.* a running instantiation of a Common Lisp *implementation*. A *Lisp image* is characterized by a single address space in which any *object* can directly refer to any another in conformance with this specification, and by a single, common, *global environment*. (External operating systems sometimes call this a "core image," "fork," "incarnation," "job," or "process." Note however, that the issue of a "process" in such an operating system is technically orthogonal to the issue of a *Lisp image* being defined here. Depending on the operating system, a single "process" might have multiple *Lisp images*, and multiple "processes" might reside in a single *Lisp image*. Hence, it is the idea of a fully shared address space for direct reference among all *objects* which is the defining characteristic. Note, too, that two "processes" which have a communication area that permits the sharing of some but not all *objects* are considered to be distinct *Lisp images*.)

Lisp printer *n. Trad.* the procedure that prints the character representation of an *object* onto a *stream*. (This procedure is implemented by the *function* **write**.)

Lisp read-eval-print loop *n. Trad.* an endless loop that *reads*[2] a *form*, *evaluates* it, and prints (i.e., *writes*[2]) the results. In many *implementations*, the default mode of interaction with Common Lisp during program development is through such a loop.

Lisp reader *n. Trad.* the procedure that parses character representations of *objects* from a *stream*, producing *objects*. (This procedure is implemented by the *function* **read**.)

list *n.* 1. a chain of *conses* in which the *car* of each *cons* is an *element* of the *list*, and the *cdr* of each *cons* is either the next link in the chain or a terminating *atom*. See also *proper list*, *dotted list*, or *circular list*. 2. the *type* that is the union of **null** and **cons**.

list designator *n.* a *designator* for a *list* of *objects*; that is, an *object* that denotes a *list* and that is one of: a *non-nil atom* (denoting a *singleton list* whose *element* is that *non-nil atom*) or a *proper list* (denoting itself).

list structure *n.* (of a *list*) the set of *conses* that make up the *list*. Note that while the *car*[1b] component of each such *cons* is part of the *list structure*, the *objects* that are *elements* of the *list* (i.e., the *objects* that are the *cars*[2] of each *cons* in the *list*) are not themselves part of its *list structure*, even if they are *conses*, except in the (*circular*[2]) case where the *list* actually contains one of its *tails* as an *element*. (The *list structure* of a *list* is sometimes redundantly referred to as its "top-level list structure" in order to emphasize that any *conses* that are *elements* of the *list* are not involved.)

literal *adj.* (of an *object*) referenced directly in a program rather than being computed by the program; that is, appearing as data in a **quote** *form*, or, if the *object* is a *self-evaluating object*, appearing as unquoted data. "In the form `(cons "one" '("two"))`, the expressions `"one"`, `("two")`, and `"two"` are literal objects."

load *v.t.* (a *file*) to cause the *code* contained in the *file* to be *executed*. See the *function* **load**.

load time *n.* the duration of time that the loader is *loading compiled code*.

load time value *n.* an *object* referred to in *code* by a **load-time-value** *form*. The *value* of such a *form* is some specific *object* which can only be computed in the run-time *environment*. In the case of *file compilation*, the *value* is computed once as part of the process of *loading* the *compiled file*, and not again. See the *special operator* **load-time-value**.

loader *n.* a facility that is part of Lisp and that *loads* a *file*. See the *function* **load**.

local declaration *n.* an *expression* which may appear only in specially designated positions of certain *forms*, and which provides information about the code contained within the containing *form*; that is, a **declare** *expression*.

local precedence order *n.* (of a *class*) a *list* consisting of the *class* followed by its *direct superclasses* in the order mentioned in the defining *form* for the *class*.

local slot *n.* (of a *class*) a *slot accessible* in only one *instance*, namely the *instance* in which the *slot* is allocated.

logical block *n.* a conceptual grouping of related output used by the *pretty printer*. See the *macro* **pprint-logical-block** and Section 22.2.1.1 (Dynamic Control of the Arrangement of Output).

logical host *n.* an *object* of *implementation-dependent* nature that is used as the representation of a "host" in a *logical pathname*, and that has an associated set of translation rules for converting *logical pathnames* belonging to that host into *physical pathnames*. See Section 19.3 (Logical Pathnames).

logical host designator *n.* a *designator* for a *logical host*; that is, an *object* that denotes a *logical host* and that is one of: a *string* (denoting the *logical host* that it names), or a *logical host* (denoting itself). (Note that because the representation of a *logical host* is *implementation-dependent*, it is possible that an *implementation* might represent a *logical host* as the *string* that names it.)

logical pathname *n.* an *object* of *type* **logical-pathname**.

long float *n.* an *object* of *type* **long-float**.

loop keyword *n. Trad.* a symbol that is a specially recognized part of the syntax of an extended **loop** *form*. Such symbols are recognized by their *name* (using **string=**), not by their identity; as such, they may be in any package. A *loop keyword* is not a *keyword*.

lowercase *adj.* (of a *character*) being among *standard characters* corresponding to the small letters a through z, or being some other *implementation-defined character* that is defined by the *implementation* to be *lowercase*. See Section 13.1.4.3 (Characters With Case).

M

macro *n.* 1. a *macro form* 2. a *macro function*. 3. a *macro name*.

macro character *n.* a *character* which, when encountered by the *Lisp reader* in its main dispatch loop, introduces a *reader macro*[1]. (*Macro characters* have nothing to do with *macros*.)

macro expansion *n.* 1. the process of translating a *macro form* into another *form*. 2. the *form* resulting from this process.

macro form *n.* a *form* that stands for another *form* (e.g., for the purposes of abstraction, information hiding, or syntactic convenience); that is, either a *compound form* whose first element is a *macro name*, or a *form* that is a *symbol* that names a *symbol macro*.

macro function *n.* a *function* of two arguments, a *form* and an *environment*, that implements *macro expansion* by producing a *form* to be evaluated in place of the original argument *form*.

macro lambda list *n.* an *extended lambda list* used in *forms* that *establish macro* definitions, such as **defmacro** and **macrolet**. See Section 3.4.4 (Macro Lambda Lists).

macro name *n.* a *name* for which **macro-function** returns *true* and which when used as the first element of a *compound form* identifies that *form* as a *macro form*.

macroexpand hook *n.* the *function* that is the *value* of ***macroexpand-hook***.

mapping *n.* 1. a type of iteration in which a *function* is successively applied to *objects* taken from corresponding entries in collections such as *sequences* or *hash tables*. 2. *Math.* a relation between two sets in which each element of the first set (the "domain") is assigned one element of the second set (the "range").

metaclass *n.* 1. a *class* whose instances are *classes*. 2. (of an *object*) the *class* of the *class* of the *object*.

Metaobject Protocol *n.* one of many possible descriptions of how a *conforming implementation* might implement various aspects of the object system. This description is beyond the scope of this document, and no *conforming implementation* is required to adhere to it except as noted explicitly in this specification. Nevertheless, its existence helps to establish normative practice, and implementors with no reason to diverge from it are encouraged to consider making their *implementation* adhere to it where possible. It is described in detail in *The Art of the Metaobject Protocol*.

method *n.* an *object* that is part of a *generic function* and which provides information about how that *generic function* should behave when its *arguments* are *objects* of certain *classes* or with certain identities.

method combination *n.* 1. generally, the composition of a set of *methods* to produce an *effective method* for a *generic function*. 2. an object of *type* **method-combination**, which represents the details of how the *method combination*[1] for one or more specific *generic functions* is to be performed.

method-defining form *n.* a *form* that defines a *method* for a *generic function*, whether explicitly or implicitly. See Section 7.6.1 (Introduction to Generic Functions).

method-defining operator *n.* an *operator* corresponding to a *method-defining form*. See Figure 7-1.

minimal compilation *n.* actions the *compiler* must take at compile time. See Section 3.2.2 (Compilation Semantics).

modified lambda list *n.* a list resembling an *ordinary lambda list* in form and purpose, but which deviates in syntax or functionality from the definition of an *ordinary lambda list*. See *ordinary lambda list*. "**deftype** uses a modified lambda list."

most recent *adj.* innermost; that is, having been *established* (and not yet *disestablished*) more recently than any other of its kind.

multiple escape *n., adj.* 1. *n.* the *syntax type* of a *character* that is used in pairs to indicate that the enclosed *characters* are to be treated as *alphabetic*[2] *characters* with their *case* preserved. For details, see Section 2.1.4.5 (Multiple Escape Characters). 2. *adj.* (of a *character*) having the *multiple escape syntax type*. 3. *n.* a *multiple escape*[2] *character*. (In the *standard readtable*, *vertical-bar* is a *multiple escape character*.)

multiple values *n.* 1. more than one *value*. "The function **truncate** returns multiple values." 2. a variable number of *values*, possibly including zero or one. "The function **values** returns multiple values." 3. a fixed number of values other than one. "The macro **multiple-value-bind** is among the few operators in Common Lisp which can detect and manipulate multiple values."

N

name *n., v.t.* 1. *n.* an *identifier* by which an *object*, a *binding*, or an *exit point* is referred to by association using a *binding*. 2. *v.t.* to give a *name* to. 3. *n.* (of an *object* having a name component) the *object* which is that component. "The string which is a symbol's name is returned by **symbol-name**." 4. *n.* (of a *pathname*) a. the name component, returned by **pathname-name**. b. the entire namestring, returned by **namestring**. 5. *n.* (of a *character*) a *string* that names the *character* and that has *length* greater than one. (All *non-graphic characters* are required to have *names* unless they have some *implementation-defined attribute* which is not *null*. Whether or not other *characters* have *names* is *implementation-dependent*.)

named constant *n.* a *variable* that is defined by Common Lisp, by the *implementation*, or by user code (see the *macro* **defconstant**) to always *yield* the same *value* when *evaluated*. "The value of a named constant may not be changed by assignment or by binding."

namespace *n.* 1. *bindings* whose denotations are restricted to a particular kind. "The bindings of names to tags is the tag namespace." 2. any *mapping* whose domain is a set of *names*. "A package defines a namespace."

namestring *n.* a *string* that represents a *filename* using either the *standardized* notation for naming *logical pathnames* described in Section 19.3.1 (Syntax of Logical Pathname Namestrings), or some *implementation-defined* notation for naming a *physical pathname*.

newline *n.* the *standard character* <Newline>, notated for the *Lisp reader* as `#\Newline`.

next method *n.* the next *method* to be invoked with respect to a given *method* for a particular set of arguments or argument *classes*. See Section 7.6.6.1.3 (Applying method combination to the sorted list of applicable methods).

nickname *n.* (of a *package*) one of possibly several *names* that can be used to refer to the *package* but that is not the primary *name* of the *package*.

nil *n.* the *object* that is at once the *symbol* named `"NIL"` in the `COMMON-LISP` package, the *empty list*, the *boolean* (or *generalized boolean*) representing *false*, and the *name* of the *empty type*.

non-atomic *adj.* being other than an *atom*; i.e., being a *cons*.

non-constant variable *n.* a *variable* that is not a *constant variable*.

non-correctable *adj.* (of an *error*) not intentionally *correctable*. (Because of the dynamic nature of *restarts*, it is neither possible nor generally useful to completely prohibit an *error* from being *correctable*. This term is used in order to express an intent that no special effort should be made by *code* signaling an *error* to make that *error correctable*; however, there is no actual requirement on *conforming programs* or *conforming implementations* imposed by this term.)

non-empty *adj.* having at least one *element*.

non-generic function *n.* a *function* that is not a *generic function*.

non-graphic *adj.* (of a *character*) not *graphic*. See Section 13.1.4.1 (Graphic Characters).

non-list *n., adj.* other than a *list*; i.e., a *non-nil atom*.

non-local exit *n.* a transfer of control (and sometimes *values*) to an *exit point* for reasons other than a *normal return*. "The operators **go**, **throw**, and **return-from** cause a non-local exit."

non-nil *n., adj.* not **nil**. Technically, any *object* which is not **nil** can be referred to as *true*, but that would tend to imply a unique view of the *object* as a *generalized boolean*. Referring to such an *object* as *non-nil* avoids this implication.

non-null lexical environment *n.* a *lexical environment* that has additional information not present in the *global environment*, such as one or more *bindings*.

non-simple *adj.* not *simple*.

non-terminating *adj.* (of a *macro character*) being such that it is treated as a constituent *character* when it appears in the middle of an extended token. See Section 2.2 (Reader Algorithm).

non-top-level form *n.* a *form* that, by virtue of its position as a *subform* of another *form*, is not a *top level form*. See Section 3.2.3.1 (Processing of Top Level Forms).

normal return *n.* the natural transfer of control and *values* which occurs after the complete *execution* of a *form*.

normalized *adj., ANSI, IEEE* (of a *float*) conforming to the description of "normalized" as described by *IEEE Standard for Binary Floating-Point Arithmetic*. See *denormalized*.

null *adj., n.* 1. *adj.* a. (of a *list*) having no *elements*: empty. See *empty list*. b. (of a *string*) having a *length* of zero. (It is common, both within this document and in observed spoken behavior, to refer to an empty string by an apparent definite reference, as in "the *null string*" even though no attempt is made to *intern*[2] null strings. The phrase "a *null string*" is technically more correct, but is generally considered awkward by most Lisp programmers. As such, the phrase "the *null string*" should be treated as an indefinite reference in all cases except for anaphoric references.) c. (of an *implementation-defined attribute* of a *character*) An *object* to which the value of that *attribute* defaults if no specific value was requested. 2. *n.* an *object* of *type* **null** (the only such *object* being **nil**).

null lexical environment *n.* the *lexical environment* which has no *bindings*.

number *n.* an *object* of *type* **number**.

numeric *adj*. (of a *character*) being one of the *standard characters* `0` through *9*, or being some other *graphic character* defined by the *implementation* to be *numeric*.

O

object *n.* 1. any Lisp datum. "The function **cons** creates an object which refers to two other objects." 2. (immediately following the name of a *type*) an *object* which is of that *type*, used to emphasize that the *object* is not just a *name* for an object of that *type* but really an *element* of the *type* in cases where *objects* of that *type* (such as **function** or **class**) are commonly referred to by *name*. "The function **symbol-function** takes a function name and returns a function object."

object-traversing *adj*. operating in succession on components of an *object*. "The operators **mapcar**, **maphash**, **with-package-iterator** and **count** perform object-traversing operations."

open *adj.*, *v.t.* (a *file*) 1. *v.t.* to create and return a *stream* to the *file*. 2. *adj*. (of a *stream*) having been *opened*[1], but not yet *closed*.

operator *n.* 1. a *function*, *macro*, or *special operator*. 2. a *symbol* that names such a *function*, *macro*, or *special operator*. 3. (in a **function** *special form*) the *cadr* of the **function** *special form*, which might be either an *operator*[2] or a *lambda expression*. 4. (of a *compound form*) the *car* of the *compound form*, which might be either an *operator*[2] or a *lambda expression*, and which is never `(setf symbol)`.

optimize quality *n.* one of several aspects of a program that might be optimizable by certain compilers. Since optimizing one such quality might conflict with optimizing another, relative priorities for qualities can be established in an **optimize** *declaration*. The *standardized optimize qualities* are `compilation-speed` (speed of the compilation process), `debug` (ease of debugging), `safety` (run-time error checking), `space` (both code size and run-time space), and `speed` (of the object code). *Implementations* may define additional *optimize qualities*.

optional parameter *n.* A *parameter* for which a corresponding positional *argument* is optional. If the *argument* is not supplied, a default value is used. See also *supplied-p parameter*.

ordinary function *n.* a *function* that is not a *generic function*.

ordinary lambda list *n.* the kind of *lambda list* used by **lambda**. See *modified lambda list* and *extended lambda list*. "**defun** uses an ordinary lambda list."

otherwise inaccessible part *n.* (of an *object*, O1) an *object*, O2, which would be made *inaccessible* if O1 were made *inaccessible*. (Every *object* is an *otherwise inaccessible part* of itself.)

output *adj*. (of a *stream*) supporting output operations (i.e., being a "data sink"). An *output stream* might also be an *input stream*, in which case it is sometimes called a *bidirectional stream*. See the *function* **output-stream-p**.

P

package *n.* an *object* of *type* **package**.

package cell *n. Trad.* (of a *symbol*) The *place* in a *symbol* that holds one of possibly several *packages* in which the *symbol* is *interned*, called the *home package*, or which holds **nil** if no such *package* exists or is known. See the *function* **symbol-package**.

package designator *n.* a *designator* for a *package*; that is, an *object* that denotes a *package* and that is one of: a *string designator* (denoting the *package* that has the *string* that it designates as its *name* or as one of its *nicknames*), or a *package* (denoting itself).

package marker *n.* a character which is used in the textual notation for a symbol to separate the package name from the symbol name, and which is *colon* in the *standard readtable*. See Section 2.1 (Character Syntax).

package prefix *n.* a notation preceding the *name* of a *symbol* in text that is processed by the *Lisp reader*, which uses a *package name* followed by one or more *package markers*, and which indicates that the symbol is looked up in the indicated *package*.

package registry *n.* A mapping of *names* to *package objects*. It is possible for there to be a *package object* which is not in this mapping; such a *package* is called an *unregistered package*. *Operators* such as **find-package** consult this mapping in order to find a *package* from its *name*. *Operators* such as **do-all-symbols**, **find-all-symbols**, and **list-all-packages** operate only on *packages* that exist in the *package registry*.

pairwise *adv.* (of an adjective on a set) applying individually to all possible pairings of elements of the set. "The types A, B, and C are pairwise disjoint if A and B are disjoint, B and C are disjoint, and A and C are disjoint."

parallel *adj. Trad.* (of *binding* or *assignment*) done in the style of **psetq**, **let**, or **do**; that is, first evaluating all of the *forms* that produce *values*, and only then *assigning* or *binding* the *variables* (or *places*). Note that this does not imply traditional computational "parallelism" since the *forms* that produce *values* are evaluated *sequentially*. See *sequential*.

parameter *n.* 1. (of a *function*) a *variable* in the definition of a *function* which takes on the *value* of a corresponding *argument* (or of a *list* of corresponding arguments) to that *function* when it is called, or which in some cases is given a default value because there is no corresponding *argument*. 2. (of a *format directive*) an *object* received as data flow by a *format directive* due to a prefix notation within the *format string* at the *format directive*'s point of use. See Section 22.3 (Formatted Output). "In `"~3,'0D"`, the number `3` and the character `#\0` are parameters to the `~D` format directive."

parameter specializer *n.* 1. (of a *method*) an *expression* which constrains the *method* to be applicable only to *argument* sequences in which the corresponding *argument* matches the *parameter specializer*. 2. a *class*, or a *list* `(eql` *`object`*`)`.

parameter specializer name *n.* 1. (of a *method* definition) an expression used in code to name a *parameter specializer*. See Section 7.6.2 (Introduction to Methods). 2. a *class*, a *symbol* naming a *class*, or a *list* `(eql` *`form`*`)`.

pathname *n.* an *object* of *type* **pathname**, which is a structured representation of the name of a *file*. A *pathname* has six components: a "host," a "device," a "directory," a "name," a "type," and a "version."

pathname designator *n.* a *designator* for a *pathname*; that is, an *object* that denotes a *pathname* and that is one of: a *pathname namestring* (denoting the corresponding *pathname*), a *stream associated with a file* (denoting the *pathname* used to open the *file*; this may be, but is not required to be, the actual name of the *file*), or a *pathname* (denoting itself). See Section 21.1.1.1.2 (Open and Closed Streams).

physical pathname *n.* a *pathname* that is not a *logical pathname*.

place *n.* 1. a *form* which is suitable for use as a *generalized reference*. 2. the conceptual location referred to by such a *place*[1].

plist [’pee,list] *n.* a *property list*.

portable *adj.* (of *code*) required to produce equivalent results and observable side effects in all *conforming implementations*.

potential copy *n.* (of an *object* O1 subject to constriants) an *object* O2 that if the specified constraints are satisfied by O1 without any modification might or might not be *identical* to O1, or else that must be a *fresh object* that resembles a *copy* of O1 except that it has been modified as necessary to satisfy the constraints.

potential number *n.* A textual notation that might be parsed by the *Lisp reader* in some *conforming implementation* as a *number* but is not required to be parsed as a *number*. No *object* is a *potential number*---either an *object* is a *number* or it is not. See Section 2.3.1.1 (Potential Numbers as Tokens).

pprint dispatch table *n.* an *object* that can be the *value* of ***print-pprint-dispatch*** and hence can control how *objects* are printed when ***print-pretty*** is *true*. See Section 22.2.1.4 (Pretty Print Dispatch Tables).

predicate *n.* a *function* that returns a *generalized boolean* as its first value.

present *n.* 1. (of a *feature* in a *Lisp image*) a state of being that is in effect if and only if the *symbol* naming the *feature* is an *element* of the *features list*. 2. (of a *symbol* in a *package*) being accessible in that *package* directly, rather than being inherited from another *package*.

pretty print *v.t.* (an *object*) to invoke the *pretty printer* on the *object*.

pretty printer *n.* the procedure that prints the character representation of an *object* onto a *stream* when the *value* of ***print-pretty*** is *true*, and that uses layout techniques (e.g., indentation) that tend to highlight the structure of the *object* in a way that makes it easier for human readers to parse visually. See the *variable* ***print-pprint-dispatch*** and Section 22.2 (The Lisp Pretty Printer).

pretty printing stream *n.* a *stream* that does pretty printing. Such streams are created by the *function* **pprint-logical-block** as a link between the output stream and the logical block.

primary method *n.* a member of one of two sets of *methods* (the set of *auxiliary methods* is the other) that form an exhaustive partition of the set of *methods* on the *method*'s *generic function*. How these sets are determined is dependent on the *method combination* type; see Section 7.6.2 (Introduction to Methods).

primary value *n.* (of *values* resulting from the *evaluation* of a *form*) the first *value*, if any, or else **nil** if there are no *values*. "The primary value returned by **truncate** is an integer quotient, truncated toward zero."

principal *adj.* (of a value returned by a Common Lisp *function* that implements a mathematically irrational or transcendental function defined in the complex domain) of possibly many (sometimes an infinite number of) correct values for the mathematical function, being the particular *value* which the corresponding Common Lisp *function* has been defined to return.

print name *n. Trad.* (usually of a *symbol*) a *name*[3].

printer control variable *n.* a *variable* whose specific purpose is to control some action of the *Lisp printer*; that is, one of the *variables* in Figure 22-1, or else some *implementation-defined variable* which is defined by the *implementation* to be a *printer control variable*.

printer escaping *n.* The combined state of the *printer control variables* ***print-escape*** and ***print-readably***. If the value of either ***print-readably*** or ***print-escape*** is *true*, then *printer escaping* is "enabled"; otherwise (if the values of both ***print-readably*** and ***print-escape*** are *false*), then *printer escaping* is "disabled".

printing *adj.* (of a *character*) being a *graphic character* other than *space*.

process *v.t.* (a *form* by the *compiler*) to perform *minimal compilation*, determining the time of evaluation for a *form*, and possibly *evaluating* that *form* (if required).

processor *n., ANSI* an *implementation*.

proclaim *v.t.* (a *proclamation*) to *establish* that *proclamation*.

proclamation *n.* a *global declaration*.

prog tag *n. Trad.* a *go tag*.

program *n. Trad.* Common Lisp *code*.

programmer *n.* an active entity, typically a human, that writes a *program*, and that might or might not also be a *user* of the *program*.

programmer code *n. code* that is supplied by the programmer; that is, *code* that is not *system code*.

proper list *n.* A *list* terminated by the *empty list*. (The *empty list* is a *proper list*.) See *improper list*.

proper name *n.* (of a *class*) a *symbol* that *names* the *class* whose *name* is that *symbol*. See the *functions* **class-name** and **find-class**.

proper sequence *n.* a *sequence* which is not an *improper list*; that is, a *vector* or a *proper list*.

proper subtype *n.* (of a *type*) a *subtype* of the *type* which is not the *same type* as the *type* (i.e., its *elements* are a "proper subset" of the *type*).

property *n.* (of a *property list*) 1. a conceptual pairing of a *property indicator* and its associated *property value* on a *property list*. 2. a *property value*.

property indicator *n.* (of a *property list*) the *name* part of a *property*, used as a *key* when looking up a *property value* on a *property list*.

property list *n.* 1. a *list* containing an even number of *elements* that are alternating *names* (sometimes called *indicators* or *keys*) and *values* (sometimes called *properties*). When there is more than one *name* and *value* pair with the *identical name* in a *property list*, the first such pair determines the *property*. 2. (of a *symbol*) the component of the *symbol* containing a *property list*.

property value *n.* (of a *property indicator* on a *property list*) the *object* associated with the *property indicator* on the *property list*.

purports to conform *v.* makes a good-faith claim of conformance. This term expresses intention to conform, regardless of whether the goal of that intention is realized in practice. For example, language implementations have been known to have bugs, and while an *implementation* of this specification with bugs might not be a *conforming implementation*, it can still *purport to conform*. This is an important distinction in certain specific cases; e.g., see the *variable* ***features***.

Q

qualified method *n.* a *method* that has one or more *qualifiers*.

qualifier *n.* (of a *method* for a *generic function*) one of possibly several *objects* used to annotate the *method* in a way that identifies its role in the *method combination*. The *method combination type* determines how many *qualifiers* are permitted for each *method*, which *qualifiers* are permitted, and the semantics of those *qualifiers*.

query I/O *n.* the *bidirectional stream* that is the *value* of the *variable* ***query-io****.

quoted object *n.* an *object* which is the second element of a **quote** *form*.

R

radix *n.* an *integer* between 2 and 36, inclusive, which can be used to designate a base with respect to which certain kinds of numeric input or output are performed. (There are n valid digit characters for any given *radix* n, and those digits are the first n digits in the sequence `0`, `1`, ..., `9`, `A`, `B`, ..., `Z`, which have the weights `0`, `1`, ..., `9`, `10`, `11`, ..., `35`, respectively. Case is not significant in parsing numbers of radix greater than `10`, so "9b8a" and "9B8A" denote the same *radix* `16` number.)

random state *n.* an *object* of *type* **random-state**.

rank *n.* a non-negative *integer* indicating the number of *dimensions* of an *array*.

ratio *n.* an *object* of *type* **ratio**.

ratio marker *n.* a character which is used in the textual notation for a *ratio* to separate the numerator from the denominator, and which is *slash* in the *standard readtable*. See Section 2.1 (Character Syntax).

rational *n.* an *object* of *type* **rational**.

read *v.t.* 1. (a *binding* or *slot* or component) to obtain the *value* of the *binding* or *slot*. 2. (an *object* from a *stream*) to parse an *object* from its representation on the *stream*.

readably *adv.* (of a manner of printing an *object* O1) in such a way as to permit the *Lisp Reader* to later *parse* the printed output into an *object* O2 that is *similar* to O1.

reader *n.* 1. a *function* that *reads*[1] a *variable* or *slot*. 2. the *Lisp reader*.

reader macro *n.* 1. a textual notation introduced by dispatch on one or two *characters* that defines special-purpose syntax for use by the *Lisp reader*, and that is implemented by a *reader macro function*. See Section 2.2 (Reader Algorithm). 2. the *character* or *characters* that introduce a *reader macro*[1]; that is, a *macro character* or the conceptual pairing of a *dispatching macro character* and the *character* that follows it. (A *reader macro* is not a kind of *macro*.)

reader macro function *n.* a *function designator* that denotes a *function* that implements a *reader macro*[2]. See the *functions* **set-macro-character** and **set-dispatch-macro-character**.

readtable *n.* an *object* of *type* **readtable**.

readtable case *n.* an attribute of a *readtable* whose value is a *case sensitivity mode*, and that selects the manner in which *characters* in a *symbol*'s *name* are to be treated by the *Lisp reader* and the *Lisp printer*. See Section 23.1.2 (Effect of Readtable Case on the Lisp Reader) and Section 22.1.3.3.2 (Effect of Readtable Case on the Lisp Printer).

readtable designator *n.* a *designator* for a *readtable*; that is, an *object* that denotes a *readtable* and that is one of: **nil** (denoting the *standard readtable*), or a *readtable* (denoting itself).

recognizable subtype *n.* (of a *type*) a *subtype* of the *type* which can be reliably detected to be such by the *implementation*. See the *function* **subtypep**.

reference *n., v.t.* 1. *n.* an act or occurrence of referring to an *object*, a *binding*, an *exit point*, a *tag*, or an *environment*. 2. *v.t.* to refer to an *object*, a *binding*, an *exit point*, a *tag*, or an *environment*, usually by *name*.

registered package *n.* a *package object* that is installed in the *package registry*. (Every *registered package* has a *name* that is a *string*, as well as zero or more *string* nicknames. All *packages* that are initially specified by Common Lisp or created by **make-package** or **defpackage** are *registered packages*. *Registered packages* can be turned into *unregistered packages* by **delete-package**.)

relative *adj.* 1. (of a *time*) representing an offset from an *absolute time* in the units appropriate to that time. For example, a *relative internal time* is the difference between two *absolute internal times*, and is measured in *internal time units*. 2. (of a *pathname*) representing a position in a directory hierarchy by motion from a position other than the root, which might therefore vary. "The notation `#P"../foo.text"` denotes a relative pathname if the host file system is Unix." See *absolute*.

repertoire *n.*, *ISO* a *subtype* of **character**. See Section 13.1.2.2 (Character Repertoires).

report *n.* (of a *condition*) to *call* the *function* **print-object** on the *condition* in an *environment* where the *value* of ***print-escape*** is *false*.

report message *n.* the text that is output by a *condition reporter*.

required parameter *n.* A *parameter* for which a corresponding positional *argument* must be supplied when *calling* the *function*.

rest list *n.* (of a *function* having a *rest parameter*) The *list* to which the *rest parameter* is *bound* on some particular *call* to the *function*.

rest parameter *n.* A *parameter* which was introduced by `&rest`.

restart *n.* an *object* of *type* **restart**.

restart designator *n.* a *designator* for a *restart*; that is, an *object* that denotes a *restart* and that is one of: a *non-nil symbol* (denoting the most recently established *active restart* whose *name* is that *symbol*), or a *restart* (denoting itself).

restart function *n.* a *function* that invokes a *restart*, as if by **invoke-restart**. The primary purpose of a *restart function* is to provide an alternate interface. By convention, a *restart function* usually has the same name as the *restart* which it invokes. The next figure shows a list of the *standardized restart functions*.

```
abort     muffle-warning  use-value
continue  store-value
```

Figure 26-4. Standardized Restart Functions

return *v.t.* (of *values*) 1. (from a *block*) to transfer control and *values* from the *block*; that is, to cause the *block* to *yield* the *values* immediately without doing any further evaluation of the *forms* in its body. 2. (from a *form*) to *yield* the *values*.

return value *n. Trad.* a *value*[1]

right-parenthesis *n.* the *standard character* `")"`, that is variously called "right parenthesis" or "close parenthesis" See Figure 2-5.

run time *n.* 1. *load time* 2. *execution time*

run-time compiler *n.* refers to the **compile** function or to *implicit compilation*, for which the compilation and run-time *environments* are maintained in the same *Lisp image*.

run-time definition *n.* a definition in the *run-time environment*.

run-time environment *n.* the *environment* in which a program is *executed*.

S

safe *adj.* 1. (of *code*) processed in a *lexical environment* where the the highest **safety** level (`3`) was in effect. See **optimize**. 2. (of a *call*) a *safe call*.

safe call *n.* a *call* in which the *call*, the *function* being *called*, and the point of *functional evaluation* are all *safe*[1] *code*. For more detailed information, see Section 3.5.1.1 (Safe and Unsafe Calls).

same *adj.* 1. (of *objects* under a specified *predicate*) indistinguishable by that *predicate*. "The symbol `car`, the string `"car"`, and the string `"CAR"` are the `same` under **string-equal**". 2. (of *objects* if no predicate is implied by context) indistinguishable by **eql**. Note that **eq** might be capable of distinguishing some *numbers* and *characters* which **eql** cannot distinguish, but the nature of such, if any, is *implementation-dependent*. Since **eq** is used only rarely in this specification, **eql** is the default predicate when none is mentioned explicitly. "The conses returned by two successive calls to **cons** are never the same." 3. (of *types*) having the same set of *elements*; that is, each *type* is a *subtype* of the others. "The types specified by `(integer 0 1)`, `(unsigned-byte 1)`, and `bit` are the same."

satisfy the test *v.* (of an *object* being considered by a *sequence function*) 1. (for a one *argument* test) to be in a state such that the *function* which is the *predicate argument* to the *sequence function* returns *true* when given a single *argument* that is the result of calling the *sequence function*'s *key argument* on the *object* being considered. See Section 17.2.2 (Satisfying a One-Argument Test). 2. (for a two *argument* test) to be in a state such that the two-place *predicate* which is the *sequence function*'s *test argument* returns *true* when given a first *argument* that is the *object* being considered, and when given a second *argument* that is the result of calling the *sequence function*'s *key argument* on an *element* of the *sequence function*'s *sequence argument* which is being tested for equality; or to be in a state such that the *test-not function* returns *false* given the same *arguments*. See Section 17.2.1 (Satisfying a Two-Argument Test).

scope *n.* the structural or textual region of code in which *references* to an *object*, a *binding*, an *exit point*, a *tag*, or an *environment* (usually by *name*) can occur.

script *n. ISO* one of possibly several sets that form an *exhaustive partition* of the type **character**. See Section 13.1.2.1 (Character Scripts).

secondary value *n.* (of *values* resulting from the *evaluation* of a *form*) the second *value*, if any, or else **nil** if there are fewer than two *values*. "The secondary value returned by **truncate** is a remainder."

section *n.* a partitioning of output by a *conditional newline* on a *pretty printing stream*. See Section 22.2.1.1 (Dynamic Control of the Arrangement of Output).

self-evaluating object *n.* an *object* that is neither a *symbol* nor a *cons*. If a *self-evaluating object* is *evaluated*, it *yields* itself as its only *value*. "Strings are self-evaluating objects."

semi-standard *adj.* (of a language feature) not required to be implemented by any *conforming implementation*, but nevertheless recommended as the canonical approach in situations where an *implementation* does plan to support such a feature. The presence of *semi-standard* aspects in the language is intended to lessen portability problems and reduce the risk of gratuitous divergence among *implementations* that might stand in the way of future standardization.

semicolon *n.* the *standard character* that is called "semicolon" (`;`). See Figure 2-5.

sequence *n.* 1. an ordered collection of elements 2. a *vector* or a *list*.

sequence function *n.* one of the *functions* in Figure 17-1, or an *implementation-defined function* that operates on one or more *sequences*. and that is defined by the *implementation* to be a *sequence function*.

sequential *adj. Trad.* (of *binding* or *assignment*) done in the style of **setq**, **let***, or **do***; that is, interleaving the evaluation of the *forms* that produce *values* with the *assignments* or *bindings* of the *variables* (or *places*). See *parallel*.

sequentially *adv.* in a *sequential* way.

serious condition *n.* a *condition* of *type* **serious-condition**, which represents a *situation* that is generally sufficiently severe that entry into the *debugger* should be expected if the *condition* is *signaled* but not *handled*.

session *n.* the conceptual aggregation of events in a *Lisp image* from the time it is started to the time it is terminated.

set *v.t. Trad.* (any *variable* or a *symbol* that is the *name* of a *dynamic variable*) to *assign* the *variable*.

setf expander *n.* a function used by **setf** to compute the *setf expansion* of a *place*.

setf expansion *n.* a set of five *expressions*[1] that, taken together, describe how to store into a *place* and which *subforms* of the macro call associated with the *place* are evaluated. See Section 5.1.1.2 (Setf Expansions).

setf function *n.* a *function* whose *name* is `(setf symbol)`.

setf function name *n.* (of a *symbol S*) the *list* `(setf S)`.

shadow *v.t.* 1. to override the meaning of. "That binding of `X` shadows an outer one." 2. to hide the presence of. "That **macrolet** of `F` shadows the outer **flet** of `F`." 3. to replace. "That package shadows the symbol `cl:car` with its own symbol `car`."

shadowing symbol *n.* (in a *package*) an *element* of the *package*'s *shadowing symbols list*.

shadowing symbols list *n.* (of a *package*) a *list*, associated with the *package*, of *symbols* that are to be exempted from 'symbol conflict errors' detected when packages are *used*. See the *function* **package-shadowing-symbols**.

shared slot *n.* (of a *class*) a *slot accessible* in more than one *instance* of a *class*; specifically, such a *slot* is *accessible* in all *direct instances* of the *class* and in those *indirect instances* whose *class* does not *shadow*[1] the *slot*.

sharpsign *n.* the *standard character* that is variously called "number sign," "sharp," or "sharp sign" (#). See Figure 2-5.

short float *n.* an *object* of *type* **short-float**.

sign *n.* one of the *standard characters* "`+`" or "`-`".

signal *v.* to announce, using a standard protocol, that a particular situation, represented by a *condition*, has been detected. See Section 9.1 (Condition System Concepts).

signature *n.* (of a *method*) a description of the *parameters* and *parameter specializers* for the *method* which determines the *method*'s applicability for a given set of required *arguments*, and which also describes the *argument* conventions for its other, non-required *arguments*.

similar *adj.* (of two *objects*) defined to be equivalent under the *similarity* relationship.

similarity *n.* a two-place conceptual equivalence predicate, which is independent of the *Lisp image* so that two *objects* in different *Lisp images* can be understood to be equivalent under this predicate. See Section 3.2.4 (Literal Objects in Compiled Files).

simple *adj.* 1. (of an *array*) being of *type* **simple-array**. 2. (of a *character*) having no *implementation-defined attributes*, or else having *implementation-defined attributes* each of which has the *null* value for that *attribute*.

simple array *n.* an *array* of *type* **simple-array**.

simple bit array *n.* a *bit array* that is a *simple array*; that is, an *object* of *type* `(simple-array bit)`.

simple bit vector *n.* a *bit vector* of *type* **simple-bit-vector**.

simple condition *n.* a *condition* of *type* **simple-condition**.

simple general vector *n.* a *simple vector*.

simple string *n.* a *string* of *type* **simple-string**.

simple vector *n.* a *vector* of *type* **simple-vector**, sometimes called a "*simple general vector*." Not all *vectors* that are *simple* are *simple vectors*---only those that have *element type* **t**.

single escape *n., adj.* 1. *n.* the *syntax type* of a *character* that indicates that the next *character* is to be treated as an *alphabetic*[2] *character* with its *case* preserved. For details, see Section 2.1.4.6 (Single Escape Character). 2. *adj.* (of a *character*) having the *single escape syntax type*. 3. *n.* a *single escape*[2] *character*. (In the *standard readtable*, *slash* is the only *single escape*.)

single float *n.* an *object* of *type* **single-float**.

single-quote *n.* the *standard character* that is variously called "apostrophe," "acute accent," "quote," or "single quote" (`'`). See Figure 2-5.

singleton *adj.* (of a *sequence*) having only one *element*. "`(list 'hello)` returns a singleton list."

situation *n.* the *evaluation* of a *form* in a specific *environment*.

slash *n.* the *standard character* that is variously called "solidus" or "slash" (`/`). See Figure 2-5.

slot *n.* a component of an *object* that can store a *value*.

slot specifier *n.* a representation of a *slot* that includes the *name* of the *slot* and zero or more *slot* options. A *slot* option pertains only to a single *slot*.

source code *n.* *code* representing *objects* suitable for *evaluation* (e.g., *objects* created by **read**, by *macro expansion*, or by *compiler macro expansion*).

source file *n.* a *file* which contains a textual representation of *source code*, that can be edited, *loaded*, or *compiled*.

space *n.* the *standard character* <Space>, notated for the *Lisp reader* as `#\Space`.

special form *n.* a *list*, other than a *macro form*, which is a *form* with special syntax or special *evaluation* rules or both, possibly manipulating the *evaluation environment* or control flow or both. The first element of a *special form* is a *special operator*.

special operator *n.* one of a fixed set of *symbols*, enumerated in Figure 3-2, that may appear in the *car* of a *form* in order to identify the *form* as a *special form*.

special variable *n. Trad.* a *dynamic variable*.

specialize *v.t.* (a *generic function*) to define a *method* for the *generic function*, or in other words, to refine the behavior of the *generic function* by giving it a specific meaning for a particular set of *classes* or *arguments*.

specialized *adj.* 1. (of a *generic function*) having *methods* which *specialize* the *generic function*. 2. (of an *array*) having an *actual array element type* that is a *proper subtype* of the *type* **t**; see Section 15.1.1 (Array Elements). "`(make-array 5 :element-type 'bit)` makes an array of length five that is specialized for bits."

specialized lambda list *n.* an *extended lambda list* used in *forms* that *establish method* definitions, such as **defmethod**. See Section 3.4.3 (Specialized Lambda Lists).

spreadable argument list designator *n.* a *designator* for a *list* of *objects*; that is, an *object* that denotes a *list* and that is a *non-null list* L1 of length n, whose last element is a *list* L2 of length m (denoting a list L3 of length m+n-1 whose *elements* are L1i for i < n-1 followed by L2j for j < m). "The list (1 2 (3 4 5)) is a spreadable argument list designator for the list (1 2 3 4 5)."

stack allocate *v.t. Trad.* to allocate in a non-permanent way, such as on a stack. Stack-allocation is an optimization technique used in some *implementations* for allocating certain kinds of *objects* that have *dynamic extent*. Such *objects* are allocated on the stack rather than in the heap so that their storage can be freed as part of unwinding the stack rather than taking up space in the heap until the next garbage collection. What *types* (if any) can have *dynamic extent* can vary from *implementation* to *implementation*. No *implementation* is ever required to perform stack-allocation.

stack-allocated *adj. Trad.* having been *stack allocated*.

standard character *n.* a *character* of *type* **standard-char**, which is one of a fixed set of 96 such *characters* required to be present in all *conforming implementations*. See Section 2.1.3 (Standard Characters).

standard class *n.* a *class* that is a *generalized instance* of *class* **standard-class**.

standard generic function a *function* of *type* **standard-generic-function**.

standard input *n.* the *input stream* which is the *value* of the *dynamic variable* ***standard-input***.

standard method combination *n.* the *method combination* named **standard**.

standard object *n.* an *object* that is a *generalized instance* of *class* **standard-object**.

standard output *n.* the *output stream* which is the *value* of the *dynamic variable* ***standard-output***.

standard pprint dispatch table *n.* A *pprint dispatch table* that is *different* from the *initial pprint dispatch table*, that implements *pretty printing* as described in this specification, and that, unlike other *pprint dispatch tables*, must never be modified by any program. (Although the definite reference "the *standard pprint dispatch table*" is generally used within this document, it is actually *implementation-dependent* whether a single *object* fills the role of the *standard pprint dispatch table*, or whether there might be multiple such objects, any one of which could be used on any given occasion where "the *standard pprint dispatch table*" is called for. As such, this phrase should be seen as an indefinite reference in all cases except for anaphoric references.)

standard readtable *n.* A *readtable* that is *different* from the *initial readtable*, that implements the *expression* syntax defined in this specification, and that, unlike other *readtables*, must never be modified by any program. (Although the definite reference "the *standard readtable*" is generally used within this document, it is actually *implementation-dependent* whether a single *object* fills the role of the *standard readtable*, or whether there might

be multiple such objects, any one of which could be used on any given occasion where "the *standard readtable*" is called for. As such, this phrase should be seen as an indefinite reference in all cases except for anaphoric references.)

standard syntax *n.* the syntax represented by the *standard readtable* and used as a reference syntax throughout this document. See Section 2.1 (Character Syntax).

standardized *adj.* (of a *name*, *object*, or definition) having been defined by Common Lisp. "All standardized variables that are required to hold bidirectional streams have "`-io*`" in their name."

startup environment *n.* the *global environment* of the running *Lisp image* from which the *compiler* was invoked.

step *v.t.*, *n.* 1. *v.t.* (an iteration *variable*) to *assign* the *variable* a new *value* at the end of an iteration, in preparation for a new iteration. 2. *n.* the *code* that identifies how the next value in an iteration is to be computed. 3. *v.t.* (*code*) to specially execute the *code*, pausing at intervals to allow user confirmation or intervention, usually for debugging.

stream *n.* an *object* that can be used with an input or output function to identify an appropriate source or sink of *characters* or *bytes* for that operation.

stream associated with a file *n.* a *file stream*, or a *synonym stream* the *target* of which is a *stream associated with a file*. Such a *stream* cannot be created with **make-two-way-stream**, **make-echo-stream**, **make-broadcast-stream**, **make-concatenated-stream**, **make-string-input-stream**, or **make-string-output-stream**.

stream designator *n.* a *designator* for a *stream*; that is, an *object* that denotes a *stream* and that is one of: **t** (denoting the *value* of ***terminal-io***), **nil** (denoting the *value* of ***standard-input*** for *input stream designators* or denoting the *value* of ***standard-output*** for *output stream designators*), or a *stream* (denoting itself).

stream element type *n.* (of a *stream*) the *type* of data for which the *stream* is specialized.

stream variable *n.* a *variable* whose *value* must be a *stream*.

stream variable designator *n.* a *designator* for a *stream variable*; that is, a *symbol* that denotes a *stream variable* and that is one of: **t** (denoting ***terminal-io***), **nil** (denoting ***standard-input*** for *input stream variable designators* or denoting ***standard-output*** for *output stream variable designators*), or some other *symbol* (denoting itself).

string *n.* a specialized *vector* that is of *type* **string**, and whose elements are of *type* **character** or a *subtype* of *type* **character**.

string designator *n.* a *designator* for a *string*; that is, an *object* that denotes a *string* and that is one of: a *character* (denoting a *singleton string* that has the *character* as its only *element*), a *symbol* (denoting the *string* that is its *name*), or a *string* (denoting itself). The intent is that this term be consistent with the behavior of **string**; *implementations* that extend **string** must extend the meaning of this term in a compatible way.

string equal *adj.* the *same* under **string-equal**.

string stream *n.* a *stream* of *type* **string-stream**.

structure *n.* an *object* of *type* **structure-object**.

structure class *n.* a *class* that is a *generalized instance* of *class* **structure-class**.

structure name *n.* a *name* defined with **defstruct**. Usually, such a *type* is also a *structure class*, but there may be *implementation-dependent* situations in which this is not so, if the `:type` option to **defstruct** is used.

style warning *n.* a *condition* of *type* **style-warning**.

subclass *n.* a *class* that *inherits* from another *class*, called a *superclass*. (No *class* is a *subclass* of itself.)

subexpression *n.* (of an *expression*) an *expression* that is contained within the *expression*. (In fact, the state of being a *subexpression* is not an attribute of the *subexpression*, but really an attribute of the containing *expression* since the *same object* can at once be a *subexpression* in one context, and not in another.)

subform *n.* (of a *form*) an *expression* that is a *subexpression* of the *form*, and which by virtue of its position in that *form* is also a *form*. "`(f x)` and `x`, but not `exit`, are subforms of `(return-from exit (f x))`."

subrepertoire *n.* a subset of a *repertoire*.

subtype *n.* a *type* whose membership is the same as or a proper subset of the membership of another *type*, called a *supertype*. (Every *type* is a *subtype* of itself.)

superclass *n.* a *class* from which another *class* (called a *subclass*) *inherits*. (No *class* is a *superclass* of itself.) See *subclass*.

supertype *n.* a *type* whose membership is the same as or a proper superset of the membership of another *type*, called a *subtype*. (Every *type* is a *supertype* of itself.) See *subtype*.

supplied-p parameter *n.* a *parameter* which recieves its *generalized boolean* value implicitly due to the presence or absence of an *argument* corresponding to another *parameter* (such as an *optional parameter* or a *rest parameter*). See Section 3.4.1 (Ordinary Lambda Lists).

symbol *n.* an *object* of *type* **symbol**.

symbol macro *n.* a *symbol* that stands for another *form*. See the *macro* **symbol-macrolet**.

synonym stream *n.* 1. a *stream* of *type* **synonym-stream**, which is consequently a *stream* that is an alias for another *stream*, which is the *value* of a *dynamic variable* whose *name* is the *synonym stream symbol* of the *synonym stream*. See the *function* **make-synonym-stream**. 2. (to a *stream*) a *synonym stream* which has the *stream* as the *value* of its *synonym stream symbol*. 3. (to a *symbol*) a *synonym stream* which has the *symbol* as its *synonym stream symbol*.

synonym stream symbol *n.* (of a *synonym stream*) the *symbol* which names the *dynamic variable* which has as its *value* another *stream* for which the *synonym stream* is an alias.

syntax type *n.* (of a *character*) one of several classifications, enumerated in Figure 2-6, that are used for dispatch during parsing by the *Lisp reader*. See Section 2.1.4 (Character Syntax Types).

system class *n.* a *class* that may be of *type* **built-in-class** in a *conforming implementation* and hence cannot be inherited by *classes* defined by *conforming programs*.

system code *n. code* supplied by the *implementation* to implement this specification (e.g., the definition of **mapcar**) or generated automatically in support of this specification (e.g., during method combination); that is, *code* that is not *programmer code*.

T

t *n.* 1. a. the *boolean* representing true. b. the canonical *generalized boolean* representing true. (Although any *object* other than **nil** is considered *true* as a *generalized boolean*, `t` is generally used when there is no special reason to prefer one such *object* over another.) 2. the *name* of the *type* to which all *objects* belong---the *supertype* of all *types* (including itself). 3. the *name* of the *superclass* of all *classes* except itself.

tag *n.* 1. a *catch tag*. 2. a *go tag*.

tail *n.* (of a *list*) an *object* that is the *same* as either some *cons* which makes up that *list* or the *atom* (if any) which terminates the *list*. "The empty list is a tail of every proper list."

target *n.* 1. (of a *constructed stream*) a *constituent* of the *constructed stream*. "The target of a synonym stream is the value of its synonym stream symbol." 2. (of a *displaced array*) the *array* to which the *displaced array* is displaced. (In the case of a chain of *constructed streams* or *displaced arrays*, the unqualified term "*target*" always refers to the immediate *target* of the first item in the chain, not the immediate target of the last item.)

terminal I/O *n.* the *bidirectional stream* that is the *value* of the *variable* ***terminal-io***.

terminating *n.* (of a *macro character*) being such that, if it appears while parsing a token, it terminates that token. See Section 2.2 (Reader Algorithm).

tertiary value *n.* (of *values* resulting from the *evaluation* of a *form*) the third *value*, if any, or else **nil** if there are fewer than three *values*.

throw *v.* to transfer control and *values* to a *catch*. See the *special operator* **throw**.

tilde *n.* the *standard character* that is called "tilde" (~). See Figure 2-5.

time a representation of a point (*absolute time*) or an interval (*relative time*) on a time line. See *decoded time*, *internal time*, and *universal time*.

time zone *n.* a *rational* multiple of `1/3600` between `-24` (inclusive) and `24` (inclusive) that represents a time zone as a number of hours offset from Greenwich Mean Time. Time zone values increase with motion to the west, so Massachusetts, U.S.A. is in time zone `5`, California, U.S.A. is time zone `8`, and Moscow, Russia is time zone *-3*. (When "daylight savings time" is separately represented as an *argument* or *return value*, the *time zone* that accompanies it does not depend on whether daylight savings time is in effect.)

token *n.* a textual representation for a *number* or a *symbol*. See Section 2.3 (Interpretation of Tokens).

top level form *n.* a *form* which is processed specially by **compile-file** for the purposes of enabling *compile time evaluation* of that *form*. *Top level forms* include those *forms* which are not *subforms* of any other *form*, and certain other cases. See Section 3.2.3.1 (Processing of Top Level Forms).

trace output *n.* the *output stream* which is the *value* of the *dynamic variable* ***trace-output***.

tree *n.* 1. a binary recursive data structure made up of *conses* and *atoms*: the *conses* are themselves also *trees* (sometimes called "subtrees" or "branches"), and the *atoms* are terminal nodes (sometimes called *leaves*). Typically, the *leaves* represent data while the branches establish some relationship among that data. 2. in general, any recursive data structure that has some notion of "branches" and *leaves*.

tree structure *n.* (of a *tree*[1]) the set of *conses* that make up the *tree*. Note that while the *car*[1b] component of each such *cons* is part of the *tree structure*, the *objects* that are the *cars*[2] of each *cons* in the *tree* are not themselves part of its *tree structure* unless they are also *conses*.

true *n.* any *object* that is not *false* and that is used to represent the success of a *predicate* test. See *t*[1].

truename *n.* 1. the canonical *filename* of a *file* in the *file system*. See Section 20.1.3 (Truenames). 2. a *pathname* representing a *truename*[1].

two-way stream *n.* a *stream* of *type* **two-way-stream**, which is a *bidirectional composite stream* that receives its input from an associated *input stream* and sends its output to an associated *output stream*.

type *n.* 1. a set of *objects*, usually with common structure, behavior, or purpose. (Note that the expression "*X* is of type *Sa*" naturally implies that "*X* is of type *Sb*" if *Sa* is a *subtype* of *Sb*.) 2. (immediately following the name of a *type*) a *subtype* of that *type*. "The type **vector** is an array type."

type declaration *n.* a *declaration* that asserts that every reference to a specified *binding* within the scope of the *declaration* results in some *object* of the specified *type*.

type equivalent *adj.* (of two *types* X and Y) having the same *elements*; that is, X is a *subtype* of Y and Y is a *subtype* of X.

type expand *n.* to fully expand a *type specifier*, removing any references to *derived types*. (Common Lisp provides no program interface to cause this to occur, but the semantics of Common Lisp are such that every *implementation* must be able to do this internally, and some situations involving *type specifiers* are most easily described in terms of a fully expanded *type specifier*.)

type specifier *n.* an *expression* that denotes a *type*. "The symbol `random-state`, the list `(integer 3 5)`, the list `(and list (not null))`, and the class named `standard-class` are type specifiers."

U

unbound *adj.* not having an associated denotation in a *binding*. See *bound*.

unbound variable *n.* a *name* that is syntactically plausible as the name of a *variable* but which is not *bound* in the *variable namespace*.

undefined function *n.* a *name* that is syntactically plausible as the name of a *function* but which is not *bound* in the *function namespace*.

unintern *v.t.* (a *symbol* in a *package*) to make the *symbol* not be *present* in that *package*. (The *symbol* might continue to be *accessible* by inheritance.)

uninterned *adj.* (of a *symbol*) not *accessible* in any *package*; i.e., not *interned*[1].

universal time *n. time*, represented as a non-negative *integer* number of seconds. *Absolute universal time* is measured as an offset from the beginning of the year 1900 (ignoring *leap seconds*). See Section 25.1.4.2 (Universal Time).

unqualified method *n.* a *method* with no *qualifiers*.

unregistered package *n.* a *package object* that is not present in the *package registry*. An *unregistered package* has no *name*; i.e., its *name* is **nil**. See the *function* **delete-package**.

unsafe *adj.* (of *code*) not *safe*. (Note that, unless explicitly specified otherwise, if a particular kind of error checking is guaranteed only in a *safe* context, the same checking might or might not occur in that context if it were *unsafe*; describing a context as *unsafe* means that certain kinds of error checking are not reliably enabled but does not guarantee that error checking is definitely disabled.)

unsafe call *n.* a *call* that is not a *safe call*. For more detailed information, see Section 3.5.1.1 (Safe and Unsafe Calls).

upgrade *v.t.* (a declared *type* to an actual *type*) 1. (when creating an *array*) to substitute an *actual array element type* for an *expressed array element type* when choosing an appropriately *specialized array* representation. See the *function* **upgraded-array-element-type**. 2. (when creating a *complex*) to substitute an *actual complex part type* for an *expressed complex part type* when choosing an appropriately *specialized complex* representation. See the *function* **upgraded-complex-part-type**.

upgraded array element type *n.* (of a *type*) a *type* that is a *supertype* of the *type* and that is used instead of the *type* whenever the *type* is used as an *array element type* for object creation or type discrimination. See Section 15.1.2.1 (Array Upgrading).

upgraded complex part type *n.* (of a *type*) a *type* that is a *supertype* of the *type* and that is used instead of the *type* whenever the *type* is used as a *complex part type* for object creation or type discrimination. See the *function* **upgraded-complex-part-type**.

uppercase *adj.* (of a *character*) being among *standard characters* corresponding to the capital letters `A` through `Z`, or being some other *implementation-defined character* that is defined by the *implementation* to be *uppercase*. See Section 13.1.4.3 (Characters With Case).

use *v.t.* (a *package* P1) to *inherit* the *external symbols* of P1. (If a package P2 uses P1, the *external symbols* of P1 become *internal symbols* of P2 unless they are explicitly *exported*.) "The package `CL-USER` uses the package `CL`."

use list *n.* (of a *package*) a (possibly empty) *list* associated with each *package* which determines what other *packages* are currently being *used* by that *package*.

user *n.* an active entity, typically a human, that invokes or interacts with a *program* at run time, but that is not necessarily a *programmer*.

V

valid array dimension *n.* a *fixnum* suitable for use as an *array dimension*. Such a *fixnum* must be greater than or equal to zero, and less than the *value* of **array-dimension-limit**. When multiple *array dimensions* are to be used together to specify a multi-dimensional *array*, there is also an implied constraint that the product of all of the *dimensions* be less than the *value* of **array-total-size-limit**.

valid array index *n.* (of an *array*) a *fixnum* suitable for use as one of possibly several indices needed to name an *element* of the *array* according to a multi-dimensional Cartesian coordinate system. Such a *fixnum* must be greater than or equal to zero, and must be less than the corresponding *dimension*[1] of the *array*. (Unless otherwise explicitly specified, the phrase "a *list* of *valid array indices*" further implies that the *length* of the *list* must be the same as the *rank* of the *array*.) "For a `2` by `3` array, valid array indices for the first dimension are `0` and `1`, and valid array indices for the second dimension are `0`, `1` and `2`."

valid array row-major index *n.* (of an *array*, which might have any number of *dimensions*[2]) a single *fixnum* suitable for use in naming any *element* of the *array*, by viewing the array's storage as a linear series of *elements* in row-major order. Such a *fixnum* must be greater than or equal to zero, and less than the *array total size* of the *array*.

valid fill pointer *n.* (of an *array*) a *fixnum* suitable for use as a *fill pointer* for the *array*. Such a *fixnum* must be greater than or equal to zero, and less than or equal to the *array total size* of the *array*.

valid logical pathname host *n.* a *string* that has been defined as the name of a *logical host*. See the *function* **load-logical-pathname-translations**.

valid pathname device *n.* a *string*, **nil**, `:unspecific`, or some other *object* defined by the *implementation* to be a *valid pathname device*.

valid pathname directory *n.* a *string*, a *list* of *strings*, **nil**, `:wild`, `:unspecific`, or some other *object* defined by the *implementation* to be a *valid directory component*.

valid pathname host *n.* a *valid physical pathname host* or a *valid logical pathname host*.

valid pathname name *n.* a *string*, **nil**, `:wild`, `:unspecific`, or some other *object* defined by the *implementation* to be a *valid pathname name*.

valid pathname type *n.* a *string*, **nil**, `:wild`, `:unspecific`.

valid pathname version *n.* a non-negative *integer*, or one of `:wild`, `:newest`, `:unspecific`, or **nil**. The symbols `:oldest`, `:previous`, and `:installed` are *semi-standard* special version symbols.

valid physical pathname host *n.* any of a *string*, a *list* of *strings*, or the symbol `:unspecific`, that is recognized by the implementation as the name of a host.

valid sequence index *n.* (of a *sequence*) an *integer* suitable for use to name an *element* of the *sequence*. Such an *integer* must be greater than or equal to zero, and must be less than the *length* of the *sequence*. (If the *sequence* is an *array*, the *valid sequence index* is further constrained to be a *fixnum*.)

value *n.* 1. a. one of possibly several *objects* that are the result of an *evaluation*. b. (in a situation where exactly one value is expected from the *evaluation* of a *form*) the *primary value* returned by the *form*. c. (of *forms* in an *implicit progn*) one of possibly several *objects* that result from the *evaluation* of the last *form*, or **nil** if there are no *forms*. 2. an *object* associated with a *name* in a *binding*. 3. (of a *symbol*) the *value* of the *dynamic variable* named by that symbol. 4. an *object* associated with a *key* in an *association list*, a *property list*, or a *hash table*.

value cell *n. Trad.* (of a *symbol*) The *place* which holds the *value*, if any, of the *dynamic variable* named by that *symbol*, and which is *accessed* by **symbol-value**. See *cell*.

variable *n.* a *binding* in the "variable" *namespace*. See Section 3.1.2.1.1 (Symbols as Forms).

vector *n.* a one-dimensional *array*.

vertical-bar *n.* the *standard character* that is called "vertical bar" (|). See Figure 2-5.

W

whitespace *n.* 1. one or more *characters* that are either the *graphic character* `#\Space` or else *non-graphic* characters such as `#\Newline` that only move the print position. 2. a. *n.* the *syntax type* of a *character* that is a *token* separator. For details, see Section 2.1.4.7 (Whitespace Characters). b. *adj.* (of a *character*) having the *whitespace*[2a] *syntax type*[2]. c. *n.* a *whitespace*[2b] *character*.

wild *adj.* 1. (of a *namestring*) using an *implementation-defined* syntax for naming files, which might "match" any of possibly several possible *filenames*, and which can therefore be used to refer to the aggregate of the *files* named by those *filenames*. 2. (of a *pathname*) a structured representation of a name which might "match" any of possibly several *pathnames*, and which can therefore be used to refer to the aggregate of the *files* named by those *pathnames*. The set of *wild pathnames* includes, but is not restricted to, *pathnames* which have a component which is `:wild`, or which have a directory component which contains `:wild` or `:wild-inferors`. See the *function* **wild-pathname-p**.

write *v.t.* 1. (a *binding* or *slot* or component) to change the *value* of the *binding* or *slot*. 2. (an *object* to a *stream*) to output a representation of the *object* to the *stream*.

writer *n.* a *function* that *writes*[1] a *variable* or *slot*.

Y

yield *v.t.* (*values*) to produce the *values* as the result of *evaluation*. "The form `(+ 2 3)` yields `5`."

A. Appendix

A.1 Removed Language Features

A.1.1 Requirements for removed and deprecated features

For this standard, some features from the language described in *Common Lisp: The Language* have been removed, and others have been deprecated (and will most likely not appear in future Common Lisp standards). Which features were removed and which were deprecated was decided on a case-by-case basis by the X3J13 committee.

Conforming implementations that wish to retain any removed features for compatibility must assure that such compatibility does not interfere with the correct function of *conforming programs*. For example, symbols corresponding to the names of removed functions may not appear in the the `COMMON-LISP` package. (Note, however, that this specification has been devised in such a way that there can be a package named `LISP` which can contain such symbols.)

Conforming implementations must implement all deprecated features. For a list of deprecated features, see Section 1.8 (Deprecated Language Features).

A.1.2 Removed Types

The *type* `string-char` was removed.

A.1.3 Removed Operators

The functions `int-char`, `char-bits`, `char-font`, `make-char`, `char-bit`, `set-char-bit`, `string-char-p`, and `commonp` were removed.

The *special operator* `compiler-let` was removed.

A.1.4 Removed Argument Conventions

The *font* argument to **digit-char** was removed. The *bits* and *font* arguments to **code-char** were removed.

A.1.5 Removed Variables

The variables `char-font-limit`, `char-bits-limit`, `char-control-bit`, `char-meta-bit`, `char-super-bit`, `char-hyper-bit`, and `*break-on-warnings*` were removed.

A.1.6 Removed Reader Syntax

The "`#,`" *reader macro* in *standard syntax* was removed.

A.1.7 Packages No Longer Required

The *packages* `LISP`, `USER`, and `SYSTEM` are no longer required. It is valid for *packages* with one or more of these names to be provided by a *conforming implementation* as extensions.