draft proposed American National Standard for Information Systems—Programming Language—Common Lisp

X3J13/94-101R
Versions 15.17 (X3J13/94-101) and 15.17R (X3J13/94-101R) are absolutely identical except for changes to the document number and date on the cover and in each page’s header, addition of this disclaimer on the back of the cover page, additions to the Edit and Review History on page Credits iv, and changes to the names and headings in the credits section on pages Credits v-vii. There are NO technical changes between these versions; from a normative point of view, the documents X3J13/94-101 and X3J13/94-101R are entirely interchangeable.

Programming Language—Common Lisp
Figures

Chapter 1 (Introduction)
1–1. Notations for NIL ........................................ 1–14
1–2. Deprecated Functions .................................. 1–33
1–3. Functions with Deprecated :TEST-NOT Arguments ........ 1–33
1–4. Symbols in the COMMON-LISP package (part one of twelve). . 1–35
1–6. Symbols in the COMMON-LISP package (part three of twelve). . 1–37
1–7. Symbols in the COMMON-LISP package (part four of twelve). . 1–38
1–10. Symbols in the COMMON-LISP package (part seven of twelve). . 1–41
1–11. Symbols in the COMMON-LISP package (part eight of twelve). . 1–42
1–12. Symbols in the COMMON-LISP package (part nine of twelve). . 1–43

Chapter 2 (Syntax)
2–1. Readtable defined names ................................ 2–1
2–2. Variables that influence the Lisp reader .................. 2–2
2–5. Standard Character Subrepertoire (Part 3 of 3: Special Characters) . 2–4
2–6. Possible Character Syntax Types ......................... 2–5
2–8. Constituent Traits of Standard Characters and Semi-Standard Characters . 2–7
2–9. Syntax for Numeric Tokens .............................. 2–14
2–10. Examples of reserved tokens ......................... 2–15
2–11. Examples of symbols ................................ 2–15
2–12. Examples of symbols or potential numbers .............. 2–16
2–13. Examples of Ratios ................................ 2–17
2–14. Examples of Floating-point numbers .................. 2–18
2–15. Examples of the printed representation of symbols (Part 1 of 2) .... 2–19
2–16. Examples of the printed representation of symbols (Part 2 of 2) .... 2–20
2–17. Valid patterns for tokens ................................ 2–21
2–18. Examples of the use of double-quote .................. 2–26
5–1. Examples of setf ................................. 5–1
5–2. Operators relating to places and generalized reference. 5–1
5–3. Sample Setf Expansion of a Variable 5–4
5–4. Sample Setf Expansion of a CAR Form 5–4
5–5. Sample Setf Expansion of a SUBSEQ Form 5–4
5–6. Sample Setf Expansion of a LDB Form 5–5
5–7. Functions that setf can be used with—1 5–6
5–8. Functions that setf can be used with—2 5–7
5–9. Read-Modify-Write Macros .......................... 5–12
5–10. Macros that have implicit tagbodies. 5–49
5–11. Operators that always prefer EQ over EQL 5–57
5–12. Summary and priorities of behavior of equal 5–61
5–13. Summary and priorities of behavior of equalp 5–63

Chapter 6 (Iteration)

Chapter 7 (Objects)
7–1. Standardized Method-Defining Operators 7–18
7–2. Built-in Method Combination Types 7–26

Chapter 8 (Structures)

Chapter 9 (Conditions)
9–1. Standardized Condition Types 9–2
9–2. Operators that define and create conditions. 9–2
9–3. Operators that read condition slots. 9–3
9–4. Operators relating to handling conditions. 9–7
9–5. Defined names relating to signaling conditions. 9–7
9–6. Defined names relating to restarts. 9–9
9–7. Operators relating to assertions. 9–10

Chapter 10 (Symbols)
10–1. Property list defined names 10–1
10–2. Symbol creation and inquiry defined names 10–1

Chapter 11 (Packages)
11–1. Some Defined Names related to Packages 11–1
11–2. Standardized Package Names 11–4

Chapter 12 (Numbers)
12–1. Operators relating to Arithmetic. 12–1

Figures iii
12–2. Defined names relating to Exponentials, Logarithms, and Trigonometry. 12–1
12–3. Operators for numeric comparison and predication. 12–1
12–4. Defined names relating to numeric type manipulation and coercion. 12–2
12–5. Defined names relating to logical operations on numbers. 12–3
12–6. Defined names relating to byte manipulation. 12–4
12–7. Defined names relating to implementation-dependent details about numbers. 12–4
12–8. Functions Affected by Rule of Float Substitutability 12–6
12–9. Trigonometric Identities for Complex Domain 12–9
12–10. Quadrant Numbering for Branch Cuts 12–9
12–11. Random-state defined names 12–10
12–12. Recommended Minimum Floating-Point Precision and Exponent Size 12–14
12–13. Uses of /=, =, <, >, <=, and => 12–22
12–14. Mathematical definition of arc sine, arc cosine, and arc tangent 12–29
12–15. Quadrant information for arc tangent 12–31
12–16. Mathematical definitions for hyperbolic functions 12–33
12–17. Bit-Wise Logical Operations 12–68
12–18. Bit-wise Logical Operations on Integers 12–72

Chapter 13 (Characters)
13–1. Character defined names – 1 13–1
13–2. Character defined names – 2 13–1

Chapter 14 (Conses)
14–1. Some defined names relating to conses. 14–1
14–2. Some defined names relating to trees. 14–1
14–3. Some defined names relating to lists. 14–2
14–4. Some defined names related to association lists. 14–2
14–5. Some defined names related to sets. 14–3
14–6. CAR and CDR variants 14–11

Chapter 15 (Arrays)
15–1. General Purpose Array-Related Defined Names 15–2
15–2. Operators that Manipulate Strings 15–3
15–3. Operators that Manipulate Bit Arrays 15–4
15–4. Bit-wise Logical Operations on Bit Arrays 15–38

Chapter 16 (Strings)

Chapter 17 (Sequences)
17–1. Standardized Sequence Functions 17–1
17–2. Operators that have Two-Argument Tests to be Satisfied 17–2
17–3. Operators that have One-Argument Tests to be Satisfied 17–3
25–3. Defined names relating to environment inquiry. 25–2
25–5. Defined names involving time in Decoded Time. 25–3
25–6. Defined names involving time in Universal Time. 25–4
25–7. Defined names involving time in Internal Time. 25–4
25–8. Defined names involving time in Seconds. 25–4

Chapter 26 (Glossary)
26–1. Exponent Markers 26–22
26–2. Standardized I/O Customization Variables 26–30
26–3. Standardized Iteration Forms 26–33
26–4. Standardized Restart Functions 26–49

Chapter A (Appendix)
Contents

Chapter 1 (Introduction)
1.1 Scope, Purpose, and History .............................................. 1–1
1.1.1 Scope and Purpose .................................................. 1–1
1.1.2 History .................................................................. 1–1
1.2 Organization of the Document ............................................ 1–4
1.3 Referenced Publications ................................................. 1–5
1.4 Definitions .................................................................... 1–7
1.4.1 Notational Conventions ............................................ 1–7
1.4.1.1 Font Key .......................................................... 1–7
1.4.1.2 Modified BNF Syntax ....................................... 1–7
1.4.1.2.1 Splicing in Modified BNF Syntax ................. 1–8
1.4.1.2.2 Indirection in Modified BNF Syntax .............. 1–9
1.4.1.2.3 Additional Uses for Indirect Definitions in Modified BNF Syntax ................................................................. 1–10
1.4.1.3 Special Symbols .................................................. 1–10
1.4.1.4 Objects with Multiple Notations ............................ 1–12
1.4.1.4.1 Case in Symbols .......................................... 1–12
1.4.1.4.2 Numbers ...................................................... 1–13
1.4.1.4.3 Use of the Dot Character ................................ 1–13
1.4.1.4.4 NIL ............................................................ 1–13
1.4.1.5 Designators ....................................................... 1–14
1.4.1.6 Nonsense Words ................................................ 1–15
1.4.2 Error Terminology .................................................... 1–15
1.4.3 Sections Not Formally Part Of This Standard ............... 1–18
1.4.4 Interpreting Dictionary Entries .................................... 1–18
1.4.4.1 The “Affected By” Section of a Dictionary Entry .... 1–19
1.4.4.2 The “Arguments” Section of a Dictionary Entry .... 1–19
1.4.4.3 The “Arguments and Values” Section of a Dictionary Entry ................................................................. 1–19
1.4.4.4 The “Binding Types Affected” Section of a Dictionary Entry ................................................................. 1–19
1.4.4.5 The “Class Precedence List” Section of a Dictionary Entry ................................................................. 1–19
1.4.4.6 Dictionary Entries for Type Specifiers .................. 1–20
1.4.4.6.1 The “Compound Type Specifier Kind” Section of a Dictionary Entry ................................................................. 1–20
1.4.4.6.2 The “Compound Type Specifier Syntax” Section of a Dictionary Entry ................................................................. 1–20
1.4.4.6.3 The “Compound Type Specifier Arguments” Section of a Dictionary Entry ................................................................. 1–21
1.4.4.6.4 The “Compound Type Specifier Description” Section of a Dictionary Entry ................................................................. 1–21
1.4.4.7 The “Constant Value” Section of a Dictionary Entry .... 1–21
1.4.4.8 The “Description” Section of a Dictionary Entry ....... 1–21
1.4.4.9 The “Examples” Section of a Dictionary Entry ....... 1–21
Chapter 3 (Evaluation and Compilation)
3.1 Evaluation .................................................. 3–1
3.1.1 Introduction to Environments ......................... 3–1
3.1.1.1 The Global Environment ......................... 3–1
3.1.1.2 Dynamic Environments ........................ 3–2
3.1.1.3 Lexical Environments ............................ 3–2
3.1.1.3.1 The Null Lexical Environment ............... 3–2
3.1.1.4 Environment Objects ............................ 3–3
3.1.2 The Evaluation Model ............................... 3–3
3.1.2.1 Form Evaluation ................................ 3–3
3.1.2.1.1 Symbols as Forms ........................ 3–3
3.1.2.1.1.1 Lexical Variables ........................ 3–4
3.1.2.1.2 Dynamic Variables ........................ 3–4
3.1.2.1.1.3 Constant Variables ........................ 3–5
Chapter 4 (Types and Classes)
4.1 Introduction ............................................. 4–1
4.2 Types ..................................................... 4–2
4.2.1 Data Type Definition ................................. 4–2
4.2.2 Type Relationships ................................. 4–2
4.2.3 Type Specifiers ................................. 4–3

3.4.2 Generic Function Lambda Lists ................. 3–41
3.4.3 Specialized Lambda Lists .......................... 3–42
3.4.4 Macro Lambda Lists ................................ 3–43
3.4.4.1 Destructuring by Lambda Lists .............. 3–43
3.4.4.1.1 Data-directed Destructuring by Lambda Lists 3–43
3.4.4.1.1.1 Examples of Data-directed Destructuring by Lambda Lists 3–43
3.4.4.1.2 Lambda-list-directed Destructuring by Lambda Lists 3–43
3.4.5 Destructuring Lambda Lists .................... 3–45
3.4.6 Boa Lambda Lists .................................. 3–45
3.4.7 Defsetf Lambda Lists ............................. 3–47
3.4.8 Deftype Lambda Lists ............................. 3–47
3.4.9 Define-modify-macro Lambda Lists ............ 3–48
3.4.10 Define-method-combination Arguments Lambda Lists 3–48
3.4.11 Syntactic Interaction of Documentation Strings and Declarations ................. 3–48
3.5 Error Checking in Function Calls .................. 3–50
3.5.1 Argument Mismatch Detection ................ 3–50
3.5.1.1 Safe and Unsafe Calls ......................... 3–50
3.5.1.1.1 Error Detection Time in Safe Calls .... 3–51
3.5.1.2 Too Few Arguments ............................ 3–51
3.5.1.3 Too Many Arguments ......................... 3–51
3.5.1.4 Unrecognized Keyword Arguments ......... 3–51
3.5.1.5 Invalid Keyword Arguments ................. 3–51
3.5.1.6 Odd Number of Keyword Arguments ....... 3–52
3.5.1.7 Destructuring Mismatch ....................... 3–52
3.5.1.8 Errors When Calling a Next Method .......... 3–52
3.6 Traversal Rules and Side Effects ................. 3–53
3.7 Destructive Operations .............................. 3–54
3.7.1 Modification of Literal Objects .................. 3–54
3.7.2 Transfer of Control during a Destructive Operation ............................ 3–55
3.7.2.1 Examples of Transfer of Control during a Destructive Operation .......... 3–55
3.8 Evaluation and Compilation Dictionary .......... 3–56
Chapter 7 (Objects)

7.1 Object Creation and Initialization .......................... 7–1
  7.1.1 Initialization Arguments ................................. 7–2
  7.1.2 Declaring the Validity of Initialization Arguments .... 7–2
  7.1.3 Defaulting of Initialization Arguments ................. 7–3
  7.1.4 Rules for Initialization Arguments ..................... 7–4
  7.1.5 Shared-Initialize ........................................ 7–5
  7.1.6 Initialize-Instance ....................................... 7–6
  7.1.7 Definitions of Make-Instance and Initialize-Instance 7–7
  7.2 Changing the Class of an Instance ......................... 7–9
  7.2.1 Modifying the Structure of the Instance ............... 7–9
  7.2.2 Initializing Newly Added Local Slots .................. 7–9
  7.2.3 Customizing the Change of Class of an Instance .... 7–10
  7.3 Reinitializing an Instance ................................ 7–11
  7.3.1 Customizing Reinitialization ........................... 7–11
  7.4 Meta-Objects ................................................. 7–12
  7.4.1 Standard Meta-objects ................................ 7–12
  7.5 Slots .......................................................... 7–13
  7.5.1 Introduction to Slots .................................... 7–13
  7.5.2 Accessing Slots .......................................... 7–13
  7.5.3 Inheritance of Slots and Slot Options ................ 7–14
  7.6 Generic Functions and Methods ........................... 7–17
  7.6.1 Introduction to Generic Functions ..................... 7–17
  7.6.2 Introduction to Methods ................................ 7–18
  7.6.3 Agreement on Parameter Specializers and Qualifiers 7–20
  7.6.4 Congruent Lambda-lists for all Methods of a Generic Function 7–20
  7.6.5 Keyword Arguments in Generic Functions and Methods 7–21
  7.6.5.1 Examples of Keyword Arguments in Generic Functions and Methods 7–21
  7.6.6 Method Selection and Combination ...................... 7–22
  7.6.6.1 Determining the Effective Method ................... 7–22
  7.6.6.1.1 Selecting the Applicable Methods ................ 7–22
  7.6.6.1.2 Sorting the Applicable Methods by Precedence Order 7–23
  7.6.6.1.3 Applying method combination to the sorted list of applicable methods 7–23
  7.6.6.2 Standard Method Combination ....................... 7–24
  7.6.6.3 Declarative Method Combination ...................... 7–25
  7.6.6.4 Built-in Method Combination Types .................. 7–26
  7.6.7 Miscellaneous Clauses ................................. 6–30
    7.6.7.1 Control Transfer Clauses ........................... 6–27
    7.6.7.1.1 Examples of NAMED clause ........................ 6–27
    7.6.7.2 Initial and Final Execution ........................ 6–27
  7.6.8 Examples of Miscellaneous Loop Features ............. 6–27
  7.6.9 Examples of clause grouping ............................ 6–28
  7.6.10 Notes about Loop ........................................ 6–30
  6.2 Iteration Dictionary ....................................... 6–31
12.2 Numbers Dictionary
12.1.7 Random-State Operations
12.1.6 Interval Designators
12.1.5.4 Principal Values and Branch Cuts
12.1.5.3.1 Examples of Rule of Canonical Representation for Complex Rationals
12.1.5.1 Rule of Complex Substitutability
12.1.4.4 Rule of Float Precision Contagion
12.1.4.3 Rule of Float Underflow and Overflow
12.1.4.2 Rule of Float Approximation
12.1.4.1.1 Examples of Rule of Float and Rational Contagion
12.1.3.3 Rule of Float Substitutability
12.1.3.2 Rule of Canonical Representation for Rationals
12.1.3.1 Rule of Unbounded Rational Precision
12.1.2 Implementation-Dependent Numeric Constants
12.1.1.3.2 Byte Operations on Integers
12.1.1.3.1 Logical Operations on Integers
12.1.1.3 Viewing Integers as Bits and Bytes
12.1.1.2 Contagion in Numeric Operations
12.1.1.1.1 Examples of Associativity and Commutativity in Numeric Operations
12.1.1.1 Associativity and Commutativity in Numeric Operations
12.1.1 Numeric Operations

Chapter 12 (Numbers)
12.1 Number Concepts
12.1.1 Numeric Operations
12.1.1.1 Associativity and Commutativity in Numeric Operations
12.1.1.1.1 Examples of Associativity and Commutativity in Numeric Operations
12.1.1.2 Contagion in Numeric Operations
12.1.1.3 Viewing Integers as Bits and Bytes
12.1.1.2 Contagion in Numeric Operations
12.1.1.1.1 Examples of Associativity and Commutativity in Numeric Operations
12.1.1 Numeric Operations
11.2 Packages Dictionary
11.1.2.4 Implementation-Defined Packages
11.1.2.3.2 Notes about The KEYWORD Package
11.1.2.3.1 Interning a Symbol in the KEYWORD Package
11.1.2.3 The KEYWORD Package
11.1.2.2 The COMMON-LISP-USER Package
11.1.2.1.2.1 Some Exceptions to Constraints on the COMMON-LISP Package for Conforming
11.1.2.1.2 Constraints on the COMMON-LISP Package for Conforming Programs
11.1.2.1 Standardized Packages
11.1.1.2.5 Prevention of Name Conflicts in Packages
11.1.1.2.4 Locating a Symbol in a Package
11.1.1.2.3 The COMMON-LISP Package
11.1.1.2 Implementation-Defined Packages
11.1.1.2.4 Notes about The KEYWORD Package
11.1.1.2.3.1 Interning a Symbol in the KEYWORD Package
11.1.1.2.3 The KEYWORD Package
11.1.1.2.2 The COMMON-LISP-USER Package
11.1.1.2 Constraints on the COMMON-LISP Package for Conforming Programs
11.1.1.1.2.4 Locating a Symbol in a Package
11.1.1.1.2.3 Notes about The COMMON-LISP Package
11.1.1.1.2 Implementation-Defined Packages
11.1.1.1 Standardized Packages
11.1.1.2.5 Prevention of Name Conflicts in Packages
11.1.1.2.4 Locating a Symbol in a Package
11.1.1.2.3 The COMMON-LISP Package
11.1.1.2 Implementation-Defined Packages
11.1.1.2 Constraints on the COMMON-LISP Package for Conforming Programs
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.3.5 FORMAT Pretty Printer Operations</td>
<td>22–32</td>
</tr>
<tr>
<td>22.3.5.1 Tilde Underscore: Conditional Newline</td>
<td>22–32</td>
</tr>
<tr>
<td>22.3.5.2 Tilde Less-Than-Sign: Logical Block</td>
<td>22–32</td>
</tr>
<tr>
<td>22.3.5.3 Tilde I: Indent</td>
<td>22–33</td>
</tr>
<tr>
<td>22.3.5.4 Tilde Slash: Call Function</td>
<td>22–33</td>
</tr>
<tr>
<td>22.3.5.6 Tilde Layout Control</td>
<td>22–34</td>
</tr>
<tr>
<td>22.3.5.6.1 Tilde T: Tabulate</td>
<td>22–34</td>
</tr>
<tr>
<td>22.3.5.6.2 Tilde Less-Than-Sign: Justification</td>
<td>22–34</td>
</tr>
<tr>
<td>22.3.5.6.3 Tilde Greater-Than-Sign: End of Justification</td>
<td>22–36</td>
</tr>
<tr>
<td>22.3.5.7 Tilde Control-Flow Operations</td>
<td>22–36</td>
</tr>
<tr>
<td>22.3.5.7.1 Tilde Asterisk: Go-To</td>
<td>22–36</td>
</tr>
<tr>
<td>22.3.5.7.2 Tilde Left-Bracket: Conditional Expression</td>
<td>22–36</td>
</tr>
<tr>
<td>22.3.5.7.3 Tilde Right-Bracket: End of Conditional Expression</td>
<td>22–37</td>
</tr>
<tr>
<td>22.3.5.7.4 Tilde Left-Brace: Iteration</td>
<td>22–37</td>
</tr>
<tr>
<td>22.3.5.7.5 Tilde Right-Brace: End of Iteration</td>
<td>22–38</td>
</tr>
<tr>
<td>22.3.5.7.6 Tilde Question-Mark: Recursive Processing</td>
<td>22–39</td>
</tr>
<tr>
<td>22.3.5.8 FORMAT Miscellaneous Operations</td>
<td>22–39</td>
</tr>
<tr>
<td>22.3.5.8.1 Tilde Left-Paren: Case Conversion</td>
<td>22–39</td>
</tr>
<tr>
<td>22.3.5.8.2 Tilde Right-Paren: End of Case Conversion</td>
<td>22–40</td>
</tr>
<tr>
<td>22.3.5.8.3 Tilde P: Plural</td>
<td>22–40</td>
</tr>
<tr>
<td>22.3.5.9 FORMAT Miscellaneous Pseudo-Operations</td>
<td>22–40</td>
</tr>
<tr>
<td>22.3.5.9.1 Tilde Semicolon: Clause Separator</td>
<td>22–40</td>
</tr>
<tr>
<td>22.3.5.9.2 Tilde Circumflex: Escape Upward</td>
<td>22–40</td>
</tr>
<tr>
<td>22.3.5.9.3 Tilde Newline: Ignored Newline</td>
<td>22–42</td>
</tr>
<tr>
<td>22.3.10 Additional Information about FORMAT Operations</td>
<td>22–42</td>
</tr>
<tr>
<td>22.3.10.1 Nesting of FORMAT Operations</td>
<td>22–42</td>
</tr>
<tr>
<td>22.3.10.2 Missing and Additional FORMAT Arguments</td>
<td>22–43</td>
</tr>
<tr>
<td>22.3.10.3 Additional FORMAT Parameters</td>
<td>22–43</td>
</tr>
<tr>
<td>22.3.10.4 Undefined FORMAT Modifier Combinations</td>
<td>22–43</td>
</tr>
<tr>
<td>22.3.11 Examples of FORMAT</td>
<td>22–43</td>
</tr>
<tr>
<td>22.3.12 Notes about FORMAT</td>
<td>22–45</td>
</tr>
<tr>
<td>22.4 Printer Dictionary</td>
<td>22–46</td>
</tr>
</tbody>
</table>

Chapter 23 (Reader)

23.1 Reader Concepts 23–1
23.1.1 Dynamic Control of the Lisp Reader 23–1
23.1.2 Effect of Readtable Case on the Lisp Reader 23–1
23.1.2.1 Examples of Effect of Readtable Case on the Lisp Reader 23–1
23.1.3 Argument Conventions of Some Reader Functions 23–2
23.1.3.1 The EOF-ERROR-P argument 23–2
23.1.3.2 The RECURSIVE-P argument 23–2
23.2 Reader Dictionary 23–4

Chapter 24 (System Construction)
Index

#  2–29
&allow-other-keys  3–35
&aux  3–37
&body  3–42
&environment  3–42
&key  3–35
&optional  3–35
&rest  3–35
&whole  3–42
'  2–24
(  2–23
()/  1–13, 26–2
(setf class-name)  7–89
(setf documentation)  25–17
)  2–23
*  12–35, 25–26
**  25–26
***  25–26
*break-on-signals*  9–35
*break-on-warnings*  A–1
*compile-file-pathname*  24–12
*compile-file-truename*  24–12
*compile-print*  24–13
*compile-verbose*  24–13
*debug-io*  21–57
*debugger-hook*  9–34
*default-pathname-defaults*  19–28
*error-output*  21–57
*features*  1–30, 2–37, 24–10
*gensym-counter*  10–9
*load-pathname*  24–13
*load-print*  24–14
*load-truename*  24–13
*load-verbose*  24–14
*macroexpand-hook*  3–80
*modules*  24–14
*package*  11–40

*print-array*  22–66
*print-base*  22–67
*print-case*  22–68
*print-circle*  2–36, 2–37, 22–69
*print-escape*  22–70
*print-gensym*  22–71
*print-length*  22–72
*print-level*  22–72
*print-lines*  22–73
*print-miser-width*  22–74
*print-pprint-dispatch*  22–75
*print-pretty*  22–75
*print-radix*  22–67
*print-readably*  22–76
*print-right-margin*  22–78
*query-io*  21–57
*random-state*  12–53
*read-base*  2–33, 2–34, 23–19
*read-default-float-format*  23–19
*read-eval*  2–33, 23–20
*read-suppress*  23–21
*readtable*  23–22
*standard-input*  21–57
*standard-output*  21–57
*terminal-io*  21–59
*trace-output*  21–57
+  T–26, 12–36, 25–25
++  25–25
+++  25–25
  ,  2–28
  2–28
  12–36, 25–24
  2–23
  ..  2–40, 22–74
  .  2–40, 22–55
  /  12–37, 25–27
  //  25–27
  ///  25–27
/=  12–20
1+  12–38
1–  12–38
:absolute  19–7
:back  19–8
:common  19–5
:compile-toplevel  3–60
:execute  3–60
Index
Index
condition 9–11
condition designator 9–3, 26–13
condition handler 26–13
condition reporter 9–4, 26–13
conditional newline 26–13
conditional newlines 22–14
conformance 26–13
conforming code 1–29, 26–13
conforming implementation 1–28, 26–13
conforming processor 26–14
conforming program 1–29, 26–14
congruence 7–19
congruent 26–14
conjugate 12–57
cons 14–1, 26–14
cons 2–26, 2–28, 14–5, 14–6
consequences 1–17
consp 14–6
constant 26–14
constant form 26–14
constant object 26–14
constant variable 26–14
constantly 5–66
constantp 3–104
constituent 26–14
constituent trait 26–14
constructed stream 26–15
contagion 26–15
continuable 26–15
continue 9–66, 9–68
control form 26–15
control-error 5–104
copy 26–15
copy-alist 14–45
copy-list 14–19
copy-pprint-dispatch 22–46
copy-readable 23–4
copy-seq 17–5
copy-structure 8–18
copy-symbol 10–7
copy-tree 14–12
correctable 26–15
cos 12–28
cosh 12–33
count 17–15
count-if 17–15
count-if-not 17–15
cctypecase 5–76
current input base 23–19, 26–15
current logical block 26–16
current output base 22–67, 26–16
current package 11–1, 26–16
current pprint dispatch table 22–16, 26–16
current random state 26–16
current readable 2–1, 26–16
D (format directive) 22–26
data type 26–16
debug 3–97
debug I/O 26–16
*debug-io* 21–57
debugger 26–16
*debugger-hook* 9–34
deef 12–43
declare 3–83
declaration 3–29, 26–16
declaration 3–29, 3–96
declaration identifier 3–29, 26–16
declaration specifier 3–29, 26–16
declare 26–17
declare 3–83
decline 26–17
decode–float 12–82
decode-universal-time 25–5
decoded time 25–2, 26–17
default method 7–19, 26–17
*default-pathname-defaults* 19–28
defaulted initialization argument list 26–17
defclass 7–60
defconstant 5–31
defgeneric 7–65
define–compiler-macro 3–67
define-condition 9–42
define-method-combination 7–76
define-method-combination arguments lambda list 3–48, 26–17
define-modify-macro 5–92
define-modify-macro lambda list 3–48, 26–17
define-setf-expander 5–96
define-symbol-macro 3–77
defined name 26–17
Index vii
empty type 26–20
encode-universal-time 25–6
end of file 26–20
end-of-file 21–61
endp 14–28
enough-namestring 19–28
ensure-directories-exist 20–4
ensure-generic-function 7–29
environment 3–1, 26–21
environment object 3–3, 26–21
environment parameter 26–21
eq 5–56
eql 4–30, 5–58
equal 5–59
Equal-Sign (sharpsign reader macro) 2–36
equalp 5–62
error 26–21
error output 26–21
error terminology 1–15
*error-output* 21–57
escape 26–21
establish 26–21
etypecase 5–76
eval 2–33, 3–59, 3–60
eval-when 3–20, 3–60
evaluate 26–21
evaluation 3–1, 26–21
evaluation environment 3–14, 26–21
evaluation order 3–63, 5–2, 5–43, 5–83, 6–5, 6–9, 7–4, 9–16, 12–79
evenp 12–40
every 5–67
execute 26–22
:execute 3–60
execution time 26–22
exhaustive partition 26–22
exhaustive union 26–22
exit point 26–22
exp 12–41
explicit return 26–22
explicit use 26–22
exponent marker 26–22
export 26–22
export 11–8
exported 26–23
expressed adjustability 26–23
expressed array element type 15–3, 26–23
expressed complex part type 26–23
expression 26–23
expressly adjustable 26–23
expt 12–41
extended character 26–23
extended function designator 26–23
extended lambda list 26–23
extended-char 13–9
extension 26–24
extensions 1–17, 1–18
extent 26–24
external file format 26–24
external file format designator 26–24
external symbol 11–2, 26–24
externalizable object 3–23, 26–24
F (format directive) 22–27
false 26–24
fbound 26–24
fboundp 5–18
fceiling 12–25
fdefinition 5–17
feature 24–1, 26–24
feature expression 24–1, 26–25
features list 24–1, 26–25
*features* 1–30, 2–37, 24–10
ffloor 12–25
fifth 14–25
file 20–1, 26–25
file compiler 26–25
file position 26–25
file position designator 26–25
file stream 21–3, 26–25
file system 26–25
file-author 20–6
file-error 20–10
file-error-pathname 20–11
file-length 21–28
file-namestring 19–28
file-position 21–29
file-stream 21–9
file-string-length 21–31
file-write-date 20–7

viii  Programming Language—Common Lisp
filename 20–1, 26–25
fill 17–7
fill pointer 15–2, 26–25
fill-pointer 15–27
fill-style conditional newline 22–18, 22–54
find 17–20
find-all-symbols 11–12
find-class 7–71
find-if 17–20
find-if-not 17–20
find-method 7–85
find-package 11–11
find-restart 9–50
find-symbol 11–9
finish-output 21–43
finite 26–25
first 14–25
fixnum 26–26
fixnum 12–20
flet 5–20
float 26–26
float 12–13, 12–85, 22–3
float-digits 12–82
float-precision 12–82
float-radix 12–82
float-sign 12–82
floating-point-inexact 12–91
floating-point-invalid-operation 12–90
floating-point-overflow 12–91
floating-point-underflow 12–91
flootp 12–86
floor 12–25
fmakunbound 5–19
font key 1–7
foo 1–15
for-value 26–26
force-output 21–43
form 26–26
formal argument 26–26
formal parameter 26–26
format 26–26
format 22–80
format argument 26–26
format control 22–16, 26–26
format directive 26–26
format string 26–26
formatter 22–46
fourth 14–25
free declaration 3–30, 26–26
fresh 26–26
fresh-line 21–21
froghine 26–27
fround 12–25
ftruncate 12–25
ftype 3–95
funbound 26–27
funcall 5–24
function 26–27
function 2–31, 4–20, 5–25, 25–18
function block name 26–27
function cell 26–27
function designator 26–27
function form 26–27
function name 26–27
function-keywords 7–28
function-lambda-expression 5–26
functional evaluation 26–27
functional value 26–27
functionp 5–28
further compilation 3–14, 26–28
G (format directive) 22–30
gcd 12–42
general 26–28
generalized boolean 26–28
generalized instance 26–28
generalized reference 5–1, 26–28
generalized synonym stream 26–28
generic function 4–22, 7–17, 26–28
generic function lambda list 3–39, 26–28
generic-function 4–22
gensym 26–28
gensym 10–8
*gensym-counter* 10–9
genenv 10–20
get 10–17
get-decoded-time 25–6
get-dispatch-macro-character 23–13
get-internal-real-time 25–15
get-internal-run-time 25–16
get-macro-character 23–14

Index ix
get-output-stream-string  21–52
get-properties          14–48
get-setf-expansion      5–98
get-universal-time      25–6
getf                    14–49
gethash                 18–10
global declaration     3–29, 26–29
global environment      3–1, 26–29
global variable         26–29
glyph                   26–29
go                      26–29
go point                26–29
go tag                  26–29
graphic                 13–3, 26–29
graphic-char-p          13–17
Greater-Than-Sign (format directive)  22–36
handle                  26–29
handler                 26–29
handler-bind            9–37
handler-case            9–38
hash table              26–29
hash-table              18–4
hash-table-count        18–6
hash-table-p            18–5
hash-table-rehash-size  18–7
hash-table-rehash-threshold 18–8
hash-table-size         18–9
hash-table-test         18–9
home package            26–29
host-namestring         19–28
I (format directive)    22–33
I/O customization variable  26–29
identical               26–30
identifier              26–30
identity                5–64
if                       5–70
ignorable               3–85
ignore                   3–85
ignore-errors           9–41
imagpart                12–59
immutable               26–30
implementation           26–30
implementation-limit    26–30
implementation-defined  26–30
implementation-dependent 26–30
implementation-independent 26–30
implicit block          26–30
implicit compilation   3–14, 26–30
implicit progn          26–30
implicit tagbody        26–31
import                  26–31
import                   11–12
improper list           14–2, 26–31
in-package              11–26
inaccessible            26–31
incf                    12–43
indefinite extent       26–31
indefinite scope        26–31
indicator               26–31
indirect instance       26–31
inherit                 26–31
initial pprint dispatch table 26–31
initial readtable       2–2, 26–31
initialization argument list  7–1, 26–31
initialization form     26–31
initialize-instance     7–88
inline                  3–93
input                   21–1, 26–31
input-stream-p          21–11
inspect                 25–23
instance                4–8, 26–31
int-char                A–1
integer                 26–31
integer                 12–17
integer-decode-float   12–82
integer-length          12–65
integerp               12–66
interactive stream      21–2, 26–32
interactive-stream-p   21–12
intern                  26–32
intern                   11–35
internal symbol         11–2, 26–32
internal time           25–4, 26–32
internal time unit      26–32
internal-time-units-per-second 25–15
interned                26–32
interpreted function   26–32
interpreted implementation 26–32
intersection            14–52

x  Programming Language—Common Lisp
Index  xi
mapc 14–40
mapcan 14–40
mapcar 14–40
mapcon 14–40
maphash 18–12
map 14–40
maplist 14–40
mapping 26–37
mask-field 12–81
max 7–26, 12–22
member 4–27, 14–39
member-if 14–39
member-if-not 14–39
merge 17–30
merge-pathnames 19–38
metaclass 4–1, 4–9, 26–37
Metaobject Protocol 26–37
method 26–38
method 4–24
method combination 26–38
method-combination 4–26, 25–18
method-combination-error 9–26
method-defining form 26–38
method-defining operator 7–18, 26–38
method-qualifiers 7–47
might signal 1–17
min 7–26, 12–22
minimal compilation 3–14, 26–38
Minus (sharpsign reader macro) 2–37
minusp 12–24
miser-style conditional newline 22–17, 22–54
mismatch 17–24
mod 12–19, 12–47
modified lambda list 26–38
*modules* 24–14
most recent 26–38
most-negative-double-float 12–87
most-negative-fixnum 12–82
most-negative-long-float 12–87
most-negative-short-float 12–87
most-possible-double-float 12–87
most-possible-fixnum 12–82
most-possible-long-float 12–87
most-possible-short-float 12–87
most-positive-double-float 12–87
most-positive-fixnum 12–82
most-positive-long-float 12–87
most-positive-short-float 12–87
muffle-warning 9–66, 9–68
multiple escape 2–9, 26–38
multiple values 26–38
multiple-value-bind 5–79
multiple-value-call 5–81
multiple-value-list 5–81
multiple-value-prog1 5–82
multiple-value-setq 5–83
multiple-values-limit 5–86
must signal 1–16
name 1–7, 26–39
name-char 13–25
named constant 26–39
namespace 3–1, 26–39
namestring 19–1, 26–39
namestring 19–28
nbutlast 14–33
nconc 7–26, 14–29
newline 26–39
Newline (format directive) 22–42
next method 7–23, 26–39
next-method-p 7–72
nickname 26–39
nil 1–13, 26–39
nil 1–13, 4–19, 5–54
nintersection 14–52
ninth 14–25
no-applicable-method 7–47
no-next-method 7–48
non-atomic 26–39
non-constant variable 26–39
non-correctable 26–40
non-empty 26–40
non-generic function 26–40
non-graphic 13–3, 26–40
non-list 26–40
non-local exit 26–40
non-nil 26–40
non-null lexical environment 26–40
non-simple 26–40
non-terminating 2–8, 26–40
non-top-level form 26–40
normal return 26–40
normalized 26–40

Index xiii
peek-char 21–17
Percent (format directive) 22–25
phase 12–58
physical pathname 26–43
pi 12–32
place 5–1, 26–43
plist 26–43
Plus (sharpsign reader macro) 2–37
plusp 12–24
pop 14–24
portable 26–43
position 17–21
position-if 17–21
position-if-not 17–21
potential copy 26–44
potential number 26–44
pprint 22–61
pprint dispatch table 22–16, 26–44
pprint-dispatch 22–47
pprint-exit-if-list-exhausted 22–48
pprint-fill 22–49
pprint-indent 22–50
pprint-linear 22–49
pprint-logical-block 22–51
pprint-newline 22–53
pprint-pop 22–55
pprint-tab 22–57
pprint-tabular 22–49
predicate 26–44
prepared to signal 1–16
present 11–2, 24–1, 26–44
pretty print 26–44
pretty printer 22–14, 26–44
pretty printing stream 26–44
primary method 26–44
primary value 26–44
prin1 22–61
prin1-to-string 22–64
princ 22–61
princ-to-string 22–64
principal 26–44
print 22–61
print name 26–45
*print-array* 22–66
*print-base* 22–67
*print-case* 22–68
*print-circle* 2–36, 2–37, 22–69
*print-escape* 22–70
*print-gensym* 22–71
*print-length* 22–72
*print-level* 22–72
*print-lines* 22–73
*print-miser-width* 22–74
print-not-readable 22–79
print-not-readable-object 22–79
print-object 22–57
*print-pprint-dispatch* 22–79
*print-pretty* 22–75
*print-radix* 22–67
*print-readably* 22–76
*print-right-margin* 22–78
print-unreadable-object 22–59
printer control variable 22–1, 26–45
printer escaping 22–2, 26–45
printing 26–45
probe-file 20–3
process 3–14, 26–45
processor 26–45
proclaim 26–45
proclamation 3–29, 26–45
prog 5–87
prog tag 26–45
prog* 5–87
prog1 5–89
prog2 5–89
progn 5–91, 7–26
program 26–45
program-error 5–104
programmer 26–45
programmer code 26–45
progv 5–38
proper list 14–2, 14–4, 26–45
proper name 26–45
proper sequence 26–46
proper subtype 26–46
property 26–46
property indicator 26–46
property list 26–46
property value 26–46
provide 24–15
psetf 5–99
psetq 5–40
purports to conform 26–46
push 14–23
pushnew 14–55
qualified method 26–46
qualifier 26–46
query I/O 26–46
*query-io* 21–57
Question-Mark (format directive) 22–39
quotation (of forms) 2–24, 2–26, 2–28
quotation (of strings) 2–26
quote 2–24, 2–26, 2–28, 3–65
quoted object 26–46
quux 1–15
R (format directive) 22–25
R (sharpsign reader macro) 2–34
radix 26–47
random 12–52
random state 26–47
random-state 12–50
*random-state* 12–53
random-state-p 12–53
rank 15–1, 26–47
rassoc 14–47
rassoc-if 14–47
rassoc-if-not 14–47
ratio 26–47
ratio marker 26–47
rational 26–47
rationalize 12–62
rationalp 12–63
read 26–47
read-line 21–24
read-preserving-whitespace 23–6
read-sequence 21–26
*read-suppress* 23–21
readably 26–47
reader 26–47
reader macro 26–47
reader macro function 2–8, 26–47
reader-error 23–23
readtable 2–1, 26–47
readtable 23–4
readtable case 26–47
readtable designator 26–48
*readtable* 23–22
readtable-case 23–11
readtablep 23–12
real 12–12
realp 12–60
realpart 12–59
recognizable subtype 26–48
redefinition 11–5
reduce 17–13
reference 26–48
registered package 26–48
reinitialize-instance 7–31
relative 26–48
:relative 19–7
rem 12–47
remf 14–51
remhash 18–11
remove 17–32
remove-duplicates 17–35
remove-if 17–32
remove-if-not 17–32
remove-method 7–49
remprop 10–19
rename-file 20–8
rename-package 11–14
repertoire 13–2, 26–48
replace 17–25
report 26–48
report message 9–4, 26–48
require 24–15
required parameter 26–48
rest 14–38
rest list 26–48
rest parameter 26–48
restart 26–48
restart 9–48
restart designator 26–49
restart function 26–49
restart-bind 9–54
restart-case 9–56
restart-name 9–61
return 26–49
return 5–47
return value 26–49
return-from 5–45
revappend 14–31
reverse 17–17
Right-Bracket (format directive) 22–37
Right-Paren (format directive) 22–40
right-parenthesis 26–49
S (format directive) 22–31
S (sharpsign reader macro) 2–36
safe 1–15, 26–49
safe call 3–50, 26–49
safe 3–29, 3–97
same 26–49
satisfies 4–26
satisfy the test 17–2, 17–4, 26–50
sbit 15–36
scale-float 12–82
schar 16–5
scope 26–50
script 26–50
search 17–23
second 14–25
secondary value 26–50
section 26–50
sections 22–14
self-evaluating object 26–50
semi-standard 26–50
semicolon 26–51
Semicolon (format directive) 22–40
Semicolon (reader macro) 2–24
sequence 17–1, 26–51
sequence 17–5
sequence function 17–1, 26–51
sequential 26–51
sequentially 26–51
serious condition 26–51
serious-condition 9–12
session 26–51
set 26–51
set 10–22
set-char-bit A–1
set-difference 14–56
set-dispatch-macro-character 23–13
set-exclusive-or 14–58
set-macro-character 23–14
set-pprint-dispatch 22–60
set-syntax-from-char 23–16
setf 5–99, 25–18
self expander 26–51
self expansion 26–51
self function 26–51
self function name 26–51
setq 5–39
seventh 14–25
shadow 3–11, 4–8, 26–51
shadow 11–15
shadowing symbol 11–3, 11–4, 26–51
shadowing symbols list 26–51
shared slot 26–52
shared-initialize 7–32
sharpsign 26–52
Sharpsign (reader macro) 2–29
Sharpsign (sharpsign reader macro) 2–37
Sharpsign A (reader macro) 2–35
Sharpsign Asterisk (reader macro) 2–32
Sharpsign B (reader macro) 2–33
special form  26–53
special operator  26–53
special variable  26–53
special-operator-p  3–103
specialize  26–54
specialized  26–54
specialized lambda list  3–40, 26–54
speed  3–97
spreadable argument list designator  26–54
sqrt  12–49
stable-sort  17–18
stack allocate  26–54
stack-allocated  26–54
standard  7–24, 7–26
standard character  2–2, 26–54
standard class  26–54
standard generic function  26–54
standard input  26–54
standard method combination  26–54
standard object  26–54
standard output  26–55
standard pprint dispatch table  26–55
standard readtable  2–1, 26–55
standard syntax  2–1, 26–55
standard-char  13–9
standard-char-p  13–18
standard-class  4–24
standard-generic-function  4–23
*standard-input*  21–57
standard-method  4–25
standard-object  4–25
*standard-output*  21–57
standardized  26–55
startup environment  3–14, 26–55
step  26–55
step  25–13
storage-condition  9–15
store-value  9–67, 9–68
stream  20–1, 21–1, 21–2, 26–55
stream  21–7
stream associated with a file  20–1, 26–55
stream designator  26–56
stream element type  26–56
stream variable  21–4, 26–56
stream variable designator  26–56
stream-element-type  21–13
stream-error  21–60
stream-error-stream  21–60
stream-external-format  21–36
streamp  21–14
string  15–3, 26–56
string  2–26, 16–2, 16–6
string designator  26–56
string equal  26–56
string stream  26–56
string-capitalize  16–7
string-char  A–1
string-char-p  A–1
string-downcase  16–7
string-equal  16–10
string-greaterp  16–10
string-left-trim  16–9
string-lessp  16–10
string-not-equal  16–10
string-not-greaterp  16–10
string-not-lessp  16–10
string-right-trim  16–9
string-stream  21–10
string-trim  16–9
string-upcase  16–7
string/=  16–10
string<=  16–10
string=<  16–10
string>=  16–10
string=>  16–10
stringp  16–12
structure  26–56
structure  2–36, 25–19
structure class  26–56
structure name  26–56
structure-class  4–24
structure-object  4–25
style warning  26–56
style-warning  9–12
subclass  4–8, 26–56
subexpression  26–57
subform  26–57
sublis  14–13
subrepertoire  26–57

Index  xix
subseq 17–9
subsetp 14–59
subst 14–15
subt-iff 14–15
substitute 17–26
substitute-if 17–26
substitute-if-not 17–26
subtype 26–57
subtypep 4–34
superclass 4–8, 26–57
supertype 26–57
supplied-p parameter 26–57
svref 15–32
sxhash 18–15
symbol 26–57
symbol 2–33, 10–2
symbol macro 3–18, 26–57
symbol-function 10–11
symbol-macrolet 3–18, 3–79
symbol-name 10–13
symbol-package 10–14
symbol-plist 10–15
symbol-value 10–16
symbolp 10–4
synonym stream 26–57
synonym stream symbol 26–57
synonym-stream 21–10
synonym-stream-symbol 21–46
syntax type 2–5, 26–57
SYSTEM package A–1
system class 26–58
system code 26–58
t 26–58
T (format directive) 22–34
tag 26–58
tagbody 5–48
tail 26–58
tailp 14–35
tan 12–28
tanh 12–33
target 26–58
tenth 14–25
*terminal-io* 21–59
terminating 2–8, 26–58
terpri 21–21
tertiary value 26–58
the 3–102
third 14–25
throw 26–58
tilde 26–58
Tilde (format directive) 22–25
Tilde A (format directive) 22–31
Tilde Ampersand (format directive) 22–25
Tilde Asterisk (format directive) 22–36
Tilde B (format directive) 22–26
Tilde C (format directive) 22–24
Tilde Circumflex (format directive) 22–41
Tilde D (format directive) 22–26
Tilde Dollarsign (format directive) 22–30
Tilde E (format directive) 22–28
Tilde F (format directive) 22–27
Tilde G (format directive) 22–30
Tilde Greater-Than-Sign (format directive) 22–36
Tilde I (format directive) 22–33
Tilde Left-Brace (format directive) 22–37
Tilde Left-Bracket (format directive) 22–36
Tilde Left-Paren (format directive) 22–39
Tilde Less-Than-Sign (format directive) 22–32, 22–35
Tilde Newline (format directive) 22–42
Tilde O (format directive) 22–27
Tilde P (format directive) 22–40
Tilde Percent (format directive) 22–25
Tilde Question-Mark (format directive) 22–39
Tilde R (format directive) 22–25
Tilde Right-Brace (format directive) 22–39
Tilde Right-Bracket (format directive) 22–37
Tilde Right-Paren (format directive) 22–40
Tilde S (format directive) 22–31
Tilde Semicolon (format directive) 22–40
Tilde Slash (format directive) 22–33
Tilde T (format directive) 22–34
Tilde Tilde (format directive) 22–25
Tilde Underscore (format directive) 22–32
Tilde Vertical-Bar (format directive) 22–25

xx Programming Language—Common Lisp
valid pathname version 26–62
valid physical pathname host 26–62
valid sequence index 26–62
value 26–63
value cell 26–63
values 4–29, 5–84
values-list 5–85
variable 26–63
variable 25–19
vector 15–1, 26–63
vector 2–31, 15–7, 15–33
vector-push 15–34
vector-push-extend 15–34
vectorp 15–36
vertical-bar 26–63
Vertical-Bar (format directive) 22–25
Vertical-Bar (sharpsign reader macro) 2–38
W (format directive) 22–32
warn 9–29
warning 9–11
warning 1–18
when 5–72
whitespace 26–63
wild 26–63
:wild 19–5, 19–8
:wild-inferiors 19–5, 19–8
wild-pathname-p 19–33
with-accessors 7–56
with-compilation-unit 24–8
with-condition-restarts 9–62
with-hash-table-iterator 18–13
with-input-from-string 21–54
with-open-file 21–37
with-open-stream 21–40
with-output-to-string 21–55
with-package-iterator 11–21
with-simple-restart 9–63
with-slots 7–58
with-standard-io-syntax 23–17
write 26–63
write 22–61
write-byte 21–16
write-char 21–23
write-line 21–25
write-sequence 21–27
write-string 21–25
write-to-string 22–64
writer 26–63
X (format directive) 22–27
X (sharpsign reader macro) 2–34
y-or-n-p 21–44
yes-or-no-p 21–44
yield 26–64
zerop 12–25
' 2–26
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.
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.
26-Apr-91 Margolin Review of 8.81/Chapters 1–12.
15-May-91 Barrett Review of 8.81/Chapters 5 (Misc), 11 (Conditions).
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).
21-Sep-91 Barrett Review of 10.16/Chapter 7 (Evaluation/Compilation).
28-Sep-91 Barrett Review of 10.95/Chapter 25 (Printer).

Programming Language—Common Lisp
13-Oct-91 Barrett Review (and help editing) of 10.104/Chapter 4
  (Evaluation and Compilation)
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).
02-Dec-91 Barrett Review of 10.156/Chapter 1 (Introduction),
  Chapter 2 (Syntax), and Chapter 3 (Evaluation and Compilation).
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-Mar-92 Loosemore Review of 11.82/Chapter 1, 3, 4, 5, 7, 8, 9, 10, 11, 12, 18, 22, 23, 24, 25, and 26.
04-Oct-93 X3J13 Minor fixes to Draft 13.65 before sending to X3.
08-Nov-93 Dalton “reply to reply to pr comments”. Document X3J13/94-311.
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.
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-viii.

iv Programming Language—Common Lisp
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:

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

Major Administrative Contributions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barrett, Kim</td>
<td>Mathis, Robert</td>
</tr>
<tr>
<td>Brown, Gary L.</td>
<td>Pitman, Kent M.</td>
</tr>
<tr>
<td>Eiron, Hanoch</td>
<td>Steele, Guy L., Jr.</td>
</tr>
<tr>
<td>Gabriel, Richard P.</td>
<td>Tyson, Mabry</td>
</tr>
<tr>
<td>Haflich, Steven M.</td>
<td>Van Deusen, Mary</td>
</tr>
<tr>
<td>Ida, Masayuki</td>
<td>White, JonL</td>
</tr>
<tr>
<td>Loeffler, David D.</td>
<td>Whittemore, Susan</td>
</tr>
<tr>
<td>Loosemore, Sandra</td>
<td>Woodyatt, Anne</td>
</tr>
<tr>
<td>Masinter, Larry</td>
<td>Zubkoff, Jan L.</td>
</tr>
</tbody>
</table>
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 Gakuen University  | MIT                |
| Apple Computer            | MITRE Corporation  |
| Boeing Advanced Technology Center | MSC          |
| Carnegie-Mellon University | NASA Ames Research Center |
| Chestnut Software         | National Bureau of Standards |
| Computer Sciences         | Nihon Symbolics    |
| Computer & Business Equipment Manufacturing Association (X3 Secretariat) | ParcPlace Systems, Inc. |
| CONTEL                    | Prime Computer     |
| CSC                       |                   |
| DARPA                     | Siemens            |
| Digital Equipment Corporation | Southern Illinois University |
| Encore                    | Sperry             |
| Evans & Sutherland        | SRI International  |
| Franz, Inc.               | Sun Microsystems   |
| Gigamos                   | Symbolics          |
| GMD                       | Tektronix          |
| Gold Hill                 | Texas Instruments  |
| Grumman Data Systems Corporation | The Aerospace Corporation |
| Harlequin, Ltd.           | Thinking Machines Corporation |
| Hewlett-Packard           | Unisys             |
| Honeywell                 | University of Bath |
| IBM                       | University of Edinburgh |
| Ibuki                     | University of Maryland |
| Integrated Inference Machines | University of Utah  |
| International LISP Associates | US Army          |
| Johnson Controls, Inc.    | USC/ISI            |
| LMI                       | Xerox              |
Individual Participants:

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

1. Introduction
Programming Language—Common Lisp
1.1 Scope, Purpose, and History

1.1.1 Scope and Purpose

The specification set forth in this document is designed to promote the portability of Common Lisp programs among a variety of data processing systems. It is a language specification aimed at an audience of implementors and knowledgeable programmers. It is neither a tutorial nor an implementation guide.

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 research 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 Chapter 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) and Section 1.7 (Language Subsets).

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

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

For information about data types, see Chapter 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 Chapter 12 (Numbers). For a complete list of *standardized types*, see Figure 4–2.

For information about general purpose control and data flow, see Chapter 5 (Data and Control Flow) or Chapter 6 (Iteration).
1.3 Referenced Publications


• *Smalltalk-80: The Language and its Implementation*, A. Goldberg and D. Robson, Addison-Wesley, 1983.


• *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.

1–6  Programming Language—Common Lisp
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\textsubscript{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\textsubscript{1}, package names, and loop keywords.

name

Denotes the name of a parameter or value.

In some situations the notation “\texttt{(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

\[
O_1 \mid \ldots \mid O_l
\]

where each \(O_i\) 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

\[
(O_{i_1} \ldots O_{i_j}) \quad 1 \leq j
\]

is spliced into the enclosing expression, such that if \(n \neq m\) and \(1 \leq n, m \leq j\), then either \(O_{i_n} \neq O_{i_m}\) or \(O_{i_n} = O_{i_m} = Q_k\), where for some \(1 \leq k \leq n\), \(O_k\) is of the form \(Q_k^*\). Furthermore, for each \(O_{i_n}\) that is of the form \(Q_k^1\), that element is required to appear somewhere in the list to be spliced.

For example, the expression

\[
(x \ [A \mid B^* \mid C] \ y)
\]

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:

\[
(x \ y)
(x \ B \ A \ C \ y)
(x \ A \ B \ B \ B \ B \ C \ y)
(x \ C \ B \ A \ B \ B \ y)
\]

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 \([O_1 \mid O_2 \mid \ldots]^+\) adds the additional restriction that at least one item from among the possible choices must be used. For example:

\[(x \ [A \mid B^* \mid C]^+ y)\]

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:

\[(x \ [A^1 \mid B^1 \mid C]^ y)\]

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:

\[\downarrow O\]

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

\[(x \ [\downarrow O] y)\]
\[ O ::= A \mid B^* \mid C \]

### 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\}^*`
- `ecase keyplace \{↓ normal-clause\}^* → \{result\}^*`
- `ecase keyform \{↓ normal-clause\}^* → \{result\}^*`

\[ normal-clause ::= (keys \{form\}^*) \]
\[ otherwise-clause ::= (\{otherwise \mid t\} \{form\}^*) \]
\[ clause ::= normal-clause \mid otherwise-clause \]

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:

\[ (\text{truncate} 7 5) → 1 2 \]
\[ (\text{truncate} 7 5) → 1, 2 \]

Each of the above three examples is equivalent, and specifies that \((\text{truncate} 7 5)\) 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.

\[ \text{or} \]

The notation ‘\(\text{or}\)’ is used to denote one of several possible alternate results. The example

\[
\text{char-name } \#\text{\(a\)} \\
\rightarrow \text{NIL} \\
\rightarrow \text{"LOWERCASE-a"} \\
\rightarrow \text{"Small-A"} \\
\rightarrow \text{"LA01"}
\]

indicates that \text{NIL}, “LOWERCASE-a”, “Small-A”, “LA01” are among the possible results of \(\text{char-name } \#\text{\(a\)}\)—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

\[
\text{char-name } \#\text{\(a\)} \rightarrow \text{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.

\[ \text{not} \]

The notation ‘\(\text{not}\)’ 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,

\[
\text{function-lambda-expression} \\
\quad \text{funcall } \#\text{'\((lambda \ (x) \ #\text{'}\((\lambda \ () \ x)) \text{ nil})\)} \\
\rightarrow \text{NIL, true, NIL} \\
\rightarrow \text{(LAMBDA () X), true, NIL} \\
\rightarrow \text{NIL, false, NIL} \\
\rightarrow \text{(LAMBDA () X), false, NIL}
\]

\[ \equiv \]

This indicates code equivalence. For example:

\[
\text{gcd } x \ (\text{gcd } y \ z) \equiv \ (\text{gcd } (\text{gcd } x \ y) \ z)
\]

This means that the results and observable side-effects of evaluating the form \(\text{gcd } x \ (\text{gcd } y \ z)\) are always the same as the results and observable side-effects of \(\text{gcd } (\text{gcd } x \ y) \ z\) for any \(x, y, \text{ and } z\).
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 “←” might be used.

\[(progn (format t "&Who? ") (read-line))\]

\[⊃ Who? Fred, Mary, and Sally←\]

\[→ "Fred, Mary, and Sally", false\]

### 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:

```lisp
(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) ; 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 "F00" 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)
⇒ (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 prevalence 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.

\[ F (x \text{ class}) (y \ t) \&\text{optional} \ z \&\text{key} \ k \]

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 \( \text{class} \); 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:

\[ \text{F} \ x \ y \ \text{&optional} \ z \ \text{&key} \ k \]

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:

\[ \text{file-position} \ \text{stream} \rightarrow \text{position} \]
\[ \text{file-position} \ \text{stream position-spec} \rightarrow \text{success-p} \]

is operationally equivalent to this line:

\[ \text{file-position} \ \text{stream} \ \text{&optional} \ \text{position-spec} \rightarrow \text{result} \]

and differs only in that it provides an 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 \( f \)) 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:

\[ \text{F} \ \text{&rest} \ \text{arguments} \]

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,

\[ F \&rest \text{ arguments}\]

means the same as

\[ F \&rest \text{ arguments} \]

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 “\(\to\)” precedes a list of values to be returned. For example:

\[ F a b c \to x \]

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:

\[ F a b c \to x, y, z \]

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,

\[ F \langle \text{no arguments} \rangle \to \langle \text{no values} \rangle \]

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:

\[ F a b c \to | \]

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 declare 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 \texttt{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 \texttt{foo} receives an argument of 17. Then the second passage (the one specifically about \texttt{foo}) would dominate because the description of the situational context is the most specific, and it would not be required that \texttt{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

\[ \langle \langle \text{Implementation} \rangle \rangle \text{ conforms with the requirements of ANSI } \langle \langle \text{standard number} \rangle \rangle \]

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

\[ \langle \langle \text{Implementation} \rangle \rangle \text{ conforms with the requirements of ANSI } \langle \langle \text{standard number} \rangle \rangle \text{ with the following exceptions: } \langle \langle \text{reference to or complete list of the requirements of the standard with which the implementation does not conform} \rangle \rangle \]

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:

```lisp
(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

Portionate 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.

A language that conforms to this requirement shall be described as being a “subset of Common Lisp as specified by ANSI ⟨⟨standard number⟩⟩.”
1.8 Deprecated Language Features

Deprecated language features are not expected to appear in future Common Lisp standards, 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 Figure 1–2 are deprecated.

<table>
<thead>
<tr>
<th>assoc-if-not</th>
<th>nsubst-if-not</th>
<th>require</th>
</tr>
</thead>
<tbody>
<tr>
<td>count-if-not</td>
<td>nsubstitute-if-not</td>
<td>set</td>
</tr>
<tr>
<td>delete-if-not</td>
<td>position-if-not</td>
<td>subst-if-not</td>
</tr>
<tr>
<td>find-if-not</td>
<td>provide</td>
<td>substitute-if-not</td>
</tr>
<tr>
<td>gentemp</td>
<td>rassoc-if-not</td>
<td></td>
</tr>
<tr>
<td>member-if-not</td>
<td>remove-if-not</td>
<td></td>
</tr>
</tbody>
</table>

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 Figure 1–3 are deprecated.

<table>
<thead>
<tr>
<th>adjoin</th>
<th>nset-difference</th>
<th>search</th>
</tr>
</thead>
<tbody>
<tr>
<td>assoc</td>
<td>nset-exclusive-or</td>
<td>set-difference</td>
</tr>
<tr>
<td>count</td>
<td>nsublis</td>
<td>set-exclusive-or</td>
</tr>
<tr>
<td>delete</td>
<td>nsubst</td>
<td>subst</td>
</tr>
<tr>
<td>delete-duplicates</td>
<td>nsubstitute</td>
<td>subsetp</td>
</tr>
<tr>
<td>find</td>
<td>nunion</td>
<td>subst</td>
</tr>
<tr>
<td>intersection</td>
<td>position</td>
<td>substitute</td>
</tr>
<tr>
<td>member</td>
<td>rassoc</td>
<td>tree-equal</td>
</tr>
<tr>
<td>mismatch</td>
<td>remove</td>
<td>union</td>
</tr>
<tr>
<td>nintersection</td>
<td>remove-duplicates</td>
<td></td>
</tr>
</tbody>
</table>

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 (Sharp-sign 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.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;allow-other-keys</td>
<td><em>print-miser-width</em></td>
</tr>
<tr>
<td>&amp;aux</td>
<td><em>print-pprint-dispatch</em></td>
</tr>
<tr>
<td>&amp;body</td>
<td><em>print-pretty</em></td>
</tr>
<tr>
<td>&amp;environment</td>
<td><em>print-radix</em></td>
</tr>
<tr>
<td>&amp;key</td>
<td><em>print-readably</em></td>
</tr>
<tr>
<td>&amp;optional</td>
<td><em>print-right-margin</em></td>
</tr>
<tr>
<td>&amp;rest</td>
<td><em>query-io</em></td>
</tr>
<tr>
<td>&amp;whole</td>
<td><em>random-state</em></td>
</tr>
<tr>
<td>*</td>
<td><em>read-base</em></td>
</tr>
<tr>
<td>**</td>
<td><em>read-default-float-format</em></td>
</tr>
<tr>
<td>***</td>
<td><em>read-eval</em></td>
</tr>
<tr>
<td><em>break-on-signals</em></td>
<td><em>read-suppress</em></td>
</tr>
<tr>
<td><em>compile-file-pathname</em></td>
<td><em>readtable</em></td>
</tr>
<tr>
<td><em>compile-file-truename</em></td>
<td><em>standard-input</em></td>
</tr>
<tr>
<td><em>compile-print</em></td>
<td><em>standard-output</em></td>
</tr>
<tr>
<td><em>compile-verbose</em></td>
<td><em>terminal-io</em></td>
</tr>
<tr>
<td><em>debug-io</em></td>
<td><em>trace-output</em></td>
</tr>
<tr>
<td><em>debugger-hook</em></td>
<td>+</td>
</tr>
<tr>
<td><em>default-pathname-defaults</em></td>
<td>++</td>
</tr>
<tr>
<td><em>error-output</em></td>
<td>+++</td>
</tr>
<tr>
<td><em>features</em></td>
<td>-</td>
</tr>
<tr>
<td><em>gensym-counter</em></td>
<td>/</td>
</tr>
<tr>
<td><em>load-pathname</em></td>
<td>//</td>
</tr>
<tr>
<td><em>load-print</em></td>
<td>///</td>
</tr>
<tr>
<td><em>load-truename</em></td>
<td>/=</td>
</tr>
<tr>
<td><em>load-verbose</em></td>
<td>1+</td>
</tr>
<tr>
<td><em>macroexpand-hook</em></td>
<td>1-</td>
</tr>
<tr>
<td><em>modules</em></td>
<td>&lt;</td>
</tr>
<tr>
<td><em>package</em></td>
<td>&lt;=</td>
</tr>
<tr>
<td><em>print-array</em></td>
<td>=</td>
</tr>
<tr>
<td><em>print-base</em></td>
<td>&gt;</td>
</tr>
<tr>
<td><em>print-case</em></td>
<td>&gt;=</td>
</tr>
<tr>
<td><em>print-circle</em></td>
<td>abort</td>
</tr>
<tr>
<td><em>print-escape</em></td>
<td>abs</td>
</tr>
<tr>
<td><em>print-gensym</em></td>
<td>acons</td>
</tr>
<tr>
<td><em>print-length</em></td>
<td>acos</td>
</tr>
<tr>
<td><em>print-level</em></td>
<td>acosh</td>
</tr>
<tr>
<td><em>print-lines</em></td>
<td>add-method</td>
</tr>
</tbody>
</table>

Figure 1–4. Symbols in the COMMON-LISP package (part one of twelve).
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjoin</td>
<td>atom</td>
</tr>
<tr>
<td>adjust-array</td>
<td>base-char</td>
</tr>
<tr>
<td>adjustable-array-p</td>
<td>base-string</td>
</tr>
<tr>
<td>allocate-instance</td>
<td>bignum</td>
</tr>
<tr>
<td>alpha-char-p</td>
<td>bit</td>
</tr>
<tr>
<td>alphanumericp</td>
<td>bit-and</td>
</tr>
<tr>
<td>and</td>
<td>bit-andc1</td>
</tr>
<tr>
<td>append</td>
<td>bit-andc2</td>
</tr>
<tr>
<td>apply</td>
<td>bit-eqv</td>
</tr>
<tr>
<td>apropos</td>
<td>bit-ior</td>
</tr>
<tr>
<td>apropos-list</td>
<td>bit-nand</td>
</tr>
<tr>
<td>aref</td>
<td>bit-nor</td>
</tr>
<tr>
<td>arithmetic-error</td>
<td>bit-not</td>
</tr>
<tr>
<td>arithmetic-error-operands</td>
<td>bit-orc1</td>
</tr>
<tr>
<td>arithmetic-error-operation</td>
<td>bit-orc2</td>
</tr>
<tr>
<td>array</td>
<td>bit-vector</td>
</tr>
<tr>
<td>array-dimension</td>
<td>bit-vector-p</td>
</tr>
<tr>
<td>array-dimension-limit</td>
<td>bit-xor</td>
</tr>
<tr>
<td>array-dimensions</td>
<td>block</td>
</tr>
<tr>
<td>array-displacement</td>
<td>boole</td>
</tr>
<tr>
<td>array-element-type</td>
<td>boole-1</td>
</tr>
<tr>
<td>array-has-fill-pointer-p</td>
<td>boole-2</td>
</tr>
<tr>
<td>array-in-bounds-p</td>
<td>boole-and</td>
</tr>
<tr>
<td>array-rank</td>
<td>boole-andc1</td>
</tr>
<tr>
<td>array-rank-limit</td>
<td>boole-andc2</td>
</tr>
<tr>
<td>array-row-major-index</td>
<td>boole-c1</td>
</tr>
<tr>
<td>array-total-size</td>
<td>boole-c2</td>
</tr>
<tr>
<td>array-total-size-limit</td>
<td>boole-clr</td>
</tr>
<tr>
<td>arrayp</td>
<td>boole-eqv</td>
</tr>
<tr>
<td>ash</td>
<td>boole-ior</td>
</tr>
<tr>
<td>asin</td>
<td>boole-nand</td>
</tr>
<tr>
<td>asinh</td>
<td>boole-nor</td>
</tr>
<tr>
<td>assert</td>
<td>boole-orc1</td>
</tr>
<tr>
<td>assoc</td>
<td>boole-orc2</td>
</tr>
<tr>
<td>assoc-if</td>
<td>boole-set</td>
</tr>
<tr>
<td>assoc-if-not</td>
<td>boole-xor</td>
</tr>
<tr>
<td>atan</td>
<td>boolean</td>
</tr>
<tr>
<td>atanh</td>
<td>both-case-p</td>
</tr>
</tbody>
</table>

Figure 1–5. Symbols in the COMMON-LISP package (part two of twelve).
<table>
<thead>
<tr>
<th>cddadr</th>
<th>clear-input</th>
<th>copy-tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>cddar</td>
<td>clear-output</td>
<td>cos</td>
</tr>
<tr>
<td>cdddar</td>
<td>close</td>
<td>cosh</td>
</tr>
<tr>
<td>cddddr</td>
<td>crlhash</td>
<td>count</td>
</tr>
<tr>
<td>cdddr</td>
<td>code-char</td>
<td>count-if</td>
</tr>
<tr>
<td>cddr</td>
<td>coerce</td>
<td>count-if-not</td>
</tr>
<tr>
<td>cdr</td>
<td>compilation-speed</td>
<td>ctypecase</td>
</tr>
<tr>
<td>ceiling</td>
<td>compile</td>
<td>debug</td>
</tr>
<tr>
<td>cell-error</td>
<td>compile-file</td>
<td>defef</td>
</tr>
<tr>
<td>cell-error-name</td>
<td>compile-file-pathname</td>
<td>declare</td>
</tr>
<tr>
<td>error</td>
<td>compiled-function</td>
<td>declaration</td>
</tr>
<tr>
<td>change-class</td>
<td>compiled-function-p</td>
<td>declare</td>
</tr>
<tr>
<td>char</td>
<td>compiler-macro</td>
<td>decode-float</td>
</tr>
<tr>
<td>char-code</td>
<td>compiler-macro-function</td>
<td>decode-universal-time</td>
</tr>
<tr>
<td>char-code-limit</td>
<td>complement</td>
<td>defclass</td>
</tr>
<tr>
<td>char-downcase</td>
<td>complex</td>
<td>defconstant</td>
</tr>
<tr>
<td>char-equal</td>
<td>complexp</td>
<td>defgeneric</td>
</tr>
<tr>
<td>char-greaterp</td>
<td>compute-applicable-methods</td>
<td>define-compiler-macro</td>
</tr>
<tr>
<td>char-int</td>
<td>compute-restarts</td>
<td>define-condition</td>
</tr>
<tr>
<td>char-lessp</td>
<td>concatenate</td>
<td>define-method-combination</td>
</tr>
<tr>
<td>char-name</td>
<td>concatenated-stream</td>
<td>define-modify-macro</td>
</tr>
<tr>
<td>char-not-equal</td>
<td>concatenated-stream-streams</td>
<td>define-setf-expander</td>
</tr>
<tr>
<td>char-not-greaterp</td>
<td>cond</td>
<td>define-symbol-macro</td>
</tr>
<tr>
<td>char-not-lessp</td>
<td>condition</td>
<td>defmacro</td>
</tr>
<tr>
<td>char-upcase</td>
<td>conjugate</td>
<td>defmethod</td>
</tr>
<tr>
<td>char/=</td>
<td>cons</td>
<td>defpackage</td>
</tr>
<tr>
<td>char&lt;</td>
<td>consp</td>
<td>defparameter</td>
</tr>
<tr>
<td>char&lt;=</td>
<td>constantly</td>
<td>defsetf</td>
</tr>
<tr>
<td>char&lt;&gt;</td>
<td>constantp</td>
<td>defstruct</td>
</tr>
<tr>
<td>char&gt;=</td>
<td>continue</td>
<td>deftype</td>
</tr>
<tr>
<td>char&gt;</td>
<td>control-error</td>
<td>defun</td>
</tr>
<tr>
<td>character</td>
<td>copy-alist</td>
<td>defvar</td>
</tr>
<tr>
<td>characterp</td>
<td>copy-list</td>
<td>delete</td>
</tr>
<tr>
<td>check-type</td>
<td>copy-pprint-dispatch</td>
<td>delete-duplicates</td>
</tr>
<tr>
<td>cis</td>
<td>copy-readable</td>
<td>delete-file</td>
</tr>
<tr>
<td>class</td>
<td>copy-seq</td>
<td>delete-if</td>
</tr>
<tr>
<td>class-name</td>
<td>copy-structure</td>
<td>delete-if-not</td>
</tr>
<tr>
<td>class-of</td>
<td>copy-symbol</td>
<td>delete-package</td>
</tr>
</tbody>
</table>

Figure 1–6. Symbols in the COMMON-LISP package (part three of twelve).
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>denominator</td>
<td>eq</td>
</tr>
<tr>
<td>deposit-field</td>
<td>eql</td>
</tr>
<tr>
<td>describe</td>
<td>equal</td>
</tr>
<tr>
<td>describe-object</td>
<td>equalp</td>
</tr>
<tr>
<td>destructuring-bind</td>
<td>error</td>
</tr>
<tr>
<td>digit-char</td>
<td>etypecase</td>
</tr>
<tr>
<td>digit-char-p</td>
<td>eval</td>
</tr>
<tr>
<td>directory</td>
<td>eval-when</td>
</tr>
<tr>
<td>directory-namestring</td>
<td>evenp</td>
</tr>
<tr>
<td>disassemble</td>
<td>every</td>
</tr>
<tr>
<td>division-by-zero</td>
<td>exp</td>
</tr>
<tr>
<td>do</td>
<td>export</td>
</tr>
<tr>
<td>do*</td>
<td>expt</td>
</tr>
<tr>
<td>do-all-symbols</td>
<td>extended-char</td>
</tr>
<tr>
<td>do-external-symbols</td>
<td>fboundp</td>
</tr>
<tr>
<td>do-symbols</td>
<td>fceiling</td>
</tr>
<tr>
<td>documentation</td>
<td>fdefinition</td>
</tr>
<tr>
<td>dolist</td>
<td>ffloor</td>
</tr>
<tr>
<td>dotimes</td>
<td>fifth</td>
</tr>
<tr>
<td>double-float</td>
<td>file-author</td>
</tr>
<tr>
<td>double-float-epsilon</td>
<td>file-error</td>
</tr>
<tr>
<td>double-float-negative-epsilon</td>
<td>file-error-pathname</td>
</tr>
<tr>
<td>dpb</td>
<td>file-length</td>
</tr>
<tr>
<td>dribble</td>
<td>file-namestring</td>
</tr>
<tr>
<td>dynamic-extent</td>
<td>file-position</td>
</tr>
<tr>
<td>ecase</td>
<td>file-stream</td>
</tr>
<tr>
<td>echo-stream</td>
<td>file-string-length</td>
</tr>
<tr>
<td>echo-stream-input-stream</td>
<td>file-write-date</td>
</tr>
<tr>
<td>echo-stream-output-stream</td>
<td>fill</td>
</tr>
<tr>
<td>ed</td>
<td>fill-pointer</td>
</tr>
<tr>
<td>eighth</td>
<td>find</td>
</tr>
<tr>
<td>elt</td>
<td>find-all-symbols</td>
</tr>
<tr>
<td>encode-universal-time</td>
<td>find-class</td>
</tr>
<tr>
<td>end-of-file</td>
<td>find-if</td>
</tr>
<tr>
<td>endp</td>
<td>find-if-not</td>
</tr>
<tr>
<td>enough-namestring</td>
<td>find-method</td>
</tr>
<tr>
<td>ensure-directories-exist</td>
<td>find-package</td>
</tr>
<tr>
<td>ensure-generic-function</td>
<td>find-restart</td>
</tr>
</tbody>
</table>

Figure 1–7. Symbols in the COMMON-LISP package (part four of twelve).
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>find-symbol</td>
<td>get-internal-run-time</td>
</tr>
<tr>
<td>finish-output</td>
<td>get-macro-character</td>
</tr>
<tr>
<td>first</td>
<td>get-output-stream-string</td>
</tr>
<tr>
<td>fixnum</td>
<td>get-properties</td>
</tr>
<tr>
<td>flet</td>
<td>get-setf-expansion</td>
</tr>
<tr>
<td>float</td>
<td>get-universal-time</td>
</tr>
<tr>
<td>float-digits</td>
<td>getf</td>
</tr>
<tr>
<td>float-precision</td>
<td>gethash</td>
</tr>
<tr>
<td>float-radix</td>
<td>go</td>
</tr>
<tr>
<td>float-sign</td>
<td>graphic-char-p</td>
</tr>
<tr>
<td>floating-point-inexact</td>
<td>handler-bind</td>
</tr>
<tr>
<td>floating-point-invalid-operation</td>
<td>handler-case</td>
</tr>
<tr>
<td>floating-point-overflow</td>
<td>hash-table</td>
</tr>
<tr>
<td>floating-point-underflow</td>
<td>hash-table-count</td>
</tr>
<tr>
<td>floatp</td>
<td>hash-table-p</td>
</tr>
<tr>
<td>floor</td>
<td>hash-table-rehash-size</td>
</tr>
<tr>
<td>fmakunbound</td>
<td>hash-table-rehash-threshold</td>
</tr>
<tr>
<td>force-output</td>
<td>hash-table-size</td>
</tr>
<tr>
<td>format</td>
<td>hash-table-test</td>
</tr>
<tr>
<td>formatter</td>
<td>host-namestring</td>
</tr>
<tr>
<td>fourth</td>
<td>identity</td>
</tr>
<tr>
<td>fresh-line</td>
<td>if</td>
</tr>
<tr>
<td>fround</td>
<td>ignorable</td>
</tr>
<tr>
<td>ftruncate</td>
<td>ignore</td>
</tr>
<tr>
<td>ftype</td>
<td>ignore-errors</td>
</tr>
<tr>
<td>funcall</td>
<td>imagoart</td>
</tr>
<tr>
<td>function</td>
<td>import</td>
</tr>
<tr>
<td>function-keywords</td>
<td>in-package</td>
</tr>
<tr>
<td>function-lambda-expression</td>
<td>inef</td>
</tr>
<tr>
<td>functionp</td>
<td>initialize-instance</td>
</tr>
<tr>
<td>gcd</td>
<td>inline</td>
</tr>
<tr>
<td>generic-function</td>
<td>input-stream-p</td>
</tr>
<tr>
<td>gensym</td>
<td>inspect</td>
</tr>
<tr>
<td>gentemp</td>
<td>integer</td>
</tr>
<tr>
<td>get</td>
<td>integer-decode-float</td>
</tr>
<tr>
<td>get-decoded-time</td>
<td>integer-length</td>
</tr>
<tr>
<td>get-dispatch-macro-character</td>
<td>integererp</td>
</tr>
<tr>
<td>get-internal-real-time</td>
<td>interactive-stream-p</td>
</tr>
</tbody>
</table>

Figure 1–8. Symbols in the COMMON-LISP package (part five of twelve).
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intern</td>
<td>lisp-implementation-type</td>
</tr>
<tr>
<td>internal-time-units-per-second</td>
<td>lisp-implementation-version</td>
</tr>
<tr>
<td>intersection</td>
<td>list</td>
</tr>
<tr>
<td>invalid-method-error</td>
<td>list*</td>
</tr>
<tr>
<td>invoke-debugger</td>
<td>list-all-packages</td>
</tr>
<tr>
<td>invoke-restart</td>
<td>list-length</td>
</tr>
<tr>
<td>invoke-restart-interactively</td>
<td>listen</td>
</tr>
<tr>
<td>isqrt</td>
<td>listp</td>
</tr>
<tr>
<td>keyword</td>
<td>load</td>
</tr>
<tr>
<td>keywordp</td>
<td>load-logical-pathname-translations</td>
</tr>
<tr>
<td>labels</td>
<td>load-time-value</td>
</tr>
<tr>
<td>lambda</td>
<td>locally</td>
</tr>
<tr>
<td>lambda-list-keywords</td>
<td>log</td>
</tr>
<tr>
<td>lambda-parameters-limit</td>
<td>logand</td>
</tr>
<tr>
<td>last</td>
<td>logandc1</td>
</tr>
<tr>
<td>lcm</td>
<td>logandc2</td>
</tr>
<tr>
<td>ldb</td>
<td>logbitp</td>
</tr>
<tr>
<td>ldb-test</td>
<td>logcount</td>
</tr>
<tr>
<td>ldiff</td>
<td>logeqv</td>
</tr>
<tr>
<td>least-negative-double-float</td>
<td>logical-pathname</td>
</tr>
<tr>
<td>least-negative-long-float</td>
<td>logical-pathname-translations</td>
</tr>
<tr>
<td>least-negative-normalized-double-float</td>
<td>logior</td>
</tr>
<tr>
<td>least-negative-normalized-long-float</td>
<td>lognand</td>
</tr>
<tr>
<td>least-negative-normalized-short-float</td>
<td>lognor</td>
</tr>
<tr>
<td>least-negative-normalized-single-float</td>
<td>lognot</td>
</tr>
<tr>
<td>least-negative-short-float</td>
<td>logorc1</td>
</tr>
<tr>
<td>least-negative-single-float</td>
<td>logorc2</td>
</tr>
<tr>
<td>least-positive-double-float</td>
<td>logtest</td>
</tr>
<tr>
<td>least-positive-long-float</td>
<td>logxor</td>
</tr>
<tr>
<td>least-positive-normalized-double-float</td>
<td>long-test</td>
</tr>
<tr>
<td>least-positive-normalized-long-float</td>
<td>long-float-epsilon</td>
</tr>
<tr>
<td>least-positive-normalized-short-float</td>
<td>long-float-negative-epsilon</td>
</tr>
<tr>
<td>least-positive-normalized-single-float</td>
<td>long-site-name</td>
</tr>
<tr>
<td>least-positive-short-float</td>
<td>loop</td>
</tr>
<tr>
<td>least-positive-single-float</td>
<td>loop-finish</td>
</tr>
<tr>
<td>length</td>
<td>lower-case-p</td>
</tr>
<tr>
<td>let</td>
<td>machine-instance</td>
</tr>
<tr>
<td>let*</td>
<td>machine-type</td>
</tr>
</tbody>
</table>

Figure 1–9. Symbols in the COMMON-LISP package (part six of twelve).
<table>
<thead>
<tr>
<th>machine-version</th>
<th>mask-field</th>
</tr>
</thead>
<tbody>
<tr>
<td>macro-function</td>
<td>max</td>
</tr>
<tr>
<td>macroexpand</td>
<td>member</td>
</tr>
<tr>
<td>macroexpand-1</td>
<td>member-if</td>
</tr>
<tr>
<td>macrolet</td>
<td>member-if-not</td>
</tr>
<tr>
<td>make-array</td>
<td>merge</td>
</tr>
<tr>
<td>make-broadcast-stream</td>
<td>merge-pathnames</td>
</tr>
<tr>
<td>make-concatenated-stream</td>
<td>method</td>
</tr>
<tr>
<td>make-condition</td>
<td>method-combination</td>
</tr>
<tr>
<td>make-dispatch-macro-character</td>
<td>method-combination-error</td>
</tr>
<tr>
<td>make-echo-stream</td>
<td>method-qualifiers</td>
</tr>
<tr>
<td>make-hash-table</td>
<td>min</td>
</tr>
<tr>
<td>make-instance</td>
<td>minusp</td>
</tr>
<tr>
<td>make-instances-obsolete</td>
<td>mismatch</td>
</tr>
<tr>
<td>make-list</td>
<td>mod</td>
</tr>
<tr>
<td>make-load-form</td>
<td>most-negative-double-float</td>
</tr>
<tr>
<td>make-load-form-saving-slots</td>
<td>most-negative-fixnum</td>
</tr>
<tr>
<td>make-method</td>
<td>most-negative-long-float</td>
</tr>
<tr>
<td>make-package</td>
<td>most-negative-short-float</td>
</tr>
<tr>
<td>make-pathname</td>
<td>most-negative-single-float</td>
</tr>
<tr>
<td>make-random-state</td>
<td>most-positive-double-float</td>
</tr>
<tr>
<td>make-sequence</td>
<td>most-positive-fixnum</td>
</tr>
<tr>
<td>make-string</td>
<td>most-positive-long-float</td>
</tr>
<tr>
<td>make-string-input-stream</td>
<td>most-positive-short-float</td>
</tr>
<tr>
<td>make-string-output-stream</td>
<td>most-positive-single-float</td>
</tr>
<tr>
<td>make-symbol</td>
<td>muffle-warning</td>
</tr>
<tr>
<td>make-synonym-stream</td>
<td>multiple-value-bind</td>
</tr>
<tr>
<td>make-two-way-stream</td>
<td>multiple-value-call</td>
</tr>
<tr>
<td>makunbound</td>
<td>multiple-value-list</td>
</tr>
<tr>
<td>map</td>
<td>multiple-value-prog1</td>
</tr>
<tr>
<td>map-into</td>
<td>multiple-value-setq</td>
</tr>
<tr>
<td>mapc</td>
<td>multiple-values-limit</td>
</tr>
<tr>
<td>mapcan</td>
<td>name-char</td>
</tr>
<tr>
<td>mapcar</td>
<td>namestring</td>
</tr>
<tr>
<td>mapcon</td>
<td>nbutlast</td>
</tr>
<tr>
<td>maphash</td>
<td>nconc</td>
</tr>
<tr>
<td>mapl</td>
<td>next-method-p</td>
</tr>
<tr>
<td>maplist</td>
<td>nil</td>
</tr>
</tbody>
</table>

Figure 1–10. Symbols in the COMMON-LISP package (part seven of twelve).
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>nintersection</td>
<td>package-error</td>
</tr>
<tr>
<td>ninth</td>
<td>package-error-package</td>
</tr>
<tr>
<td>no-applicable-method</td>
<td>package-name</td>
</tr>
<tr>
<td>no-next-method</td>
<td>package-nicknames</td>
</tr>
<tr>
<td>not</td>
<td>package-shadowing-symbols</td>
</tr>
<tr>
<td>notany</td>
<td>package-use-list</td>
</tr>
<tr>
<td>notevery</td>
<td>package-used-by-list</td>
</tr>
<tr>
<td>notininline</td>
<td>packagep</td>
</tr>
<tr>
<td>nreconc</td>
<td>pairlis</td>
</tr>
<tr>
<td>nreverse</td>
<td>parse-error</td>
</tr>
<tr>
<td>nset-difference</td>
<td>parse-integer</td>
</tr>
<tr>
<td>nset-exclusive-or</td>
<td>parse-namestring</td>
</tr>
<tr>
<td>nstring-capitalize</td>
<td>pathname</td>
</tr>
<tr>
<td>nstring-downcase</td>
<td>pathname-device</td>
</tr>
<tr>
<td>nstring-upcase</td>
<td>pathname-directory</td>
</tr>
<tr>
<td>nsublis</td>
<td>pathname-host</td>
</tr>
<tr>
<td>nsubst</td>
<td>pathname-match-p</td>
</tr>
<tr>
<td>nsubst-if</td>
<td>pathname-name</td>
</tr>
<tr>
<td>nsubst-if-not</td>
<td>pathname-type</td>
</tr>
<tr>
<td>nsubstitute</td>
<td>pathname-version</td>
</tr>
<tr>
<td>nsubstitute-if</td>
<td>pathnamep</td>
</tr>
<tr>
<td>nsubstitute-if-not</td>
<td>peek-char</td>
</tr>
<tr>
<td>nth</td>
<td>phase</td>
</tr>
<tr>
<td>nth-value</td>
<td>pi</td>
</tr>
<tr>
<td>nthcdr</td>
<td>plusp</td>
</tr>
<tr>
<td>null</td>
<td>pop</td>
</tr>
<tr>
<td>number</td>
<td>position</td>
</tr>
<tr>
<td>numberp</td>
<td>position-if</td>
</tr>
<tr>
<td>numerator</td>
<td>position-if-not</td>
</tr>
<tr>
<td>nunion</td>
<td>pprint</td>
</tr>
<tr>
<td>oddp</td>
<td>pprint-dispatch</td>
</tr>
<tr>
<td>open</td>
<td>pprint-exit-if-list-exhausted</td>
</tr>
<tr>
<td>open-stream-p</td>
<td>pprint-fill</td>
</tr>
<tr>
<td>optimize</td>
<td>pprint-indent</td>
</tr>
<tr>
<td>or</td>
<td>pprint-linear</td>
</tr>
<tr>
<td>otherwise</td>
<td>pprint-logical-block</td>
</tr>
<tr>
<td>output-stream-p</td>
<td>pprint-newline</td>
</tr>
<tr>
<td>package</td>
<td>pprint-pop</td>
</tr>
</tbody>
</table>

Figure 1–11. Symbols in the COMMON-LISP package (part eight of twelve).
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pprint-tab</td>
<td>read-char</td>
</tr>
<tr>
<td>pprint-tabular</td>
<td>read-char-no-hang</td>
</tr>
<tr>
<td>prin1</td>
<td>read-delimited-list</td>
</tr>
<tr>
<td>prin1-to-string</td>
<td>read-from-string</td>
</tr>
<tr>
<td>princ</td>
<td>read-line</td>
</tr>
<tr>
<td>princ-to-string</td>
<td>read-preserving-whitespace</td>
</tr>
<tr>
<td>print</td>
<td>read-sequence</td>
</tr>
<tr>
<td>print-not-readable</td>
<td>reader-error</td>
</tr>
<tr>
<td>print-not-readable-object</td>
<td>readtable</td>
</tr>
<tr>
<td>print-object</td>
<td>readtable-case</td>
</tr>
<tr>
<td>print-unreadable-object</td>
<td>readtablep</td>
</tr>
<tr>
<td>probe-file</td>
<td>real</td>
</tr>
<tr>
<td>proclaim</td>
<td>realp</td>
</tr>
<tr>
<td>prog</td>
<td>realpart</td>
</tr>
<tr>
<td>prog*</td>
<td>reduce</td>
</tr>
<tr>
<td>prog1</td>
<td>reinitialize-instance</td>
</tr>
<tr>
<td>prog2</td>
<td>rem</td>
</tr>
<tr>
<td>progn</td>
<td>remf</td>
</tr>
<tr>
<td>program-error</td>
<td>remhash</td>
</tr>
<tr>
<td>progv</td>
<td>remove</td>
</tr>
<tr>
<td>provide</td>
<td>remove-duplicates</td>
</tr>
<tr>
<td>psetf</td>
<td>remove-if</td>
</tr>
<tr>
<td>psetq</td>
<td>remove-if-not</td>
</tr>
<tr>
<td>push</td>
<td>remove-method</td>
</tr>
<tr>
<td>pushnew</td>
<td>remprop</td>
</tr>
<tr>
<td>quote</td>
<td>rename-file</td>
</tr>
<tr>
<td>random</td>
<td>rename-package</td>
</tr>
<tr>
<td>random-state</td>
<td>replace</td>
</tr>
<tr>
<td>random-state-p</td>
<td>require</td>
</tr>
<tr>
<td>rassoc</td>
<td>rest</td>
</tr>
<tr>
<td>rassoc-if</td>
<td>restart</td>
</tr>
<tr>
<td>rassoc-if-not</td>
<td>restart-bind</td>
</tr>
<tr>
<td>ratio</td>
<td>restart-case</td>
</tr>
<tr>
<td>rational</td>
<td>restart-name</td>
</tr>
<tr>
<td>rationalize</td>
<td>return</td>
</tr>
<tr>
<td>rationalize</td>
<td>return-from</td>
</tr>
<tr>
<td>read</td>
<td>revappend</td>
</tr>
<tr>
<td>read-alp</td>
<td>reverse</td>
</tr>
</tbody>
</table>

Figure 1–12. Symbols in the COMMON-LISP package (part nine of twelve).
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>room</td>
<td>simple-bit-vector</td>
</tr>
<tr>
<td>rotatef</td>
<td>simple-bit-vector-p</td>
</tr>
<tr>
<td>round</td>
<td>simple-condition</td>
</tr>
<tr>
<td>row-major-aref</td>
<td>simple-condition-format-arguments</td>
</tr>
<tr>
<td>rplaca</td>
<td>simple-condition-format-control</td>
</tr>
<tr>
<td>rplacd</td>
<td>simple-error</td>
</tr>
<tr>
<td>safety</td>
<td>simple-string</td>
</tr>
<tr>
<td>satisfies</td>
<td>simple-string-p</td>
</tr>
<tr>
<td>sbit</td>
<td>simple-type-error</td>
</tr>
<tr>
<td>scale-float</td>
<td>simple-vector</td>
</tr>
<tr>
<td>schar</td>
<td>simple-vector-p</td>
</tr>
<tr>
<td>search</td>
<td>simple-warning</td>
</tr>
<tr>
<td>second</td>
<td>sin</td>
</tr>
<tr>
<td>sequence</td>
<td>single-float</td>
</tr>
<tr>
<td>serious-condition</td>
<td>single-float-epsilon</td>
</tr>
<tr>
<td>set</td>
<td>single-float-negative-epsilon</td>
</tr>
<tr>
<td>set-difference</td>
<td>sinh</td>
</tr>
<tr>
<td>set-dispatch-macro-character</td>
<td>sixth</td>
</tr>
<tr>
<td>set-exclusive-or</td>
<td>sleep</td>
</tr>
<tr>
<td>set-macro-character</td>
<td>slot-boundp</td>
</tr>
<tr>
<td>set-pprint-dispatch</td>
<td>slot-exists-p</td>
</tr>
<tr>
<td>set-syntax-from-char</td>
<td>slot-makunbound</td>
</tr>
<tr>
<td>setf</td>
<td>slot-missing</td>
</tr>
<tr>
<td>setq</td>
<td>slot-unbound</td>
</tr>
<tr>
<td>seventh</td>
<td>slot-value</td>
</tr>
<tr>
<td>shadow</td>
<td>software-type</td>
</tr>
<tr>
<td>shadowing-import</td>
<td>software-version</td>
</tr>
<tr>
<td>shared-initialize</td>
<td>some</td>
</tr>
<tr>
<td>shiftf</td>
<td>sort</td>
</tr>
<tr>
<td>short-float</td>
<td>space</td>
</tr>
<tr>
<td>short-float-epsilon</td>
<td>special</td>
</tr>
<tr>
<td>short-float-negative-epsilon</td>
<td>special-operator-p</td>
</tr>
<tr>
<td>short-site-name</td>
<td>speed</td>
</tr>
<tr>
<td>signal</td>
<td>sqrt</td>
</tr>
<tr>
<td>signed-byte</td>
<td>stable-sort</td>
</tr>
<tr>
<td>signum</td>
<td>standard</td>
</tr>
<tr>
<td>simple-array</td>
<td>standard-char</td>
</tr>
<tr>
<td>simple-base-string</td>
<td>standard-char-p</td>
</tr>
</tbody>
</table>

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
stream                    symbol
streamamp                 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            tan
string-trim              tanh
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).
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>truncate</td>
<td>values-list</td>
</tr>
<tr>
<td>two-way-stream</td>
<td>variable</td>
</tr>
<tr>
<td>two-way-stream-input-stream</td>
<td>vector</td>
</tr>
<tr>
<td>two-way-stream-output-stream</td>
<td>vector-pop</td>
</tr>
<tr>
<td>type</td>
<td>vector-push</td>
</tr>
<tr>
<td>type-error</td>
<td>vector-push-extend</td>
</tr>
<tr>
<td>type-error-datum</td>
<td>vectorp</td>
</tr>
<tr>
<td>type-error-expected-type</td>
<td>warn</td>
</tr>
<tr>
<td>type-of</td>
<td>warning</td>
</tr>
<tr>
<td>typecase</td>
<td>when</td>
</tr>
<tr>
<td>typep</td>
<td>wild-pathname-p</td>
</tr>
<tr>
<td>unbound-slot</td>
<td>with-accessors</td>
</tr>
<tr>
<td>unbound-slot-instance</td>
<td>with-compilation-unit</td>
</tr>
<tr>
<td>unbound-variable</td>
<td>with-condition-restarts</td>
</tr>
<tr>
<td>undefined-function</td>
<td>with-hash-table-iterator</td>
</tr>
<tr>
<td>unexport</td>
<td>with-input-from-string</td>
</tr>
<tr>
<td>unintern</td>
<td>with-open-file</td>
</tr>
<tr>
<td>union</td>
<td>with-open-stream</td>
</tr>
<tr>
<td>unless</td>
<td>with-output-to-string</td>
</tr>
<tr>
<td>unread-char</td>
<td>with-package-iterator</td>
</tr>
<tr>
<td>unsigned-byte</td>
<td>with-simple-restart</td>
</tr>
<tr>
<td>untrace</td>
<td>with-slots</td>
</tr>
<tr>
<td>unuse-package</td>
<td>with-standard-io-syntax</td>
</tr>
<tr>
<td>unwind-protect</td>
<td>write</td>
</tr>
<tr>
<td>update-instance-for-different-class</td>
<td>write-byte</td>
</tr>
<tr>
<td>update-instance-for-redefined-class</td>
<td>write-char</td>
</tr>
<tr>
<td>upgraded-array-element-type</td>
<td>write-line</td>
</tr>
<tr>
<td>upgraded-complex-part-type</td>
<td>write-string</td>
</tr>
<tr>
<td>upper-case-p</td>
<td>write-to-string</td>
</tr>
<tr>
<td>use-package</td>
<td>y-or-n-p</td>
</tr>
<tr>
<td>use-value</td>
<td>yes-or-no-p</td>
</tr>
<tr>
<td>user-homedir-pathname</td>
<td></td>
</tr>
<tr>
<td>values</td>
<td>zerop</td>
</tr>
</tbody>
</table>

Figure 1–15. Symbols in the COMMON-LISP package (part twelve of twelve).
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 Chapter 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.

Figure 2–1 lists some defined names that are applicable to readtables.

<table>
<thead>
<tr>
<th>Defined Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>readtable</em></td>
<td></td>
</tr>
<tr>
<td>copy-readtable</td>
<td></td>
</tr>
<tr>
<td>get-dispatch-macro-character</td>
<td></td>
</tr>
<tr>
<td>get-macro-character</td>
<td></td>
</tr>
<tr>
<td>make-dispatch-macro-character</td>
<td></td>
</tr>
<tr>
<td>readable</td>
<td></td>
</tr>
<tr>
<td>readable-case</td>
<td></td>
</tr>
<tr>
<td>set-dispatch-macro-character</td>
<td></td>
</tr>
<tr>
<td>set-macro-character</td>
<td></td>
</tr>
<tr>
<td>set-syntax-from-char</td>
<td></td>
</tr>
</tbody>
</table>

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 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. Figure 2–2 lists the variables that influence the behavior of the Lisp reader.

<table>
<thead>
<tr>
<th><em>package</em></th>
<th><em>read-default-float-format</em></th>
<th><em>readtable</em></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>read-base</em></td>
<td><em>read-suppress</em></td>
<td></td>
</tr>
</tbody>
</table>

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:
<table>
<thead>
<tr>
<th>Graphic ID</th>
<th>Glyph</th>
<th>Description</th>
<th>Graphic ID</th>
<th>Glyph</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA01</td>
<td>a</td>
<td>small a</td>
<td>LN01</td>
<td>n</td>
<td>small n</td>
</tr>
<tr>
<td>LA02</td>
<td>A</td>
<td>capital A</td>
<td>LN02</td>
<td>ᵊ</td>
<td>capital N</td>
</tr>
<tr>
<td>LB01</td>
<td>b</td>
<td>small b</td>
<td>LO01</td>
<td>o</td>
<td>small o</td>
</tr>
<tr>
<td>LB02</td>
<td>B</td>
<td>capital B</td>
<td>LO02</td>
<td>₀</td>
<td>capital O</td>
</tr>
<tr>
<td>LC01</td>
<td>c</td>
<td>small c</td>
<td>LP01</td>
<td>p</td>
<td>small p</td>
</tr>
<tr>
<td>LC02</td>
<td>C</td>
<td>capital C</td>
<td>LP02</td>
<td>ᵂ</td>
<td>capital P</td>
</tr>
<tr>
<td>LD01</td>
<td>d</td>
<td>small d</td>
<td>LQ01</td>
<td>ᵥ</td>
<td>small q</td>
</tr>
<tr>
<td>LD02</td>
<td>D</td>
<td>capital D</td>
<td>LQ02</td>
<td>ᵉ</td>
<td>capital Q</td>
</tr>
<tr>
<td>LE01</td>
<td>e</td>
<td>small e</td>
<td>LR01</td>
<td>r</td>
<td>small r</td>
</tr>
<tr>
<td>LE02</td>
<td>E</td>
<td>capital E</td>
<td>LR02</td>
<td>ᵇ</td>
<td>capital R</td>
</tr>
<tr>
<td>LF01</td>
<td>f</td>
<td>small f</td>
<td>LS01</td>
<td>s</td>
<td>small s</td>
</tr>
<tr>
<td>LF02</td>
<td>F</td>
<td>capital F</td>
<td>LS02</td>
<td>ᵧ</td>
<td>capital S</td>
</tr>
<tr>
<td>LG01</td>
<td>g</td>
<td>small g</td>
<td>LT01</td>
<td>t</td>
<td>small t</td>
</tr>
<tr>
<td>LG02</td>
<td>G</td>
<td>capital G</td>
<td>LT02</td>
<td>ᵯ</td>
<td>capital T</td>
</tr>
<tr>
<td>LH01</td>
<td>h</td>
<td>small h</td>
<td>LU01</td>
<td>u</td>
<td>small u</td>
</tr>
<tr>
<td>LH02</td>
<td>H</td>
<td>capital H</td>
<td>LU02</td>
<td>ᵰ</td>
<td>capital U</td>
</tr>
<tr>
<td>LI01</td>
<td>i</td>
<td>small i</td>
<td>LV01</td>
<td>v</td>
<td>small v</td>
</tr>
<tr>
<td>LI02</td>
<td>I</td>
<td>capital I</td>
<td>LV02</td>
<td>ᵱ</td>
<td>capital V</td>
</tr>
<tr>
<td>LJ01</td>
<td>j</td>
<td>small j</td>
<td>LW01</td>
<td>w</td>
<td>small w</td>
</tr>
<tr>
<td>LJ02</td>
<td>J</td>
<td>capital J</td>
<td>LW02</td>
<td>ᵲ</td>
<td>capital W</td>
</tr>
<tr>
<td>LK01</td>
<td>k</td>
<td>small k</td>
<td>LX01</td>
<td>x</td>
<td>small x</td>
</tr>
<tr>
<td>LK02</td>
<td>K</td>
<td>capital K</td>
<td>LX02</td>
<td>ᵳ</td>
<td>capital X</td>
</tr>
<tr>
<td>LL01</td>
<td>l</td>
<td>small l</td>
<td>LY01</td>
<td>y</td>
<td>small y</td>
</tr>
<tr>
<td>LL02</td>
<td>L</td>
<td>capital L</td>
<td>LY02</td>
<td>ᵴ</td>
<td>capital Y</td>
</tr>
<tr>
<td>LM01</td>
<td>m</td>
<td>small m</td>
<td>LZ01</td>
<td>z</td>
<td>small z</td>
</tr>
<tr>
<td>LM02</td>
<td>N</td>
<td>capital M</td>
<td>LZ02</td>
<td>ᵵ</td>
<td>capital Z</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Graphic ID</th>
<th>Glyph</th>
<th>Description</th>
<th>Graphic ID</th>
<th>Glyph</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ND01</td>
<td>1</td>
<td>digit 1</td>
<td>ND06</td>
<td>6</td>
<td>digit 6</td>
</tr>
<tr>
<td>ND02</td>
<td>2</td>
<td>digit 2</td>
<td>ND07</td>
<td>7</td>
<td>digit 7</td>
</tr>
<tr>
<td>ND03</td>
<td>3</td>
<td>digit 3</td>
<td>ND08</td>
<td>8</td>
<td>digit 8</td>
</tr>
<tr>
<td>ND04</td>
<td>4</td>
<td>digit 4</td>
<td>ND09</td>
<td>9</td>
<td>digit 9</td>
</tr>
<tr>
<td>ND05</td>
<td>5</td>
<td>digit 5</td>
<td>ND10</td>
<td>0</td>
<td>digit 0</td>
</tr>
</tbody>
</table>

Figure 2–4. Standard Character Subrepertoire (Part 2 of 3: Numeric Characters)
Graphic ID   Glyph   Description
---          ---       ---
SP02        !        exclamation mark
SP03        $        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.

<table>
<thead>
<tr>
<th>character</th>
<th>syntax type</th>
<th>character</th>
<th>syntax type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backspace</td>
<td>constituent</td>
<td>0–9</td>
<td>constituent</td>
</tr>
<tr>
<td>Tab</td>
<td>whitespace₂</td>
<td>:</td>
<td>constituent</td>
</tr>
<tr>
<td>Newline</td>
<td>whitespace₂</td>
<td>;</td>
<td>terminating macro char</td>
</tr>
<tr>
<td>Linefeed</td>
<td>whitespace₂</td>
<td>&lt;</td>
<td>constituent</td>
</tr>
<tr>
<td>Page</td>
<td>whitespace₂</td>
<td>=</td>
<td>constituent</td>
</tr>
<tr>
<td>Return</td>
<td>whitespace₂</td>
<td>&gt;</td>
<td>constituent</td>
</tr>
<tr>
<td>Space</td>
<td>whitespace₂</td>
<td>?</td>
<td>constituent*</td>
</tr>
<tr>
<td>!</td>
<td>constituent*</td>
<td>@</td>
<td>constituent</td>
</tr>
<tr>
<td>*</td>
<td>terminating macro char</td>
<td>$</td>
<td>constituent*</td>
</tr>
<tr>
<td>#</td>
<td>non-terminating macro char</td>
<td>%</td>
<td>single escape</td>
</tr>
<tr>
<td>$</td>
<td>constituent</td>
<td>\</td>
<td>constituent*</td>
</tr>
<tr>
<td>%</td>
<td>constituent</td>
<td>]</td>
<td>constituent</td>
</tr>
<tr>
<td>&amp;</td>
<td>constituent</td>
<td>^</td>
<td>constituent</td>
</tr>
<tr>
<td>,</td>
<td>terminating macro char</td>
<td>(</td>
<td>terminating macro char</td>
</tr>
<tr>
<td>(</td>
<td>terminating macro char</td>
<td>)</td>
<td>constituent</td>
</tr>
<tr>
<td>)</td>
<td>terminating macro char</td>
<td>*</td>
<td>constituent</td>
</tr>
<tr>
<td>+</td>
<td>constituent</td>
<td>+</td>
<td>multiple escape</td>
</tr>
<tr>
<td>.</td>
<td>terminating macro char</td>
<td>.</td>
<td>constituent*</td>
</tr>
<tr>
<td>-</td>
<td>constituent</td>
<td>-</td>
<td>constituent</td>
</tr>
<tr>
<td>.</td>
<td>constituent</td>
<td>/</td>
<td>Rubout constituent</td>
</tr>
<tr>
<td>/</td>
<td>constituent</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>character</th>
<th>syntax type</th>
<th>character</th>
<th>syntax type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backspace</td>
<td>constituent</td>
<td>0–9</td>
<td>constituent</td>
</tr>
<tr>
<td>Tab</td>
<td>whitespace₂</td>
<td>:</td>
<td>constituent</td>
</tr>
<tr>
<td>Newline</td>
<td>whitespace₂</td>
<td>;</td>
<td>terminating macro char</td>
</tr>
<tr>
<td>Linefeed</td>
<td>whitespace₂</td>
<td>&lt;</td>
<td>constituent</td>
</tr>
<tr>
<td>Page</td>
<td>whitespace₂</td>
<td>=</td>
<td>constituent</td>
</tr>
<tr>
<td>Return</td>
<td>whitespace₂</td>
<td>&gt;</td>
<td>constituent</td>
</tr>
<tr>
<td>Space</td>
<td>whitespace₂</td>
<td>?</td>
<td>constituent*</td>
</tr>
<tr>
<td>!</td>
<td>constituent*</td>
<td>@</td>
<td>constituent</td>
</tr>
<tr>
<td>*</td>
<td>terminating macro char</td>
<td>$</td>
<td>constituent*</td>
</tr>
<tr>
<td>#</td>
<td>non-terminating macro char</td>
<td>%</td>
<td>single escape</td>
</tr>
<tr>
<td>$</td>
<td>constituent</td>
<td>\</td>
<td>constituent*</td>
</tr>
<tr>
<td>%</td>
<td>constituent</td>
<td>]</td>
<td>constituent</td>
</tr>
<tr>
<td>&amp;</td>
<td>constituent</td>
<td>^</td>
<td>constituent</td>
</tr>
<tr>
<td>,</td>
<td>terminating macro char</td>
<td>(</td>
<td>terminating macro char</td>
</tr>
<tr>
<td>(</td>
<td>terminating macro char</td>
<td>)</td>
<td>constituent</td>
</tr>
<tr>
<td>)</td>
<td>terminating macro char</td>
<td>*</td>
<td>constituent</td>
</tr>
<tr>
<td>+</td>
<td>constituent</td>
<td>+</td>
<td>multiple escape</td>
</tr>
<tr>
<td>.</td>
<td>terminating macro char</td>
<td>.</td>
<td>constituent*</td>
</tr>
<tr>
<td>-</td>
<td>constituent</td>
<td>-</td>
<td>constituent</td>
</tr>
<tr>
<td>.</td>
<td>constituent</td>
<td>/</td>
<td>Rubout constituent</td>
</tr>
<tr>
<td>/</td>
<td>constituent</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 alphabetic2 characters, but are not used in the names of any standard Common Lisp defined names.

Whitespace2 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 alphabetic2, 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 alphabetic2. Any character quoted by a single escape is treated as an alphabetic2 constituent, regardless of its normal syntax.
<table>
<thead>
<tr>
<th>constituent character</th>
<th>traits</th>
<th>constituent character</th>
<th>traits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backspace</td>
<td>invalid</td>
<td>{</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
<tr>
<td>Tab</td>
<td>invalid&lt;sup&gt;*&lt;/sup&gt;</td>
<td>}</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
<tr>
<td>Newline</td>
<td>invalid&lt;sup&gt;*&lt;/sup&gt;</td>
<td>+</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;, plus sign</td>
</tr>
<tr>
<td>Linefeed</td>
<td>invalid&lt;sup&gt;*&lt;/sup&gt;</td>
<td>-</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;, minus sign</td>
</tr>
<tr>
<td>Page</td>
<td>invalid&lt;sup&gt;*&lt;/sup&gt;</td>
<td>.</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;, dot, decimal point</td>
</tr>
<tr>
<td>Return</td>
<td>invalid&lt;sup&gt;*&lt;/sup&gt;</td>
<td>/</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;, ratio marker</td>
</tr>
<tr>
<td>Space</td>
<td>invalid</td>
<td>A, a</td>
<td>alphadigit</td>
</tr>
<tr>
<td>!</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>B, b</td>
<td>alphadigit</td>
</tr>
<tr>
<td>&quot;</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>C, c</td>
<td>alphadigit</td>
</tr>
<tr>
<td>#</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>D, d</td>
<td>alphadigit, double-float exponent marker</td>
</tr>
<tr>
<td>$</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>E, e</td>
<td>alphadigit, float exponent marker</td>
</tr>
<tr>
<td>%</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>F, f</td>
<td>alphadigit, single-float exponent marker</td>
</tr>
<tr>
<td>&amp;</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>G, g</td>
<td>alphadigit</td>
</tr>
<tr>
<td>'</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>H, h</td>
<td>alphadigit</td>
</tr>
<tr>
<td>(</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>I, i</td>
<td>alphadigit</td>
</tr>
<tr>
<td>)</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>J, j</td>
<td>alphadigit</td>
</tr>
<tr>
<td>*</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>K, k</td>
<td>alphadigit</td>
</tr>
<tr>
<td>,</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>L, l</td>
<td>alphadigit, long-float exponent marker</td>
</tr>
<tr>
<td>0-9</td>
<td>alphadigit</td>
<td>M, m</td>
<td>alphadigit</td>
</tr>
<tr>
<td>:</td>
<td>package marker</td>
<td>N, n</td>
<td>alphadigit</td>
</tr>
<tr>
<td>;</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>O, o</td>
<td>alphadigit</td>
</tr>
<tr>
<td>&lt;</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>P, p</td>
<td>alphadigit</td>
</tr>
<tr>
<td>=</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>Q, q</td>
<td>alphadigit</td>
</tr>
<tr>
<td>&gt;</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>R, r</td>
<td>alphadigit</td>
</tr>
<tr>
<td>?</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>S, s</td>
<td>alphadigit, short-float exponent marker</td>
</tr>
<tr>
<td>@</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>T, t</td>
<td>alphadigit</td>
</tr>
<tr>
<td>[</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>U, u</td>
<td>alphadigit</td>
</tr>
<tr>
<td>\</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>V, v</td>
<td>alphadigit</td>
</tr>
<tr>
<td>]</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>W, w</td>
<td>alphadigit</td>
</tr>
<tr>
<td>^</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>X, x</td>
<td>alphadigit</td>
</tr>
<tr>
<td>_</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>Y, y</td>
<td>alphadigit</td>
</tr>
<tr>
<td>`</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>Z, z</td>
<td>alphadigit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td>Rubout</td>
</tr>
<tr>
<td>~</td>
<td>alphabetic&lt;sub&gt;2&lt;/sub&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 because the indicated characters are of syntax type whitespace<sub>2</sub>, 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 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 characters, are to be treated as alphabetic 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 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 characters are used to separate tokens.

Space and newline are whitespace 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 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. 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. 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 charac-
ter y is unread (see unread-char), and then step 10 is entered.

- If y is a whitespace 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 character, y is treated as a constituent
  whose only constituent trait is alphabetic. 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. $Z$ 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 Figure 2–9.

```
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 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 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\6E |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.

2.3.1.2 Examples of Potential Numbers

As examples, the tokens in Figure 2–10 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 | 77777q | 1.7J | -3/4+6.7J | 12/25/83 |
| 27\^19 | 3\^4/5 | 6//7 | 3.1.2.6 | ^^-43^- |
| 3.141,592,653,589,793,288,4 | -3.7+2.61-6.17j+19.6k |

Figure 2–10. Examples of reserved tokens

The tokens in Figure 2–11 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 Figure 2–12 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^c |

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 (Sharp-sign 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 Figure 2–13.
For information on how ratios are printed, see Section 22.1.3.1.2 (Printing Ratios).

### 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.

Figure 2–14 contains examples of notations for floats:

| 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
0.0 ; Floating-point zero in default format
0E0 ; As input, this is also floating-point zero in default format.
0e0 ; As input, this is also floating-point zero in default format.
-.0 ; As input, this might be a zero or a minus zero,
     ; depending on whether the implementation supports
     ; a distinct minus zero.
0.0 ; As output, this would appear as 0.0.
0.0s0 ; As input, this is also floating-point zero in default format.
0s0 ; 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 Figure 2–15. For presentational simplicity, these examples assume that the *readtable case* of the *current readtable* is :upcase.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROBBOZ</td>
<td>The symbol whose name is FROBBOZ.</td>
</tr>
<tr>
<td>frobboz</td>
<td>Another way to notate the same symbol.</td>
</tr>
<tr>
<td>fRObBoz</td>
<td>Yet another way to notate it.</td>
</tr>
<tr>
<td>unwind-protect</td>
<td>A symbol with a hyphen in its name.</td>
</tr>
<tr>
<td>+$</td>
<td>The symbol named +$.</td>
</tr>
<tr>
<td>1+</td>
<td>The symbol named 1+.</td>
</tr>
<tr>
<td>+1</td>
<td>This is the integer 1, not a symbol.</td>
</tr>
<tr>
<td>pascal_style</td>
<td>This symbol has an underscore in its name.</td>
</tr>
<tr>
<td>file.rel.43</td>
<td>This symbol has periods in its name.</td>
</tr>
<tr>
<td>(</td>
<td>The symbol whose name is (.</td>
</tr>
<tr>
<td>+1</td>
<td>The symbol whose name is +1.</td>
</tr>
<tr>
<td>+\1</td>
<td>Also the symbol whose name is +1.</td>
</tr>
<tr>
<td>\frobboz</td>
<td>The symbol whose name is fROBBOZ.</td>
</tr>
<tr>
<td>3.14159265\s0</td>
<td>The symbol whose name is 3.14159265s0.</td>
</tr>
<tr>
<td>3.14159265\S0</td>
<td>A different symbol, whose name is 3.14159265S0.</td>
</tr>
<tr>
<td>3.14159265s0</td>
<td>A possible short float approximation to (\pi).</td>
</tr>
</tbody>
</table>

Figure 2–15. Examples of the printed representation of symbols (Part 1 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 Figure 2–17.
nnnnn  a number
xxxxx  a symbol in the current package
:xxxxx  a symbol in the key word 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:aaaaa:aaaaa  undefined

<table>
<thead>
<tr>
<th>Figure 2–17. Valid patterns for tokens</th>
</tr>
</thead>
</table>

Note that nnnnn has number syntax, neither xxxxx nor ppppp has number syntax, and aaaaaa 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 key word 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 key word 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 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 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: \( '\langle\text{exp}\rangle \)

A single-quote introduces an expression to be “quoted.” Single-quote followed by an expression \( \text{exp} \) is treated by the Lisp reader as an abbreviation for and is parsed identically to the expression \( \text{(quote exp)} \). See the special operator quote.

2.4.3.1 Examples of Single-Quote

\[ '\text{foo} \rightarrow \text{FOO} \]
\[ '\text{foo} \rightarrow \text{(QUOTE FOO)} \]
\[ \text{(car 'foo)} \rightarrow \text{QUOTE} \]

2.4.4 Semicolon

Syntax: \( ;\langle\text{text}\rangle \)

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

\[ (+ 3 \text{; three} 4) \rightarrow 7 \]

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

```lisp
;;; 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.
  (t (+ (fib (- n 1)) ;The traditional formula
    (fib (- n 2)))))) ; is fib[n-1]+fib[n-2].
```

Syntax 2–25
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 Figure 2–18.

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Foo&quot;</td>
<td>A string with three characters in it</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>An empty string</td>
</tr>
<tr>
<td>&quot;&quot;APL\360?&quot; he cried.&quot;</td>
<td>A string with twenty characters</td>
</tr>
<tr>
<td>&quot;</td>
<td>x</td>
</tr>
</tbody>
</table>

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 ,(cdr 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 . ,form)` 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)` 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 \( F_2 \) 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

\[(\text{',}(\text{,}(\text{a} \text{ b}) \ ,c \ ,\text{0d})\text{,})\]

will be interpreted as if it were

\[(\text{append} \ (\text{list} \ (\text{append} \ (\text{list} \ a) \ (\text{list} \ 'b) \ '\text{nil})) \ (\text{list} \ c) \ d \ '\text{nil})\]

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

\[
\begin{align*}
(\text{append} \ (\text{list} \ (\text{append} \ (\text{list} \ a) \ (\text{list} \ 'b))) \ (\text{list} \ c) \ d) \\
(\text{append} \ (\text{list} \ (\text{append} \ (\text{list} \ a) \ '(\text{b}))) \ (\text{list} \ c) \ d) \\
(\text{list*} \ (\text{cons} \ a \ '(\text{b}))) \ c \ d) \\
(\text{append} \ (\text{list} \ (\text{cons} \ a \ '(\text{b}))) \ (\text{list} \ c) \ d) \\
(\text{list*} \ (\text{cons} \ a \ '(\text{b}))) \ c \ (\text{copy-list} \ d))
\end{align*}
\]

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 \((\text{pprint} \ '(\text{a} ,\text{b}))\) 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 Figure 2–19.
<table>
<thead>
<tr>
<th>dispatch char</th>
<th>purpose</th>
<th>dispatch char</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backspace</td>
<td>signals error</td>
<td>{</td>
<td>undefined*</td>
</tr>
<tr>
<td>Tab</td>
<td>signals error</td>
<td>}</td>
<td>undefined*</td>
</tr>
<tr>
<td>Newline</td>
<td>signals error</td>
<td>+</td>
<td>read-time conditional</td>
</tr>
<tr>
<td>Linefeed</td>
<td>signals error</td>
<td>-</td>
<td>read-time conditional</td>
</tr>
<tr>
<td>Page</td>
<td>signals error</td>
<td>.</td>
<td>read-time evaluation</td>
</tr>
<tr>
<td>Return</td>
<td>signals error</td>
<td>/</td>
<td>undefined</td>
</tr>
<tr>
<td>Space</td>
<td>signals error</td>
<td>A, a</td>
<td>array</td>
</tr>
<tr>
<td>!</td>
<td>undefined*</td>
<td>B, b</td>
<td>binary rational</td>
</tr>
<tr>
<td>#</td>
<td>reference to = label</td>
<td>D, d</td>
<td>undefined</td>
</tr>
<tr>
<td>$</td>
<td>undefined</td>
<td>E, e</td>
<td>undefined</td>
</tr>
<tr>
<td>%</td>
<td>undefined</td>
<td>F, f</td>
<td>undefined</td>
</tr>
<tr>
<td>&amp;</td>
<td>undefined</td>
<td>G, g</td>
<td>undefined</td>
</tr>
<tr>
<td>'</td>
<td>function abbreviation</td>
<td>H, h</td>
<td>undefined</td>
</tr>
<tr>
<td>(</td>
<td>simple vector</td>
<td>I, i</td>
<td>undefined</td>
</tr>
<tr>
<td>)</td>
<td>signals error</td>
<td>J, j</td>
<td>undefined</td>
</tr>
<tr>
<td>*</td>
<td>bit vector</td>
<td>K, k</td>
<td>undefined</td>
</tr>
<tr>
<td>,</td>
<td>undefined</td>
<td>L, l</td>
<td>undefined</td>
</tr>
<tr>
<td>:</td>
<td>uninterned symbol</td>
<td>M, m</td>
<td>undefined</td>
</tr>
<tr>
<td>;</td>
<td>undefined</td>
<td>N, n</td>
<td>undefined</td>
</tr>
<tr>
<td>&lt;</td>
<td>signals error</td>
<td>O, o</td>
<td>octal rational</td>
</tr>
<tr>
<td>=</td>
<td>labels following object</td>
<td>P, p</td>
<td>pathname</td>
</tr>
<tr>
<td>&gt;</td>
<td>undefined</td>
<td>Q, q</td>
<td>undefined</td>
</tr>
<tr>
<td>?</td>
<td>undefined*</td>
<td>R, r</td>
<td>radix-n rational</td>
</tr>
<tr>
<td>@</td>
<td>undefined</td>
<td>S, s</td>
<td>structure</td>
</tr>
<tr>
<td>[</td>
<td>undefined*</td>
<td>T, t</td>
<td>undefined</td>
</tr>
<tr>
<td>\</td>
<td>character object</td>
<td>U, u</td>
<td>undefined</td>
</tr>
<tr>
<td>]</td>
<td>undefined*</td>
<td>V, v</td>
<td>undefined</td>
</tr>
<tr>
<td>^</td>
<td>undefined</td>
<td>W, w</td>
<td>undefined</td>
</tr>
<tr>
<td>.</td>
<td>undefined</td>
<td>X, x</td>
<td>hexadecimal rational</td>
</tr>
<tr>
<td></td>
<td></td>
<td>undefined</td>
<td>Y, y</td>
</tr>
<tr>
<td>balanced comment</td>
<td></td>
<td>Z, z</td>
<td>undefined</td>
</tr>
<tr>
<td>-</td>
<td>undefined</td>
<td>Rubout</td>
<td>undefined</td>
</tr>
</tbody>
</table>

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:** \(#\langle\langle x\rangle\rangle\)

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,

\[(apply \#'+ l) \equiv (apply (function +) l)\]

### 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,

\[#(a b c c c c)\]
\[#6(a b c c c c)\]
\[#6(a b c)\]
\[#6(a b c)\]
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: #\langle\langle symbol-name \rangle\rangle

#: 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

#Xrational 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 ;105_{16}

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

#nr

#radixrational 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.

Figure 2–20 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 |
| #xDb5 | ;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 #rnR 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.

Figure 2–21 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

#nA

#nAobject 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:

\[
\begin{array}{ccc}
0 & 1 & 5 \\
\text{foo} & 2 & \text{(hot dog)} \\
\end{array}
\]

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

\[
\begin{array}{c}
(0 1 5) \text{(foo 2 (hot dog))} \\
\end{array}
\]

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

\[
\begin{array}{c}
((0 1 5) \text{(foo 2 (hot dog))}) \\
\end{array}
\]

#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:

\[
\begin{align*}
\text{(setq } x \text{ (list 'p 'q))} \\
\text{(setq } y \text{ (list (list 'a 'b) x 'foo x)})) \\
\text{(rplacd (last } y \text{) (cdr } y))
\end{align*}
\]

could be represented in this way:

\[
\begin{align*}
\text{((a b) . #1=(#2=(p q) foo #2# . #1#))}
\end{align*}
\]

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

\[
\begin{align*}
\text{((a b) (p q) foo (p q) (p q) foo (p q) (p q) ...)}
\end{align*}
\]

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; 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 \( \equiv \#+(\text{not test})\) 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:

```lisp
;;;; 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).
(defvar add3 (n) #(format t "Adding 3 to " 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.
(defvar mention-fun-fact-1a ()
 (format t "CL uses ; and #|...|# in comments."))
> MENTION-FUN-FACT-1A
(defvar 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.
(defvar mention-fun-fact-2a ()
 (format t "Don't use |\# unmatched or you'll get in trouble!"))
> MENTION-FUN-FACT-2A
(defvar 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 notat-
ing 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 encoun-
tering #<. 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 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 “#”).
Programming Language—Common Lisp

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 and to write 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 bound3 by using let. Figure 3–1 lists some defined names that are applicable to assigning, binding, and defining variables.

<table>
<thead>
<tr>
<th>boundp</th>
<th>let</th>
<th>progv</th>
</tr>
</thead>
<tbody>
<tr>
<td>defconstant</td>
<td>let*</td>
<td>psetq</td>
</tr>
<tr>
<td>defparameter</td>
<td>makunbound</td>
<td>set</td>
</tr>
<tr>
<td>defvar</td>
<td>multiple-value-bind</td>
<td>setq</td>
</tr>
<tr>
<td>lambda</td>
<td>multiple-value-setq</td>
<td>symbol-value</td>
</tr>
</tbody>
</table>

Figure 3–1. Some Defined Names Applicable to Variables

The following is a description of each kind of variable.

3.1.2.1.1 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 shadowed2 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.

3.1.2.1.2 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 shadowed2 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.

3.1.2.1.4 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. Figure 3–2 lists all of the Common Lisp symbols that have definitions as special operators.

<table>
<thead>
<tr>
<th>block</th>
<th>let*</th>
<th>return-from</th>
</tr>
</thead>
<tbody>
<tr>
<td>catch</td>
<td>load-time-value</td>
<td>setq</td>
</tr>
<tr>
<td>eval-when</td>
<td>locally</td>
<td>symbol-macrolet</td>
</tr>
<tr>
<td>flet</td>
<td>macrolet</td>
<td>tagbody</td>
</tr>
<tr>
<td>function</td>
<td>multiple-value-call</td>
<td>the</td>
</tr>
<tr>
<td>go</td>
<td>multiple-value-prog1</td>
<td>throw</td>
</tr>
<tr>
<td>if</td>
<td>prog</td>
<td>unwind-protect</td>
</tr>
<tr>
<td>labels</td>
<td>progv</td>
<td></td>
</tr>
<tr>
<td>let</td>
<td>quote</td>
<td></td>
</tr>
</tbody>
</table>

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.

Figure 3–3 lists some defined names that are applicable to macros.

<table>
<thead>
<tr>
<th><em>macroexpand-hook</em></th>
<th>macro-function</th>
<th>macroexpand-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>defmacro</td>
<td>macroexpand</td>
<td>macrolet</td>
</tr>
</tbody>
</table>

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.

```lisp
(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`.

Figure 3–4 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))
)
```

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)
```

3–10 Programming Language—Common Lisp
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

\[
\begin{align*}
\text{(let ((funs '()))} \\
\text{(dotimes (j 10)} \\
\text{ \ (push (function (lambda (z) (+ 5 z)))} \\
\text{ \ funs))} \\
\end{align*}
\]

where the closures may be identical.

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

\[
\text{(mapcar (function (lambda (x) (+ x 2))) y)}
\]

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:

\[
\text{(defun test (x z)} \\
\text{ \ (let ((z (* x 2)))} \\
\text{ \ (print z)} \\
\text{ \ z))}
\]

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:

\[
\text{(defun contorted-example (f g x)} \\
\text{ \ (if (= x 0)} \\
\text{ \ \ (funcall f)} \\
\text{ \ \ \ (block here)} \\
\text{ \ \ \ \ (+ 5 (contorted-example g \ \ \ \ #'(lambda () (return-from here 4)) \ \ \ \ (- x 1))))))
\]

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:
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:

```lisp
(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.

---

3–12 Programming Language—Common Lisp
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:

```lisp
(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

```lisp
(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.

Figure 3–5 lists some operators for receiving multiple values. These operators can be used to specify one or more forms to evaluate and where to put the values returned by those forms.

<table>
<thead>
<tr>
<th>multiple-value-bind</th>
<th>multiple-value-prog1</th>
<th>return-from</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiple-value-call</td>
<td>multiple-value-setq</td>
<td>throw</td>
</tr>
<tr>
<td>multiple-value-list</td>
<td>return</td>
<td></td>
</tr>
</tbody>
</table>

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

The function `values` can produce multiple values. `(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`.

---

Evaluation and Compilation  3–13
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 term **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* 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., `boundp`) 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`.

Evaluation and Compilation 3–17
### 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 Figure 3–7.
Figure 3–7. EVAL-WHEN processing

<table>
<thead>
<tr>
<th>CT</th>
<th>LT</th>
<th>E</th>
<th>Mode</th>
<th>Action</th>
<th>New Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>Process</td>
<td>compile-time-too</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>CTT</td>
<td>Process</td>
<td>compile-time-too</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>NCT</td>
<td>Process</td>
<td>not-compile-time</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>—</td>
<td>Process</td>
<td>not-compile-time</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>—</td>
<td>—</td>
<td>Evaluate</td>
<td>—</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>CTT</td>
<td>Evaluate</td>
<td>—</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>NCT</td>
<td>Discard</td>
<td>—</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
<td>—</td>
<td>Discard</td>
<td>—</td>
</tr>
</tbody>
</table>

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:

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

Figure 3–8 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.

<table>
<thead>
<tr>
<th>declaim</th>
<th>define-modify-macro</th>
<th>defsetf</th>
</tr>
</thead>
<tbody>
<tr>
<td>defclass</td>
<td>define-setf-expander</td>
<td>defstruct</td>
</tr>
<tr>
<td>defconstant</td>
<td>defmacro</td>
<td>deftype</td>
</tr>
<tr>
<td>define-compiler-macro</td>
<td>defpackage</td>
<td>defvar</td>
</tr>
<tr>
<td>define-condition</td>
<td>defparameter</td>
<td></td>
</tr>
</tbody>
</table>

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 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 of S is similar to the car of C, and the cdr of S is similar to the cdr 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 of S is similar to the corresponding dimension 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 eqness of substructures within a file. Preserving eqness 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 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

Figure 3–9 shows a list of all declaration identifiers defined by this standard.

<table>
<thead>
<tr>
<th>declaration</th>
<th>ignore</th>
<th>special</th>
</tr>
</thead>
<tbody>
<tr>
<td>dynamic-extent</td>
<td>inline</td>
<td>type</td>
</tr>
<tr>
<td>ftype</td>
<td>notinline</td>
<td>optimize</td>
</tr>
<tr>
<td>ignorable</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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. \((\text{type-specifier } \{\text{var}\}^*)\) is taken as shorthand for \((\text{type type-specifier } \{\text{var}\}^*)\).

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 \(F_1\) that applies to a binding for a name \(N\) established by some form \(F_2\) of which \(F_1\) is a subform affects only references to \(N\) within \(F_1\); it does not to apply to other references to \(N\) outside of \(F_1\), nor does it affect the manner in which the binding of \(N\) by \(F_2\) 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:

- \textbf{do, do*: step-forms, end-test-form, and result-forms.}
- \textbf{dolist, dotimes: result-form}
- \textbf{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` 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]
```

Evaluation and Compilation  3–31
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.

<table>
<thead>
<tr>
<th>Context</th>
<th>Kind of Lambda List</th>
</tr>
</thead>
<tbody>
<tr>
<td>defun form</td>
<td>ordinary lambda list</td>
</tr>
<tr>
<td>defmacro form</td>
<td>macro lambda list</td>
</tr>
<tr>
<td>lambda expression</td>
<td>ordinary lambda list</td>
</tr>
<tr>
<td>flet local function definition</td>
<td>ordinary lambda list</td>
</tr>
<tr>
<td>labels local function definition</td>
<td>ordinary lambda list</td>
</tr>
<tr>
<td>handler-case clause specification</td>
<td>ordinary lambda list</td>
</tr>
<tr>
<td>restart-case clause specification</td>
<td>ordinary lambda list</td>
</tr>
<tr>
<td>macrolet local macro definition</td>
<td>macro lambda list</td>
</tr>
<tr>
<td>define-method-combination</td>
<td>ordinary lambda list</td>
</tr>
<tr>
<td>define-method-combination :arguments option</td>
<td>define-method-combination arguments lambda list</td>
</tr>
<tr>
<td>defstruct :constructor option</td>
<td>boa lambda list</td>
</tr>
<tr>
<td>defgeneric form</td>
<td>generic function lambda list</td>
</tr>
<tr>
<td>defgeneric method clause</td>
<td>specialized lambda list</td>
</tr>
<tr>
<td>defmethod form</td>
<td>specialized lambda list</td>
</tr>
<tr>
<td>defsetf form</td>
<td>defsetf lambda list</td>
</tr>
<tr>
<td>define-setf-expander form</td>
<td>macro lambda list</td>
</tr>
<tr>
<td>deftype form</td>
<td>deftype lambda list</td>
</tr>
<tr>
<td>destructuring-bind form</td>
<td>destructuring lambda list</td>
</tr>
<tr>
<td>define-compiler-macro form</td>
<td>macro lambda list</td>
</tr>
<tr>
<td>define-modify-macro form</td>
<td>define-modify-macro lambda list</td>
</tr>
</tbody>
</table>

Figure 3–10. What Kind of Lambda Lists to Use

Figure 3–11 lists some defined names that are applicable to lambda lists.

<table>
<thead>
<tr>
<th>lambda-list-keywords</th>
<th>lambda-parameters-limit</th>
</tr>
</thead>
</table>

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 Figure 3–12 are those which use ordinary lambda lists:
Figure 3–12. Standardized Operators that use Ordinary Lambda Lists

An ordinary lambda list can contain the lambda list keywords shown in Figure 3–13.

<table>
<thead>
<tr>
<th>&amp;allow-other-keys</th>
<th>&amp;key</th>
<th>&amp;rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;aux</td>
<td>&amp;optional</td>
<td></td>
</tr>
</tbody>
</table>

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; 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 argument checking is suppressed in calls to that function.

If the :allow-other-keys argument is true in a call to a function, keyword argument checking is suppressed in that call.

The :allow-other-keys argument is permissible in all situations involving keyword 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 (kkey x &allow-other-keys) x) :x 1 :y 2) → 1

;;; :ALLOW-OTHER-KEYS NIL is always permitted.
((lambda (kkey) t) :allow-other-keys nil) → T

;;; As with other keyword arguments, only the left-most pair
;;; named :ALLOW-OTHER-KEYS has any effect.
((lambda (kkey 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 (kkey 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.

(lambdax y &aux (a (car x)) (b 2) c) (list x y a b c))
≡ (lambdax 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 (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)) 6 3)
→ (6 T 3 T NIL)
((lambda (a &optional (b 2)) (+ a (* b 3))) 4 5)
→ (19)
((lambda (a &optional (b 2)) (+ a (* b 3))) 4 5 6)
→ (19)
((lambda (a &optional (b 2)) (+ a (* b 3))) 4 5 6 3)
→ (19)
((lambda (a &optional (b 2)) (+ a (* b 3))) 4 5 6 3 8)
→ (19)
((lambda (a &optional (b 2)) (+ a (* b 3))) 4 5 6 3 8 9 10 11)
→ (19)

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 8 NIL)
((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 :d 8)
((lambda (a b &key c d) (list a b c d)) :a :b :c :d) → (:a :b :c :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))

3–38 Programming Language—Common Lisp
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:

```lisp
(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:

```lisp
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 Figure 3–14.

<table>
<thead>
<tr>
<th>&amp;allow-other-keys</th>
<th>&amp;optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;key</td>
<td>&amp;rest</td>
</tr>
</tbody>
</table>

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 Figure 3–15 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 Figure 3–16.

<table>
<thead>
<tr>
<th>&amp;allow-other-keys</th>
<th>&amp;key</th>
<th>&amp;optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;aux</td>
<td>&amp;rest</td>
<td></td>
</tr>
</tbody>
</table>

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.

\[
\text{lambda-list::=} (\{ \text{var} \mid (\text{var} \ [\text{specializer}]) \}\*) \\
(\&\text{optional} \{ \text{var} \mid (\text{var} \ [\text{init-form} \ [\text{supplied-p-parameter}]])) \}\*) \\
(\&\text{rest} \text{var}) \\
(\&\text{key} \{ \text{var} \mid (\{ \text{var} \mid (\text{keyword-name} \ \text{var}) \} \ [\text{init-form} \ [\text{supplied-p-parameter}]])) \}\*) \ [\&\text{allow-other-keys}] \\
(\&\text{aux} \{ \text{var} \mid (\text{var} \ [\text{init-form}])\}*)
\]

### 3.4.4 Macro Lambda Lists

A macro lambda list is used in describing macros defined by the operators in Figure 3–17.

<table>
<thead>
<tr>
<th>define-compiler-macro</th>
<th>defmacro</th>
<th>macrolet</th>
</tr>
</thead>
<tbody>
<tr>
<td>define-setf-expander</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

\[
\text{reqvars::=} (\text{var} \mid \downarrow \text{pattern})* \\
\text{optvars::=} (\&\text{optional} \{ \text{var} \mid (\{ \text{var} \mid \downarrow \text{pattern} \} \ [\text{init-form} \ [\text{supplied-p-parameter}]])) \}\*) \\
\text{restvar::=} (\&\text{rest} \mid \&\text{body} \{ \text{var} \mid \downarrow \text{pattern} \}) \\
\text{keyvars::=} (\&\text{key} \{ \text{var} \mid (\{ \text{var} \mid (\text{keyword-name} \ \text{var} \mid \downarrow \text{pattern})) \} \ [\text{init-form} \ [\text{supplied-p-parameter}]]))\* \\
(\&\text{allow-other-keys}) \\
\text{auxvars::=} (\&\text{aux} \{ \text{var} \mid (\text{var} \ [\text{init-form}])\}*) \\
\text{envvar::=} (\&\text{environment} \ \text{var}) \\
\text{wholevar::=} (\&\text{whole} \ \text{var}) \\
\text{lambda-list::=} (\downarrow \text{wholevar} \mid \text{envvar} \mid \text{reqvars} \mid \text{envvar} \mid \text{optvars} \mid \text{envvar} \\
(\&\text{restvar} \mid \text{envvar} \mid \text{keyvars} \mid \text{envvar} \mid \text{auxvars} \mid \text{envvar} \mid \text{wholevar} \mid \text{envvar} \mid \text{reqvars} \mid \text{envvar} \mid \text{optvars} \mid \text{envvar} \mid \text{var})
\]
A macro lambda list can contain the lambda list keywords shown in Figure 3–18.

<table>
<thead>
<tr>
<th>&amp;allow-other-keys</th>
<th>&amp;environment</th>
<th>&amp;rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;aux</td>
<td>&amp;key</td>
<td>&amp;whole</td>
</tr>
<tr>
<td>&amp;body</td>
<td>&amp;optional</td>
<td></td>
</tr>
</tbody>
</table>

**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 lambdas nested within a macro lambda list have the syntax of destructuring lambda lists.

A destructuring lambda list has the following syntax:

\[
\begin{align*}
\text{reqvars} & := \{ \text{var} \mid \lambda\text{lambda-list} \}^* \\
\text{optvars} & := [\&optional \{ \text{var} \mid (\{\text{var} \mid \lambda\text{lambda-list} \} \ [\text{init-form} \ [\text{supplied-p-parameter}])])^*] \\
\text{restvar} & := [\{\&rest \text{ var} \mid \&body \{ \text{var} \mid \lambda\text{lambda-list} \}] \\
\text{keyvars} & := [\&key \{ \text{var} \mid (\{\text{var} \mid (\text{keyword-name} \{ \text{var} \mid \lambda\text{lambda-list} \}) \ [\text{init-form} \ [\text{supplied-p-parameter}])]])^* \\
&\&allow-other-keys] \\
\text{auxvars} & := [\&aux \{ \text{var} \mid (\text{var} \ [\text{init-form}])^*] \\
\text{envvar} & := [\&environment \text{ var}] \\
\text{wholevar} & := [\&whole \text{ var}] \\
\text{lambda-list} & := (\{\&wholevar \mid \text{reqvars} \mid \text{optvars} \mid \text{restvar} \mid \text{keyvars} \mid \text{auxvars}) \mid \\
&\{\&wholevar \mid \text{reqvars} \mid \text{optvars} \mid \text{var})
\end{align*}
\]
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 “by order of argument” 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:

(defun my-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 IMPLMENTATION-DEPENDENT F EFF)
(create-foo 10 'bee 'see :d 'dee) → #S(FOO A 10 B BEE C SEE D DEE E IMPLMENTATION-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, 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 Figure 3–19.

<table>
<thead>
<tr>
<th>&amp;allow-other-keys</th>
<th>&amp;key</th>
<th>&amp;rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;environment</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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`. 

Evaluation and Compilation   3–47
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 Figure 3–20.

| &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 Figure 3–21.

| &allow-other-keys | &optional | &rest |
| &aux | &key | &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 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` or `cdr` of the `cons`, or performing a destructive operation on an object which is either the `car` or the `cdr` 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 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)
→ (2 1 4 3 7 6 FIVE)
→ (2)

(prog foo ((a (list 1 2 3 4 5 6 8 9 10)))
  (sort a #'(lambda (x y) (if (zerop (random 5)) (return-from foo a) (> x y)))))
→ (1 2 3 4 5 6 8 9 10)
→ (3 4 5 6 2 7 8 9 10 1)
→ (1 2 4 3)
```
### lambda

**Symbol**

#### Syntax:

\[
\text{lambda } \text{lambda-list} \ [ \{\text{declaration}\}^{*} \mid \text{documentation}\} \ \{\text{form}\}^{*}
\]

#### Arguments:

- **lambda-list**: an ordinary lambda list.
- **declaration**: a `declare` expression; not evaluated.
- **documentation**: a `string`; not evaluated.
- **form**: a `form`.

#### Description:

A *lambda expression* is a *list* that 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*.

*Documentation* is attached to the denoted *function* (if any is actually created) as a *documentation string*.

#### See Also:

- `function`, `documentation`, Section 3.1.3 (Lambda Expressions), Section 3.1.2.1.2.4 (Lambda Forms), Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

#### Notes:

The *lambda form*

\[
((\text{lambda } \text{lambda-list} \ . \ \text{body}) \ . \ \text{arguments})
\]

is semantically equivalent to the *function form*

\[
(\text{funcall } \#'(\text{lambda } \text{lambda-list} \ . \ \text{body}) \ . \ \text{arguments})
\]

### lambda

**Macro**

#### Syntax:

\[
\text{lambda } \text{lambda-list} \ [ \{\text{declaration}\}^{*} \mid \text{documentation}\} \ \{\text{form}\}^{*} \rightarrow \text{function}
\]

#### Arguments and Values:

- **lambda-list**: an ordinary lambda list.

---

3–56  Programming Language—Common Lisp
**declaration**—a **declare** expression; not evaluated.

**documentation**—a string; not evaluated.

**form**—a form.

**function**—a function.

**Description:**

Provides a shorthand notation for a **function** special form involving a **lambda expression** such that:

\[
\text{(lambda}\ \text{lambda-list}\ [\{\text{declaration}\}^* \mid \text{documentation}]\ \{\text{form}\}^*)
\equiv\ (\text{function}\ (\text{lambda}\ \text{lambda-list}\ [\{\text{declaration}\}^* \mid \text{documentation}]\ \{\text{form}\}^*))
\equiv\ #'(\text{lambda}\ \text{lambda-list}\ [\{\text{declaration}\}^* \mid \text{documentation}]\ \{\text{form}\}^*)
\]

**Examples:**

\[(\text{funcall}\ (\text{lambda}\ (x)\ (+\ x\ 3))\ 4)\ \to\ 7\]

**See Also:**

lambda (symbol)

**Notes:**

This macro could be implemented by:

\[
\text{(defmacro}\ \text{lambda}\ (&\text{whole}\ \text{form}\ \&\text{optional}\ \text{definition}\ \&\text{rest}\ \text{bvl-decls-and-body})
\text{\quad}(\text{declare}\ (\text{ignore}\ \text{bvl-decls-and-body}))
\text{\quad}'#',\text{form})
\]

---

**compile**

**Function**

**Syntax:**

\[\text{compile}\ \text{name}\ \&\text{optional}\ \text{definition}\ \to\ \text{function},\ \text{warnings-p},\ \text{failure-p}\]

**Arguments and Values:**

\[\text{name}\—\text{a function name},\ \text{or}\ \text{nil}.

\text{definition}\—\text{a lambda expression} \text{or} \text{a function}. The default is the function definition of \text{name} if it names a \text{function}, or the \text{macro function} of \text{name} if it names a \text{macro}. The consequences are undefined if no \text{definition} is supplied when the \text{name} is \text{nil}.

\text{function}\—\text{the function-name}, \text{or} \text{a compiled function}.

---

Evaluation and Compilation  3–57
compile

warnings-p—a generalized boolean.

failure-p—a generalized boolean.

Description:
Compiles an interpreted function.

compile produces a compiled function from definition. If the definition is a lambda expression, it is coerced to a function. If the definition is already a compiled function, compile either produces that function itself (i.e., is an identity operation) or an equivalent function.

If the name is nil, the resulting compiled function is returned directly as the primary value. If a non-nil name is given, then the resulting compiled function replaces the existing function definition of name and the name is returned as the primary value; if name is a symbol that names a macro, its macro function is updated and the name is returned as the primary value.

Literal objects appearing in code processed by the compile function are neither copied nor coalesced. The code resulting from the execution of compile references objects that are eql to the corresponding objects in the source code.

compile is permitted, but not required, to establish a handler for conditions of type error. For example, the handler might issue a warning and restart compilation from some implementation-dependent point in order to let the compilation proceed without manual intervention.

The secondary value, warnings-p, is false if no conditions of type error or warning were detected by the compiler, and true otherwise.

The tertiary value, failure-p, is false if no conditions of type error or warning (other than style-warning) were detected by the compiler, and true otherwise.

Examples:

(defun foo () "bar") → FOO
(compiled-function-p #'foo) → implementation-dependent
(compile 'foo) → FOO
(compiled-function-p #'foo) → true
(setf (symbol-function 'foo)
     (compile nil '(lambda () "replaced"))) → #<Compiled-Function>
(foo) → "replaced"

Affected By:
*error-output*, *macroexpand-hook*.

The presence of macro definitions and proclamations.

Exceptional Situations:
The consequences are undefined if the lexical environment surrounding the function to be compiled contains any bindings other than those for macros, symbol macros, or declarations.
For information about errors detected during the compilation process, see Section 3.2.5 (Exceptional Situations in the Compiler).

See Also:
compile-file

### eval

**Function**

**Syntax:**

eval form → {result}*

**Arguments and Values:**

- **form**—a form.
- **results**—the values yielded by the evaluation of form.

**Description:**

Evaluates form in the current dynamic environment and the null lexical environment.

eval is a user interface to the evaluator.

The evaluator expands macro calls as if through the use of macroexpand-1.

Constants appearing in code processed by eval are not copied nor coalesced. The code resulting from the execution of eval references objects that are eql to the corresponding objects in the source code.

**Examples:**

- (setq form '((1+ a) a 999)) → 999
- (eval form) → 1000
- (eval 'form) → (1+ A)
- (let ((a '((this would break if eval used local value))) (eval form)) → 1000

**See Also:**
macroexpand-1, Section 3.1.2 (The Evaluation Model)

**Notes:**

To obtain the current dynamic value of a symbol, use of **symbol-value** is equivalent (and usually preferable) to use of eval.

Note that an eval form involves two levels of evaluation for its argument. First, form is evaluated by the normal argument evaluation mechanism as would occur with any call. The object that
results from this normal argument evaluation becomes the value of the form parameter, and is then evaluated as part of the eval form. For example:

(eval (list 'cdr (car '((quote (a . b)) c)))) → b

The argument form (list 'cdr (car '(((quote (a . b)) c))) is evaluated in the usual way to produce the argument (cdr (quote (a . b))); eval then evaluates its argument, (cdr (quote (a . b))), to produce b. Since a single evaluation already occurs for any argument form in any function form, eval is sometimes said to perform “an extra level of evaluation.”

eval-when

Special Operator

Syntax:

eval-when ((situation)* {form}* → {result}*)

Arguments and Values:

situation—One of the symbols :compile-toplevel, :load-toplevel, :execute, compile, load, or eval.

The use of eval, compile, and load is deprecated.

forms—an implicit progn.

results—the values of the forms if they are executed, or nil if they are not.

Description:

The body of an eval-when form is processed as an implicit progn, but only in the situations listed.

The use of the situations :compile-toplevel (or compile) and :load-toplevel (or load) controls whether and when evaluation occurs when eval-when appears as a top level form in code processed by compile-file. See Section 3.2.3 (File Compilation).

The use of the situation :execute (or eval) controls whether evaluation occurs for other eval-when forms; that is, those that are not top level forms, or those in code processed by eval or compile. If the :execute situation is specified in such a form, then the body forms are processed as an implicit progn; otherwise, the eval-when form returns nil.

eval-when normally appears as a top level form, but it is meaningful for it to appear as a non-top-level form. However, the compile-time side effects described in Section 3.2 (Compilation) only take place when eval-when appears as a top level form.

Examples:

One example of the use of eval-when is that for the compiler to be able to read a file properly when it uses user-defined reader macros, it is necessary to write
(eval-when (:compile-toplevel :load-toplevel :execute)
  (set-macro-character #\$ #'(lambda (stream char)
      (declare (ignore char))
      (list 'dollar (read stream))))) → T

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

;;; The EVAL-WHEN in this case is not at toplevel, so only the :EXECUTE
;;; keyword is considered. At compile time, this has no effect.
;;; At load time (if the LET is at toplevel), or at execution time
;;; (if the LET is embedded in some other form which does not execute
;;; until later) this sets (SYMBOL-FUNCTION 'FOO1) to a function which
;;; returns 1.
(let ((x 1))
  (eval-when (:execute :load-toplevel :compile-toplevel)
    (setf (symbol-function 'foo1) #'(lambda () x))))

;;; If this expression occurs at the toplevel of a file to be compiled,
;;; it has BOTH a compile time AND a load-time effect of setting
;;; (SYMBOL-FUNCTION 'FOO2) to a function which returns 2.
(eval-when (:execute :load-toplevel :compile-toplevel)
  (let ((x 2))
    (eval-when (:execute :load-toplevel :compile-toplevel)
      (setf (symbol-function 'foo2) #'(lambda () x))))))

;;; If this expression occurs at the toplevel of a file to be compiled,
;;; it has BOTH a compile time AND a load-time effect of setting the
;;; function cell of FOO3 to a function which returns 3.
(eval-when (:execute :load-toplevel :compile-toplevel)
  (setf (symbol-function 'foo3) #'(lambda () 3)))))

;;; #4: This always does nothing. It simply returns NIL.
(eval-when (:compile-toplevel)
  (eval-when (:compile-toplevel)
    (print 'foo4)))

;;; If this form occurs at toplevel of a file to be compiled, FOO6 is
;;; printed at compile time. If this form occurs in a non-top-level
;;; position, nothing is printed at compile time. Regardless of context,
;;; nothing is ever printed at load time or execution time.
(eval-when (:compile-toplevel)
  (eval-when (:execute)
    (print 'foo5)))

;;; If this form occurs at toplevel of a file to be compiled, FOO6 is
eval-when

;;; printed at compile time. If this form occurs in a non-top-level
;;; position, nothing is printed at compile time. Regardless of context,
;;; nothing is ever printed at load time or execution time.
(eval-when (:execute :load-toplevel)
  (eval-when (:compile-toplevel)
    (print 'foo6)))

See Also:
compile-file, Section 3.2 (Compilation)

Notes:
The following effects are logical consequences of the definition of eval-when:

- Execution of a single eval-when expression executes the body code at most once.

- Macros intended for use in top level forms should be written so that side-effects are
  done by the forms in the macro expansion. The macro-expander itself should not do the
  side-effects.

For example:
Wrong:
(defmacro foo ()
  (really-foo)
  '(really-foo))

Right:
(defmacro foo ()
  '(eval-when (:compile-toplevel :execute :load-toplevel) (really-foo)))

Adherence to this convention means that such macros behave intuitively when appearing
as non-top-level forms.

- Placing a variable binding around an eval-when reliably captures the binding because the
  compile-time-too mode cannot occur (i.e., introducing a variable binding means that the
  eval-when is not a top level form). For example,

  (let ((x 3))
    (eval-when (:execute :load-toplevel :compile-toplevel) (print x)))

prints 3 at execution (i.e., load) time, and does not print anything at compile time.
This is important so that expansions of defun and defmacro can be done in terms of
eval-when and can correctly capture the lexical environment.

(defun bar (x) (defun foo () (+ x 3)))
might expand into

\[
\text{(defun bar (x)} \\
\quad \text{(progn (eval-when (:compile-toplevel)} \\
\quad \quad \text{(compiler::notice-function-definition 'foo '(x))}) \\
\quad \text{(eval-when (:execute :load-toplevel)} \\
\quad \quad \text{(setf (symbol-function 'foo) #'(lambda () (+ x 3))))})
\]

which would be treated by the above rules the same as

\[
\text{(defun bar (x)} \\
\quad \text{(setf (symbol-function 'foo) #'(lambda () (+ x 3))))}
\]

when the definition of bar is not a top level form.

---

**load-time-value**

*Special Operator*

**Syntax:**

load-time-value form &optional read-only-p → object

**Arguments and Values:**

- **form**—a form; evaluated as described below.
- **read-only-p**—a boolean; not evaluated.
- **object**—the primary value resulting from evaluating form.

**Description:**

load-time-value provides a mechanism for delaying evaluation of form until the expression is in the run-time environment; see Section 3.2 (Compilation).

Read-only-p designates whether the result can be considered a constant object. If t, the result is a read-only quantity that can, if appropriate to the implementation, be copied into read-only space and/or coalesced with similar constant objects from other programs. If nil (the default), the result must be neither copied nor coalesced; it must be considered to be potentially modifiable data.

If a load-time-value expression is processed by compile-file, the compiler performs its normal semantic processing (such as macro expansion and translation into machine code) on form, but arranges for the execution of form to occur at load time in a null lexical environment, with the result of this evaluation then being treated as a literal object at run time. It is guaranteed that the evaluation of form will take place only once when the file is loaded, but the order of evaluation with respect to the evaluation of top level forms in the file is implementation-dependent.
**load-time-value**

If a `load-time-value` expression appears within a function compiled with `compile`, the `form` is evaluated at compile time in a *null lexical environment*. The result of this compile-time evaluation is treated as a *literal object* in the compiled code.

If a `load-time-value` expression is processed by `eval`, `form` is evaluated in a *null lexical environment*, and one value is returned. Implementations that implicitly compile (or partially compile) expressions processed by `eval` might evaluate `form` only once, at the time this compilation is performed.

If the *same list* (`load-time-value form`) is evaluated or compiled more than once, it is *implementation-dependent* whether `form` is evaluated only once or is evaluated more than once. This can happen both when an expression being evaluated or compiled shares substructure, and when the *same form* is processed by `eval` or `compile` multiple times. Since a `load-time-value` expression can be referenced in more than one place and can be evaluated multiple times by `eval`, it is *implementation-dependent* whether each execution returns a fresh *object* or returns the same *object* as some other execution. Users must use caution when destructively modifying the resulting *object*.

If two lists (`load-time-value form`) that are the *same* under `equal` but are not *identical* are evaluated or compiled, their values always come from distinct evaluations of `form`. Their *values* may not be coalesced unless `read-only-p` is `t`.

**Examples:**

```lisp
;;; The function INCR1 always returns the same value, even in different images.
;;; The function INCR2 always returns the same value in a given image,
;;; but the value it returns might vary from image to image.
(defun incr1 (x) (+ x #.(random 17)))
(defun incr2 (x) (+ x (load-time-value (random 17))))

;;; The function FOO1-REF references the nth element of the first of
;;; the *FOO-ARRAYS* that is available at load time. It is permissible for
;;; that array to be modified (e.g., by SET-FOO1-REF); FOO1-REF will see the
;;; updated values.
(defvar *foo-arrays* (list (make-array 7) (make-array 8)))
(defun fool-ref (n) (aref (load-time-value (first *my-arrays*) nil) n))
(defun set-foo1-ref (n val)
  (setf (aref (load-time-value (first *my-arrays*) nil) n) val))

;;; The function BAR1-REF references the nth element of the first of
;;; the *BAR-ARRAYS* that is available at load time. The programmer has
;;; promised that the array will be treated as read-only, so the system
;;; can copy or coalesce the array.
(defvar *bar-arrays* (list (make-array 7) (make-array 8)))
(defun bar1-ref (n) (aref (load-time-value (first *my-arrays*) t) n))
```
;;; This use of LOAD-TIME-VALUE permits the indicated vector to be coalesced
;;; even though NIL was specified, because the object was already read-only
;;; when it was written as a literal vector rather than created by a constructor.
;;; User programs must treat the vector v as read-only.
(defun baz-ref (n)
  (let ((v (load-time-value #(A B C) nil)))
    (values (svref v n) v))

;;; This use of LOAD-TIME-VALUE permits the indicated vector to be coalesced
;;; even though NIL was specified in the outer situation because T was specified
;;; in the inner situation. User programs must treat the vector v as read-only.
(defun baz-ref (n)
  (let ((v (load-time-value (load-time-value (vector 1 2 3) t) nil)))
    (values (svref v n) v)))

See Also:
compile-file, compile, eval, Section 3.2.2.2 (Minimal Compilation), Section 3.2 (Compilation)

Notes:
load-time-value must appear outside of quoted structure in a “for evaluation” position. In situations which would appear to call for use of load-time-value within a quoted structure, the backquote reader macro is probably called for; see Section 2.4.6 (Backquote).

Specifying nil for read-only-p is not a way to force an object to become modifiable if it has already been made read-only. It is only a way to say that, for an object that is modifiable, this operation is not intended to make that object read-only.

---

quote

Special Operator

Syntax:

quote object → object

Arguments and Values:

object—an object; not evaluated.

Description:

The quote special operator just returns object.

The consequences are undefined if literal objects (including quoted objects) are destructively modified.
Examples:

(setq a 1) → 1
(quote (setq a 3)) → (SETQ A 3)
a → 1
'a → A
''a → (QUOTE A)
''''a → (QUOTE (QUOTE A))
(setq a 43) → 43
(list a (cons a 3)) → (43 (43 . 3))
(list (quote a) (quote (cons a 3))) → (A (CONS A 3))
1 → 1
'1 → 1
"foo" → "foo"
'"foo" → "foo"
(car '(a b)) → A
'(car '(a b)) → (CAR (QUOTE (A B)))
#(car '(a b)) → #(CAR (QUOTE (A B)))
'#(car '(a b)) → #(#(CAR (QUOTE (A B))))

See Also:
Section 3.1 (Evaluation), Section 2.4.3 (Single-Quote), Section 3.2.1 (Compiler Terminology)

Notes:
The textual notation 'object is equivalent to (quote object); see Section 3.2.1 (Compiler Terminology).

Some objects, called self-evaluating objects, do not require quotation by quote. However, symbols and lists are used to represent parts of programs, and so would not be usable as constant data in a program without quote. Since quote suppresses the evaluation of these objects, they become data rather than program.

compiler-macro-function

Syntax:

compiler-macro-function name &optional environment → function
(setf (compiler-macro-function name &optional environment) new-function)

Arguments and Values:

name—a function name.
environment—an environment object.
function, new-function—a compiler macro function, or nil.

Description:
Accesses the compiler macro function named name, if any, in the environment.
A value of nil denotes the absence of a compiler macro function named name.

Exceptional Situations:
The consequences are undefined if environment is non-nil in a use of setf of compiler-macro-function.

See Also:
define-compiler-macro, Section 3.2.2.1 (Compiler Macros)

define-compiler-macro

Macro

Syntax:
define-compiler-macro name lambda-list [ {declaration}* | documentation ] {form}* → name

Arguments and Values:
name—a function name.
lambda-list—a macro lambda list.
declaration—a declare expression; not evaluated.
documentation—a string; not evaluated.
form—a form.

Description:
This is the normal mechanism for defining a compiler macro function. Its manner of definition is the same as for defmacro; the only differences are:

- The name can be a function name naming any function or macro.
- The expander function is installed as a compiler macro function for the name, rather than as a macro function.
define-compiler-macro

- The \&whole argument is bound to the form argument that is passed to the compiler macro function. The remaining lambda-list parameters are specified as if this form contained the function name in the car and the actual arguments in the cdr, but if the car of the actual form is the symbol funcall, then the destructuring of the arguments is actually performed using its cddr instead.

- Documentation is attached as a documentation string to name (as kind compiler-macro) and to the compiler macro function.

- Unlike an ordinary macro, a compiler macro can decline to provide an expansion merely by returning a form that is the same as the original (which can be obtained by using \&whole).

Examples:

\[
\begin{align*}
(\text{defun} \text{square} \text{ (}x\text{)} (\text{expt} \text{x} 2)) & \rightarrow \text{SQUARE} \\
(\text{define-compiler-macro} \text{square} (\&whole \text{form} \ \text{arg}) & \rightarrow \text{SQUARE} \\
(\text{if} (\text{atom} \text{arg}) & \rightarrow \text{SQUARE} \\
\quad (\text{expt} ,\text{arg} 2) & \rightarrow \text{SQUARE} \\
\quad (\text{case} (\text{car} \text{arg}) & \rightarrow \text{SQUARE} \\
\quad \quad (\text{square} (\text{if} (= (\text{length} \text{arg}) 2) & \rightarrow \text{SQUARE} \\
\quad \quad \quad (\text{expt} ,(\text{nth} 1 \text{arg}) 4) & \rightarrow \text{SQUARE} \\
\quad \quad \quad \text{form})) & \rightarrow \text{SQUARE} \\
\quad \quad (\text{expt} (\text{if} (= (\text{length} \text{arg}) 3) & \rightarrow \text{SQUARE} \\
\quad \quad \quad (\text{if} (\text{numberp} (\text{nth} 2 \text{arg})))) & \rightarrow \text{SQUARE} \\
\quad \quad \quad \quad (\text{expt} ,(\text{nth} 1 \text{arg} ) ,(* 2 (\text{nth} 2 \text{arg})))) & \rightarrow \text{SQUARE} \\
\quad \quad \quad \quad \text{form}) & \rightarrow \text{SQUARE} \\
\quad \quad \quad (\text{otherwise} '(\text{expt} ,\text{arg} 2)))) & \rightarrow \text{SQUARE} \\
\text{(square} (\text{square} 3)) & \rightarrow 81 \\
\text{(macroexpand} '((\text{square} \text{x})) & \rightarrow (\text{SQUARE} \text{X}), \text{false} \\
\text{(funcall} (\text{compiler-macro-function} '\text{square}) '((\text{square} \text{x}) \text{nil}) & \rightarrow (\text{EXPT} \text{X} 2) \\
\text{(funcall} (\text{compiler-macro-function} '\text{square}) '((\text{square} (\text{square} \text{x})) \text{nil}) & \rightarrow (\text{EXPT} \text{X} 4) \\
\text{(funcall} (\text{compiler-macro-function} '\text{square}) '((\text{funcall} #\text{\'square} \text{x}) \text{nil}) & \rightarrow (\text{EXPT} \text{X} 2) \\
\text{(defun distance-positional} (\text{x1} \text{y1} \text{x2} \text{y2}) & \rightarrow \text{DISTANCE-POSITIONAL} \\
\text{(sqrt} (+ (\text{expt} (- \text{x2} \text{x1}) 2) (\text{expt} (- \text{y2} \text{y1}) 2)))) & \rightarrow \text{DISTANCE-POSITIONAL} \\
\text{(defun distance} (\&key (\text{x1} 0) \text{(y1} 0) (\text{x2} \text{x1}) \text{(y2} \text{y1}) & \rightarrow \text{DISTANCE} \\
\text{(distance-positional} \text{x1} \text{y1} \text{x2} \text{y2}) & \rightarrow \text{DISTANCE} \\
\text{(define-compiler-macro distance} (\&whole \text{form} &rest \text{key-value-pairs}
\end{align*}
\]
define-compiler-macro

&key (x1 0 x1-p)
  (y1 0 y1-p)
  (x2 x1 x2-p)
  (y2 y1 y2-p)
&allow-other-keys
&environment env)
(flet ((key (n) (nth (* n 2) key-value-pairs))
  (arg (n) (nth (1+ (* n 2)) key-value-pairs))
  (simplep (x)
    (let ((expanded-x (macroexpand x env))
        (or (constantp expanded-x env)
            (symbolp expanded-x)))))))
(let ((n (/ (length key-value-pairs) 2)))
  (multiple-value-bind (x1s y1s x2s y2s others)
      (loop for (key) on key-value-pairs by #'cddr
        count (eq key ':x1) into x1s
        count (eq key ':y1) into y1s
        count (eq key ':x2) into x2s
        count (eq key ':y1) into y2s
        count (not (member key '(:x1 :x2 :y1 :y2)))
        into others
        finally (return (values x1s y1s x2s y2s others)))
    (cond ((and (= n 4) (eq (key 0) :x1)
        (eq (key 1) :y1)
        (eq (key 2) :x2)
        (eq (key 3) :y2))
      ,(distance-positional ,x1 ,y1 ,x2 ,y2)))
    ((and (if x1-p (and (= x1s 1) (simplep x1)) t)
        (if y1-p (and (= y1s 1) (simplep y1)) t)
        (if x2-p (and (= x2s 1) (simplep x2)) t)
        (if y2-p (and (= y2s 1) (simplep y2)) t)
        (zerop others))
      ,(distance-positional ,x1 ,y1 ,x2 ,y2))
    ((and (< x1s 2) (< y1s 2) (< x2s 2) (< y2s 2)
        (zerop others))
      (let ((temps (loop repeat n collect (gensym))))
        ,(distance
          ,(loop for i below n
            collect (list (nth i temps) (arg i))))
        (t form)))))))
→ DISTANCE
(dolist (form Evaluation and Compilation 3–69
'((distance :x1 (setq x 7) :x2 (decf x) :y1 (incf x) :y2 (incf x))
 (distance :x1 (setq x 7) :y1 (decf x) :x2 (decf x) :y2 (decf x))
 (distance :x1 (setq x 7) :y1 (incf x) :x1 (incf x))
 (distance :x1 :y1 b1 :x2 a2 :y2 b2)
 (distance :x1 :a1 :x2 :a2 :y1 b1 :y2 b2)
 (print (funcall (compiler-macro-function 'distance) form nil)))

⊿

LET ((#:G6558 (SETQ X 7))
 ⊿
 (#:G6559 (DECF X))
 ⊿
 (#:G6560 (DECF X))
 ⊿
 (#:G6561 (DECF X)))
 ⊿
 (DISTANCE :X1 #:G6558 :X2 #:G6559 :Y1 #:G6560 :Y2 #:G6561))
 ⊿
 (DISTANCE-POSITIONAL (SETQ X 7) (DECF X) (DECF X))
 ⊿
 (LET ((#:G6567 (SETQ X 7))
 ⊿
 (#:G6568 (INCF X)))
 ⊿
 (DISTANCE :X1 #:G6567 :Y1 #:G6568))
 ⊿
 (DISTANCE :X1 (SETQ X 7) :Y1 (INCF X) :X1 (INCF X))
 ⊿
 (DISTANCE-POSITIONAL A1 B1 A2 B2)
 ⊿
 (DISTANCE-POSITIONAL A1 B1 A2 B2)
 ⊿
 → NIL

See Also:
compiler-macro-function, defmacro, documentation, Section 3.4.11 (Syntactic Interaction of
Documentation Strings and Declarations)

Notes:
The consequences of writing a compiler macro definition for a function in the COMMON-LISP package
are undefined; it is quite possible that in some implementations such an attempt would override
an equivalent or equally important definition. In general, it is recommended that a programmer
only write compiler macro definitions for functions he or she personally maintains—writing a
compiler macro definition for a function maintained elsewhere is normally considered a violation
of traditional rules of modularity and data abstraction.

defmacro

Macro

Syntax:
defmacro name lambda-list [ {declaration}* | documentation ] {form}* → name

Arguments and Values:
name—a symbol.

3–70 Programming Language—Common Lisp
defmacro

lambda-list—a macro lambda list.

declaration—a declare expression; not evaluated.

documentation—a string; not evaluated.

form—a form.

Description:
Defines name as a macro by associating a macro function with that name in the global environment. The macro function is defined in the same lexical environment in which the defmacro form appears.

The parameter variables in lambda-list are bound to destructured portions of the macro call.

The expansion function accepts two arguments, a form and an environment. The expansion function returns a form. The body of the expansion function is specified by forms. Forms are executed in order. The value of the last form executed is returned as the expansion of the macro. The body forms of the expansion function (but not the lambda-list) are implicitly enclosed in a block whose name is name.

The lambda-list conforms to the requirements described in Section 3.4.4 (Macro Lambda Lists).

Documentation is attached as a documentation string to name (as kind function) and to the macro function.

defmacro can be used to redefine a macro or to replace a function definition with a macro definition.

Recursive expansion of the form returned must terminate, including the expansion of other macros which are subforms of other forms returned.

The consequences are undefined if the result of fully macroexpanding a form contains any circular list structure except in literal objects.

If a defmacro form appears as a top level form, the compiler must store the macro definition at compile time, so that occurrences of the macro later on in the file can be expanded correctly. Users must ensure that the body of the macro can be evaluated at compile time if it is referenced within the file being compiled.

Examples:

(defmacro mac1 (a b) "Mac1 multiplies and adds" 
  ’(+ a (* b 3))) → MAC1
(mac1 4 5) → 19
(documentation ’mac1 ’function) → "Mac1 multiplies and adds"
(defmacro mac2 (&optional (a 2 b) (c 3 d) &rest x) ’(,a ,b ,c ,d ,x)) → MAC2
(mac2 6) → (6 T 3 NIL NIL)
defmacro

(defmacro mac2 6 3 8) → (6 T 3 T (8))
(defmacro mac3 (&whole r a &optional (b 3) &rest x &key c (d a))
  ''((r ,a ,b ,c ,d ,x))) → MAC3
(mac3 i 6 :d 8 :c 9 :d 10) → ((MAC3 I 6 :D 8 :C 9 :D 10) I 6 9 8 (:D 8 :C 9 :D 10))

The stipulation that an embedded destructuring lambda list is permitted only where ordinary lambda list syntax would permit a parameter name but not a list is made to prevent ambiguity. For example, the following is not valid:

(defmacro loser (x &optional (a b &rest c) &rest z)
  ...)

because ordinary lambda list syntax does permit a list following &optional; the list (a b &rest c) would be interpreted as describing an optional parameter named a whose default value is that of the form b, with a supplied-p parameter named &rest (not valid), and an extraneous symbol c in the list (also not valid). An almost correct way to express this is

(defmacro loser (x &optional ((a b &rest c)) &rest z)
  ...)

The extra set of parentheses removes the ambiguity. However, the definition is now incorrect because a macro call such as (loser (car pool)) would not provide any argument form for the lambda list (a b &rest c), and so the default value against which to match the lambda list would be nil because no explicit default value was specified. The consequences of this are unspecified since the empty list, nil, does not have forms to satisfy the parameters a and b. The fully correct definition would be either

(defmacro loser (x &optional ((a b &rest c) '(nil nil)) &rest z)
  ...)

or

(defmacro loser (x &optional ((&optional a b &rest c)) &rest z)
  ...)

These differ slightly: the first requires that if the macro call specifies a explicitly then it must also specify b explicitly, whereas the second does not have this requirement. For example,

(loser (car pool) ((+ x 1)))

would be a valid call for the second definition but not for the first.

(defmacro dm1a (&whole x) ''(x))
(macroexpand '(dm1a)) → (QUOTE (DM1A))
(macroexpand '(dm1a a)) is an error.

(defmacro dm1b (&whole x a &optional b) ''((x ,a ,b))
(macroexpand '(dm1b q)) is an error.
(macroexpand '(dm1b q)) → (QUOTE ((DM1B Q) Q NIL))
(macroexpand '(dm1b q r)) → (QUOTE ((DM1B Q R) Q R))
(macroexpand '(dm1b q r s)) is an error.

(defmacro dm2a (&whole form a b) ''(form ,form a ,a b ,b))
(macroexpand '(dm2a x y)) → (QUOTE (FORM (DM2A X Y) A X B Y))
(dm2a x y) → (FORM (DM2A X Y) A X B Y)

(defmacro dm2b (&whole form a (&whole b (c . d) &optional (e 5)) &body f &environment env)
  '(,form ,,a ,,b ,,(macroexpand c env) ,,d ,,e ,,f))
; Note that because backquote is involved, implementations may differ
; slightly in the nature (though not the functionality) of the expansion.
(macroexpand '(dm2b x1 (((incf x2) x3 x4)) x5 x6)) → (LIST* '(DM2B X1 (((INCF X2) X3 X4))
  X5 X6)
  X1
  '(((INCF X2) X3 X4)) (SETQ X2 (+ X2 1)) (X3 X4) 5 (X5 X6)),
  T
)(let ((x1 5))
(macrolet ((segundo (x) '(cadr ,x)))
  (dm2b x1 (((segundo x2) x3 x4)) x5 x6)))
→ ((DM2B X1 (((SEGUNDO X2) X3 X4)) X5 X6)
  5 (((SEGUNDO X2) X3 X4)) (CADR X2) (X3 X4) 5 (X5 X6))

See Also:
define-compiler-macro, destructuring-bind, documentation, macroexpand,
*macroexpand-hook*, macrolet, macro-function, Section 3.1 (Evaluation), Section 3.2 (Compilation), Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

---

**macro-function**

**Accessor**

**Syntax:**

macro-function symbol &optional environment → function

(setf (macro-function symbol &optional environment) new-function)

**Arguments and Values:**

symbol—a symbol.

environment—an environment object.
macro-function

function—a macro function or nil.

new-function—a macro function.

Description:
Determines whether symbol has a function definition as a macro in the specified environment.
If so, the macro expansion function, a function of two arguments, is returned. If symbol has no function definition in the lexical environment environment, or its definition is not a macro, macro-function returns nil.

It is possible for both macro-function and special-operator-p to return true of symbol. The macro definition must be available for use by programs that understand only the standard Common Lisp special forms.

Examples:

(defmacro macfun (x) '(macro-function 'macfun)) → MACFUN
(not (macro-function 'macfun)) → false

(macrolet ((foo (&environment env)
  (if (macro-function 'bar env)
    ''yes
    ''no)))
  (list (foo)
    (macrolet ((bar () :beep))
      (foo))))
→ (NO YES)

Affected By:
(setf macro-function), defmacro, and macrolet.

Exceptional Situations:
The consequences are undefined if environment is non-nil in a use of setf of macro-function.

See Also:
defmacro, Section 3.1 (Evaluation)

Notes:
setf can be used with macro-function to install a macro as a symbol’s global function definition:

(setf (macro-function symbol) fn)

The value installed must be a function that accepts two arguments, the entire macro call and
an environment, and computes the expansion for that call. Performing this operation causes
symbol to have only that macro definition as its global function definition; any previous definition,
whether as a macro or as a function, is lost.

**macroexpand, macroexpand-1**

*Function*

**Syntax:**

macroexpand form &optional env → expansion, expanded-p
macroexpand-1 form &optional env → expansion, expanded-p

**Arguments and Values:**

form—a form.
env—an environment object. The default is nil.
expansion—a form.
expanded-p—a generalized boolean.

**Description:**

macroexpand and macroexpand-1 expand macros.

If form is a macro form, then macroexpand-1 expands the macro form call once.

macroexpand repeatedly expands form until it is no longer a macro form. In effect,
macroexpand calls macroexpand-1 repeatedly until the secondary value it returns is nil.

If form is a macro form, then the expansion is a macro expansion and expanded-p is true. Other-
wise, the expansion is the given form and expanded-p is false.

Macro expansion is carried out as follows. Once macroexpand-1 has determined that the form
is a macro form, it obtains an appropriate expansion function for the macro or symbol macro.
The value of *macroexpand-hook* is coerced to a function and then called as a function of three
arguments: the expansion function, the form, and the env. The value returned from this call is
taken to be the expansion of the form.

In addition to macro definitions in the global environment, any local macro definitions established
within env by macrolet or symbol-macrolet are considered. If only form is supplied as an argu-
ment, then the environment is effectively null, and only global macro definitions as established by
defmacro are considered. Macro definitions are shadowed by local function definitions.

**Examples:**

(defmacro alpha (x y) '(beta ,x ,y)) → ALPHA

Evaluation and Compilation  3–75
**macroexpand, macroexpand-1**

(defmacro beta (x y) `(gamma ,x ,y)) → BETA
(defmacro delta (x y) `(gamma ,x ,y)) → EPSILON
(defmacro expand (form &environment env)
  (multiple-value-bind (expansion expanded-p)
      (macroexpand form env)
    `(values ',expansion ',expanded-p))) → EXPAND
(defmacro expand-1 (form &environment env)
  (multiple-value-bind (expansion expanded-p)
      (macroexpand-1 form env)
    `(values ',expansion ',expanded-p))) → EXPAND-1

;; Simple examples involving just the global environment
(macroexpand-1 '(alpha a b)) → (BETA A B), true
(expand-1 (alpha a b)) → (BETA A B), true
(macroexpand '(alpha a b)) → (GAMMA A B), true
(expand (alpha a b)) → (GAMMA A B), true
(macroexpand-1 'not-a-macro) → NOT-A-MACRO, false
(expand-1 not-a-macro) → NOT-A-MACRO, false
(macroexpand '(not-a-macro a b)) → (NOT-A-MACRO A B), false
(expand (not-a-macro a b)) → (NOT-A-MACRO A B), false

;; Examples involving lexical environments
(macrolet ((alpha (x y) `(delta ,x ,y)))
  (macroexpand-1 '(alpha a b)) → (BETA A B), true
  (expand-1 (alpha a b)) → (DELTA A B), true
  (macroexpand '(alpha a b)) → (GAMMA A B), true
  (expand (alpha a b)) → (GAMMA A B), true
  (macrolet ((alpha (x y) `(delta ,x ,y)))
    (macroexpand '(alpha a b)) → (GAMMA A B), true
    (expand (alpha a b)) → (GAMMA A B), true
    (macrolet ((beta (x y) `(epsilon ,x ,y)))
      (macroexpand-1 (alpha a b)) → (EPSILON A B), true
      (expand (alpha a b)) → (EPSILON A B), true
      (let ((x (list 1 2 3)))
        (symbol-macrolet ((a (first x)))
          (expand a)) → (FIRST X), true
        (let ((x (list 1 2 3)))
          (symbol-macrolet ((a (first x)))
            (macroexpand 'a)) → a, false
          (symbol-macrolet ((b (alpha x y)))
            (expand-1 b)) → (ALPHA X Y), true
          (symbol-macrolet ((b (alpha x y)))
            (expand b)) → (GAMMA X Y), true
          (symbol-macrolet ((b (alpha x y)))
            (exp...
((a b))
(expand-1 a)) → B, true
(symbol-macrolet ((b (alpha x y))
 (a b))
(expand a)) → (GAMMA X Y), true

;; Examples of shadowing behavior
(flet ((beta (x y) (+ x y)))
 (expand (alpha a b))) → (BETA A B), true
(macrolet ((alpha (x y) '(delta ,x ,y)))
 (flet ((alpha (x y) (+ x y)))
 (expand (alpha a b))) → (ALPHA A B), false
(let ((x (list 1 2 3)))
 (symbol-macrolet ((a (first x)))
 (let ((a x))
 (expand a))) → A, false

Affected By:
defmacro, setf of macro-function, macrolet, symbol-macrolet

See Also:
*macroexpand-hook*, defmacro, setf of macro-function, macrolet, symbol-macrolet, Section 3.1 (Evaluation)

Notes:
Neither macroexpand nor macroexpand-1 makes any explicit attempt to expand macro forms that are either subforms of the form or subforms of the expansion. Such expansion might occur implicitly, however, due to the semantics or implementation of the macro function.

define-symbol-macro

Macro

Syntax:
define-symbol-macro symbol expansion
→ symbol

Arguments and Values:
symbol—a symbol.
expansion—a form.
define-symbol-macro

Description:

Provides a mechanism for globally affecting the macro expansion of the indicated symbol.

Globally establishes an expansion function for the symbol macro named by symbol. The only guaranteed property of an expansion function for a symbol macro is that when it is applied to the form and the environment it returns the correct expansion. (In particular, it is implementation-dependent whether the expansion is conceptually stored in the expansion function, the environment, or both.)

Each global reference to symbol (i.e., not shadowed by a binding for a variable or symbol macro named by the same symbol) is expanded by the normal macro expansion process; see Section 3.1.2.1.1 (Symbols as Forms). The expansion of a symbol macro is subject to further macro expansion in the same lexical environment as the symbol macro reference, exactly analogous to normal macros.

The consequences are unspecified if a special declaration is made for symbol while in the scope of this definition (i.e., when it is not shadowed by a binding for a variable or symbol macro named by the same symbol).

Any use of setq to set the value of the symbol while in the scope of this definition is treated as if it were a setf. psetq of symbol is treated as if it were a psetf, and multiple-value-setq is treated as if it were a setf of values.

A binding for a symbol macro can be shadowed by let or symbol-macrolet.

Examples:

```
(define-symbol-macro thing1 (first *things*)) → THING1
(define-symbol-macro thing2 (second *things*)) → THING2
(define-symbol-macro thing3 (third *things*)) → THING3
```

```
(thing1) → ALPHA
(setq thing1 'ONE) → ONE
*things* → (ONE BETA GAMMA)
(multiple-value-setq (thing2 thing3) (values 'two 'three)) → TWO
(thing3) → THREE
*things* → (ONE TWO THREE)
```

```
(list thing2 (let ((thing2 2)) thing2)) → (TWO 2)
```

Exceptional Situations:

If symbol is already defined as a global variable, an error of type program-error is signaled.

See Also:

symbol-macrolet, macroexpand
symbol-macrolet

**Syntax:**

\[
\text{symbol-macrolet}\; (\{\text{symbol expansion}\}^*\,\{\text{declaration}\}^*\,\{\text{form}\}^*)
\]

\[
\rightarrow\;\{\text{result}\}^*
\]

**Arguments and Values:**

- **symbol**—a symbol.
- **expansion**—a form.
- **declaration**—a declare expression; not evaluated.
- **forms**—an implicit progn.
- **results**—the values returned by the forms.

**Description:**

symbol-macrolet provides a mechanism for affecting the macro expansion environment for symbols.

symbol-macrolet lexically establishes expansion functions for each of the symbol macros named by symbols. The only guaranteed property of an expansion function for a symbol macro is that when it is applied to the form and the environment it returns the correct expansion. (In particular, it is implementation-dependent whether the expansion is conceptually stored in the expansion function, the environment, or both.)

Each reference to symbol as a variable within the lexical scope of symbol-macrolet is expanded by the normal macro expansion process; see Section 3.1.2.1.1 (Symbols as Forms). The expansion of a symbol macro is subject to further macro expansion in the same lexical environment as the symbol macro invocation, exactly analogous to normal macros.

Exactly the same declarations are allowed as for let with one exception: symbol-macrolet signals an error if a special declaration names one of the symbols being defined by symbol-macrolet.

When the forms of the symbol-macrolet form are expanded, any use of setq to set the value of one of the specified variables is treated as if it were a setf. psetq of a symbol defined as a symbol macro is treated as if it were a psetf, and multiple-value-setq is treated as if it were a setf of values.

The use of symbol-macrolet can be shadowed by let. In other words, symbol-macrolet only substitutes for occurrences of symbol that would be in the scope of a lexical binding of symbol surrounding the forms.
Examples:

;;; The following is equivalent to
;;; (list 'foo (let ((x 'bar)) x)),
;;; not
;;; (list 'foo (let ((foo 'bar)) 'foo))
(symbol-macrolet ((x 'foo))
  (list x (let ((x 'bar)) x)))
→ (foo bar)
not
→ (foo foo)

(symbol-macrolet ((x '(foo x)))
  (list x))
→ ((FOO X))

Exceptional Situations:

If an attempt is made to bind a symbol that is defined as a global variable, an error of type program-error is signaled.

If declaration contains a special declaration that names one of the symbols being bound by symbol-macrolet, an error of type program-error is signaled.

See Also:
with-slots, macroexpand

Notes:

The special form symbol-macrolet is the basic mechanism that is used to implement with-slots.

If a symbol-macrolet form is a top level form, the forms are also processed as top level forms. See Section 3.2.3 (File Compilation).

*macroexpand-hook*

Variable

Value Type:
a designator for a function of three arguments: a macro function, a macro form, and an environment object.

Initial Value:
a designator for a function that is equivalent to the function funcall, but that might have additional implementation-dependent side-effects.
Description:

Used as the expansion interface hook by `macroexpand-1` to control the macro expansion process. When a macro form is to be expanded, this function is called with three arguments: the macro function, the macro form, and the environment in which the macro form is to be expanded. The environment object has dynamic extent; the consequences are undefined if the environment object is referred to outside the dynamic extent of the macro expansion function.

Examples:

```lisp
(defun hook (expander form env)
  (format t "Now expanding: ~S~%" form)
  (funcall expander form env))
→ HOOK

(defmacro machook (x y) '(/ (+ ,x ,y) 2)) → MACHOOK

(macroexpand '(machook 1 2)) → (/ (+ 1 2) 2), true

(let ((*macroexpand-hook* #'hook))
  (macroexpand '(machook 1 2)))
▷ Now expanding (MACHOOK 1 2)
  → (/ (+ 1 2) 2), true
```

See Also:

macroexpand, macroexpand-1, funcall, Section 3.1 (Evaluation)

Notes:

The net effect of the chosen initial value is to just invoke the macro function, giving it the macro form and environment as its two arguments.

Users or user programs can assign this variable to customize or trace the macro expansion mechanism. Note, however, that this variable is a global resource, potentially shared by multiple programs; as such, if any two programs depend for their correctness on the setting of this variable, those programs may not be able to run in the same Lisp image. For this reason, it is frequently best to confine its uses to debugging situations.

Users who put their own function into *macroexpand-hook* should consider saving the previous value of the hook, and calling that value from their own.

---

proclaim

Function

Syntax:

proclaim declaration-specifier → implementation-dependent

Arguments and Values:

description-specifier—a declaration specifier.
proclaim

Description:

Establishes the declaration specified by declaration-specifier in the global environment.

Such a declaration, sometimes called a global declaration or a proclamation, is always in force unless locally shadowed.

Names of variables and functions within declaration-specifier refer to dynamic variables and global function definitions, respectively.

Figure 3–22 shows a list of declaration identifiers that can be used with proclaim.

<table>
<thead>
<tr>
<th>declaration</th>
<th>inline</th>
<th>optimize</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>inline</td>
<td>optimize</td>
<td>type</td>
</tr>
<tr>
<td>notinline</td>
<td></td>
<td>special</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3–22. Global Declaration Specifiers

An implementation is free to support other (implementation-defined) declaration identifiers as well.

Examples:

```
(defun declare-variable-types-globally (type vars)
  (proclaim '(type ,type ,@vars))
  type)
```

;; Once this form is executed, the dynamic variable *TOLERANCE*
;; must always contain a float.

```
(declare-variable-types-globally 'float '(*tolerance*))
```

→ FLOAT

See Also:

declare, Section 3.2 (Compilation)

Notes:

Although the execution of a proclaim form has effects that might affect compilation, the compiler does not make any attempt to recognize and specially process proclaim forms. A proclamation such as the following, even if a top level form, does not have any effect until it is executed:

```
(proclaim '*(special *x*))
```

If compile time side effects are desired, eval-when may be useful. For example:

```
(eval-when (:execute :compile-toplevel :load-toplevel)
  (proclaim '*(special *x*))
)
```

In most such cases, however, it is preferrable to use declare for this purpose.
Since proclaim forms are ordinary function forms, macro forms can expand into them.

**declare**

*Macro*

**Syntax:**

```elisp
declare {declaration-specifier}*
```

→ implementation-dependent

**Arguments and Values:**

- **declaration-specifier**—a declaration specifier; not evaluated.

**Description:**

Establishes the declarations specified by the declaration-specifiers.

If a use of this macro appears as a top level form in a file being processed by the file compiler, the proclamations are also made at compile-time. As with other defining macros, it is unspecified whether or not the compile-time side-effects of a declare persist after the file has been compiled.

**Examples:**

**See Also:**

declare, proclaim

**declare**

*Symbol*

**Syntax:**

```elisp
declare {declaration-specifier}*
```

**Arguments:**

- **declaration-specifier**—a declaration specifier; not evaluated.

**Description:**

A declare expression, sometimes called a declaration, can occur only at the beginning of the bodies of certain forms; that is, it may be preceded only by other declare expressions, or by a documentation string if the context permits.

A declare expression can occur in a lambda expression or in any of the forms listed in Figure 3–23.
A `declare` expression can only occur where specified by the syntax of these forms. The consequences of attempting to evaluate a `declare` expression are undefined. In situations where such expressions can appear, explicit checks are made for their presence and they are never actually evaluated; it is for this reason that they are called “declare expressions” rather than “declare forms.”

Macro forms cannot expand into declarations; declare expressions must appear as actual subexpressions of the form to which they refer.

Figure 3-24 shows a list of declaration specifiers that can be used with declare.

An implementation is free to support other (implementation-defined) declaration identifiers as well.

**Examples:**

```lisp
(defun nonsense (k x z)
  (foo z x) ;First call to foo
  (let ((j (foo k x)) ;Second call to foo
        (x (* k k)))
    (declare (inline foo) (special x z))
    (foo x j z))) ;Third call to foo
```
In this example, the inline declaration applies only to the third call to \texttt{foo}, but not to the first or second ones. The special declaration of \texttt{x} causes \texttt{let} to make a dynamic binding for \texttt{x}, and causes the reference to \texttt{x} in the body of \texttt{let} to be a dynamic reference. The reference to \texttt{x} in the second call to \texttt{foo} is a local reference to the second parameter of \texttt{nonsense}. The reference to \texttt{x} in the first call to \texttt{foo} is a local reference, not a special one. The special declaration of \texttt{z} causes the reference to \texttt{z} in the third call to \texttt{foo} to be a dynamic reference; it does not refer to the parameter \texttt{nonsense} named \texttt{z}, because that parameter binding has not been declared to be special. (The special declaration of \texttt{z} does not appear in the body of \texttt{defun}, but in an inner form, and therefore does not affect the binding of the parameter.)

**Exceptional Situations:**

The consequences of trying to use a declare expression as a form to be evaluated are undefined.

**See Also:**
proclaim, Section 4.2.3 (Type Specifiers), declaration, dynamic-extent, ftype, ignorable, ignore, inline, notinline, optimize, type

---

**ignore, ignorable**

**Declaration**

**Syntax:**

\[
\text{(ignore \{var | (function fn)\}*  \\
\text{ ignorable \{var | (function fn)\}*  
\text{ ignore \{var | (function fn)\}*  \\
\text{ ignorable \{var | (function fn)\}*  
\]

**Arguments:**

- \texttt{var}—a variable name.
- \texttt{fn}—a function name.

**Valid Context:**

declaration

**Binding Types Affected:**

variable, function

**Description:**

The ignore and ignorable declarations refer to for-value references to variable bindings for the \texttt{vars} and to function bindings for the \texttt{fns}.

An ignore declaration specifies that for-value references to the indicated bindings will not occur within the scope of the declaration. Within the scope of such a declaration, it is desirable for a compiler to issue a warning about the presence of either a for-value reference to any \texttt{var} or \texttt{fn}, or a special declaration for any \texttt{var}. 

Evaluation and Compilation 3–85
An ignorable declaration specifies that for-value references to the indicated bindings might or might not occur within the scope of the declaration. Within the scope of such a declaration, it is not desirable for a compiler to issue a warning about the presence or absence of either a for-value reference to any var or fn, or a special declaration for any var.

When not within the scope of a ignore or ignorable declaration, it is desirable for a compiler to issue a warning about any var for which there is neither a for-value reference nor a special declaration, or about any fn for which there is no for-value reference.

Any warning about a “used” or “unused” binding must be of type style-warning, and may not affect program semantics.

The stream variables established by with-open-file, with-open-stream, with-input-from-string, and with-output-to-string, and all iteration variables are, by definition, always “used”. Using (declare (ignore v)), for such a variable v has unspecified consequences.

See Also:
  declare

---

dynamic-extent

**Declaration**

Syntax:

\[\text{(dynamic-extent [{\text{var}}\,* | \text{(function fn)}\,*]}]\]

Arguments:

- var—a variable name.
- fn—a function name.

Valid Context:

- declaration

Binding Types Affected:

- variable, function

Description:

In some containing form, F, this declaration asserts for each var, (which need not be bound by F), and for each value \(v_i\) that var takes on, and for each object \(x_{ijk}\) that is an otherwise inaccessible part of \(v_i\) at any time when \(v_i\) becomes the value of var, that just after the execution of F terminates, \(x_{ijk}\) is either inaccessible (if F established a binding for var) or still an otherwise inaccessible part of the current value of var (if F did not establish a binding for var). The same relation holds for each fn, except that the bindings are in the function namespace.
The compiler is permitted to use this information in any way that is appropriate to the implementation and that does not conflict with the semantics of Common Lisp.

**dynamic-extent** declarations can be *free declarations* or *bound declarations*.

The **vars** and **fns** named in a **dynamic-extent** declaration must not refer to *symbol macro* or *macro* bindings.

**Examples:**

Since stack allocation of the initial value entails knowing at the **object**’s creation time that the **object** can be **stack-allocated**, it is not generally useful to make a **dynamic-extent declaration** for **variables** which have no lexically apparent initial value. For example, it is probably useful to write:

```
(defun f ()
  (let ((x (list 1 2 3)))
    (declare (dynamic-extent x))
    ...
))
```

This would permit those compilers that wish to do so to **stack allocate** the list held by the local variable `x`. It is permissible, but in practice probably not as useful, to write:

```
(defun g (x) (declare (dynamic-extent x)) ...) 
(defun f () (g (list 1 2 3)))
```

Most compilers would probably not **stack allocate** the argument to `g` in `f` because it would be a modularity violation for the compiler to assume facts about `g` from within `f`. Only an implementation that was willing to be responsible for recompiling `f` if the definition of `g` changed incompatibly could legitimately **stack allocate** the list argument to `g` in `f`.

Here is another example:

```
(declaim (inline g))
(defun g (x) (declare (dynamic-extent x)) ...) 
(defun f () (g (list 1 2 3)))
```

```
(defun f ()
  (flet ((g (x) (declare (dynamic-extent x)) ...))
    (g (list 1 2 3)))
)
```

In the previous example, some compilers might determine that optimization was possible and others might not.

A variant of this is the so-called “stack allocated rest list” that can be achieved (in implementations supporting the optimization) by:

```
(defun f (&rest x)
```

```
...)
```
dynamic-extent

(declare (dynamic-extent x))
...)

Note that although the initial value of x is not explicit, the f function is responsible for assembling the list x from the passed arguments, so the f function can be optimized by the compiler to construct a stack-allocated list instead of a heap-allocated list in implementations that support such.

In the following example,

(let ((x (list 'a1 'b1 'c1))
       (y (cons 'a2 (cons 'b2 (cons 'c2 nil))))))
(declare (dynamic-extent x y))
...

The otherwise inaccessible parts of x are three conses, and the otherwise inaccessible parts of y are three other conses. None of the symbols a1, b1, c1, a2, b2, c2, or nil is an otherwise inaccessible part of x or y because each is interned and hence accessible by the package (or packages) in which it is interned. However, if a freshly allocated uninterned symbol had been used, it would have been an otherwise inaccessible part of the list which contained it.

;; In this example, the implementation is permitted to stack allocate
;; the list that is bound to X.
(let ((x (list 1 2 3)))
 (declare (dynamic-extent x))
 (print x)
 :done)
> (1 2 3)
→ :DONE

;; In this example, the list to be bound to L can be stack-allocated.
(defun zap (x y z)
 (do ((l (list x y z) (cdr l))
      ((null l))
      (declare (dynamic-extent l))
      (print (car l)))) → ZAP
(zap 1 2 3)
> 123
→ NIL

;; Some implementations might open-code LIST-ALL-PACKAGES in a way
;; that permits using stack allocation of the list to be bound to L.
(do ((l (list-all-packages) (cdr l)))
 (null l))
(declare (dynamic-extent l))
(let ((name (package-name (car l))))
 (when (string-search "COMMON-LISP" name) (print name))))

3–88 Programming Language—Common Lisp
Some implementations might have the ability to stack allocate rest lists. A declaration such as the following should be a cue to such implementations that stack-allocation of the rest list would be desirable.

```
(defun add (&rest x)
  (declare (dynamic-extent x))
  (apply #'+ x))
```

```
(add 1 2 3) → 6
```

```
(defun zap (n m)
  ;; Computes (RANDOM (+ M 1)) at relative speed of roughly O(N).
  ;; It may be slow, but with a good compiler at least it
  ;; doesn’t waste much heap storage. :-)  
  (let ((a (make-array n)))
    (declare (dynamic-extent a))
    (do (i n)
      (declare (dynamic-extent i))
      (setf (aref a i) (random (+ i 1))))
    (aref a m)))
```

```
(< (zap 5 3) 3) → true
```

The following are in error, since the value of x is used outside of its extent:

```
(length (list (let ((x (list 1 2 3))) ; Invalid
  (declare (dynamic-extent x))
  x)))
```

```
(progn (let ((x (list 1 2 3))) ; Invalid
  (declare (dynamic-extent x))
  x)
  nil)
```

See Also:
- declare

Notes:
The most common optimization is to stack allocate the initial value of the objects named by the vars.

It is permissible for an implementation to simply ignore this declaration.
**type**

**Declaration**

**Syntax:**

```
(type typespec {var}*)
(typespec {var}*)
```

**Arguments:**

- `typespec`—a type specifier.
- `var`—a variable name.

**Valid Context:**

declaration or proclamation

**Binding Types Affected:**

variable

**Description:**

Affects only variable bindings and specifies that the `var` take on values only of the specified `typespec`. In particular, values assigned to the variables by `setq`, as well as the initial values of the `var` must be of the specified `typespec`. Type declarations never apply to function bindings (see `ftype`).

A type declaration of a symbol defined by `symbol-macrolet` is equivalent to wrapping a `the` expression around the expansion of that symbol, although the symbol's macro expansion is not actually affected.

The meaning of a type declaration is equivalent to changing each reference to a variable (`var`) within the scope of the declaration to `(the typespec var)`, changing each expression assigned to the variable (`new-value`) within the scope of the declaration to `(the typespec new-value)`, and executing `(the typespec var)` at the moment the scope of the declaration is entered.

A `type` declaration is valid in all declarations. The interpretation of a type declaration is as follows:

1. During the execution of any reference to the declared variable within the scope of the declaration, the consequences are undefined if the value of the declared variable is not of the declared `type`.

2. During the execution of any `setq` of the declared variable within the scope of the declaration, the consequences are undefined if the newly assigned value of the declared variable is not of the declared `type`.

3. At the moment the scope of the declaration is entered, the consequences are undefined if
the value of the declared variable is not of the declared type.

A type declaration affects only variable references within its scope.

If nested type declarations refer to the same variable, then the value of the variable must be a member of the intersection of the declared types.

If there is a local type declaration for a dynamic variable, and there is also a global type proclamation for that same variable, then the value of the variable within the scope of the local declaration must be a member of the intersection of the two declared types.

Type declarations can be free declarations or bound declarations.

A symbol cannot be both the name of a type and the name of a declaration. Defining a symbol as the name of a class, structure, condition, or type, when the symbol has been declared as a declaration name, or vice versa, signals an error.

Within the lexical scope of an array type declaration, all references to array elements are assumed to satisfy the expressed array element type (as opposed to the upgraded array element type). A compiler can treat the code within the scope of the array type declaration as if each access of an array element were surrounded by an appropriate the form.

Examples:

```
(defun f (x y)
  (declare (type fixnum x y))
  (let ((z (+ x y)))
    (declare (type fixnum z))
    z)) → F
(f 1 2) → 3
;; The previous definition of F is equivalent to
(defun f (x y)
  ;; This declaration is a shorthand form of the TYPE declaration
  (declare (fixnum x y))
  ;; To declare the type of a return value, it’s not necessary to
  ;; create a named variable. A THE special form can be used instead.
  (the fixnum (+ x y))) → F
(f 1 2) → 3

(defvar *one-array* (make-array 10 :element-type '(signed-byte 5)))
(defvar *another-array* (make-array 10 :element-type '(signed-byte 8)))

(defun frob (an-array)
  (declare (type (array (signed-byte 5) 1) an-array))
  (setf (aref an-array 1) 31)
  (setf (aref an-array 2) 127)
```
**type**

```
(setq (aref an-array 3) (* 2 (aref an-array 3)))
(let ((foo 0))
  (declare (type (signed-byte 5) foo))
  (setf foo (aref an-array 0)))

(frob *one-array*)
(frob *another-array*)

The above definition of `frob` is equivalent to:

```
(defun frob (an-array)
  (setf (the (signed-byte 5) (aref an-array 1)) 31)
  (setf (the (signed-byte 5) (aref an-array 2)) 127)
  (setf (the (signed-byte 5) (aref an-array 3))
    (* 2 (the (signed-byte 5) (aref an-array 3))))
  (let ((foo 0))
    (declare (type (signed-byte 5) foo))
    (setf foo (the (signed-byte 5) (aref an-array 0)))))
```

Given an implementation in which **fixnums** are 29 bits but **fixnum arrays** are upgraded to signed 32-bit **arrays**, the following could be compiled with all **fixnum** arithmetic:

```
(defun bump-counters (counters)
  (declare (type (array fixnum *) bump-counters))
  (dotimes (i (length counters))
    (incf (aref counters i)))
```

See Also:
declare, declaim, proclaim

Notes:

`<typespec {var}>` is an abbreviation for `(type <typespec {var}>`.

A **type** declaration for the arguments to a function does not necessarily imply anything about the type of the result. The following function is not permitted to be compiled using **implementation-dependent fixnum-only arithmetic**:

```
(defun f (x y) (declare (fixnum x y)) (+ x y))
```

To see why, consider `(f most-positive-fixnum 1)`. Common Lisp defines that F must return a **bignum** here, rather than signal an error or produce a mathematically incorrect result. If you have special knowledge such as "fixnum overflow" cases will not come up, you can declare the result value to be in the **fixnum** range, enabling some compilers to use more efficient arithmetic:

```
(defun f (x y)
  (declare (fixnum x y))
  (the fixnum (+ x y)))
```

3–92  Programming Language—Common Lisp
Note, however, that in the three-argument case, because of the possibility of an implicit intermediate value growing too large, the following will not cause implementation-dependent fixnum-only arithmetic to be used:

```
(defun f (x y)
  (declare (fixnum x y z))
  (the fixnum (+ x y z)))
```

To see why, consider `(f most-positive-fixnum 1 -1)`. Although the arguments and the result are all fixnums, an intermediate value is not a fixnum. If it is important that implementation-dependent fixnum-only arithmetic be selected in implementations that provide it, consider writing something like this instead:

```
(defun f (x y)
  (declare (fixnum x y z))
  (the fixnum (+ (the fixnum (+ x y)) z)))
```

---

**inline, notinline**

*Declaration*

**Syntax:**

```
(inline {function-name}* )

(notinline {function-name}* )
```

**Arguments:**

*function-name*—a function name.

**Valid Context:**

declaration or proclamation

**Binding Types Affected:**

function

**Description:**

*inline* specifies that it is desirable for the compiler to produce inline calls to the *functions* named by *function-names*; that is, the code for a specified *function-name* should be integrated into the calling routine, appearing “in line” in place of a procedure call. A compiler is free to ignore this declaration. *inline* declarations never apply to variable bindings.

If one of the *functions* mentioned has a lexically apparent local definition (as made by *flet* or *labels*), then the declaration applies to that local definition and not to the global function definition.
inline, notinline

While no conforming implementation is required to perform inline expansion of user-defined functions, those implementations that do attempt to recognize the following paradigm:

To define a function \( f \) that is not inline by default but for which (\texttt{declare (inline f)}) will make \( f \) be locally inlined, the proper definition sequence is:

\[
\begin{align*}
&\texttt{(declare (inline f))} \\
&\texttt{(defun f ...)} \\
&\texttt{(declare (notinline f))}
\end{align*}
\]

The \texttt{inline} proclamation preceding the \texttt{defun} form ensures that the compiler has the opportunity to save the information necessary for inline expansion, and the \texttt{notinline} proclamation following the \texttt{defun} form prevents \( f \) from being expanded inline everywhere.

\texttt{notinline} specifies that it is undesirable to compile the \texttt{functions} named by \texttt{function-names} inline. A compiler is not free to ignore this declaration; calls to the specified functions must be implemented as out-of-line subroutine calls.

If one of the \texttt{functions} mentioned has a lexically apparent local definition (as made by \texttt{flet} or \texttt{labels}), then the declaration applies to that local definition and not to the global function definition.

In the presence of a \texttt{compiler macro} definition for \texttt{function-name}, a \texttt{notinline} declaration prevents that \texttt{compiler macro} from being used. An \texttt{inline} declaration may be used to encourage use of \texttt{compiler macro} definitions. \texttt{inline} and \texttt{notinline} declarations otherwise have no effect when the lexically visible definition of \texttt{function-name} is a \texttt{macro} definition.

\texttt{inline} and \texttt{notinline} declarations can be free declarations or bound declarations. \texttt{inline} and \texttt{notinline} declarations of functions that appear before the body of a \texttt{flet} or \texttt{labels} form that defines that function are bound declarations. Such declarations in other contexts are free declarations.

Examples:

\[
\begin{align*}
&\texttt{;; The globally defined function DISPATCH should be open-coded,} \\
&\texttt{;; if the implementation supports inlining, unless a NOTINLINE} \\
&\texttt{;; declaration overrides this effect.} \\
&\texttt{(declare (inline dispatch))} \\
&\texttt{(defun dispatch (x) (funcall (get (car x) 'dispatch) x))} \\
&\texttt{;; Here is an example where inlining would be encouraged.} \\
&\texttt{(defun top-level-1 () (dispatch (read-command)))} \\
&\texttt{;; Here is an example where inlining would be prohibited.} \\
&\texttt{(defun top-level-2 ()} \\
&\texttt{  (declare (notinline dispatch))} \\
&\texttt{  (dispatch (read-command)))} \\
&\texttt{;; Here is an example where inlining would be prohibited.} \\
&\texttt{(declare (notinline dispatch))} \\
&\texttt{(defun top-level-3 () (dispatch (read-command)))}
\end{align*}
\]
;; Here is an example where inlining would be encouraged.
(defun top-level-4 ()
  (declare (inline dispatch))
  (dispatch (read-command)))

See Also:
declare, declare, proclaim

ftype

Syntax:
(ftype type [function-name]*)

Arguments:
function-name—a function name.
type—a type specifier.

Valid Context:
declaration or proclamation

Binding Types Affected:
function

Description:
Specifies that the functions named by function-names are of the functional type type. For example:

(declare (ftype (function (integer list) t) ith)
  (ftype (function (number) float) sine cosine))

If one of the functions mentioned has a lexically apparent local definition (as made by flet or labels), then the declaration applies to that local definition and not to the global function definition. ftype declarations never apply to variable bindings (see type).

The lexically apparent bindings of function-names must not be macro definitions. (This is because ftype declares the functional definition of each function name to be of a particular subtype of function, and macros do not denote functions.)

ftype declarations can be free declarations or bound declarations. ftype declarations of functions that appear before the body of a flet or labels form that defines that function are bound declarations. Such declarations in other contexts are free declarations.
See Also:
declare, declaim, proclaim

declaration

Syntax:
(declaration {name}*)

Arguments:
name—a symbol.

Valid Context:
proclamation only

Description:
Advises the compiler that each name is a valid but potentially non-standard declaration name.
The purpose of this is to tell one compiler not to issue warnings for declarations meant for
another compiler or other program processor.

Examples:
(declare (declaration author target-language target-machine))
(declare (target-language ada))
(declare (target-machine IBM-650))
(defun strangep (x)
  (declare (author "Harry Tweaker")))
  (member x '(('strange weird odd peculiar)))

See Also:
declaim, proclaim
Syntax:

\[
(\text{optimize } \{ \text{quality } | (\text{quality value}) \}^*)
\]

Arguments:

- \text{quality}—an optimize quality.
- \text{value}—one of the integers 0, 1, 2, or 3.

Valid Context:

declaration or proclamation

Description:

Advises the compiler that each \text{quality} should be given attention according to the specified corresponding \text{value}. Each \text{quality} must be a symbol naming an optimize quality; the names and meanings of the standard optimize qualities are shown in Figure 3–25.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>compilation-speed</td>
<td>speed of the compilation process</td>
</tr>
<tr>
<td>debug</td>
<td>ease of debugging</td>
</tr>
<tr>
<td>safety</td>
<td>run-time error checking</td>
</tr>
<tr>
<td>space</td>
<td>both code size and run-time space</td>
</tr>
<tr>
<td>speed</td>
<td>speed of the object code</td>
</tr>
</tbody>
</table>

Figure 3–25. Optimize qualities

There may be other, implementation-defined optimize qualities.

A \text{value} 0 means that the corresponding \text{quality} is totally unimportant, and 3 that the \text{quality} is extremely important; 1 and 2 are intermediate values, with 1 the neutral value. (\text{quality} 3) can be abbreviated to \text{quality}.

Note that \text{code} which has the optimization (\text{safety} 3), or just \text{safety}, is called \text{safe code}.

The consequences are unspecified if a \text{quality} appears more than once with different \text{values}.

Examples:

\begin{verbatim}
(defun often-used-subroutine (x y)
  (declare (optimize (safety 2)))
  (error-check x y)
  (hairy-setup x)
  (do ((i 0 (+ i 1)))
    (z x (cdr z)))
\end{verbatim}
((null z))
;; This inner loop really needs to burn.
(declare (optimize speed))
(declare (fixnum i))
))

See Also:
declare, declaim, proclaim, Section 3.3.4 (Declaration Scope)

Notes:
An optimize declaration never applies to either a variable or a function binding. An optimize declaration can only be a free declaration. For more information, see Section 3.3.4 (Declaration Scope).

---

**special**

*Declaration*

**Syntax:**

(special {var}* )

**Arguments:**

var—a symbol.

**Valid Context:**

declaration or proclamation

**Binding Types Affected:**

variable

**Description:**

Specifies that all of the vars named are dynamic. This specifier affects variable bindings and affects references. All variable bindings affected are made to be dynamic bindings, and affected variable references refer to the current dynamic binding. For example:

```
(defun hack (thing *mod*) ;The binding of the parameter
  (declare (special *mod*)); *mod* is visible to hack1,
  (hack1 (car thing))); but not that of thing.
(defun hack1 (arg)
  (declare (special *mod*)); Declare references to *mod*
  ;within hack1 to be special.
  (if (atom arg) *mod*
      (cons (hack1 (car arg)) (hack1 (cdr arg)))))
```

---

3–98 Programming Language—Common Lisp
A *special* declaration does not affect inner bindings of a `var`; the inner bindings implicitly shadow a `special` declaration and must be explicitly re-declared to be `special`. `special` declarations never apply to function bindings.

`special` declarations can be either *bound declarations*, affecting both a binding and references, or *free declarations*, affecting only references, depending on whether the declaration is attached to a variable binding.

When used in a *proclamation*, a `special` declaration specifier applies to all bindings as well as to all references of the mentioned variables. For example, after

```
(declare (special x))
```

then in a function definition such as

```
(defun example (x) ...)
```

the parameter `x` is bound as a dynamic variable rather than as a lexical variable.

**Examples:**

```
(defun declare-eg (y) ; this y is special
  (declare (special y))
  (let ((y t)) ; this y is lexical
    (list y
      (locally (declare (special y)) y)))) ; this y refers to the
                                      ; special binding of y
```

→ DECLARE-EG
```
(declare-eg nil) → (T NIL)
```

```
(setf (symbol-value 'x) 6)
(defun foo (x) ; a lexical binding of x
  (print x)
  (let ((x (1+ x))) ; a special binding of x
    (declare (special x)) ; and a lexical reference
    (bar))
  (1+ x))
(defun bar ()
  (print (locally (declare (special x))
    x)))
(foo 10)
⇒ 10
⇒ 11
⇒ 11
```

```
(setf (symbol-value 'x) 6)
(defun bar (x y) ; [1] 1st occurrence of x
  (let ((old-x x)) ; [2] 2nd occurrence of x -- same as 1st occurrence
    ...)
```

Evaluation and Compilation 3–99
The reference to *foo* in the first line of this example is not special even though there is a special declaration in the second line.

In the contorted code above, the outermost and innermost bindings of y are dynamic, but the middle binding is lexical. The two arguments to + are different, one being the value, which is 3, of the lexical variable y, and the other being the value of the dynamic variable named y (a binding of which happens, coincidentally, to lexically surround it at an outer level). All the bindings of x and references to x are dynamic, however, because of the proclamation that x is always special.

See Also:
defparameter, defvar

---

**locally**

*Special Operator*

**Syntax:**

locally {declaration}*[ form]* → {result}*

**Arguments and Values:**

*Declaration—a declare expression; not evaluated.*
forms—an implicit progn.

results—the values of the forms.

Description:
Sequentially evaluates a body of forms in a lexical environment where the given declarations have effect.

Examples:

```lisp
(defun sample-function (y) ;this y is regarded as special
  (declare (special y))
  (let ((y t)) ;this y is regarded as lexical
    (list y
      (locally (declare (special y))
        ;; this next y is regarded as special
        y))))
→ SAMPLE-FUNCTION
(sample-function nil) → (T NIL)
(setq x '(1 2 3) y '(4 . 5)) → (4 . 5)

;;; The following declarations are not notably useful in specific.
;;; They just offer a sample of valid declaration syntax using LOCALLY.
(locally (declare (inline floor) (notinline car cdr))
  (declare (optimize space))
  (floor (car x) (cdr y))) → 0, 1

;;; This example shows a definition of a function that has a particular set
;;; of OPTIMIZE settings made locally to that definition.
(locally (declare (optimize (safety 3) (space 3) (speed 0)))
  (defun frob (w x y &optional (z (foo x y)))
    (mumble x y z w)))
→ FROB

;;; This is like the previous example, except that the optimize settings
;;; remain in effect for subsequent definitions in the same compilation unit.
(declaim (optimize (safety 3) (space 3) (speed 0)))
(defun frob (w x y &optional (z (foo x y)))
  (mumble x y z w))
→ FROB
```

See Also:
declare
Notes:
The `special` declaration may be used with `locally` to affect references to, rather than `bindings` of, `variables`.

If a `locally` form is a `top level form`, the body `forms` are also processed as `top level forms`. See `Section 3.2.3 (File Compilation)`.

### the

**Special Operator**

**Syntax:**

\[
\text{the \ value-type \ form} \rightarrow \{\text{result}\}^*
\]

**Arguments and Values:**

- `value-type`—a `type specifier`; not evaluated.
- `form`—a `form`; evaluated.
- `results`—the `values` resulting from the `evaluation` of `form`. These `values` must conform to the `type` supplied by `value-type`; see below.

**Description:**

`the` specifies that the `values` returned by `form` are of the `types` specified by `value-type`. The consequences are undefined if any `result` is not of the declared type.

It is permissible for `form` to `yield` a different number of `values` than are specified by `value-type`, provided that the values for which `types` are declared are indeed of those `types`. Missing values are treated as `nil` for the purposes of checking their `types`.

Regardless of number of `values` declared by `value-type`, the number of `values` returned by the `the` `special form` is the same as the number of `values` returned by `form`.

**Examples:**

```
(the symbol (car (list (gensym)))) → #:G9876
(the fixnum (+ 5 7)) → 12
(the (values) (truncate 3.2 2)) → 1, 1.2
(the integer (truncate 3.2 2)) → 1, 1.2
(the (values integer) (truncate 3.2 2)) → 1, 1.2
(the (values integer float) (truncate 3.2 2)) → 1, 1.2
(the (values integer float symbol) (truncate 3.2 2)) → 1, 1.2
(the (values integer float symbol t null list)
  (truncate 3.2 2)) → 1, 1.2
(let ((i 100))
  (declare (fixnum i))
```

3–102 Programming Language—Common Lisp
(the fixnum (1+ i))) → 101
(let* ((x (list 'a 'b 'c))
   (y 5))
   (setf (the fixnum (car x)) y)
   x) → (5 B C)

Exceptional Situations:
The consequences are undefined if the values yielded by the form are not of the type specified by value-type.

See Also:
values

Notes:
The values type specifier can be used to indicate the types of multiple values:

(the (values integer integer) (floor x y))
(the (values string t)
   (gethash the-key the-string-table))

setf can be used with the type declarations. In this case the declaration is transferred to the form that specifies the new value. The resulting setf form is then analyzed.

---

**special-operator-p**

*Function*

**Syntax:**

special-operator-p symbol → generalized-boolean

**Arguments and Values:**

symbol—a symbol.

generalized-boolean—a generalized boolean.

**Description:**

Returns true if symbol is a special operator; otherwise, returns false.

**Examples:**

(special-operator-p 'if) → true
(special-operator-p 'car) → false
(special-operator-p 'one) → false
Exceptional Situations:
Should signal type-error if its argument is not a symbol.

Notes:
Historically, this function was called special-form-p. The name was finally declared a misnomer and changed, since it returned true for special operators, not special forms.

**constantp**

Function

Syntax:
constantp form &optional environment → generalized-boolean

Arguments and Values:
form—a form.
environment—an environment object. The default is nil.
generalized-boolean—a generalized boolean.

Description:
Returns true if form can be determined by the implementation to be a constant form in the indicated environment; otherwise, it returns false indicating either that the form is not a constant form or that it cannot be determined whether or not form is a constant form.

The following kinds of forms are considered constant forms:

- **Self-evaluating objects** (such as numbers, characters, and the various kinds of arrays) are always considered constant forms and must be recognized as such by constantp.

- **Constant variables**, such as keywords, symbols defined by Common Lisp as constant (such as nil, t, and pi), and symbols declared as constant by the user in the indicated environment using defconstant are always considered constant forms and must be recognized as such by constantp.

- **quote forms** are always considered constant forms and must be recognized as such by constantp.

- An implementation is permitted, but not required, to detect additional constant forms. If it does, it is also permitted, but not required, to make use of information in the environment. Examples of constant forms for which constantp might or might not return true are: (sqrt pi), (+ 3 2), (length '(a b c)), and (let ((x 7)) (zerop x)).
constantp

If an implementation chooses to make use of the environment information, such actions as expanding macros or performing function inlining are permitted to be used, but not required; however, expanding compiler macros is not permitted.

Examples:

- (constantp 1) → true
- (constantp 'temp) → false
- (constantp 'temp) → true
- (defconstant this-is-a-constant 'never-changing) → THIS-IS-A-CONSTANT
- (constantp 'this-is-a-constant) → true
- (constantp "temp") → true
- (setq a 6) → 6
- (constantp a) → true
- (constantp '(sin pi)) → implementation-dependent
- (constantp '(car '(x))) → implementation-dependent
- (constantp '(eql x x)) → implementation-dependent
- (constantp '(typep x 'nil)) → implementation-dependent
- (constantp '(typep x 't)) → implementation-dependent
- (constantp '(values this-is-a-constant)) → implementation-dependent
- (constantp '(values 'x 'y)) → implementation-dependent
- (constantp '(let ((a '(a b c)) (+ (length a) 6))) → implementation-dependent

Affected By:

The state of the global environment (e.g., which symbols have been declared to be the names of constant variables).

See Also:

defconstant
Programming Language—Common Lisp

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).

The class of the class of an object is called its meta-class. For more information about meta-classes, see Section 7.4 (Meta-Objects).
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.

<table>
<thead>
<tr>
<th>Section</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section 4.3 (Classes)</td>
<td>Object System types</td>
</tr>
<tr>
<td>Section 7.5 (Slots)</td>
<td>Object System types</td>
</tr>
<tr>
<td>Chapter 7 (Objects)</td>
<td>Object System types</td>
</tr>
<tr>
<td>Section 7.6 (Generic Functions and Methods)</td>
<td>Object System types</td>
</tr>
<tr>
<td>Section 9.1 (Condition System Concepts)</td>
<td>Condition System types</td>
</tr>
<tr>
<td>Chapter 4 (Types and Classes)</td>
<td>Miscellaneous types</td>
</tr>
<tr>
<td>Chapter 2 (Syntax)</td>
<td>All types—read and print syntax</td>
</tr>
<tr>
<td>Section 22.1 (The Lisp Printer)</td>
<td>All types—print syntax</td>
</tr>
<tr>
<td>Section 3.2 (Compilation)</td>
<td>All types—compilation issues</td>
</tr>
</tbody>
</table>

Figure 4–1. Cross-References to Data Type Information

4.2.2 Type Relationships

- The types cons, symbol, array, number, character, hash-table, function, readable, 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`.
<table>
<thead>
<tr>
<th>Type Specifier</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic-error</td>
<td>function</td>
<td>simple-condition</td>
</tr>
<tr>
<td>array</td>
<td>generic-function</td>
<td>simple-error</td>
</tr>
<tr>
<td>atom</td>
<td>hash-table</td>
<td>simple-string</td>
</tr>
<tr>
<td>base-char</td>
<td>integer</td>
<td>simple-type-error</td>
</tr>
<tr>
<td>base-string</td>
<td>keyword</td>
<td>simple-vector</td>
</tr>
<tr>
<td>bignum</td>
<td>list</td>
<td>simple-warning</td>
</tr>
<tr>
<td>bit</td>
<td>logical-pathname</td>
<td>single-float</td>
</tr>
<tr>
<td>bit-vector</td>
<td>long-float</td>
<td>standard-char</td>
</tr>
<tr>
<td>broadcast-stream</td>
<td>method</td>
<td>standard-class</td>
</tr>
<tr>
<td>built-in-class</td>
<td>method-combination</td>
<td>standard-generic-function</td>
</tr>
<tr>
<td>cell-error</td>
<td>nil</td>
<td>standard-method</td>
</tr>
<tr>
<td>character</td>
<td>null</td>
<td>standard-object</td>
</tr>
<tr>
<td>class</td>
<td>number</td>
<td>storage-condition</td>
</tr>
<tr>
<td>compiled-function</td>
<td>package</td>
<td>stream</td>
</tr>
<tr>
<td>complex</td>
<td>package-error</td>
<td>stream-error</td>
</tr>
<tr>
<td>concatenated-stream</td>
<td>parse-error</td>
<td>string</td>
</tr>
<tr>
<td>condition</td>
<td>pathname</td>
<td>string-stream</td>
</tr>
<tr>
<td>cons</td>
<td>print-not-readable</td>
<td>structure-class</td>
</tr>
<tr>
<td>control-error</td>
<td>program-error</td>
<td>structure-object</td>
</tr>
<tr>
<td>division-by-zero</td>
<td>random-state</td>
<td>style-warning</td>
</tr>
<tr>
<td>double-float</td>
<td>ratio</td>
<td>symbol</td>
</tr>
<tr>
<td>echo-stream</td>
<td>rational</td>
<td>synonym-stream</td>
</tr>
<tr>
<td>end-of-file</td>
<td>reader-error</td>
<td>t</td>
</tr>
<tr>
<td>error</td>
<td>readable</td>
<td>two-way-stream</td>
</tr>
<tr>
<td>extended-char</td>
<td>real</td>
<td>type-error</td>
</tr>
<tr>
<td>file-error</td>
<td>restart</td>
<td>unbound-slot</td>
</tr>
<tr>
<td>file-stream</td>
<td>sequence</td>
<td>unbound-variable</td>
</tr>
<tr>
<td>fixnum</td>
<td>serious-condition</td>
<td>undefined-function</td>
</tr>
<tr>
<td>float</td>
<td>short-float</td>
<td>unsigned-byte</td>
</tr>
<tr>
<td>floating-point-inexact</td>
<td>signed-byte</td>
<td>vector</td>
</tr>
<tr>
<td>floating-point-invalid-operation</td>
<td>simple-array</td>
<td>warning</td>
</tr>
<tr>
<td>floating-point-overflow</td>
<td>simple-base-string</td>
<td></td>
</tr>
<tr>
<td>floating-point-underflow</td>
<td>simple-bit-vector</td>
<td></td>
</tr>
</tbody>
</table>

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

Figure 4–4 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.

Figure 4–5 shows some defined names relating to types and declarations.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>coerce</td>
<td>defstruct</td>
<td>subtypep</td>
</tr>
<tr>
<td>declaim</td>
<td>deftype</td>
<td>the</td>
</tr>
<tr>
<td>declare</td>
<td>ftype</td>
<td>type</td>
</tr>
<tr>
<td>defclass</td>
<td>locally</td>
<td>type-of</td>
</tr>
<tr>
<td>define-condition</td>
<td>proclaim</td>
<td>typep</td>
</tr>
</tbody>
</table>

Figure 4–5. Defined names relating to types and declarations.

Figure 4–6 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.
<table>
<thead>
<tr>
<th>and</th>
<th>function</th>
<th>simple-array</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic-error</td>
<td>generic-function</td>
<td>simple-base-string</td>
</tr>
<tr>
<td>array</td>
<td>hash-table</td>
<td>simple-bit-vector</td>
</tr>
<tr>
<td>atom</td>
<td>integer</td>
<td>simple-condition</td>
</tr>
<tr>
<td>base-char</td>
<td>keyword</td>
<td>simple-error</td>
</tr>
<tr>
<td>base-string</td>
<td>list</td>
<td>simple-string</td>
</tr>
<tr>
<td>bignum</td>
<td>logical-pathname</td>
<td>simple-type-error</td>
</tr>
<tr>
<td>bit</td>
<td>long-float</td>
<td>simple-vector</td>
</tr>
<tr>
<td>bit-vector</td>
<td>member</td>
<td>simple-warning</td>
</tr>
<tr>
<td>broadcast-stream</td>
<td>method</td>
<td>single-float</td>
</tr>
<tr>
<td>built-in-class</td>
<td>method-combination</td>
<td>standard-char</td>
</tr>
<tr>
<td>cell-error</td>
<td>mod</td>
<td>standard-class</td>
</tr>
<tr>
<td>character</td>
<td>nil</td>
<td>standard-generic-function</td>
</tr>
<tr>
<td>class</td>
<td>not</td>
<td>standard-method</td>
</tr>
<tr>
<td>compiled-function</td>
<td>null</td>
<td>standard-object</td>
</tr>
<tr>
<td>complex</td>
<td>number</td>
<td>storage-condition</td>
</tr>
<tr>
<td>concatenated-stream</td>
<td>or</td>
<td>stream</td>
</tr>
<tr>
<td>condition</td>
<td>package</td>
<td>stream-error</td>
</tr>
<tr>
<td>cons</td>
<td>package-error</td>
<td>string</td>
</tr>
<tr>
<td>control-error</td>
<td>parse-error</td>
<td>string-stream</td>
</tr>
<tr>
<td>division-by-zero</td>
<td>pathname</td>
<td>structure-class</td>
</tr>
<tr>
<td>double-float</td>
<td>print-not-readable</td>
<td>structure-object</td>
</tr>
<tr>
<td>echo-stream</td>
<td>program-error</td>
<td>style-warning</td>
</tr>
<tr>
<td>end-of-file</td>
<td>random-state</td>
<td>symbol</td>
</tr>
<tr>
<td>eql</td>
<td>ratio</td>
<td>synonym-stream</td>
</tr>
<tr>
<td>error</td>
<td>rational</td>
<td>t</td>
</tr>
<tr>
<td>extended-char</td>
<td>reader-error</td>
<td>two-way-stream</td>
</tr>
<tr>
<td>file-error</td>
<td>real</td>
<td>type-error</td>
</tr>
<tr>
<td>file-stream</td>
<td>readable</td>
<td>unbound-slot</td>
</tr>
<tr>
<td>fixnum</td>
<td>restart</td>
<td>unbound-variable</td>
</tr>
<tr>
<td>float</td>
<td>satisfies</td>
<td>undefined-function</td>
</tr>
<tr>
<td>floating-point-inexact</td>
<td>sequence</td>
<td>unsigned-byte</td>
</tr>
<tr>
<td>floating-point-invalid-operation</td>
<td>serious-condition</td>
<td>values</td>
</tr>
<tr>
<td>floating-point-overflow</td>
<td>short-float</td>
<td>vector</td>
</tr>
<tr>
<td>floating-point-underflow</td>
<td>signed-byte</td>
<td>warning</td>
</tr>
</tbody>
</table>

Figure 4–6. Standardized TypeSpecifier Names
4.3 Classes

While the object system is general enough to describe all standardized classes (including, for example, number, hash-table, and symbol), Figure 4–7 contains a list of classes that are especially relevant to understanding the object system.

<table>
<thead>
<tr>
<th>built-in-class</th>
<th>method-combination</th>
<th>standard-object</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>standard-class</td>
<td>structure-class</td>
</tr>
<tr>
<td>generic-function</td>
<td>standard-generic-function</td>
<td>structure-object</td>
</tr>
<tr>
<td>method</td>
<td>standard-method</td>
<td></td>
</tr>
</tbody>
</table>

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, if 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 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

(defun C1 ()
  ((S1 :initform 5.4 :type number)
   (S2 :allocation :class)))

(defun 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

\[ R_C = \{(C, C_1), (C_1, C_2), \ldots, (C_{n-1}, C_n)\} \]

where \( C_1, \ldots, C_n \) 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 \( S_C \) be the set of C and its superclasses. Let R be

\[ R = \bigcup_{c \in S_C} R_c \]

The set R might or might not generate a partial ordering, depending on whether the \( R_c, c \in S_C \), are consistent; it is assumed that they are consistent and that R generates a partial ordering. When the \( R_c \) are not consistent, it is said that R is inconsistent.

To compute the class precedence list for C, topologically sort the elements of \( S_C \) 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 \( S_C \) 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 \( S_C \), and remove all pairs of the form \((C, D), D \in S_C\), 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 \( S_C \) 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 \( C_1, \ldots, C_n \) such that \( C_i \) precedes \( C_{i+1} \), \( 1 \leq i < n \), and \( C_n \) precedes \( C_1 \).

Sometimes there are several classes from \( S_C \) 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 \( R_c, c \in S_C \), are inconsistent.)
In more precise terms, let \( \{N_1, \ldots, N_m\}, m \geq 2, \) be the classes from \( S_C \) with no predecessors. Let \((C_1 \ldots C_n), n \geq 1, \) be the class precedence list constructed so far. \( C_1 \) is the most specific class, and \( C_n \) is the least specific. Let \( 1 \leq j \leq n \) be the largest number such that there exists an \( i \) where \( 1 \leq i \leq m \) and \( N_i \) is a direct superclass of \( C_j; N_i \) 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 \( T_1 \) and \( T_2 \) be subgraphs whose only element in common is the class \( J \). Suppose that no superclass of \( J \) appears in either \( T_1 \) or \( T_2 \), and that \( J \) is in the superclass chain of every class in both \( T_1 \) and \( T_2 \). Let \( C_1 \) be the bottom of \( T_1 \); and let \( C_2 \) be the bottom of \( T_2 \). Suppose \( C \) is a class whose direct superclasses are \( C_1 \) and \( C_2 \) in that order, then the class precedence list for \( C \) starts with \( C \) and is followed by all classes in \( T_1 \) except \( J \). All the classes of \( T_2 \) 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 \( \text{pie} \). The following classes are defined:

\[
\begin{align*}
(\text{defclass} & \quad \text{pie} \quad (\text{apple} \quad \text{cinnamon}) \quad ()) \\
(\text{defclass} & \quad \text{apple} \quad (\text{fruit}) \quad ()) \\
(\text{defclass} & \quad \text{cinnamon} \quad (\text{spice}) \quad ()) \\
(\text{defclass} & \quad \text{fruit} \quad (\text{food}) \quad ()) \\
(\text{defclass} & \quad \text{spice} \quad (\text{food}) \quad ()) \\
(\text{defclass} & \quad \text{food} \quad () \quad ())
\end{align*}
\]

The set \( S_{\text{pie}} = \{ \text{pie, apple, cinnamon, fruit, spice, food, standard-object, t} \} \). The set \( R = \{(\text{pie, apple}), (\text{apple, cinnamon}), (\text{apple, fruit}), (\text{cinnamon, spice}), (\text{fruit, food}), (\text{spice, food}), (\text{food, standard-object}), (\text{standard-object, t})\} \).

The class \( \text{pie} \) is not preceded by anything, so it comes first; the result so far is \( (\text{pie}) \). Remove \( \text{pie} \) from \( S \) and pairs mentioning \( \text{pie} \) from \( R \) to get \( S = \{ \text{apple, cinnamon, fruit, spice, food, standard-object, t} \} \) and \( R = \{ (\text{apple, cinnamon}), (\text{apple, fruit}), (\text{cinnamon, spice}), (\text{fruit, food}), (\text{spice, food}), (\text{food, standard-object}), (\text{standard-object, t}) \} \).

The class \( \text{apple} \) is not preceded by anything, so it is next; the result is \( (\text{pie apple}) \). Removing \( \text{apple} \) and the relevant pairs results in \( S = \{ \text{cinnamon, fruit, spice, food, standard-object, t} \} \) and \( R = \{ (\text{cinnamon, spice}), (\text{fruit, food}), (\text{spice, food}), (\text{food, standard-object}), (\text{standard-object, t}) \} \).

The classes \( \text{cinnamon} \) and \( \text{fruit} \) 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 \( \text{apple} \) 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). \( S = \{ \text{cinnamon, spice, food, standard-object, t} \}; \ R = \{ (\text{cinnamon, spice}), (\text{spice, food}),
(\text{food, standard-object}), (\text{standard-object, t}) \}.\)

The class cinnamon is next, giving the result so far as (pie apple fruit cinnamon). At this
point \( S = \{ \text{spice, food, standard-object, t} \}; \ R = \{ (\text{spice, food}), (\text{food, standard-object}),
(\text{standard-object, t}) \}.\)

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:

```lisp
(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:

```lisp
(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
matching generic function. Methods specified by the new defclass form are added.

4–14 Programming Language—Common Lisp
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 \( \text{eq} \). 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 \( \text{make-instances-obsolete} \). 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 specification, 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.
### 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`.

<table>
<thead>
<tr>
<th>arithmetic-error</th>
<th>generic-function</th>
<th>simple-error</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>hash-table</td>
<td>simple-type-error</td>
</tr>
<tr>
<td>bit-vector</td>
<td>integer</td>
<td>simple-warning</td>
</tr>
<tr>
<td>broadcast-stream</td>
<td>list</td>
<td>standard-class</td>
</tr>
<tr>
<td>built-in-class</td>
<td>logical-pathname</td>
<td>standard-generic-function</td>
</tr>
<tr>
<td>cell-error</td>
<td>method</td>
<td>standard-method</td>
</tr>
<tr>
<td>character</td>
<td>method-combination</td>
<td>standard-object</td>
</tr>
<tr>
<td>class</td>
<td>null</td>
<td>storage-condition</td>
</tr>
<tr>
<td>complex</td>
<td>number</td>
<td>stream</td>
</tr>
<tr>
<td>concatenated-stream</td>
<td>package</td>
<td>stream-error</td>
</tr>
<tr>
<td>condition</td>
<td>package-error</td>
<td>string</td>
</tr>
<tr>
<td>cons</td>
<td>parse-error</td>
<td>string-stream</td>
</tr>
<tr>
<td>control-error</td>
<td>pathname</td>
<td>structure-class</td>
</tr>
<tr>
<td>division-by-zero</td>
<td>print-not-readable</td>
<td>structure-object</td>
</tr>
<tr>
<td>echo-stream</td>
<td>program-error</td>
<td>style-warning</td>
</tr>
<tr>
<td>end-of-file</td>
<td>random-state</td>
<td>symbol</td>
</tr>
<tr>
<td>error</td>
<td>ratio</td>
<td>synonym-stream</td>
</tr>
<tr>
<td>file-error</td>
<td>rational</td>
<td>t</td>
</tr>
<tr>
<td>file-stream</td>
<td>reader-error</td>
<td>two-way-stream</td>
</tr>
<tr>
<td>float</td>
<td>readable</td>
<td>type-error</td>
</tr>
<tr>
<td>floating-point-inexact</td>
<td>real</td>
<td>unbound-slot</td>
</tr>
<tr>
<td>floating-point-invalid-operation</td>
<td>restart</td>
<td>unbound-variable</td>
</tr>
<tr>
<td>floating-point-overflow</td>
<td>sequence</td>
<td>undefined-function</td>
</tr>
<tr>
<td>floating-point-underflow</td>
<td>serious-condition</td>
<td>vector</td>
</tr>
<tr>
<td>function</td>
<td>simple-condition</td>
<td>warning</td>
</tr>
</tbody>
</table>
nil

Type

Supertypes:
   all types

Description:
The type nil contains no objects and so is also called the empty type. The type nil is a subtype of every type. No object is of type nil.

Notes:
The type containing the object nil is the type null, not the type nil.

boolean

Type

Supertypes:
   boolean, symbol, t

Description:
The type boolean contains the symbols t and nil, which represent true and false, respectively.

See Also:
t (constant variable), nil (constant variable), if, not, complement

Notes:
Conditional operations, such as if, permit the use of generalized booleans, not just booleans; any non-nil value, not just t, counts as true for a generalized boolean. However, as a matter of convention, the symbol t is considered the canonical value to use even for a generalized boolean when no better choice presents itself.

Types and Classes  4–19
function

function

System Class

Class Precedence List:
  function, t

Description:
  A function is an object that represents code to be executed when an appropriate number of arguments is supplied. A function is produced by the function special form, the function coerce, or the function compile. A function can be directly invoked by using it as the first argument to funcall, apply, or multiple-value-call.

Compound Type Specifier Kind:
  Specializing.

Compound Type Specifier Syntax:
  (function [arg-typespec [value-typespec]])

 arg-typespec::=({typespec}*  
 [optional {typespec}*]  
 [rest typespec]  
 [key {((keyword typespec)}*])

Compound Type Specifier Arguments:
  typespec—a type specifier.
  value-typespec—a type specifier.

Compound Type Specifier Description:
  The list form of the function type-specifier can be used only for declaration and not for discrimination. Every element of this type is a function that accepts arguments of the types specified by the arg-types and returns values that are members of the types specified by value-type. The optional, rest, key, and allow-other-keys markers can appear in the list of argument types. The type specifier provided with rest is the type of each actual argument, not the type of the corresponding variable.

  The key parameters should be supplied as lists of the form (keyword type). The keyword must be a valid keyword-name symbol as must be supplied in the actual arguments of a call. This is usually a symbol in the KEYWORD package but can be any symbol. When key is given in a function type specifier lambda list, the keyword parameters given are exhaustive unless allow-other-keys is also present. allow-other-keys is an indication that other keyword arguments might actually be supplied and, if supplied, can be used. For example, the type of the function make-list could be declared as follows:

4–20 Programming Language—Common Lisp
(function ((integer 0) &key (:initial-element t)) list)

The value-type can be a values type specifier in order to indicate the types of multiple values.

Consider a declaration of the following form:

(ftype (function (arg0-type arg1-type ...) val-type) f))

Any form (f arg0 arg1 ...) within the scope of that declaration is equivalent to the following:

(the val-type (f (the arg0-type arg0) (the arg1-type arg1) ...))

That is, the consequences are undefined if any of the arguments are not of the specified types or the result is not of the specified type. In particular, if any argument is not of the correct type, the result is not guaranteed to be of the specified type.

Thus, an ftype declaration for a function describes calls to the function, not the actual definition of the function.

Consider a declaration of the following form:

(type (function (arg0-type arg1-type ...) val-type) fn-valued-variable)

This declaration has the interpretation that, within the scope of the declaration, the consequences are unspecified if the value of fn-valued-variable is called with arguments not of the specified types; the value resulting from a valid call will be of type val-type.

As with variable type declarations, nested declarations imply intersections of types, as follows:

- Consider the following two declarations of ftype:

  (ftype (function (arg0-type1 arg1-type1 ...) val-type1) f))

  and

  (ftype (function (arg0-type2 arg1-type2 ...) val-type2) f))

  If both these declarations are in effect, then within the shared scope of the declarations, calls to f can be treated as if f were declared as follows:

  (ftype (function ((and arg0-type1 arg0-type2) (and arg1-type1 arg1-type2) ...)) (and val-type1 val-type2))

  f))

  It is permitted to ignore one or all of the ftype declarations in force.

- If two (or more) type declarations are in effect for a variable, and they are both function declarations, the declarations combine similarly.
compiled-function

Supertypes:
compiled-function, function, t

Description:
Any function may be considered by an implementation to be a compiled function if it contains no references to macros that must be expanded at run time, and it contains no unresolved references to load time values. See Section 3.2.2 (Compilation Semantics).

Functions whose definitions appear lexically within a file that has been compiled with compile-file and then loaded with load are of type compiled-function. Functions produced by the compile function are of type compiled-function. Other functions might also be of type compiled-function.

generic-function

System Class

Class Precedence List:
generic-function, function, t

Description:
A generic function is a function whose behavior depends on the classes or identities of the arguments supplied to it. A generic function object contains a set of methods, a lambda list, a method combination type, and other information. The methods define the class-specific behavior and operations of the generic function; a method is said to specialize a generic function. When invoked, a generic function executes a subset of its methods based on the classes or identities of its arguments.

A generic function can be used in the same ways that an ordinary function can be used; specifically, a generic function can be used as an argument to funcall and apply, and can be given a global or a local name.
standard-generic-function

System Class

Class Precedence List:
standard-generic-function, generic-function, function, t

Description:
The class standard-generic-function is the default class of generic functions established by defmethod, ensure-generic-function, defgeneric, and defclass forms.

class

System Class

Class Precedence List:
class, standard-object, t

Description:
The type class represents objects that determine the structure and behavior of their instances. Associated with an object of type class is information describing its place in the directed acyclic graph of classes, its slots, and its options.

built-in-class

System Class

Class Precedence List:
built-in-class, class, standard-object, t

Description:
A built-in class is a class whose instances have restricted capabilities or special representations. Attempting to use defclass to define subclasses of a built-in class signals an error of type error. Calling make-instance to create an instance of a built-in class signals an error of type error. Calling slot-value on an instance of a built-in class signals an error of type error. Redefining a built-in class or using change-class to change the class of an instance to or from a built-in class signals an error of type error. However, built-in classes can be used as parameter specializers in methods.
structure-class  

Class Precedence List:  
structure-class, class, standard-object, t

Description:  
All classes defined by means of defstruct are instances of the class structure-class.

standard-class  

Class Precedence List:  
standard-class, class, standard-object, t

Description:  
The class standard-class is the default class of classes defined by defclass.

method  

Class Precedence List:  
method, t

Description:  
A method is an object that represents a modular part of the behavior of a generic function.

A method contains code to implement the method’s behavior, a sequence of parameter specializers that specify when the given method is applicable, and a sequence of qualifiers that is used by the method combination facility to distinguish among methods. Each required parameter of each method has an associated parameter specializer, and the method will be invoked only on arguments that satisfy its parameter specializers.

The method combination facility controls the selection of methods, the order in which they are run, and the values that are returned by the generic function. The object system offers a default method combination type and provides a facility for declaring new types of method combination.

See Also:  
Section 7.6 (Generic Functions and Methods)
standard-method

Class Precedence List:
standard-method, method, standard-object, t

Description:
The class standard-method is the default class of methods defined by the defmethod and defgeneric forms.

structure-object

Class Precedence List:
structure-object, t

Description:
The class structure-object is an instance of structure-class and is a superclass of every class that is an instance of structure-class except itself, and is a superclass of every class that is defined by defstruct.

See Also:
defstruct, Section 2.4.8.13 (Sharpsign S), Section 22.1.3.12 (Printing Structures)

standard-object

Class Precedence List:
standard-object, t

Description:
The class standard-object is an instance of standard-class and is a superclass of every class that is an instance of standard-class except itself.
method-combination

System Class

Class Precedence List:
  method-combination, t

Description:
Every method combination object is an indirect instance of the class method-combination. A method combination object represents the information about the method combination being used by a generic function. A method combination object contains information about both the type of method combination and the arguments being used with that type.

t

System Class

Class Precedence List:
  t

Description:
The set of all objects. The type t is a supertype of every type, including itself. Every object is of type t.

satisfies

Type Specifier

Compound TypeSpecifier Kind:
  Predicating.

Compound TypeSpecifier Syntax:
  (satisfies predicate-name)

Compound TypeSpecifier Arguments:
  predicate-name—a symbol.

Compound TypeSpecifier Description:
This denotes the set of all objects that satisfy the predicate predicate-name, which must be a symbol whose global function definition is a one-argument predicate. A name is required for predicate-name; lambda expressions are not allowed. For example, the type specifier (and integer (satisfies evenp)) denotes the set of all even integers. The form (typep x '(satisfies p)) is equivalent to (if (p x) t nil).
The argument is required. The symbol * can be the argument, but it denotes itself (the symbol *), and does not represent an unspecified value.

The symbol satisfies is not valid as a type specifier.

**member**

**Type Specifier**

**Compound Type Specifier Kind:**
Combining.

**Compound Type Specifier Syntax:**
(member {object}*)

**Compound Type Specifier Arguments:**
object—an object.

**Compound Type Specifier Description:**
This denotes the set containing the named objects. An object is of this type if and only if it is eql to one of the specified objects.

The type specifiers (member) and nil are equivalent. * can be among the objects, but if so it denotes itself (the symbol *) and does not represent an unspecified value. The symbol member is not valid as a type specifier; and, specifically, it is not an abbreviation for either (member) or (member *).

**See Also:**
the type eql

**not**

**Type Specifier**

**Compound Type Specifier Kind:**
Combining.

**Compound Type Specifier Syntax:**
(not typespec)

**Compound Type Specifier Arguments:**
typespec—a type specifier.
Compound Type Specifier Description:
This denotes the set of all **objects** that are not of the **type** `typespec`.

The argument is required, and cannot be `*`.

The symbol **not** is not valid as a **type specifier**.

---

**and**

Type Specifier

Compound Type Specifier Kind:
Combining.

Compound Type Specifier Syntax:
`(and {typespec}*)`

Compound Type Specifier Arguments:
`typespec`—a **type specifier**.

Compound Type Specifier Description:
This denotes the set of all **objects** of the **type** determined by the intersection of the `typespecs`.

`*` is not permitted as an argument.

The `type specifiers` (and) and t are equivalent. The symbol **and** is not valid as a **type specifier**, and, specifically, it is not an abbreviation for `(and)`.

---

**or**

Type Specifier

Compound Type Specifier Kind:
Combining.

Compound Type Specifier Syntax:
`(or {typespec}*)`

Compound Type Specifier Arguments:
`typespec`—a **type specifier**.

---

4–28  Programming Language—Common Lisp
Compound Type Specifier Description:

This denotes the set of all objects of the type determined by the union of the typespecs. For example, the type list by definition is the same as (or null cons). Also, the value returned by position is an object of type (or null (integer 0 *)); i.e., either nil or a non-negative integer.

* is not permitted as an argument.

The type specifiers or and nil are equivalent. The symbol or is not valid as a type specifier; and, specifically, it is not an abbreviation for (or).

values

TypeSpecifier

Compound Type Specifier Kind:
Specializing.

Compound Type Specifier Syntax:
(values | value-typespec)

value-typespec::={typespec}* [optional {typespec}*] [rest typespec] [allow-other-keys]

Compound Type Specifier Arguments:
typespec—a type specifier.

Compound Type Specifier Description:

This type specifier can be used only as the value-type in a function type specifier or a the special form. It is used to specify individual types when multiple values are involved. The &optional and &rest markers can appear in the value-type list; they indicate the parameter list of a function that, when given to multiple-value-call along with the values, would correctly receive those values.

The symbol * may not be among the value-types.

The symbol values is not valid as a type specifier; and, specifically, it is not an abbreviation for (values).
**eql**

*Type Specifier*

**Compound Type Specifier Kind:**

Combining.

**Compound Type Specifier Syntax:**

\[(\text{eql} \ object)\]

**Compound Type Specifier Arguments:**

*object*—an *object*.

**Compound Type Specifier Description:**

Represents the *type* of all \[x\] for which \[(\text{eql} \ object \ x)\] is true.

The argument *object* is required. The *object* can be \[*\], but if so it denotes itself (the symbol \[*\]) and does not represent an unspecified value. The *symbol eql* is not valid as an *atomic type specifier*.

---

**coerce**

*Function*

**Syntax:**

\[
\text{coerce} \ object \ \text{result-type} \rightarrow \text{result}
\]

**Arguments and Values:**

*object*—an *object*.

*result-type*—a *type specifier*.

*result*—an *object*, of type *result-type* except in situations described in Section 12.1.5.3 (Rule of Canonical Representation for Complex Rationals).

**Description:**

*Coerces the object to type result-type.*

If *object* is already of type *result-type*, the *object* itself is returned, regardless of whether it would have been possible in general to coerce an *object* of some other *type* to *result-type*.

Otherwise, the *object* is coerced to type *result-type* according to the following rules:

---

4–30  Programming Language—Common Lisp
coerce

sequence

If the result-type is a recognizable subtype of list, and the object is a sequence, then the result is a list that has the same elements as object.

If the result-type is a recognizable subtype of vector, and the object is a sequence, then the result is a vector that has the same elements as object. If result-type is a specialized type, the result has an actual array element type that is the result of upgrading the element type part of that specialized type. If no element type is specified, the element type defaults to t. If the implementation cannot determine the element type, an error is signaled.

character

If the result-type is character and the object is a character designator, the result is the character it denotes.

complex

If the result-type is complex and the object is a real, then the result is obtained by constructing a complex whose real part is the object and whose imaginary part is the result of coercing an integer zero to the type of the object (using coerce). (If the real part is a rational, however, then the result must be represented as a rational rather than a complex; see Section 12.1.5.3 (Rule of Canonical Representation for Complex Rationals). So, for example, (coerce 3 'complex) is permissible, but will return 3, which is not a complex.)

float

If the result-type is any of float, short-float, single-float, double-float, long-float, and the object is a real, then the result is a float of type result-type which is equal in sign and magnitude to the object to whatever degree of representational precision is permitted by that float representation. (If the result-type is float and object is not already a float, then the result is a single float.)

function

If the result-type is function, and object is any function name that is fbound but that is globally defined neither as a macro name nor as a special operator, then the result is the functional value of object.

If the result-type is function, and object is a lambda expression, then the result is a closure of object in the null lexical environment.

t

Any object can be coerced to an object of type t. In this case, the object is simply returned.
Examples:

(coerce '(a b c) 'vector) → #(A B C)
(coerce 'a 'character) → #\A
(coerce 4.56 'complex) → #C(4.56 0.0)
(coerce 4.5s0 'complex) → #C(4.5s0 0.0s0)
(coerce 7/2 'complex) → 7/2
(coerce 0 'short-float) → 0.0s0
(coerce 3.5L0 'float) → 3.5L0
(coerce 7/2 'float) → 3.5
(coerce (cons 1 2) t) → (1 . 2)

All the following forms should signal an error:

(coerce '(a b c) '(vector * 4))
(coerce #(a b c) '(vector * 4))
(coerce '(a b c) '(vector * 2))
(coerce #(a b c) '(vector * 2))
(coerce "foo" '(string 2))
(coerce #(#\a #\b #\c) '(string 2))
(coerce '(0 1) '(simple-bit-vector 3))

Exceptional Situations:
If a coercion is not possible, an error of type type-error is signaled.

(coerce x 'nil) always signals an error of type type-error.

An error of type error is signaled if the result-type is function but object is a symbol that is not fbound or if the symbol names a macro or a special operator.

An error of type type-error should be signaled if result-type specifies the number of elements and object is of a different length.

See Also:
rational, floor, char-code, char-int

Notes:
Coercions from floats to rationals and from ratios to integers are not provided because of rounding problems.

(coerce x 't) ≡ (identity x) ≡ x
deftype

Syntax:

```
deftype name lambda-list [ {declaration}* | documentation ] {form}* → name
```

Arguments and Values:

- `name`—a symbol.
- `lambda-list`—a deftype lambda list.
- `declaration`—a `declare` expression; not evaluated.
- `documentation`—a string; not evaluated.
- `form`—a form.

Description:

`deftype` defines a derived type specifier named `name`.

The meaning of the new `type specifier` is given in terms of a function which expands the `type specifier` into another `type specifier`, which itself will be expanded if it contains references to another derived type specifier.

The newly defined `type specifier` may be referenced as a list of the form `(name arg_1 arg_2 ...)`.

The number of arguments must be appropriate to the `lambda-list`. If the new `type specifier` takes no arguments, or if all of its arguments are optional, the `type specifier` may be used as an atomic `type specifier`.

The argument expressions to the `type specifier`, `arg_1` ... `arg_n`, are not evaluated. Instead, these literal objects become the objects to which corresponding parameters become bound.

The body of the `deftype` form (but not the `lambda-list`) is implicitly enclosed in a `block` named `name`, and is evaluated as an `implicit progn`, returning a new `type specifier`.

The lexical environment of the body is the one which was current at the time the `deftype` form was evaluated, augmented by the variables in the `lambda-list`.

Recursive expansion of the `type specifier` returned as the expansion must terminate, including the expansion of `type specifiers` which are nested within the expansion.

The consequences are undefined if the result of fully expanding a `type specifier` contains any circular structure, except within the objects referred to by `member` and `eql` `type specifiers`.

`Documentation` is attached to `name` as a `documentation string` of kind `type`.

If a `deftype` form appears as a `top level form`, the `compiler` must ensure that the `name` is recognized in subsequent `type` declarations. The `programmer` must ensure that the body of a `deftype` form can be `evaluated` at compile time if the `name` is referenced in subsequent `type` declarations.
If the expansion of a type specifier is not defined fully at compile time (perhaps because it expands into an unknown type specifier or a satisfies of a named function that isn’t defined in the compile-time environment), an implementation may ignore any references to this type in declarations and/or signal a warning.

Examples:

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

(defun square-matrix (&optional type size)
  '(and (array ,type (,size ,size))
        (satisfies equidimensional))) ) → SQUARE-MATRIX
```

See Also:

declare, defmacro, documentation, Section 4.2.3 (Type Specifiers), Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

**subtypep**

*Function*

**Syntax:**

```
subtypep type-1 type-2 &optional environment → subtype-p, valid-p
```

**Arguments and Values:**

- **type-1**—a type specifier.
- **type-2**—a type specifier.
- **environment**—an environment object. The default is nil, denoting the null lexical environment and the current global environment.
- **subtype-p**—a generalized boolean.
- **valid-p**—a generalized boolean.

**Description:**

If type-1 is a recognizable subtype of type-2, the first value is true. Otherwise, the first value is false, indicating that either type-1 is not a subtype of type-2, or else type-1 is a subtype of type-2 but is not a recognizable subtype.

A second value is also returned indicating the ‘certainty’ of the first value. If this value is true, then the first value is an accurate indication of the subtype relationship. (The second value is always true when the first value is true.)
Figure 4–9 summarizes the possible combinations of values that might result.

<table>
<thead>
<tr>
<th>Value 1</th>
<th>Value 2</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>type-1 is definitely a subtype of type-2.</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>type-1 is definitely not a subtype of type-2.</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>subtypep could not determine the relationship, so type-1 might or might not be a subtype of type-2.</td>
</tr>
</tbody>
</table>

Figure 4–9. Result possibilities for subtypep

subtypep is permitted to return the values false and false only when at least one argument involves one of these type specifiers: and, eql, the list form of function, member, not, or, satisfies, or values. (A type specifier `involves` such a symbol if, after being type expanded, it contains that symbol in a position that would call for its meaning as a type specifier to be used.) One consequence of this is that if neither type-1 nor type-2 involves any of these type specifiers, then subtypep is obliged to determine the relationship accurately. In particular, subtypep returns the values true and true if the arguments are equal and do not involve any of these type specifiers.

subtypep never returns a second value of nil when both type-1 and type-2 involve only the names in Figure 4–2, or names of types defined by defstruct, define-condition, or defclass, or derived types that expand into only those names. While type specifiers listed in Figure 4–2 and names of defclass and defstruct can in some cases be implemented as derived types, subtypep regards them as primitive.

The relationships between types reflected by subtypep are those specific to the particular implementation. For example, if an implementation supports only a single type of floating-point numbers, in that implementation (subtypep 'float 'long-float) returns the values true and true (since the two types are identical).

For all T1 and T2 other than *, (array T1) and (array T2) are two different type specifiers that always refer to the same sets of things if and only if they refer to arrays of exactly the same specialized representation, i.e., if (upgraded-array-element-type 'T1) and (upgraded-array-element-type 'T2) return two different type specifiers that always refer to the same sets of objects. This is another way of saying that `(array type-specifier)` and `(array , (upgraded-array-element-type 'type-specifier))` refer to the same set of specialized array representations. For all T1 and T2 other than *, the intersection of (array T1) and (array T2) is the empty set if and only if they refer to arrays of different, distinct specialized representations.

Therefore,

(subtypep `(array T1) `(array T2)) → true

if and only if

(upgraded-array-element-type 'T1) and
(upgraded-array-element-type 'T2)
**subtypep**

return two different type specifiers that always refer to the same sets of objects.

For all type-specifiers \( T_1 \) and \( T_2 \) other than \( * \),

\[
\text{subtypep} \ (\text{complex } T_1) \ (\text{complex } T_2) \rightarrow \text{true}, \text{true}
\]

if:

1. \( T_1 \) is a subtype of \( T_2 \), or
2. \((\text{upgraded-complex-part-type } \ T_1)\) and \((\text{upgraded-complex-part-type } \ T_2)\) return two different type specifiers that always refer to the same sets of objects; in this case, \((\text{complex } T_1)\) and \((\text{complex } T_2)\) both refer to the same specialized representation.

The values are false and true otherwise.

The form

\[
\text{subtypep} \ (\text{complex single-float}) \ (\text{complex float}))
\]

must return true in all implementations, but

\[
\text{subtypep} \ (\text{array single-float}) \ (\text{array float}))
\]

returns true only in implementations that do not have a specialized array representation for single floats distinct from that for other floats.

Examples:

\[
\begin{align*}
(\text{subtypep} \ '\text{compiled-function} \ '\text{function}) & \rightarrow \text{true}, \text{true} \\
(\text{subtypep} \ '\text{null} \ '\text{list}) & \rightarrow \text{true}, \text{true} \\
(\text{subtypep} \ '\text{null} \ '\text{symbol}) & \rightarrow \text{true}, \text{true} \\
(\text{subtypep} \ '\text{integer} \ '\text{string}) & \rightarrow \text{false}, \text{true} \\
(\text{subtypep} \ '(\text{satisfies dummy}) \ nil) & \rightarrow \text{false}, \text{implementation-dependent} \\
(\text{subtypep} \ '(\text{integer 1 3}) \ '(\text{integer 1 4})) & \rightarrow \text{true}, \text{true} \\
(\text{subtypep} \ '(\text{integer} \ (0) \ (0)) \ 'nil) & \rightarrow \text{true}, \text{true} \\
(\text{subtypep} \ 'nil \ '(\text{integer} \ (0) \ (0))) & \rightarrow \text{true}, \text{true} \\
(\text{subtypep} \ '(\text{integer} \ (0) \ (0)) \ '(\text{member})) & \rightarrow \text{true}, \text{true}; \text{or false}, \text{false} \\
(\text{subtypep} \ '(\text{member}) \ 'nil) & \rightarrow \text{true}, \text{true}; \text{or false}, \text{false} \\
(\text{subtypep} \ 'nil \ '(\text{member})) & \rightarrow \text{true}, \text{true}; \text{or false}, \text{false}
\end{align*}
\]

Let \(<\text{aet-x}>\) and \(<\text{aet-y}>\) be two distinct type specifiers that do not always refer to the same sets of objects in a given implementation, but for which make-array, will return an object of the same array type.

Thus, in each case,

\[
\begin{align*}
(\text{subtypep} \ (\text{array-element-type} \ (\text{make-array} \ 0 \ :\text{element-type} \ '<\text{aet-x}>')))
& \rightarrow \text{true}, \text{true} \\
(\text{array-element-type} \ (\text{make-array} \ 0 \ :\text{element-type} \ '<\text{aet-y}>'))
& \rightarrow \text{true}, \text{true}
\end{align*}
\]

4–36  Programming Language—Common Lisp
(subtypep (array-element-type (make-array 0 :element-type '<aet-y>))
  (array-element-type (make-array 0 :element-type '<aet-x>)))
→ true, true

If (array <aet-x>) and (array <aet-y>) are different names for exactly the same set of objects,
these names should always refer to the same sets of objects. That implies that the following set of
tests are also true:

(subtypep '(array <aet-x>) '(array <aet-y>)) → true, true
(subtypep '(array <aet-y>) '(array <aet-x>)) → true, true

See Also:
  Section 4.2 (Types)

Notes:
The small differences between the subtypep specification for the array and complex types are
necessary because there is no creation function for complexes which allows the specification of the
resultant part type independently of the actual types of the parts. Thus in the case of the type
complex, the actual type of the parts is referred to, although a number can be a member of more
than one type. For example, 17 is of type (mod 18) as well as type (mod 256) and type integer; and
2.3f5 is of type single-float as well as type float.

**type-of**

*Function*

**Syntax:**

```
type-of object  → typespec
```

**Arguments and Values:**

- `object`—an object.
- `typespec`—a type specifier.

**Description:**

Returns a type specifier, `typespec`, for a type that has the `object` as an element. The `typespec`
satisfies the following:

1. For any `object` that is an element of some built-in type:
   a. the type returned is a recognizable subtype of that built-in type.
   b. the type returned does not involve `and`, `eql`, `member`, `not`, `or`, `satisfies`, or `values`.

Types and Classes 4–37
type-of

2. For all objects, (typep object (type-of object)) returns true. Implicit in this is that type specifiers which are not valid for use with typep, such as the list form of the function type specifier, are never returned by type-of.

3. The type returned by type-of is always a recognizable subtype of the class returned by class-of. That is,

   (subtypep (type-of object) (class-of object)) → true, true

4. For objects of metaclass structure-class or standard-class, and for conditions, type-of returns the proper name of the class returned by class-of if it has a proper name, and otherwise returns the class itself. In particular, for objects created by the constructor function of a structure defined with defstruct without a :type option, type-of returns the structure name; and for objects created by make-condition, the typespec is the name of the condition type.

5. For each of the types short-float, single-float, double-float, or long-float of which the object is an element, the typespec is a recognizable subtype of that type.

Examples:

   (type-of 'a) → SYMBOL
   (type-of '(1 . 2)) → CONS
   (CONS FIXNUM FIXNUM)
   (type-of #c(0 1)) → COMPLEX
   (COMPLEX INTEGER)
   (defstruct temp-struct x y z) → TEMP-STRUCT
   (type-of (make-temp-struct)) → TEMP-STRUCT
   (type-of "abc") → STRING
   (STRING 3)
   (subtypep (type-of "abc") 'string) → true, true
   (type-of (expt 2 40)) → BIGNUM
   INTEGER
   (INTEGER 1099511627776 1099511627776)
   SYSTEM::TWO-WORD-BIGNUM
   FIXNUM
   (subtypep (type-of 112312) 'integer) → true, true
   (defvar *foo* (make-array 5 :element-type t)) → *FOO*
   (class-name (class-of *foo*)) → VECTOR
(type-of *foo*)
  → VECTOR
  or (VECTOR T 5)

See Also:
array-element-type, class-of, defstruct, typecase, typep, Section 4.2 (Types)

Notes:
Implementors are encouraged to arrange for type-of to return a portable value.

### typep

**Function**

**Syntax:**

\[ \text{typep } \text{object } \text{type-specifier} \ \&\text{optional} \ \text{environment} \ → \ generalized-boolean \]

**Arguments and Values:**

- **object**—an object.
- **type-specifier**—any type specifier except values, or a type specifier list whose first element is either function or values.
- **environment**—an environment object. The default is nil, denoting the null lexical environment and the and current global environment.
- **generalized-boolean**—a generalized boolean.

**Description:**

Returns true if object is of the type specified by type-specifier; otherwise, returns false.

A type-specifier of the form (satisfies fn) is handled by applying the function fn to object.

\[(\text{typep } \text{object } '(\text{array } \text{type-specifier})), \text{ where type-specifier is not } *, \text{ returns true if and only if } \text{object is an array that could be the result of supplying type-specifier as the :element-type argument to make-array}. \ (\text{array } *) \text{ refers to all arrays regardless of element type, while } (\text{array } \text{type-specifier}) \text{ refers only to those arrays that can result from giving type-specifier as the :element-type argument to make-array}. \ A \text{ similar interpretation applies to } (\text{simple-array } \text{type-specifier}) \text{ and } (\text{vector } \text{type-specifier}). \text{ See Section 15.1.2.1 (Array Upgrading)}.\]

\[(\text{typep } \text{object } '(\text{complex } \text{type-specifier})) \text{ returns true for all complex numbers that can result from giving numbers of type type-specifier to the function complex, plus all other complex numbers of the same specialized representation. Both the real and the imaginary parts of any such complex number must satisfy:}\]

\[(\text{typep realpart } \text{type-specifier})\]
(typep imagpart 'type-specifier)

See the function upgraded-complex-part-type.

Examples:

(typep 12 'integer) → true
(typep (+ most-positive-fixnum) 'fixnum) → false
(typep nil t) → true
(typep nil nil) → false
(typep 1 '(mod 2)) → true
(typep #c(1 1) '(complex (eql 1))) → true
;; To understand this next example, you might need to refer to
;; Section 12.1.5.3 (Rule of Canonical Representation for Complex Rationals).
(typep #c(0 0) '(complex (eql 0))) → false

Let \( A_x \) and \( A_y \) be two type specifiers that denote different types, but for which

(upgraded-array-element-type '\( A_x \))

and

(upgraded-array-element-type '\( A_y \))

denote the same type. Notice that

(typep (make-array 0 :element-type '\( A_x \)) '(array \( A_x \))) → true
(typep (make-array 0 :element-type '\( A_y \)) '(array \( A_y \))) → true
(typep (make-array 0 :element-type '\( A_x \)) '(array \( A_y \))) → true
(typep (make-array 0 :element-type '\( A_y \)) '(array \( A_x \))) → true

Exceptional Situations:

An error of type error is signaled if type-specifier is values, or a type specifier list whose first
element is either function or values.

The consequences are undefined if the type-specifier is not a type specifier.

See Also:

type-of, upgraded-array-element-type, upgraded-complex-part-type, Section 4.2.3 (Type Specifiers)

Notes:

Implementations are encouraged to recognize and optimize the case of (typep \( x \) (the class \( y \))),
since it does not involve any need for expansion of deftype information at runtime.
type-error

Condition Type

Class Precedence List:
  type-error, error, serious-condition, condition, t

Description:
The type type-error represents a situation in which an object is not of the expected type. The “offending datum” and “expected type” are initialized by the initialization arguments named :datum and :expected-type to make-condition, and are accessed by the functions type-error-datum and type-error-expected-type.

See Also:
type-error-datum, type-error-expected-type

type-error-datum, type-error-expected-type Function

Syntax:
  type-error-datum condition → datum
  type-error-expected-type condition → expected-type

Arguments and Values:
  condition—a condition of type type-error.
  datum—an object.
  expected-type—a type specifier.

Description:
type-error-datum returns the offending datum in the situation represented by the condition.
type-error-expected-type returns the expected type of the offending datum in the situation represented by the condition.

Examples:
(defun fix-digits (condition)
  (check-type condition type-error)
  (let* ((digits '(zero one two three four five six seven eight nine)))
(val (position (type-error-datum condition) digits)))
(if (and val (subtypep 'fixnum (type-error-expected-type condition)))
  (store-value 7)))

(defun foo (x)
  (handler-bind ((type-error #'fix-digits))
    (check-type x number)
    (+ x 3)))

(foo 'seven)
→ 10

See Also:
type-error, Chapter 9 (Conditions)

simple-type-error

Class Precedence List:
simple-type-error, simple-condition, type-error, error, serious-condition, condition, t

Description:
Conditions of type simple-type-error are like conditions of type type-error, except that they provide an alternate mechanism for specifying how the condition is to be reported; see the type simple-condition.

See Also:
simple-condition, simple-condition-format-control, simple-condition-format-arguments, type-error-datum, type-error-expected-type
Programming Language—Common Lisp

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.

Figure 5–1 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.

<table>
<thead>
<tr>
<th>Access function</th>
<th>Update Function</th>
<th>Update using setf</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>(setq x datum)</td>
<td>(setf x datum)</td>
</tr>
<tr>
<td>(car x)</td>
<td>(rplaca x datum)</td>
<td>(setf (car x) datum)</td>
</tr>
<tr>
<td>(symbol-value x)</td>
<td>(set x datum)</td>
<td>(setf (symbol-value x) datum)</td>
</tr>
</tbody>
</table>

Figure 5–1. Examples of setf

Figure 5–2 shows **operators** relating to **places** and **generalized reference**.

| assert          | defsetf          | push             |
| ccase           | get-setf-expansion | remf             |
| ctypecase       | getf             | rotatef          |
| def             | 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, rotatf, 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, rotatf, psetf, setf, 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))

5–2 Programming Language—Common Lisp
(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 self 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 self 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 \((\text{ldb } bs (\text{car } exp))\):

\[
\begin{align*}
(g0001 & \ g0002) & \quad \text{:list of temporary variables} \\
(bs & \ exp) & \quad \text{:list of value forms} \\
(g0003) & \quad \text{:list of store variables} \\
(&\text{progn } (\text{rplaca } g0002 (\text{dpb } g0003 \ g0001 (\text{car } g0002))) g0003) & \quad \text{:storing form} \\
&(\text{ldb } g0001 (\text{car } g0002)) & \quad \text{:accessing form}
\end{align*}
\]

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 Figure 5–7.
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 Figure 5–8, 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.
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.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Argument that is a place</th>
<th>Update function used</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldb</td>
<td>second</td>
<td>dpb</td>
</tr>
<tr>
<td>mask-field</td>
<td>second</td>
<td>deposit-field</td>
</tr>
<tr>
<td>getf</td>
<td>first</td>
<td>implementation-dependent</td>
</tr>
</tbody>
</table>

ld

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 neces-
sarily 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 a (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

5–8 Programming Language—Common Lisp
\[ s \to ((A \ 1 \ B \ 6 \ C \ 3)) \]

;;; 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

\[
\text{(setf (values place-1 \ldots place-n) values-form)}
\]

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 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,

\[
\text{(setf (the integer (cadr x)) (+ y 3))}
\]

is processed as if it were

\[
\text{(setf (cadr x) (the integer (+ y 3)))}
\]
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:

\[
\text{(setf (apply #'name} {arg}* val)} \equiv \text{(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 \texttt{setf} form expands into a call to the function named \( \texttt{setf} \( f \) \). The first argument in the newly constructed function form is \texttt{new-value} and the remaining arguments are the remaining elements of \texttt{place}. This expansion occurs regardless of whether \( f \) or \( \texttt{setf} \( f \) \) is defined as a function locally, globally, or not at all. For example,

\[
(\texttt{setf} \ (f \ \texttt{arg1} \ \texttt{arg2} \ \ldots) \ \texttt{new-value})
\]

expands into a form with the same effect and value as

\[
(\texttt{let} \ ((#:temp-1 \ \texttt{arg1}) ;\text{force correct order of evaluation}
\ (#:temp-2 \ \texttt{arg2})
\ \ldots
\ (#:temp-0 \ \texttt{new-value}))
\ (\texttt{funcall} \ (\texttt{function} \ (\texttt{setf} \ (f))) \ #:temp-0 \ #:temp-1 \ #:temp-2))
\]

A function named \( \texttt{setf} \( f \) \) must return its first argument as its only value in order to preserve the semantics of \texttt{setf}.

5.1.3 Treatment of Other Macros Based on SETF

For each of the “read-modify-write” operators in Figure 5–9, and for any additional macros defined by the programmer using \texttt{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 \texttt{place} argument, the actual read of the “old value” from that \texttt{place} happens after all of the argument form evaluations, and just before a “new value” is computed and written back into the \texttt{place}.

Specifically, each of these operators can be viewed as involving a form with the following general syntax:

\[
(\texttt{operator} \ \{\texttt{preceding-form}\}^* \ \texttt{place} \ \{\texttt{following-form}\}^*)
\]

The evaluation of each such form proceeds like this:

1. \textit{Evaluate} each of the \texttt{preceding-forms}, in left-to-right order.
2. \textit{Evaluate} the subforms of the \texttt{place}, in the order specified by the second value of the \texttt{setf} expansion for that \texttt{place}.
3. \textit{Evaluate} each of the \texttt{following-forms}, in left-to-right order.
4. \textit{Read} the old value from \texttt{place}.
5. Compute the new value.
6. Store the new value into \texttt{place}.
<table>
<thead>
<tr>
<th>decf</th>
<th>pop</th>
<th>pushnew</th>
</tr>
</thead>
<tbody>
<tr>
<td>incf</td>
<td>push</td>
<td>remf</td>
</tr>
</tbody>
</table>

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.
apply

Function

Syntax:
apply function &rest args\(^+\) \(\rightarrow\) \{result\}\(*\)

Arguments and Values:
- function—a function designator.
- args—a spreadable argument list designator.
- results—the values returned by function.

Description:
Applies the function to the args.

When the function receives its arguments via &rest, it is permissible (but not required) for the implementation to bind the rest parameter to an object that shares structure with the last argument to apply. Because a function can neither detect whether it was called via apply nor whether (if so) the last argument to apply was a constant, conforming programs must neither rely on the list structure of a rest list to be freshly consed, nor modify that list structure.

setf can be used with apply in certain circumstances; see Section 5.1.2.5 (APPLY Forms as Places).

Examples:

```lisp
(setq f '+) \(\rightarrow\) +
(apply f '(1 2)) \(\rightarrow\) 3
(setq f '#-) \(\rightarrow\) #<FUNCTION ->
(apply f '(1 2)) \(\rightarrow\) -1
(apply #'max 3 5 '(2 7 3)) \(\rightarrow\) 7
(apply 'cons '((+ 2 3) 4)) \(\rightarrow\) ((+ 2 3) . 4)
(apply #'+ '()) \(\rightarrow\) 0
(deparameter *some-list* '(a b c))
(defun strange-test (&rest x) (eq x *some-list*))
(apply #'strange-test *some-list*) \(\rightarrow\) implementation-dependent

defun bad-boy (&rest x) (rplacd x 'y))
(bad-boy 'a 'b 'c) has undefined consequences.
(apply #'bad-boy *some-list*) has undefined consequences.

(defvar *some-list* '(a b c))
(defun foo (size &rest keys &key double &allow-other-keys)
  (let ((v (apply #'make-array size :allow-other-keys t keys)))
    (if double (concatenate (type-of v) v v) v)))
```

5–14 Programming Language—Common Lisp
(foo 4 :initial-contents '(a b c d) :double t)  
→ #(A B C D A B C D)

See Also:  
funcall, fdefinition, function, Section 3.1 (Evaluation), Section 5.1.2.5 (APPLY Forms as Places)

defun

Syntax:
defun function-name lambda-list [{declaration}* | documentation] {form}*  
→ function-name

Arguments and Values:
function-name—a function name.
lambda-list—an ordinary lambda list.
declaration—a declare expression; not evaluated.
documentation—a string; not evaluated.
forms—an implicit progn.
block-name—the function block name of the function-name.

Description:  
Defines a new function named function-name in the global environment. The body of the function defined by defun consists of forms; they are executed as an implicit progn when the function is called. defun can be used to define a new function, to install a corrected version of an incorrect definition, to redefine an already-defined function, or to redefine a macro as a function.

defun implicitly puts a block named block-name around the body forms (but not the forms in the lambda-list) of the function defined.

Documentation is attached as a documentation string to name (as kind function) and to the function object.

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

(l lambda lambda-list
  [{declaration}* | documentation]
  (block block-name {form}*)))

processed in the lexical environment in which defun was executed.
defun

( None of the arguments are evaluated at macro expansion time. )

defun is not required to perform any compile-time side effects. In particular, defun does not make the function definition available at compile time. An implementation may choose to store information about the function for the purposes of compile-time error-checking (such as checking the number of arguments on calls), or to enable the function to be expanded inline.

Examples:

(defun recur (x)
  (when (> x 0)
    (recur (1- x))) → RECUR
)(defun ex (a b &optional c (d 66) &rest keys &key test (start 0))
  (list a b c d keys test start)) → EX
(ex 1 2) → (1 2 NIL 66 NIL 0)
(ex 1 2 3 4 :test 'equal :start 50)
  → (1 2 3 4 (:TEST EQUAL :START 50) EQUAL 50)
(ex :test 1 :start 2) → (:TEST 1 :START 2 NIL NIL 0)

;; This function assumes its callers have checked the types of the
;; arguments, and authorizes the compiler to build in that assumption.
(defun discriminant (a b c)
  (declare (number a b c))
  "Compute the discriminant for a quadratic equation."
  (- (* b b) (* 4 a c)) → DISCRIMINANT
(discriminant 1 2/3 -2) → 76/9

;; This function assumes its callers have not checked the types of the
;; arguments, and performs explicit type checks before making any assumptions.
(defun careful-discriminant (a b c)
  "Compute the discriminant for a quadratic equation."
  (check-type a number)
  (check-type b number)
  (check-type c number)
  (locally (declare (number a b c))
    (- (* b b) (* 4 a c))) → CAREFUL-DISCRIMINANT
(careful-discriminant 1 2/3 -2) → 76/9

See Also:

flet, labels, block, return-from, declare, documentation, Section 3.1 (Evaluation), Section 3.4.1 (Ordinary Lambda Lists), Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

Notes:

return-from can be used to return prematurely from a function defined by defun.

5–16 Programming Language—Common Lisp
Additional side effects might take place when additional information (typically debugging information) about the function definition is recorded.

**fdefinition**

*Accessor*

**Syntax:**

```lisp
fdefinition function-name → definition
(setf (fdefinition function-name) new-definition)
```

**Arguments and Values:**

- `function-name`—a function name. In the non-setf case, the name must be fbound in the global environment.
- `definition`—Current global function definition named by `function-name`.
- `new-definition`—a function.

**Description:**

`fdefinition` accesses the current global function definition named by `function-name`. The definition may be a function or may be an object representing a special form or macro. The value returned by `fdefinition` when `fboundp` returns true but the `function-name` denotes a macro or special form is not well-defined, but `fdefinition` does not signal an error.

**Exceptional Situations:**

Should signal an error of type `type-error` if `function-name` is not a function name.

An error of type `undefined-function` is signaled in the non-setf case if `function-name` is not fbound.

**See Also:**

`fboundp`, `fmakunbound`, `macro-function`, `special-operator-p`, `symbol-function`

**Notes:**

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

- `setf` can be used with `fdefinition` to replace a global function definition when the `function-name`’s function definition does not represent a special form. `setf` of `fdefinition` requires a function as the new value. It is an error to set the `fdefinition` of a `function-name` to a symbol, a list, or the value returned by `fdefinition` on the name of a macro or special form.
fboundp

Syntax:
   fboundp name → generalized-boolean

Pronunciation:
   [ˌɪfˈbaʊndp]  

Arguments and Values:
   name—a function name.
   generalized-boolean—a generalized boolean.

Description:
   Returns true if name is fbound; otherwise, returns false.

Examples:
   (fboundp 'car) → true
   (fboundp 'nth-value) → false
   (fboundp 'with-open-file) → true
   (fboundp 'unwind-protect) → true
   (defun my-function (x) x) → MY-FUNCTION
   (fboundp 'my-function) → true
   (let ((saved-definition (symbol-function 'my-function)))
      (unwind-protect (progn (fmakunbound 'my-function)
                                 (fboundp 'my-function))
                      (setf (symbol-function 'my-function) saved-definition)))
      → false
   (fboundp 'my-function) → true
   (defmacro my-macro (x) ',x) → MY-MACRO
   (fboundp 'my-macro) → true
   (fmakunbound 'my-function) → MY-FUNCTION
   (fboundp 'my-function) → false
   (flet ((my-function (x) x))
      (fboundp 'my-function)) → false

Exceptional Situations:
   Should signal an error of type type-error if name is not a function name.

See Also:
   symbol-function, fmakunbound, fdefinition
Notes:

It is permissible to call `symbol-function` on any `symbol` that is `fbound`.

`fboundp` is sometimes used to “guard” an access to the `function cell`, as in:

```
(if (fboundp x)
  (symbol-function x))
```

Defining a `setf expander` F does not cause the `setf function` (setf F) to become defined.

---

### fmakunbound

Function

Syntax:

```
fmakunbound name  →  name
```

Pronunciation:

```
[ˌɛfˈmɑːkənˈbaʊnd] or [ˌɛfˈməkənˈbaʊnd]
```

Arguments and Values:

- `name`—a `function name`.

Description:

Removes the `function` or `macro definition`, if any, of `name` in the `global environment`.

Examples:

```
(defun add-some (x) (+ x 19))  →  ADD-SOME
(fboundp 'add-some)  →  true
(flet ((add-some (x) (+ x 37)))
  (fmakunbound 'add-some)
  (add-some 1))  →  38
(fboundp 'add-some)  →  false
```

Exceptional Situations:

Should signal an error of `type type-error` if `name` is not a `function name`.

The consequences are undefined if `name` is a `special operator`.

See Also:

- `fboundp`, `makunbound`
flet, labels, macrolet

Syntax:

flet ((function-name lambda-list [ (local-declaration)* | local-documentation ] {local-form}*))* → {result}*

labels ((function-name lambda-list [ (local-declaration)* | local-documentation ] {local-form})*)* → {result}*

macrolet ((name lambda-list [ (local-declaration)* | local-documentation ] {local-form})*)* → {result}*

Arguments and Values:

function-name—an function name.

name—a symbol.

lambda-list—a lambda list; for flet and labels, it is an ordinary lambda list; for macrolet, it is a macro lambda list.

local-declaration—a declare expression; not evaluated.

declaration—a declare expression; not evaluated.

local-documentation—a string; not evaluated.

local-forms, forms—an implicit progn.

results—the values of the forms.

Description:

flet, labels, and macrolet define local functions and macros, and execute forms using the local definitions. Forms are executed in order of occurrence.

The body forms (but not the lambda list) of each function created by flet and labels and each macro created by macrolet are enclosed in an implicit block whose name is the function block name of the function-name or name, as appropriate.

The scope of the declarations between the list of local function/macro definitions and the body forms in flet and labels does not include the bodies of the locally defined functions, except that...
for labels, any inline, notinline, or ftype declarations that refer to the locally defined functions do apply to the local function bodies. That is, their scope is the same as the function name that they affect. The scope of these declarations does not include the bodies of the macro expander functions defined by macrolet.

flet

flet defines locally named functions and executes a series of forms with these definition bindings. Any number of such local functions can be defined.

The scope of the name binding encompasses only the body. Within the body of flet, function-names matching those defined by flet refer to the locally defined functions rather than to the global function definitions of the same name. Also, within the scope of flet, global self expander definitions of the function-name defined by flet do not apply. Note that this applies to (defsetf f ...), not (defmethod (setf f) ...).

The names of functions defined by flet are in the lexical environment; they retain their local definitions only within the body of flet. The function definition bindings are visible only in the body of flet, not the definitions themselves. Within the function definitions, local function names that match those being defined refer to functions or macros defined outside the flet. flet can locally shadow a global function name, and the new definition can refer to the global definition.

Any local-documentation is attached to the corresponding local function (if one is actually created) as a documentation string.

labels

labels is equivalent to flet except that the scope of the defined function names for labels encompasses the function definitions themselves as well as the body.

macrolet

macrolet establishes local macro definitions, using the same format used by defmacro.

Within the body of macrolet, global self expander definitions of the names defined by the macrolet do not apply; rather, self expands the macro form and recursively process the resulting form.

The macro-expansion functions defined by macrolet are defined in the lexical environment in which the macrolet form appears. Declarations and macrolet and symbol-macrolet definitions affect the local macro definitions in a macrolet, but the consequences are undefined if the local macro definitions reference any local variable or function bindings that are visible in that lexical environment.

Any local-documentation is attached to the corresponding local macro function as a documentation string.
flet, labels, macrolet

Examples:

\[
\begin{align*}
(\text{defun foo (x flag)} & ) \\
& (\text{macrolet ((fudge (z))}) \\
& \quad ; \text{The parameters x and flag are not accessible} \\
& \quad ; \text{at this point; a reference to flag would be to} \\
& \quad ; \text{the global variable of that name.} \\
& \quad (\text{if flag (+ ,z ,z) ,z))) \\
& \quad ; \text{The parameters x and flag are accessible here.} \\
& \quad (+ x) \\
& \quad (\text{fudge x}) \\
& \quad (\text{fudge (+ x 1)))))
\end{align*}
\]

≡

\[
(\text{defun foo (x flag)} & ) \\
& (+ x) \\
& (\text{if flag (+ x x) x}) \\
& (\text{if flag (+ (+ x 1) (+ x 1)) (+ x 1))})
\]

after macro expansion. The occurrences of \texttt{x} and \texttt{flag} legitimately refer to the parameters of the function \texttt{foo} because those parameters are visible at the site of the macro call which produced the expansion.

\[
(\text{flet ((flet1 (n) (+ n n))}) \\
& (\text{flet ((flet1 (n) (+ 2 (flet1 n))}))} \\
& (\text{flet1 2})) \rightarrow 6
\]

\[
(\text{defun dummy-function () 'top-level}) \rightarrow \text{DUMMY-FUNCTION} \\
(\text{funcall #'dummy-function}) \rightarrow \text{TOP-LEVEL} \\
(\text{flet ((dummy-function () 'shadow) }) \\
& (\text{funcall #'dummy-function})) \rightarrow \text{SHADOW} \\
& (\text{eq (funcall #'dummy-function) (funcall #'dummy-function)}) \rightarrow \text{true} \\
& (\text{flet ((dummy-function () 'shadow) }) \\
& (\text{eq (funcall #'dummy-function) }} \\
& (\text{funcall #'dummy-function))) \rightarrow \text{false}
\]

\[
(\text{defun recursive-times (k n) } ) \\
& (\text{labels ((temp (n) }} \\
& \quad (\text{if (zerop n) 0 (+ k (temp (1- n))))})} \\
& (\text{temp n})) \rightarrow \text{RECURSIVE-TIMES} \\
& \text{(recursive-times 2 3) \rightarrow 6}
\]

\[
(\text{defmacro mlets (x &environment env) } ) \\
& (\text{(let ((form ('(abbit ,x))} \\
& \quad (\text{macroexpand form env)))}) \rightarrow \text{MLETS}
\]
(macrolet ((babbit (z) ‘(+ ,z ,z))) (mlets 5)) → 10

(flet ((safesqrt (x) (sqrt (abs x))))
  ;; The safesqrt function is used in two places.
  (safesqrt (apply #'+ (map 'list '#'safesqrt '(1 2 3 4 5 6))))
→ 3.291173)

(defun integer-power (n k)
  (declare (integer n))
  (declare (type (integer 0 *) k))
  (labels ((expt0 (x k a)
    (declare (integer x a) (type (integer 0 *) k))
    (cond ((zerop k) a)
      ((evenp k) (expt1 (* x x) (floor k 2) a))
      (t (expt0 (* x x) (floor k 2) (x a))))))
    (expt1 (x k a)
    (declare (integer x a) (type (integer 0 *) k))
    (cond ((evenp k) (expt1 (* x x) (floor k 2) a))
      (t (expt0 (* x x) (floor k 2) (x a))))))
  (expt0 n k 1))) → INTEGER-POWER

(defun example (y l)
  (flet ((attach (x)
    (setq l (append l (list x))))
    (declare (inline attach)))
    (dolist (x y)
      (unless (null (cdr x))
        (attach x))
    l))
  (example '((a apple apricot) (b banana) (c cherry) (d) (e))
    '((1) (2) (3) (4 2) (5) (6 3 2)))
→ ((1) (2) (3) (4 2) (5) (6 3 2) (A APPLE APRICOT) (B BANANA) (C CHERRY))

See Also:
declare, defmacro, defun, documentation, let, Section 3.1 (Evaluation), Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

Notes:
It is not possible to define recursive functions with flet. labels can be used to define mutually recursive functions.

If a macrolet form is a top level form, the body forms are also processed as top level forms. See Section 3.2.3 (File Compilation).
**funcall**

*Function*

**Syntax:**

```
funcall function &rest args → {result}*
```

**Arguments and Values:**

- `function`—a function designator.
- `args`—arguments to the `function`.
- `results`—the values returned by the `function`.

**Description:**

`funcall` applies `function` to `args`. If `function` is a `symbol`, it is coerced to a `function` as if by finding its functional value in the global environment.

**Examples:**

```
(funcall #'+ 1 2 3) → 6
(funcall 'car '(1 2 3)) → 1
(funcall 'position 1 '(1 2 3 2 1) :start 1) → 4
(cons 1 2) → (1 . 2)
(flet ((cons (x y) `(kons ,x ,y)))
     (let ((cons (symbol-function '+)))
       (funcall #'cons
                 (funcall 'cons 1 2)
                 (funcall cons 1 2))))
   → (KONS (1 . 2) 3)
```

**Exceptional Situations:**

An error of type `undefined-function` should be signaled if `function` is a `symbol` that does not have a global definition as a `function` or that has a global definition as a `macro` or a `special operator`.

**See Also:**

`apply`, `function`, Section 3.1 (Evaluation)

**Notes:**

```
(funcall function arg1 arg2 ...)
≡ (apply function arg1 arg2 ... nil)
≡ (apply function (list arg1 arg2 ...))
```

The difference between `funcall` and an ordinary function call is that in the former case the
function is obtained by ordinary evaluation of a form, and in the latter case it is obtained by the special interpretation of the function position that normally occurs.

### function

**Syntax:**

```
function name → function
```

**Arguments and Values:**

- **name**—a function name or lambda expression.
- **function**—a function object.

**Description:**

The value of function is the functional value of name in the current lexical environment.

If name is a function name, the functional definition of that name is that established by the innermost lexically enclosing flet, labels, or macrolet form, if there is one. Otherwise the global functional definition of the function name is returned.

If name is a lambda expression, then a lexical closure is returned. In situations where a closure over the same set of bindings might be produced more than once, the various resulting closures might or might not be eq.

It is an error to use function on a function name that does not denote a function in the lexical environment in which the function form appears. Specifically, it is an error to use function on a symbol that denotes a macro or special form. An implementation may choose not to signal this error for performance reasons, but implementations are forbidden from defining the failure to signal an error as a useful behavior.

**Examples:**

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

The result of (adder 3) is a function that adds 3 to its argument:

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

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

**See Also:**

- defun, fdefinition, flet, labels, symbol-function, Section 3.1.2.1.1 (Symbols as Forms), Section 2.4.8.2 (Sharpsign Single-Quote), Section 22.1.3.13 (Printing Other Objects)
Notes:
The notation *'name may be used as an abbreviation for (function name).

function-lambda-expression

Syntax:
function-lambda-expression function 
→ lambda-expression, closure-p, name

Arguments and Values:
function—a function.
lambda-expression—a lambda expression or nil.
closure-p—a generalized boolean.
name—an object.

Description:
Returns information about function as follows:
The primary value, lambda-expression, is function’s defining lambda expression, or nil if the information is not available. The lambda expression may have been pre-processed in some ways, but it should remain a suitable argument to compile or function. Any implementation may legitimately return nil as the lambda-expression of any function.

The secondary value, closure-p, is nil if function’s definition was enclosed in the null lexical environment or something non-nil if function’s definition might have been enclosed in some non-null lexical environment. Any implementation may legitimately return true as the closure-p of any function.

The tertiary value, name, is the “name” of function. The name is intended for debugging only and is not necessarily one that would be valid for use as a name in defun or function, for example. By convention, nil is used to mean that function has no name. Any implementation may legitimately return nil as the name of any function.

Examples:
The following examples illustrate some possible return values, but are not intended to be exhaustive:
(function-lambda-expression #'(lambda (x) x))
→ NIL, false, NIL
or
→ NIL, true, NIL
or
→ (LAMBDA (X) X), true, NIL
or
→ (LAMBDA (X) X), false, NIL

(funcall #'(lambda () #'(lambda (x) x)))
→ NIL, false, NIL
or
→ NIL, true, NIL
or
→ (LAMBDA () X), true, NIL
not
→ NIL, false, NIL
not
→ (LAMBDA () X), false, NIL

(flet ((foo (x) x))
  (setf (symbol-function 'bar) #'foo)
  (function-lambda-expression #'bar))
→ NIL, false, NIL
or
→ NIL, true, NIL
or
→ (LAMBDA (X) (BLOCK FOO X)), true, NIL
or
→ (LAMBDA (X) (BLOCK FOO X)), false, FOO
or
→ (SI::BLOCK-LAMBDA FOO (X) X), false, FOO

(defun foo ()
  (flet ((bar (x) x))
    #'bar))

(function-lambda-expression (foo))
→ NIL, false, NIL
or
→ NIL, true, NIL
or
→ (LAMBDA (X) (BLOCK BAR X)), true, NIL
or
→ (LAMBDA (X) (BLOCK BAR X)), true, (:INTERNAL FOO 0 BAR)
or
→ (LAMBDA (X) (BLOCK BAR X)), false, "BAR in FOO"

Notes:

Although implementations are free to return "nil, true, nil" in all cases, they are encouraged to return a lambda expression as the primary value in the case where the argument was created by a call to compile or eval (as opposed to being created by loading a compiled file).
**functionp**

*Function*

**Syntax:**

`functionp object  → generalized-boolean`

**Arguments and Values:**

- `object`—an *object*.
- `generalized-boolean`—a *generalized boolean*.

**Description:**

Returns *true* if `object` is of type *function*; otherwise, returns *false*.

**Examples:**

```lisp
(functionp 'append) → false
(functionp #'append) → true
(functionp (symbol-function 'append)) → true
(flet ((f () 1)) (functionp #'f)) → true
(functionp (compile nil '(lambda () 259))) → true
(functionp nil) → false
(functionp 12) → false
(functionp '(lambda (x) (* x x))) → false
(functionp #'(lambda (x) (* x x))) → true
```

**Notes:**

```
(functionp object) ≡ (typep object 'function)
```

---

**compiled-function-p**

*Function*

**Syntax:**

`compiled-function-p object  → generalized-boolean`

**Arguments and Values:**

- `object`—an *object*.
- `generalized-boolean`—a *generalized boolean*.

**Description:**

Returns *true* if `object` is of type *compiled-function*; otherwise, returns *false*.

---

5–28 Programming Language—Common Lisp
Examples:

```lisp
(defun f (x) x) → F
(compiled-function-p #'f) → false
⇒ true
(compiled-function-p 'f) → false
(compile 'f) → F
(compiled-function-p #'f) → true
(compiled-function-p 'f) → false
(compiled-function-p (compile nil '(lambda (x) x))) → true
(compiled-function-p #'(lambda (x) x)) → false
⇒ true
(compiled-function-p '(lambda (x) x)) → false
```

See Also:
compile, compile-file, compiled-function

Notes:

```lisp
(compiled-function-p object) ≡ (typep object 'compiled-function)
```

---

**call-arguments-limit**

*Constant Variable*

**Constant Value:**
An integer not smaller than 50 and at least as great as the value of `lambda-parameters-limit`, the exact magnitude of which is implementation-dependent.

**Description:**
The upper exclusive bound on the number of arguments that may be passed to a function.

**See Also:**
lambda-parameters-limit, multiple-values-limit
**lambda-list-keywords**  
*Constant Variable*

**Constant Value:**

*a list, the elements of which are implementation-dependent, but which must contain at least the symbols &allow-other-keys, &aux, &body, &environment, &key, &optional, &rest, and &whole.*

**Description:**

*A list of all the lambda list keywords used in the implementation, including the additional ones used only by macro definition forms.*

**See Also:**

defun, flet, defmacro, macrolet, Section 3.1.2 (The Evaluation Model)

---

**lambda-parameters-limit**  
*Constant Variable*

**Constant Value:**

*implementation-dependent, but not smaller than 50.*

**Description:**

*A positive integer that is the upper exclusive bound on the number of parameter names that can appear in a single lambda list.*

**See Also:**

call-arguments-limit

**Notes:**

Implementors are encouraged to make the value of lambda-parameters-limit as large as possible.

---

5–30  Programming Language—Common Lisp
defconstant

Syntax:

```
defconstant name initial-value [documentation] → name
```

Arguments and Values:

- `name`—a symbol; not evaluated.
- `initial-value`—a form; evaluated.
- `documentation`—a string; not evaluated.

Description:

`defconstant` causes the global variable named by `name` to be given a value that is the result of evaluating `initial-value`. A constant defined by `defconstant` can be redefined with `defconstant`. However, the consequences are undefined if an attempt is made to assign a value to the symbol using another operator, or to assign it to a different value using a subsequent `defconstant`. If `documentation` is supplied, it is attached to `name` as a documentation string of kind `variable`. `defconstant` normally appears as a top level form, but it is meaningful for it to appear as a non-top-level form. However, the compile-time side effects described below only take place when `defconstant` appears as a top level form.

The consequences are undefined if there are any bindings of the variable named by `name` at the time `defconstant` is executed or if the value is not `eql` to the value of `initial-value`. The consequences are undefined when constant symbols are rebound as either lexical or dynamic variables. In other words, a reference to a symbol declared with `defconstant` always refers to its global value.

The side effects of the execution of `defconstant` must be equivalent to at least the side effects of the execution of the following code:

```
(setf (symbol-value 'name) initial-value)
(setf (documentation 'name 'variable) 'documentation)
```

If a `defconstant` form appears as a top level form, the compiler must recognize that `name` names a constant variable. An implementation may choose to evaluate the value-form at compile time, load time, or both. Therefore, users must ensure that the `initial-value` can be evaluated at compile time (regardless of whether or not references to `name` appear in the file) and that it always evaluates to the same value.
Examples:

(defconstant this-is-a-constant 'never-changing "for a test") → THIS-IS-A-CONSTANT
this-is-a-constant → NEVER-CHANGING
(documentation 'this-is-a-constant 'variable) → "for a test"
(constantp 'this-is-a-constant) → true

See Also:
declare, defparameter, defvar, documentation, proclaim, Section 3.1.2.1.3 (Constant Variables), Section 3.2 (Compilation)

defparameter, defvar

Macro

Syntax:

defparameter name initial-value [documentation] → name
defvar name [initial-value [documentation]] → name

Arguments and Values:

name—a symbol; not evaluated.
initial-value—a form; for defparameter, it is always evaluated, but for defvar it is evaluated only if name is not already bound.
documentation—a string; not evaluated.

Description:

defparameter and defvar establish name as a dynamic variable.
defparameter unconditionally assigns the initial-value to the dynamic variable named name.
defvar, by contrast, assigns initial-value (if supplied) to the dynamic variable named name only if name is not already bound.

If no initial-value is supplied, defvar leaves the value cell of the dynamic variable named name undisturbed; if name was previously bound, its old value persists, and if it was previously unbound, it remains unbound.

If documentation is supplied, it is attached to name as a documentation string of kind variable.
defparameter and defvar normally appear as a top level form, but it is meaningful for them to appear as non-top-level forms. However, the compile-time side effects described below only take place when they appear as top level forms.
defparameter, defvar

Examples:

```lisp
(defparameter *p* 1) → *P*
*p* → 1
(constantp 'p) → false
(setq *p* 2) → 2
(defparameter *p* 3) → *P*
*p* → 3
```

```lisp
(defvar *v* 1) → *V*
*v* → 1
(constantp 'v) → false
(setq *v* 2) → 2
(defvar *v* 3) → *V*
*v* → 2
```

```lisp
(defun foo ()
  (let ((*p* 'p) (*v* 'v))
    (bar)))
```

The principal operational distinction between defparameter and defvar
makes an unconditional assignment to name, while defvar makes a conditional one. In practice,
this means that defparameter is useful in situations where loading or reloading the definition
would want to pick up a new value of the variable, while defvar is used in situations where the
old value would want to be retained if the file were loaded or reloaded. For example, one might
create a file which contained:

```lisp
(defun foo ()
  (let ((*p* 'p) (*v* 'v))
    (bar))) → FOO
```

Here the initial value, (), for the variable *the-interesting-numbers* is just a seed that we are
never likely to want to reset to something else once something has been grown from it. As such,
we have used defvar to avoid having the *interesting-numbers* information reset if the file is
loaded a second time. It is true that the two calls to define-interesting-number here would
be reprocessed, but if there were additional calls in another file, they would not be and that
information would be lost. On the other hand, consider the following code:

```lisp
(defparameter *default-beep-count* 3)
```

Here the initial value, (), for the variable *the-interesting-numbers* is just a seed that we are
never likely to want to reset to something else once something has been grown from it. As such,
we have used defvar to avoid having the *interesting-numbers* information reset if the file is
loaded a second time. It is true that the two calls to define-interesting-number here would
be reprocessed, but if there were additional calls in another file, they would not be and that
information would be lost. On the other hand, consider the following code:

```lisp
(defun beep (&optional (n *default-beep-count*))
  (dotimes (i n) (si:%%beep 1000. 100000.) (sleep 0.1)))
```

Data and Control Flow  5–33
defparameter, defvar

Here we could easily imagine editing the code to change the initial value of *default-beep-count*, and then reloading the file to pick up the new value. In order to make value updating easy, we have used defparameter.

On the other hand, there is potential value to using defvar in this situation. For example, suppose that someone had predefined an alternate value for *default-beep-count*, or had loaded the file and then manually changed the value. In both cases, if we had used defvar instead of defparameter, those user preferences would not be overridden by (re)loading the file.

The choice of whether to use defparameter or defvar has visible consequences to programs, but is nevertheless often made for subjective reasons.

Side Effects:

If a defvar or defparameter form appears as a top level form, the compiler must recognize that the name has been proclaimed special. However, it must neither evaluate the initial-value form nor assign the dynamic variable named name at compile time.

There may be additional (implementation-defined) compile-time or run-time side effects, as long as such effects do not interfere with the correct operation of conforming programs.

Affected By:

defvar is affected by whether name is already bound.

See Also:

declare, defconstant, documentation, Section 3.2 (Compilation)

Notes:

It is customary to name dynamic variables with an asterisk at the beginning and end of the name, e.g., *foo* is a good name for a dynamic variable, but not for a lexical variable; foo is a good name for a lexical variable, but not for a dynamic variable. This naming convention is observed for all defined names in Common Lisp; however, neither conforming programs nor conforming implementations are obliged to adhere to this convention.

The intent of the permission for additional side effects is to allow implementations to do normal “bookkeeping” that accompanies definitions. For example, the macro expansion of a defvar or defparameter form might include code that arranges to record the name of the source file in which the definition occurs.

defparameter and defvar might be defined as follows:

(defmacro defparameter (name initial-value
  &optional (documentation nil documentation-p))
  `(progn (declare (special ,name))
    (setf (symbol-value ',name) ,initial-value)
    ,(when documentation-p
      ,(setf (documentation ',name 'variable) ',documentation))
    ,name))

5–34 Programming Language—Common Lisp
(defmacro defvar (name &optional
  (initial-value nil initial-value-p)
  (documentation nil documentation-p))
 '(progn (declare (special ,name))
   ,(when initial-value-p
     ,(unless (boundp ',name)
       (setf (symbol-value ',name) ,initial-value)))
   ,(when documentation-p
     ,(setf (documentation ',name 'variable) ',documentation))
   ',name))

destructuring-bind

Macro

Syntax:

destructuring-bind lambda-list expression {declaration}* {form}* → {result}*{result}*

Arguments and Values:

lambda-list—a destructuring lambda list.
expression—a form.
declaration—a declare expression; not evaluated.
forms—an implicit progn.
results—the values returned by the forms.

Description:

destructuring-bind binds the variables specified in lambda-list to the corresponding values in
the tree structure resulting from the evaluation of expression; then destructuring-bind evaluates
forms.

The lambda-list supports destructuring as described in Section 3.4.5 (Destructuring Lambda
Lists).

Examples:

(defun iota (n) (loop for i from 1 to n collect i)) ;helper
(destructuring-bind ((a &optional (b 'bee)) one two three)
  '((alpha) ,0(iota 3))
  (list a b three two one)) → (ALPHA BEE 3 2 1)
Exceptional Situations:
If the result of evaluating the expression does not match the destructuring pattern, an error of type error should be signaled.

See Also:
macrolet, defmacro

---

**let, let***

*Special Operator*

**Syntax:**
let ([var | (var [init-form]])*) {declaration}* {form}* → {result}*  
let* ([var | (var [init-form]])*) {declaration}* {form}* → {result}*

**Arguments and Values:**
var—a symbol.  
init-form—a form.  
declaration—a declare expression; not evaluated.  
form—a form.  
results—the values returned by the forms.

**Description:**
let and let* create new variable bindings and execute a series of forms that use these bindings.  
let performs the bindings in parallel and let* does them sequentially.  
The form

(\texttt{(let ((var1 init-form-1)  
            (var2 init-form-2)  
            ...  
            (varm init-form-m))  
            declaration1  
            declaration2  
            ...  
            declarationp  
            form1  
            form2  
            ...  
            formn)}

---

5–36 Programming Language—Common Lisp
let, let*

first evaluates the expressions `init-form-1`, `init-form-2`, and so on, in that order, saving the resulting values. Then all of the variables `varj` are bound to the corresponding values; each binding is lexical unless there is a special declaration to the contrary. The expressions `formk` are then evaluated in order; the values of all but the last are discarded (that is, the body of a `let` is an implicit `progn`).

let* is similar to `let`, but the bindings of variables are performed sequentially rather than in parallel. The expression for the `init-form` of a `var` can refer to `vars` previously bound in the `let*`.

The form

```
(let* ((var1 init-form-1)
       (var2 init-form-2)
       ...
       (varm init-form-m))
  declaration1
declaration2
  ...
declarationp
  form1
  form2
  ...
  formn)
```

first evaluates the expression `init-form-1`, then binds the variable `var1` to that value; then it evaluates `init-form-2` and binds `var2`, and so on. The expressions `formj` are then evaluated in order; the values of all but the last are discarded (that is, the body of `let*` is an implicit `progn`).

For both `let` and `let*`, if there is not an `init-form` associated with a `var`, `var` is initialized to `nil`.

The special form `let` has the property that the scope of the name binding does not include any initial value form. For `let*`, a variable’s scope also includes the remaining initial value forms for subsequent variable bindings.

Examples:

```
(setq a 'top) → TOP
(defun dummy-function () a) → DUMMY-FUNCTION
(let ((a 'inside) (b a))
  (format nil "S "S "S" a b (dummy-function))) → "INSIDE TOP TOP"
(let* ((a 'inside) (b a))
  (format nil "S "S "S" a b (dummy-function))) → "INSIDE INSIDE TOP"
(let ((a 'inside) (b a))
  (declare (special a))
  (format nil "S "S "S" a b (dummy-function))) → "INSIDE TOP INSIDE"
```

The code
(let (x)
  (declare (integer x))
  (setq x (gcd y z))

...)

is incorrect; although x is indeed set before it is used, and is set to a value of the declared type integer, nevertheless x initially takes on the value nil in violation of the type declaration.

See Also:
progv

---

**progv**

*Special Operator*

**Syntax:**

progv symbols values {form}* → {result}*  

**Arguments and Values:**

symbols—a list of symbols; evaluated.  
values—a list of objects; evaluated.  
forms—an implicit progn.  
results—the values returned by the forms.

**Description:**

progv creates new dynamic variable bindings and executes each form using those bindings. Each form is evaluated in order.

progv allows binding one or more dynamic variables whose names may be determined at run time. Each form is evaluated in order with the dynamic variables whose names are in symbols bound to corresponding values. If too few values are supplied, the remaining symbols are bound and then made to have no value. If too many values are supplied, the excess values are ignored. The bindings of the dynamic variables are undone on exit from progv.

**Examples:**

(setq *x* 1) → 1  
(progv '(*x*) '(2) *x*) → 2  
* x* → 1  

Assuming *x* is not globally special,

(let ((*x* 3))
  (progv '(*x*) '(4))

5–38 Programming Language—Common Lisp
(list *x* (symbol-value 'x*))) → (3 4)

See Also:
let, Section 3.1 (Evaluation)

Notes:
Among other things, progv is useful when writing interpreters for languages embedded in Lisp; it provides a handle on the mechanism for binding dynamic variables.

---

setq

Syntax:

```
setq {\langle pair\rangle}* → result

pair::=var form
```

Pronunciation:
[ 'set,kyü]

Arguments and Values:

- `var`—a symbol naming a variable other than a constant variable.
- `form`—a form.
- `result`—the primary value of the last form, or `nil` if no `pairs` were supplied.

Description:
Assigns values to variables.

`(setq var1 form1 var2 form2 ...)` is the simple variable assignment statement of Lisp. First `form1` is evaluated and the result is stored in the variable `var1`, then `form2` is evaluated and the result stored in `var2`, and so forth. `setq` may be used for assignment of both lexical and dynamic variables.

If any `var` refers to a binding made by `symbol-macrolet`, then that `var` is treated as if `setf` (not `setq`) had been used.

Examples:

`;; A simple use of SETQ to establish values for variables.
(setq a 1 b 2 c 3) → 3
a → 1
b → 2
c → 3
;;; Use of SETQ to update values by sequential assignment.
(setq a (+ 1 b) b (+ 1 a) c (+ a b)) → 7
a → 3
b → 4
c → 7

;;; This illustrates the use of SETQ on a symbol macro.
(let ((x (list 10 20 30)))
  (symbol-macrolet ((y (car x)) (z (cadr x)))
    (setq y (+ 1 z) z (+ 1 y))
    (list x y z)))
→ ((21 22 30) 21 22)

Side Effects:
The primary value of each form is assigned to the corresponding var.

See Also:
psetq, set, setf

---

psetq

Macro

Syntax:
psetq {pair}::* → nil

pair ::= var form

Pronunciation:
psetq: [pē'set,kyü]

Arguments and Values:
var—a symbol naming a variable other than a constant variable.
form—a form.

Description:
Assigns values to variables.

This is just like setq, except that the assignments happen “in parallel.” That is, first all of the forms are evaluated, and only then are the variables set to the resulting values. In this way, the assignment to one variable does not affect the value computation of another in the way that would occur with setq’s sequential assignment.
If any \textit{var} refers to a \textit{binding} made by \texttt{symbol-macrolet}, then that \textit{var} is treated as if \texttt{psetf} (not \texttt{psetq}) had been used.

\textbf{Examples:}

\begin{verbatim}
;;; A simple use of PSETQ to establish values for variables.
;;; As a matter of style, many programmers would prefer SETQ
;;; in a simple situation like this where parallel assignment
;;; is not needed, but the two have equivalent effect.
(psetq a 1 b 2 c 3) → NIL
a → 1
b → 2
c → 3

;;; Use of PSETQ to update values by parallel assignment.
;;; The effect here is very different than if SETQ had been used.
(psetq a (1+ b) b (1+ a) c (+ a b)) → NIL
a → 3
b → 2
c → 3

;;; Use of PSETQ on a symbol macro.
(let ((x (list 10 20 30)))
  (symbol-macrolet ((y (car x)) (z (cadr x)))
    (psetq y (1+ z) z (1+ y))
    (list x y z))
→ ((21 11 30) 21 11)

;;; Use of parallel assignment to swap values of A and B.
(let ((a 1) (b 2))
  (psetq a b b a)
  (values a b))
→ 2, 1
\end{verbatim}

\textbf{Side Effects:}

The values of \textit{forms} are assigned to \textit{vars}.

\textbf{See Also:}

\texttt{psetf, setq}
block

block

Special Operator

Syntax:

\[
\text{block name form}^* \rightarrow \{\text{result}\}^*
\]

Arguments and Values:

- **name**—a symbol.
- **form**—a form.
- **results**—the values of the forms if a normal return occurs, or else, if an explicit return occurs, the values that were transferred.

Description:

block establishes a block named name and then evaluates forms as an implicit progn.

The special operators block and return-from work together to provide a structured, lexical, non-local exit facility. At any point lexically contained within forms, return-from can be used with the given name to return control and values from the block form, except when an intervening block with the same name has been established, in which case the outer block is shadowed by the inner one.

The block named name has lexical scope and dynamic extent.

Once established, a block may only be exited once, whether by normal return or explicit return.

Examples:

- \(\text{(block empty)} \rightarrow \text{NIL}\)
- \(\text{(block whocares (values 1 2) (values 3 4))} \rightarrow 3, 4\)
- \(\text{(let ((x 1))}
\text{  (block stop (setq x 2) (return-from stop) (setq x 3))}
\text{x} \rightarrow 2\)
- \(\text{(block early (return-from early (values 1 2)) (values 3 4))} \rightarrow 1, 2\)
- \(\text{(block outer (block inner (return-from outer 1)) 2)} \rightarrow 1\)
- \(\text{(block twin (block twin (return-from twin 1)) 2)} \rightarrow 2\)
  
  ;; Contrast behavior of this example with corresponding example of CATCH.
- \(\text{(block b}
\text{  (flet ((b1 () (return-from b 1)))}
\text{  (block b (b1) (print 'unreachable))}
\text{2))} \rightarrow 1\)

See Also:

return, return-from, Section 3.1 (Evaluation)
Notes:

**catch**

_Syntax:_

```
catch tag {form}* → {result}*
```

_Arguments and Values:_

- `tag`—a catch tag; evaluated.
- `forms`—an implicit `progn`.
- `results`—if the `forms` exit normally, the values returned by the `forms`; if a throw occurs to the `tag`, the values that are thrown.

_Description:_

`catch` is used as the destination of a non-local control transfer by `throw`. Tags are used to find the `catch` to which a `throw` is transferring control. `(catch 'foo form)` catches a `(throw 'foo form)` but not a `(throw 'bar form)`.

The order of execution of `catch` follows:

1. _Tag_ is evaluated. It serves as the name of the `catch`.
2. _Forms_ are then evaluated as an implicit `progn`, and the results of the last `form` are returned unless a `throw` occurs.
3. If a `throw` occurs during the execution of one of the `forms`, control is transferred to the `catch form` whose `tag` is `eq` to the tag argument of the `throw` and which is the most recently established `catch` with that `tag`. No further evaluation of `forms` occurs.
4. The `tag` established by `catch` is _disestablished_ just before the results are returned.

If during the execution of one of the `forms`, a `throw` is executed whose tag is `eq` to the `catch` tag, then the values specified by the `throw` are returned as the result of the dynamically most recently established `catch` form with that tag.

The mechanism for `catch` and `throw` works even if `throw` is not within the lexical scope of `catch`. `throw` must occur within the _dynamic extent_ of the _evaluation_ of the body of a `catch` with a corresponding `tag`.

_Examples:_

```
(catch 'dummy-tag 1 2 (throw 'dummy-tag 3) 4) → 3
```
(catch 'dummy-tag 1 2 3 4) → 4
(defun throw-back (tag) (throw tag t)) → THROW-BACK
(catch 'dummy-tag (throw-back 'dummy-tag) 2) → T

;; Contrast behavior of this example with corresponding example of BLOCK.
(catch 'c
  (flet ((c1 () (throw 'c 1)))
    (catch 'c (c1) (print 'unreachable))
  ) → 2

Exceptional Situations:
An error of type control-error is signaled if throw is done when there is no suitable catch tag.

See Also:
throw, Section 3.1 (Evaluation)

Notes:
It is customary for symbols to be used as tags, but any object is permitted. However, numbers should not be used because the comparison is done using eq.

catch differs from block in that catch tags have dynamic scope while block names have lexical scope.

---

go

Special Operator

Syntax:
    go tag

Arguments and Values:
    tag — a go tag.

Description:
go transfers control to the point in the body of an enclosing tagbody form labeled by a tag eql to tag. If there is no such tag in the body, the bodies of lexically containing tagbody forms (if any) are examined as well. If several tags are eql to tag, control is transferred to whichever matching tag is contained in the innermost tagbody form that contains the go. The consequences are undefined if there is no matching tag lexically visible to the point of the go.

The transfer of control initiated by go is performed as described in Section 5.2 (Transfer of Control to an Exit Point).

Examples:

(tagbody}
(setq val 2)
(go lp)
(incf val 3)
lp (incf val 4)) → NIL
val → 6

The following is in error because there is a normal exit of the\ntagbody\ before the\ngo\ is executed.

(let ((a nil))
  (tagbody t (setq a #'(lambda () (go t))))
  (funcall a))

The following is in error because the\ntagbody\ is passed over before the\go form\ is executed.

(funcall (block nil
  (tagbody a (return #'(lambda () (go a)))))

See Also:
tagbody

---

**return-from**

*Special Operator*

**Syntax:**

```
return-from name [result] →
```

**Arguments and Values:**

- `name`: a block tag; not evaluated.
- `result`: a form; evaluated. The default is `nil`.  

**Description:**

Returns control and *multiple values* from a lexically enclosing block.

A block form named `name` must lexically enclose the occurrence of `return-from`; any values yielded by the evaluation of `result` are immediately returned from the innermost such lexically enclosing block.

The transfer of control initiated by `return-from` is performed as described in Section 5.2 (Transfer of Control to an Exit Point).

**Examples:**

```
(block alpha (return-from alpha) 1) → NIL
(block alpha (return-from alpha 1) 2) → 1
```
return-from

(block alpha (return-from alpha (values 1 2)) 3) → 1, 2
(let ((a 0))
    (dotimes (i 10) (incf a) (when (oddp i) (return)))
    a) → 2
(defun temp (x)
    (if x (return-from temp 'dummy))
    44) → TEMP
(temp nil) → 44
(temp t) → DUMMY
(block out
    (flet ((exit (n) (return-from out n)))
        (block out (exit 1)))
    2) → 1
(block nil
    (unwind-protect (return-from nil 1)
        (return-from nil 2)))
→ 2
(dolist (flag '(nil t))
    (block nil
        (let ((x 5))
            (declare (special x))
            (unwind-protect (return-from nil)
                (print x)))))
    (print 'here))
▷ 5
▷ HERE
▷ 5
▷ HERE
→ NIL
(dolist (flag '(nil t))
    (block nil
        (let ((x 5))
            (declare (special x))
            (unwind-protect
                (if flag (return-from nil))
                (print x)))))
    (print 'here))
▷ 5
▷ HERE
▷ 5
▷ HERE
→ NIL

The following has undefined consequences because the block form exits normally before the return-from form is attempted.

5–46 Programming Language—Common Lisp
(funcall (block nil #'(lambda () (return-from nil)))()) is an error.

See Also:
block, return, Section 3.1 (Evaluation)

---

**return**  
*Macro*

**Syntax:**

```
return [result]  →
```

**Arguments and Values:**

- **result**—a form; evaluated. The default is nil.

**Description:**

Returns, as if by return-from, from the block named nil.

**Examples:**

```
(block nil (return 1)) → NIL
(block nil (return 1) 2) → 1
(block nil (return (values 1 2)) 3) → 1, 2
(block nil (block alpha (return 1) 2)) → 1
(block alpha (block nil (return 1)) 2) → 2
(block nil (block nil (return 1) 2)) → 1
```

See Also:
block, return-from, Section 3.1 (Evaluation)

**Notes:**

```
(return) ≡ (return-from nil)
(return form) ≡ (return-from nil form)
```

The **implicit blocks established** by macros such as do are often named nil, so that return can be used to exit from such forms.
tagbody

Syntax:

\[ \text{tagbody}\{\text{tag} | \text{statement}\}^* \rightarrow \text{nil} \]

Arguments and Values:

- **tag** — a *go tag*; not evaluated.
- **statement** — a *compound form*; evaluated as described below.

Description:

Executes zero or more *statements* in a *lexical environment* that provides for control transfers to labels indicated by the *tags*.

The *statements* in a *tagbody* are evaluated in order from left to right, and their *values* are discarded. If at any time there are no remaining *statements*, *tagbody* returns *nil*. However, if \((\text{go tag})\) is evaluated, control jumps to the part of the body labeled with the *tag*. (Tags are compared with eql.)

A *tag* established by *tagbody* has *lexical scope* and has *dynamic extent*. Once *tagbody* has been exited, it is no longer valid to *go* to a *tag* in its body. It is permissible for *go* to jump to a *tagbody* that is not the innermost *tagbody* containing that *go*; the *tags* established by a *tagbody* only shadow other *tags* of like name.

The determination of which elements of the body are *tags* and which are *statements* is made prior to any *macro expansion* of that element. If a *statement* is a *macro form* and its *macro expansion* is an *atom*, that *atom* is treated as a *statement*, not a *tag*.

Examples:

\[
\begin{align*}
\text{(let (val)} \\
\text{ (tagbody)} \\
\text{ (setq val 1)} \\
\text{ (go point-a)} \\
\text{ (incf val 16)} \\
\text{ point-c} \\
\text{ (incf val 04)} \\
\text{ (go point-b)} \\
\text{ (incf val 32)} \\
\text{ point-a} \\
\text{ (incf val 02)} \\
\text{ (go point-c)} \\
\text{ (incf val 64)} \\
\text{ point-b} \\
\text{ (incf val 08))}
\end{align*}
\]
val)
→ 15
(defun f1 (flag)
  (let ((n 1))
    (tagbody
      (setq n (f2 flag #'(lambda () (go out))))
    out
      (prin1 n))))
→ F1
(defun f2 (flag escape)
  (if flag (funcall escape) 2))
→ F2
(f1 nil)
▷ 2
→ NIL
(f1 t)
▷ 1
→ NIL

See Also:
go

Notes:
The macros in Figure 5–10 have implicit tagbodies.

<table>
<thead>
<tr>
<th>do</th>
<th>do-external-symbols</th>
<th>dotimes</th>
</tr>
</thead>
<tbody>
<tr>
<td>do*</td>
<td>do-symbols</td>
<td>prog</td>
</tr>
<tr>
<td>do-all-symbols</td>
<td>dolist</td>
<td>prog*</td>
</tr>
</tbody>
</table>

Figure 5–10. Macros that have implicit tagbodies.
	hrow

Special Operator

Syntax:
throw tag result-form →

Arguments and Values:
tag—a catch tag; evaluated.
result-form—a form; evaluated as described below.
throw

Description:

*throw* causes a non-local control transfer to a *catch* whose tag is *eq* to *tag*.

*Tag* is evaluated first to produce an object called the throw tag; then *result-form* is evaluated, and its results are saved. If the *result-form* produces multiple values, then all the values are saved. The most recent outstanding *catch* whose *tag* is *eq* to the throw tag is exited; the saved results are returned as the value or values of *catch*.

The transfer of control initiated by *throw* is performed as described in Section 5.2 (Transfer of Control to an Exit Point).

Examples:

```lisp
(catch 'result
  (setq i 0 j 0)
  (loop (incf j 3) (incf i)
        (if (= i 3) (throw 'result (values i j)))))) → 3, 9
```

```lisp
(catch nil
  (unwind-protect (throw nil 1)
                  (throw nil 2))) → 2
```

The consequences of the following are undefined because the *catch* of *b* is passed over by the first *throw*, hence portable programs must assume that its *dynamic extent* is terminated. The binding of the *catch tag* is not yet disestablished and therefore it is the target of the second *throw*.

```lisp
(catch 'a
  (catch 'b
            (unwind-protect (throw 'a 1)
                            (throw 'b 2)))))
```

The following prints “The inner catch returns :SECOND-THROW” and then returns :outer-catch.

```lisp
(catch 'foo
  (format t "The inner catch returns ~s." %)
        (catch 'foo
                  (unwind-protect (throw 'foo :first-throw)
                                  (throw 'foo :second-throw)))))
    :outer-catch)
▷ The inner catch returns :SECOND-THROW
⇒ :OUTER-CATCH
```

Exceptional Situations:

If there is no outstanding *catch tag* that matches the throw tag, no unwinding of the stack
is performed, and an error of type control-error is signaled. When the error is signaled, the dynamic environment is that which was in force at the point of the throw.

See Also:
block, catch, return-from, unwind-protect, Section 3.1 (Evaluation)

Notes:
catch and throw are normally used when the exit point must have dynamic scope (e.g., the throw is not lexically enclosed by the catch), while block and return are used when lexical scope is sufficient.

unwind-protect

Special Operator

Syntax:
unwind-protect protected-form {cleanup-form}* → {result}*  

Arguments and Values:
protected-form—a form.
cleanup-form—a form.
results—the values of the protected-form.

Description:
unwind-protect evaluates protected-form and guarantees that cleanup-forms are executed before unwind-protect exits, whether it terminates normally or is aborted by a control transfer of some kind. unwind-protect is intended to be used to make sure that certain side effects take place after the evaluation of protected-form.

If a non-local exit occurs during execution of cleanup-forms, no special action is taken. The cleanup-forms of unwind-protect are not protected by that unwind-protect.

unwind-protect protects against all attempts to exit from protected-form, including go, handler-case, ignore-errors, restart-case, return-from, throw, and with-simple-restart.

Undoing of handler and restart bindings during an exit happens in parallel with the undoing of the bindings of dynamic variables and catch tags, in the reverse order in which they were established. The effect of this is that cleanup-form sees the same handler and restart bindings, as well as dynamic variable bindings and catch tags, as were visible when the unwind-protect was entered.

Examples:

(tagbody
The following code is not correct:

```lisp
(unwind-protect
  (progn (incf *access-count*)
         (perform-access))
  (decf *access-count*))
```

If an exit occurs before completion of `incf`, the `decf` form is executed anyway, resulting in an incorrect value for `*access-count*`. The correct way to code this is as follows:

```lisp
(let ((old-count *access-count*))
  (unwind-protect
    (progn (incf *access-count*)
           (perform-access))
    (setq *access-count* old-count)))
```

```lisp
;;; The following returns 2.
(block nil
  (unwind-protect (return 1)
      (return 2))))

;;; The following has undefined consequences.
(block a
  (block b

```
unwind-protect

;;; The following returns 2.
(catch nil
  (unwind-protect (throw nil 1)
    (throw nil 2)))

;;; The following has undefined consequences because the catch of B is
;;; passed over by the first THROW, hence portable programs must assume
;;; its dynamic extent is terminated. The binding of the catch tag is not
;;; yet disestablished and therefore it is the target of the second throw.
(catch 'a
  (catch 'b
    (unwind-protect (throw 'a 1)
      (throw 'b 2))))

;;; The following prints "The inner catch returns :SECOND-THROW"
;;; and then returns :OUTER-CATCH.
(catch 'foo
  (format t "The inner catch returns ~s." %
    (catch 'foo
      (unwind-protect (throw 'foo :first-throw)
        (throw 'foo :second-throw)))
    :outer-catch)

;;; The following returns 10. The inner CATCH of A is passed over, but
;;; because that CATCH is disestablished before the THROW to A is executed,
;;; it isn’t seen.
(catch 'a
  (catch 'b
    (unwind-protect (1+ (catch 'a (throw 'b 1)))
      (throw 'a 10)))

;;; The following has undefined consequences because the extent of
;;; the (CATCH 'BAR ...) exit ends when the (THROW 'FOO ...) commences.
(catch 'foo
  (catch 'bar
    (unwind-protect (throw 'foo 3)
      (throw 'bar 4)
      (print 'xxx))))
;;; The following returns 4; XXX is not printed.
;;; The (THROW 'FOO ...) has no effect on the scope of the BAR
;;; catch tag or the extent of the (CATCH 'BAR ...) exit.
(catch 'bar
  (catch 'foo
    (unwind-protect (throw 'foo 3)
      (throw 'bar 4)
      (print 'xxx))))

;;; The following prints 5.
(block nil
  (let ((x 5))
    (declare (special x))
    (unwind-protect (return)
      (print x))))

See Also:
catch, go, handler-case, restart-case, return, return-from, throw, Section 3.1 (Evaluation)

---

nil

Constant Variable

Constant Value:
nil.

Description:
nil represents both boolean (and generalized boolean) false and the empty list.

Examples:
nil → NIL

See Also:
t
not

Function

Syntax:

```
not x → boolean
```

Arguments and Values:

- **x**—a *generalized boolean* (i.e., any *object*).
- **boolean**—a *boolean*.

Description:

Returns *t* if *x* is *false*; otherwise, returns *nil*.

Examples:

```
(not nil) → T
(not '()) → T
(not (integerp 'sss)) → T
(not (integerp 1)) → NIL
(not 3.7) → NIL
(not 'apple) → NIL
```

See Also:

- null

Notes:

*not* is intended to be used to invert the ‘truth value’ of a *boolean* (or *generalized boolean*) whereas *null* is intended to be used to test for the *empty list*. Operationally, *not* and *null* compute the same result; which to use is a matter of style.

---

**t**

Constant Variable

Constant Value:

*t*.

Description:

The *boolean* representing true, and the canonical *generalized boolean* representing true. Although any *object* other than *nil* is considered *true*, *t* is generally used when there is no special reason to prefer one such *object* over another.
The symbol \texttt{t} is also sometimes used for other purposes as well. For example, as the name of a class, as a designator (\textit{e.g.}, a stream designator) or as a special symbol for some syntactic reason (\textit{e.g.}, in \texttt{case} and \texttt{typcase} to label the otherwise-clause).

**Examples:**

\[
\begin{align*}
\texttt{t} & \rightarrow \texttt{T} \\
(eq \texttt{t} \ 't') & \rightarrow \texttt{true} \\
(find-class \ 't') & \rightarrow \ #<\texttt{CLASS T 610703333}> \\
(case \ 'a' (a 1) (t 2)) & \rightarrow 1 \\
(case \ 'b' (a 1) (t 2)) & \rightarrow 2 \\
(prin1 \ 'hello' \ t) & \rightarrow \texttt{HELLO}
\end{align*}
\]

**See Also:**

\texttt{nil}

---

**\texttt{eq}**

*Function*

**Syntax:**

\[
\texttt{eq } x \ y \rightarrow \texttt{generalized-boolean}
\]

**Arguments and Values:**

\[
\begin{align*}
x & \text{— an object.} \\
y & \text{— an object.}
\end{align*}
\]

\texttt{generalized-boolean}—a generalized boolean.

**Description:**

Returns \texttt{true} if its arguments are the same, identical \texttt{object}; otherwise, returns \texttt{false}.

**Examples:**

\[
\begin{align*}
(eq \ 'a' \ 'b') & \rightarrow \texttt{false} \\
(eq \ 'a' \ 'a') & \rightarrow \texttt{true} \\
(eq \ 3 \ 3) & \rightarrow \texttt{true} \\
(eq \ 3.0 \ 3.0) & \rightarrow \texttt{false}
\end{align*}
\]

5–56 Programming Language—Common Lisp
(eq #c(3 -4) #c(3 -4))  
→ true  
or  
→ false

(eq #c(3 -4.0) #c(3 -4))  → false
(eq (cons 'a 'b) (cons 'a 'c))  → false
(eq (cons 'a 'b) (cons 'a 'b))  → false
(eq '(a . b) '(a . b))  
→ true  
or  
→ false

(progn (setq x (cons 'a 'b)) (eq x x))  → true
(progn (setq x '(a . b)) (eq x x))  → true
(eq #\A #\A)  
→ true  
or  
→ false

(let ((x "Foo")) (eq x x))  → true
(eq "Foo" "Foo")  
→ true  
or  
→ false

(eq "Foo" (copy-seq "Foo"))  → false
(eq "FOO" "foo")  → false
(eq "string-seq" (copy-seq "string-seq"))  → false
(let ((x 5)) (eq x x))  
→ true  
or  
→ false

See Also:
eql, equal, equalp, =, Section 3.2 (Compilation)

Notes:

Objects that appear the same when printed are not necessarily eq to each other. Symbols that print the same usually are eq to each other because of the use of the intern function. However, numbers with the same value need not be eq, and two similar lists are usually not identical.

An implementation is permitted to make “copies” of characters and numbers at any time. The effect is that Common Lisp makes no guarantee that eq is true even when both its arguments are “the same thing” if that thing is a character or number.

Most Common Lisp operators use eql rather than eq to compare objects, or else they default to eql and only use eq if specifically requested to do so. However, the following operators are defined to use eq rather than eql in a way that cannot be overridden by the code which employs them:

<table>
<thead>
<tr>
<th>catch</th>
<th>get</th>
<th>getf</th>
<th>throw</th>
</tr>
</thead>
<tbody>
<tr>
<td>remf</td>
<td>remprop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5–11. Operators that always prefer EQ over EQL
### eql

**Syntax:**

\[
eql \ x \ y \ \rightarrow\ \text{generalized-boolean}
\]

**Arguments and Values:**

- \(x\)—an object.
- \(y\)—an object.

**generalized-boolean**—a generalized boolean.

**Description:**

The value of `eql` is `true` of two objects, \(x\) and \(y\), in the following cases:

1. If \(x\) and \(y\) are eq.
2. If \(x\) and \(y\) are both numbers of the same type and the same value.
3. If they are both characters that represent the same character.

Otherwise the value of `eql` is `false`.

If an implementation supports positive and negative zeros as distinct values, then `(eql 0.0 -0.0)` returns `false`. Otherwise, when the syntax `-0.0` is read it is interpreted as the value `0.0`, and so `(eql 0.0 -0.0)` returns `true`.

**Examples:**

\[
\begin{align*}
(eql \ 'a \ 'b) & \rightarrow \ false \\
(eql \ 'a \ 'a) & \rightarrow \ true \\
(eql \ 3 \ 3) & \rightarrow \ true \\
(eql \ 3.0 \ 3.0) & \rightarrow \ false \\
(eql \ #c(3 \ -4) \ #c(3 \ -4)) & \rightarrow \ true \\
(eql \ #c(3 \ -4.0) \ #c(3 \ -4)) & \rightarrow \ false \\
(eql \ (\text{cons} \ 'a \ 'b) \ (\text{cons} \ 'a \ 'c)) & \rightarrow \ false \\
(eql \ (\text{cons} \ 'a \ 'b) \ (\text{cons} \ 'a \ 'b)) & \rightarrow \ false \\
(eql \ '(a . b) \ '(a . b)) & \rightarrow \ true \\
\end{align*}
\]

\[
\begin{align*}
\text{or} \rightarrow \ false \\
\text{(progn \ (setq x \ (\text{cons} \ 'a \ 'b)) \ (eql x x))} & \rightarrow \ true \\
\text{(progn \ (setq x \ '(a . b)) \ (eql x x))} & \rightarrow \ true \\
\text{(eql \ #\A \ #\A)} & \rightarrow \ true \\
\end{align*}
\]

5–58 Programming Language—Common Lisp
(eql "Foo" "Foo") → true
or false
(eql "Foo" (copy-seq "Foo")) → false
(eql "FOO" "foo") → false

Normally (eql 1.0e0 1.0d0) is false, under the assumption that 1.0e0 and 1.0d0 are of distinct data types. However, implementations that do not provide four distinct floating-point formats are permitted to “collapse” the four formats into some smaller number of them; in such an implementation (eql 1.0e0 1.0d0) might be true.

See Also:
eq, equal, equalp, =, char=

Notes:
eql is the same as eq, except that if the arguments are characters or numbers of the same type then their values are compared. Thus eql tells whether two objects are conceptually the same, whereas eq tells whether two objects are implementationally identical. It is for this reason that eql, not eq, is the default comparison predicate for operators that take sequences as arguments.

eql may not be true of two floats even when they represent the same value. = is used to compare mathematical values.

Two complex numbers are considered to be eql if their real parts are eql and their imaginary parts are eql. For example, (eql #C(4 5) #C(4 5)) is true and (eql #C(4 5) #C(4.0 5.0)) is false. Note that while (eql #C(5.0 0.0) 5.0) is false, (eql #C(5 0) 5) is true. In the case of (eql #C(5.0 0.0) 5.0) the two arguments are of different types, and so cannot satisfy eql. In the case of (eql #C(5 0) 5), #C(5 0) is not a complex number, but is automatically reduced to the integer 5.

equal

Function

Syntax:

equal x y → generalized-boolean

Arguments and Values:

x—an object.
y—an object.
generalized-boolean—a generalized boolean.
equal

Description:

Returns true if x and y are structurally similar (isomorphic) objects. Objects are treated as follows by equal.

Symbols, Numbers, and Characters

equal is true of two objects if they are symbols that are eq, if they are numbers that are eql, or if they are characters that are eql.

Conses

For conses, equal is defined recursively as the two cars being equal and the two cdrs being equal.

Arrays

Two arrays are equal only if they are eq, with one exception: strings and bit vectors are compared element-by-element (using eql). If either x or y has a fill pointer, the fill pointer limits the number of elements examined by equal. Uppercase and lowercase letters in strings are considered by equal to be different.

Pathnames

Two pathnames are equal if and only if all the corresponding components (host, device, and so on) are equivalent. Whether or not uppercase and lowercase letters are considered equivalent in strings appearing in components is implementation-dependent. pathnames that are equal should be functionally equivalent.

Other (Structures, hash-tables, instances, ...)

Two other objects are equal only if they are eq.

equal does not descend any objects other than the ones explicitly specified above. Figure 5–12 summarizes the information given in the previous list. In addition, the figure specifies the priority of the behavior of equal, with upper entries taking priority over lower ones.
equal

<table>
<thead>
<tr>
<th>Type</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>uses eql</td>
</tr>
<tr>
<td>character</td>
<td>uses eql</td>
</tr>
<tr>
<td>cons</td>
<td>descends</td>
</tr>
<tr>
<td>bit vector</td>
<td>descends</td>
</tr>
<tr>
<td>string</td>
<td>descends</td>
</tr>
<tr>
<td>pathname</td>
<td>“functionally equivalent”</td>
</tr>
<tr>
<td>structure</td>
<td>uses eq</td>
</tr>
<tr>
<td>Other array</td>
<td>uses eq</td>
</tr>
<tr>
<td>hash table</td>
<td>uses eq</td>
</tr>
<tr>
<td>Other object</td>
<td>uses eq</td>
</tr>
</tbody>
</table>

Figure 5–12. Summary and priorities of behavior of equal

Any two objects that are eql are also equal.

equal may fail to terminate if x or y is circular.

Examples:

- (equal 'a 'b) → false
- (equal 'a 'a) → true
- (equal 3 3) → true
- (equal 3 3.0) → false
- (equal 3.0 3.0) → true
- (equal #c(3 -4) #c(3 -4)) → true
- (equal #c(3 -4.0) #c(3 -4)) → false
- (equal (cons 'a 'b) (cons 'a 'c)) → false
- (equal (cons 'a 'b) (cons 'a 'b)) → true
- (equal #\A #\A) → true
- (equal #\A #\a) → false
- (equal "Foo" "Foo") → true
- (equal "Foo" (copy-seq "Foo")) → true
- (equal "FOO" "foo") → false
- (equal "This-string" "This-string") → true
- (equal "This-string" "this-string") → false

See Also:
eq, eql, equalp, =, string=, string-equal, char=, char-equal, tree-equal

Notes:
Object equality is not a concept for which there is a uniquely determined correct algorithm. The appropriateness of an equality predicate can be judged only in the context of the needs of some particular program. Although these functions take any type of argument and their names sound
very generic, **equal** and **equalp** are not appropriate for every application.

A rough rule of thumb is that two *objects* are **equal** if and only if their printed representations are the same.

**equalp**  

*Function*

**Syntax:**

\[
\text{equalp} \ x \ y \ \rightarrow \ \text{generalized-boolean}
\]

**Arguments and Values:**

- \(x\)—an *object*.
- \(y\)—an *object*.
- \(\text{generalized-boolean}\)—a *generalized boolean*.

**Description:**

Returns **true** if \(x\) and \(y\) are **equal**, or if they have components that are of the same *type* as each other and if those components are **equalp**; specifically, **equalp** returns **true** in the following cases:

**Characters**

If two *characters* are **char-equal**.

**Numbers**

If two *numbers* are the *same* under \(=\).

**Conses**

If the two *carts* in the *conses* are **equalp** and the two *cdrs* in the *conses* are **equalp**.

**Arrays**

If two *arrays* have the same number of dimensions, the dimensions match, and the corresponding *active elements* are **equalp**. The *types* for which the *arrays* are *specialized* need not match; for example, a *string* and a general *array* that happens to contain the same *characters* are **equalp**. Because **equalp** performs element-by-element comparisons of *strings* and ignores the case of characters, case distinctions are ignored when **equalp** compares *strings*.
**Structures**

If two structures $S_1$ and $S_2$ have the same class and the value of each slot in $S_1$ is the same under **equalp** as the value of the corresponding slot in $S_2$.

**Hash Tables**

**equalp** descends hash-tables by first comparing the count of entries and the :test function; if those are the same, it compares the keys of the tables using the :test function and then the values of the matching keys using **equalp** recursively.

**equalp** does not descend any objects other than the ones explicitly specified above. Figure 5–13 summarizes the information given in the previous list. In addition, the figure specifies the priority of the behavior of **equalp**, with upper entries taking priority over lower ones.

<table>
<thead>
<tr>
<th>Type</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>uses =</td>
</tr>
<tr>
<td>character</td>
<td>uses char-equal</td>
</tr>
<tr>
<td>cons</td>
<td>descends</td>
</tr>
<tr>
<td>bit vector</td>
<td>descends</td>
</tr>
<tr>
<td>string</td>
<td>descends</td>
</tr>
<tr>
<td>pathname</td>
<td>same as <strong>equal</strong></td>
</tr>
<tr>
<td>structure</td>
<td>descends, as described above</td>
</tr>
<tr>
<td>Other array</td>
<td>descend, as described above</td>
</tr>
<tr>
<td>hash table</td>
<td>descends, as described above</td>
</tr>
<tr>
<td>Other object</td>
<td>uses <strong>eq</strong></td>
</tr>
</tbody>
</table>

Figure 5–13. Summary and priorities of behavior of **equalp**

**Examples:**

- `(equalp 'a 'b)` → false
- `(equalp 'a 'a)` → true
- `(equalp 3 3)` → true
- `(equalp 3.0 3)` → true
- `(equalp 3.0 3.0)` → true
- `(equalp #c(3 -4) #c(3 -4))` → true
- `(equalp #c(3 -4.0) #c(3 -4))` → true
- `(equalp (cons 'a 'b) (cons 'a 'c))` → false
- `(equalp (cons 'a 'b) (cons 'a 'b))` → true
- `(equalp #\A #\A)` → true
- `(equalp #\A #\a)` → true
- `(equalp "Foo" "Foo")` → true
- `(equalp "Foo" (copy-seq "Foo"))` → true
(equalp "FOO" "foo") → true
(setq array1 (make-array 6 :element-type 'integer
  :initial-contents '(1 1 1 3 5 7)))
→ #(1 1 1 3 5 7)
(setq array2 (make-array 8 :element-type 'integer
  :initial-contents '(1 1 1 3 5 7 2 6)
  :fill-pointer 6))
→ #(1 1 1 3 5 7)
(equalp array1 array2) → true
(setq vector1 (vector 1 1 1 3 5 7)) → #(1 1 1 3 5 7)
(equalp array1 vector1) → true

See Also:
eq, eql, equal, =, string=, string-equal, char=, char-equal

Notes:
Object equality is not a concept for which there is a uniquely determined correct algorithm. The appropriateness of an equality predicate can be judged only in the context of the needs of some particular program. Although these functions take any type of argument and their names sound very generic, equal and equalp are not appropriate for every application.

identity

Function

Syntax:
  identity object → object

Arguments and Values:
  object—an object.

Description:
  Returns its argument object.

Examples:
  (identity 101) → 101
  (mapcan #'identity (list (list 1 2 3) '(4 5 6))) → (1 2 3 4 5 6)

Notes:
  identity is intended for use with functions that require a function as an argument.
  (eql x (identity x)) returns true for all possible values of x, but (eq x (identity x)) might return false when x is a number or character.
identity could be defined by

(defun identity (x) x)

complement

Function

Syntax:
complement function  → complement-function

Arguments and Values:
function—a function.
complement-function—a function.

Description:
Returns a function that takes the same arguments as function, and has the same side-effect behavior as function, but returns only a single value: a generalized boolean with the opposite truth value of that which would be returned as the primary value of function. That is, when the function would have returned true as its primary value the complement-function returns false, and when the function would have returned false as its primary value the complement-function returns true.

Examples:

(funcall (complement #'zerop) 1) → true
(funcall (complement #'characterp) #\A) → false
(funcall (complement #'member) 'a '(a b c)) → false
(funcall (complement #'member) 'd '(a b c)) → true

See Also:
not

Notes:

(complement x) ≡ #'(lambda (&rest arguments) (not (apply x arguments)))

In Common Lisp, functions with names like “xxx-if-not” are related to functions with names like “xxx-if” in that

(xxx-if-not f . arguments) ≡ (xxx-if (complement f) . arguments)

For example,

(find-if-not #'zerop '(0 0 3)) ≡
(find-if (complement #'zerop) ’(0 0 3)) → 3

Note that since the "xxx-if-not" functions and the :test-not arguments have been deprecated, uses of "xxx-if" functions or :test arguments with \texttt{complement} are preferred.

\textbf{constantly} \hspace{1cm} \textit{Function}

\textbf{Syntax:}
\[
\text{constantly } value \rightarrow \text{function}
\]

\textbf{Arguments and Values:}
\begin{itemize}
  \item \texttt{value}—an \textit{object}.
  \item \texttt{function}—a \textit{function}.
\end{itemize}

\textbf{Description:}
\texttt{constantly} returns a \textit{function} that accepts any number of arguments, that has no side-effects, and that always returns \texttt{value}.

\textbf{Examples:}
\begin{verbatim}
(mapcar (constantly 3) ’(a b c d)) → (3 3 3 3)
(defmacro with-vars (vars &body forms)
  ’((lambda ,vars ,@forms) ,@(mapcar (constantly nil) vars)))
→ \texttt{WITH-VARS}
(macroexpand ’(with-vars (a b) (setq a 3 b (* a a)) (list a b)))
→ ((\texttt{LAMBDA} \texttt{(A B)} \texttt{(SETQ A 3 B \texttt{(* A A)) \texttt{(LIST A B)}} NIL NIL), \texttt{true}}
\end{verbatim}

\textbf{See Also:}
\begin{itemize}
  \item \texttt{identity}
\end{itemize}

\textbf{Notes:}
\texttt{constantly} could be defined by:
\begin{verbatim}
(defun constantly (object)
  #'(lambda (&rest arguments) object))
\end{verbatim}
every, some, notevery, notany

every, some, notevery, notany

Function

Syntax:
  every predicate &rest sequences+ → generalized-boolean
  some predicate &rest sequences+ → result
  notevery predicate &rest sequences+ → generalized-boolean
  notany predicate &rest sequences+ → generalized-boolean

Arguments and Values:
  predicate—a designator for a function of as many arguments as there are sequences.
  sequence—a sequence.
  result—an object.
  generalized-boolean—a generalized boolean.

Description:
  every, some, notevery, and notany test elements of sequences for satisfaction of a given predicate. The first argument to predicate is an element of the first sequence; each succeeding argument is an element of a succeeding sequence.

  Predicate is first applied to the elements with index 0 in each of the sequences, and possibly then to the elements with index 1, and so on, until a termination criterion is met or the end of the shortest of the sequences is reached.

  every returns false as soon as any invocation of predicate returns false. If the end of a sequence is reached, every returns true. Thus, every returns true if and only if every invocation of predicate returns true.

  some returns the first non-nil value which is returned by an invocation of predicate. If the end of a sequence is reached without any invocation of the predicate returning true, some returns false. Thus, some returns true if and only if some invocation of predicate returns true.

  notany returns false as soon as any invocation of predicate returns true. If the end of a sequence is reached, notany returns true. Thus, notany returns true if and only if it is not the case that any invocation of predicate returns true.

  notevery returns true as soon as any invocation of predicate returns false. If the end of a sequence is reached, notevery returns false. Thus, notevery returns true if and only if it is not the case that every invocation of predicate returns true.
Examples:

\[
\begin{align*}
(every \#'characterp "abc") & \rightarrow true \\
(some \#'=' \{(1 2 3 4 5) \,(5 4 3 2 1)\}) & \rightarrow true \\
(notevery \#'< \{(1 2 3 4) \,(5 6 7 8) \,(9 10 11 12)\}) & \rightarrow false \\
(notany \#'> \{(1 2 3 4) \,(5 6 7 8) \,(9 10 11 12)\}) & \rightarrow true
\end{align*}
\]

Exceptional Situations:

Should signal **type-error** if its first argument is neither a symbol nor a function or if any subsequent argument is not a proper sequence.

Other exceptional situations are possible, depending on the nature of the **predicate**.

See Also:

and, or, Section 3.6 (Traversal Rules and Side Effects)

Notes:

\[
\begin{align*}
\text{(notany predicate \{sequence\}*)} & \equiv \text{(not (some predicate \{sequence\}*)}) \\
\text{(notevery predicate \{sequence\}*)} & \equiv \text{(not (every predicate \{sequence\}*)})
\end{align*}
\]

**and**

**Macro**

Syntax:

\[
\text{and \{form\}*} \rightarrow \{result\}*
\]

Arguments and Values:

form—a **form**.

results—the **values** resulting from the evaluation of the last **form**, or the symbols nil or t.

Description:

The macro **and** evaluates each **form** one at a time from left to right. As soon as any **form** evaluates to nil, **and** returns nil without evaluating the remaining **forms**. If all **forms** but the last evaluate to true values, **and** returns the results produced by evaluating the last **form**.

If no **forms** are supplied, (and) returns t.

**and** passes back multiple values from the last **subform** but not from subforms other than the last.

Examples:

\[
\text{(if (and (> n 0))}
\]

5–68 Programming Language—Common Lisp
(< n (length a-simple-vector))
(eq (elt a-simple-vector n) 'foo))
(princ "Foo!")

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

(setq temp1 1 temp2 1 temp3 1) → 1
(and (incf temp1) (incf temp2) (incf temp3)) → 2
(and (eql 2 temp1) (eql 2 temp2) (eql 2 temp3)) → true
(decf temp3) → 1
(and (decf temp1) (decf temp2) (eq temp3 'nil) (decf temp3)) → NIL
(and (eql temp1 temp2) (eql temp2 temp3)) → true
(and) → T

See Also:
cond, every, if, or, when

Notes:

(and form) ≡ (let () form)
(and form1 form2 ...) ≡ (when form1 (and form2 ...))

cond

Macro

Syntax:
cond {↓ clause}* → {result}*  

clause::=(test-form {form}* )

Arguments and Values:
test-form—a form.
forms—an implicit progn.
results—the values of the forms in the first clause whose test-form yields true, or the primary value of the test-form if there are no forms in that clause, or else nil if no test-form yields true.

Description:
cond allows the execution of forms to be dependent on test-form.
Test-forms are evaluated one at a time in the order in which they are given in the argument list until a test-form is found that evaluates to true.

If there are no forms in that clause, the primary value of the test-form is returned by the cond form. Otherwise, the forms associated with this test-form are evaluated in order, left to right, as an implicit progn, and the values returned by the last form are returned by the cond form.

Once one test-form has yielded true, no additional test-forms are evaluated. If no test-form yields true, nil is returned.

Examples:

(defun select-options ()
  (cond ((= a 1) (setq a 2))
        ((= a 2) (setq a 3))
        ((and (= a 3) (floor a 2)))
        (t (floor a 3)))
  \rightarrow SELECT-OPTIONS

(setq a 1) \rightarrow 1
(select-options) \rightarrow 2
a \rightarrow 2
(select-options) \rightarrow 3
a \rightarrow 3
(select-options) \rightarrow 1
(setq a 5) \rightarrow 5
(select-options) \rightarrow 1, 2

See Also:
if, case.

if

Special Operator

Syntax:
if test-form then-form | else-form] \rightarrow \{result\}*

Arguments and Values:
Test-form—a form.
Then-form—a form.
Else-form—a form. The default is nil.
results—if the test-form yielded true, the values returned by the then-form; otherwise, the values returned by the else-form.
Description:

if allows the execution of a form to be dependent on a single test-form.

First test-form is evaluated. If the result is true, then then-form is selected; otherwise else-form is selected. Whichever form is selected is then evaluated.

Examples:

(if t 1) → 1
(if nil 1 2) → 2
(defun test ()
  (dolist (truth-value '(t nil 1 (a b c)))
    (if truth-value (print 'true) (print 'false))
    (print truth-value))) → TEST
(test)
▷ TRUE T
▷ FALSE NIL
▷ TRUE 1
▷ TRUE (A B C)
→ NIL

See Also:

cond, unless, when

Notes:

(if test-form then-form else-form)
≡ (cond (test-form then-form) (t else-form))

Or

Syntax:

or {form}* → {results}*  

Arguments and Values:

form—a form.

results—the values or primary value (see below) resulting from the evaluation of the last form executed or nil.

Description:

or evaluates each form, one at a time, from left to right. The evaluation of all forms terminates when a form evaluates to true (i.e., something other than nil).
If the *evaluation* of any *form* other than the last returns a *primary value* that is *true*, or *immediately* returns that *value* (but no additional *values*) without evaluating the remaining *forms*. If every *form* but the last returns *false* as its *primary value*, or returns all *values* returned by the last *form*. If no *forms* are supplied, or returns *nil*.

**Examples:**

```
(or) → NIL
(or temp0 temp1 (setq temp2 37)) → 10
(temp2 → 20
(or (incf temp1) (incf temp2) (incf temp3)) → 11
(temp1 → 11
(temp2 → 20
(temp3 → 30
(or (values) templ) → 11
(or (values temp1 temp2) temp3) → 11
(or temp0 (values temp1 temp2)) → 11, 20
(or (values temp0 temp1) (values temp2 temp3)) → 20, 30
```

**See Also:**

`and`, `some`, `unless`

---

**when**, **unless**

*Macro*

**Syntax:**

```
when test-form {form}* → {result}*  
unless test-form {form}* → {result}*  
```

**Arguments and Values:**

- **test-form**—a *form*.
- **forms**—an implicit *progn*.
- **results**—the *values* of the *forms* in a *when* *form* if the *test-form* *yields* *true* or in an *unless* *form* if the *test-form* *yields* *false*; otherwise *nil*.

**Description:**

`when` and `unless` allow the execution of *forms* to be dependent on a single *test-form*.

In a *when* *form*, if the *test-form* *yields* *true*, the *forms* are *evaluated* in order from left to right and the *values* returned by the *forms* are returned from the *when* *form*. Otherwise, if the *test-form* *yields* *false*, the *forms* are not *evaluated*, and the *when* *form* returns *nil*.  

---

5–72 Programming Language—Common Lisp
when, unless

In an unless form, if the test-form yields false, the forms are evaluated in order from left to right and the values returned by the forms are returned from the unless form. Otherwise, if the test-form yields false, the forms are not evaluated, and the unless form returns nil.

Examples:

\[
\begin{align*}
&(\text{when } t \ \text{'hello}) \rightarrow \text{HELLO} \\
&(\text{unless } t \ \text{'hello}) \rightarrow \text{NIL} \\
&(\text{when } \text{nil } \text{'hello}) \rightarrow \text{NIL} \\
&(\text{unless } \text{nil } \text{'hello}) \rightarrow \text{HELLO} \\
&(\text{when } t) \rightarrow \text{NIL} \\
&(\text{unless } \text{nil}) \rightarrow \text{NIL} \\
&(\text{when } t \ (\text{prin1 } 1) \ (\text{prin1 } 2) \ (\text{prin1 } 3)) \\
\triangleright 123 \\
\rightarrow 3 \\
&(\text{unless } t \ (\text{prin1 } 1) \ (\text{prin1 } 2) \ (\text{prin1 } 3)) \rightarrow \text{NIL} \\
&(\text{when } \text{nil } \ (\text{prin1 } 1) \ (\text{prin1 } 2) \ (\text{prin1 } 3)) \rightarrow \text{NIL} \\
&(\text{unless } \text{nil } \ (\text{prin1 } 1) \ (\text{prin1 } 2) \ (\text{prin1 } 3)) \\
\triangleright 123 \\
\rightarrow 3 \\
&(\text{let } ((x 3)) \\
\quad (\text{list } (\text{when } (\text{oddp } x) \ (\text{incf } x) \ (\text{list } x)) \\
\quad (\text{when } (\text{oddp } x) \ (\text{incf } x) \ (\text{list } x)) \\
\quad (\text{unless } (\text{oddp } x) \ (\text{incf } x) \ (\text{list } x)) \\
\quad (\text{unless } (\text{oddp } x) \ (\text{incf } x) \ (\text{list } x)) \\
\quad (\text{if } (\text{oddp } x) \ (\text{incf } x) \ (\text{list } x)) \\
\quad (\text{if } (\text{oddp } x) \ (\text{incf } x) \ (\text{list } x)) \\
\quad (\text{if } (\text{not } (\text{oddp } x)) \ (\text{incf } x) \ (\text{list } x)) \\
\quad (\text{if } (\text{not } (\text{oddp } x)) \ (\text{incf } x) \ (\text{list } x))) \\
\rightarrow ((\text{4}) \ \text{NIL} \ (\text{5}) \ \text{NIL} \ 6 \ (6) \ 7 \ (7)) \\
\end{align*}
\]

See Also:
and, cond, if, or

Notes:

\[
\begin{align*}
&(\text{when } \text{test } \{\text{form}\}^+) \equiv (\text{and } \text{test } (\text{progn } \{\text{form}\}^+)) \\
&(\text{when } \text{test } \{\text{form}\}^+) \equiv (\text{cond } (\text{test } \{\text{form}\}^+)) \\
&(\text{when } \text{test } \{\text{form}\}^+) \equiv (\text{if } \text{test } (\text{progn } \{\text{form}\}^+) \ \text{nil}) \\
&(\text{when } \text{test } \{\text{form}\}^+) \equiv (\text{unless } (\text{not } \text{test } \{\text{form}\}^+)) \\
&(\text{unless } \text{test } \{\text{form}\}^+) \equiv (\text{cond } ((\text{not } \text{test } \{\text{form}\}^+)) \\
&(\text{unless } \text{test } \{\text{form}\}^+) \equiv (\text{if } \text{test } \text{nil } (\text{progn } \{\text{form}\}^+)) \\
&(\text{unless } \text{test } \{\text{form}\}^+) \equiv (\text{when } (\text{not } \text{test } \{\text{form}\}^+))
\end{align*}
\]
case, ccase, ecase  

Macro

Syntax:

\[
\text{case} \quad \text{keyform} \quad \{\downarrow \text{normal-clause}\}^* \quad \{\downarrow \text{otherwise-clause}\} \quad \rightarrow \quad \{\text{result}\}^* \\
\text{ccase} \quad \text{keyplace} \quad \{\downarrow \text{normal-clause}\}^* \quad \rightarrow \quad \{\text{result}\}^* \\
\text{ecase} \quad \text{keyform} \quad \{\downarrow \text{normal-clause}\}^* \quad \rightarrow \quad \{\text{result}\}^* \\
\text{normal-clause}::=(\text{keys} \quad \{\text{form}\}^*) \\
\text{otherwise-clause}::=(\{\text{otherwise} \mid \text{t}\} \quad \{\text{form}\}^*) \\
\text{clause}::=\text{normal-clause} \mid \text{otherwise-clause}
\]

Arguments and Values:

- **keyform**—a *form*; evaluated to produce a *test-key*.
- **keyplace**—a *form*; evaluated initially to produce a *test-key*. Possibly also used later as a *place* if no *keys* match.
- **test-key**—an object produced by evaluating *keyform* or *keyplace*.
- **keys**—a *designator* for a *list of objects*. In the case of *case*, the *symbols* *t* and *otherwise* may not be used as the *keys designator*. To refer to these *symbols* by themselves as *keys*, the designators (*t*) and (otherwise), respectively, must be used instead.
- **forms**—an *implicit prog*.
- **results**—the *values* returned by the *forms* in the matching *clause*.

Description:

These *macros* allow the conditional execution of a body of *forms* in a *clause* that is selected by matching the *test-key* on the basis of its identity.

The *keyform* or *keyplace* is evaluated to produce the *test-key*.

Each of the *normal-clauses* is then considered in turn. If the *test-key* is the *same* as any *key* for that *clause*, the *forms* in that *clause* are evaluated as an *implicit prog*, and the *values* it returns are returned as the value of the *case*, *ccase*, or *ecase* form.

These *macros* differ only in their *behavior* when no *normal-clause* matches; specifically:

- **case**

  If no *normal-clause* matches, and there is an *otherwise-clause*, then that *otherwise-clause*
case, ccase, ecase

automatically matches; the forms in that clause are evaluated as an implicit progn, and the values it returns are returned as the value of the case.

If there is no otherwise-clause, case returns nil.

ccase

If no normal-clause matches, a correctable error of type type-error is signaled. The offending datum is the test-key and the expected type is type equivalent to (member key1 key2 ...). The store-value restart can be used to correct the error.

If the store-value restart is invoked, its argument becomes the new test-key, and is stored in keyplace as if by (setf keyplace test-key). Then ccase starts over, considering each clause anew.

The subforms of keyplace might be evaluated again if none of the cases holds.

ecase

If no normal-clause matches, a non-correctable error of type type-error is signaled. The offending datum is the test-key and the expected type is type equivalent to (member key1 key2 ...).

Note that in contrast with ccase, the caller of ecase may rely on the fact that ecase does not return if a normal-clause does not match.

Examples:

```
(dolist (k '(1 2 3 :four #\v () t 'other))
  (format t "~S "
    (case k ((1 2) 'clause1)
      (3 'clause2)
      (nil 'no-keys-so-never-seen)
      ((nil) 'nilslot)
      (((:four #\v) 'clause4)
       ((t) 'tslot)
       (otherwise 'others))))))
> CLAUSE1 CLAUSE1 CLAUSE2 CLAUSE4 CLAUSE4 NILSLOT TSLOT OTHERS
→ NIL
(defun add-em (x) (apply #'+ (mapcar #'decode x)))
→ ADD-EM
(defun decode (x)
  (ccase x
    ((i uno) 1)
    ((ii dos) 2)
    ((iii tres) 3)
    (otherwise (iv cuatro) 4)))
→ DECODE
```

Data and Control Flow 5–75
(add-em '(uno iii)) → 4
(add-em '(uno iiii))
▷ Error: The value of X, III, is not I, UNO, II, DOS, III,
▷ TRES, IV, or CUATRO.
▷ 1: Supply a value to use instead.
▷ 2: Return to Lisp Toplevel.
▷ Debug> :CONTINUE 1
▷ Value to evaluate and use for X: 'IV
→ 5

Side Effects:
The debugger might be entered. If the store-value restart is invoked, the value of keyplace might be changed.

Affected By:
ccase and ecase, since they might signal an error, are potentially affected by existing handlers and *debug-io*.

Exceptional Situations:
ccase and ecase signal an error of type type-error if no normal-clause matches.

See Also:
cond, typecase, setf, Section 5.1 (Generalized Reference)

Notes:

(case test-key
  {([key]* {form})*})*
≡
(let ((#1=#:g0001 test-key))
  (cond {((member #1# '{key}*) {form})*})*)

The specific error message used by ecase and ccase can vary between implementations. In situations where control of the specific wording of the error message is important, it is better to use case with an otherwise-clause that explicitly signals an error with an appropriate message.

typecase, ctypecase, etypecase

Macro

Syntax:
typecase keyform {normal-clause}* [otherwise-clause] → {result}*
ctypecase keyplace {normal-clause}* → {result}*
etypecase keyform {normal-clause}* → {result}*

5–76 Programming Language—Common Lisp
**typecase, ctypecase, etypecase**

\[\text{normal-clause::}=\text{type}\ \{\text{form}\}^{*}\]
\[\text{otherwise-clause::}=\{\text{otherwise}\ |\ t\} \ \{\text{form}\}^{*}\]
\[\text{clause::}=\text{normal-clause} | \text{otherwise-clause}\]

**Arguments and Values:**
- **keyform**—a form; evaluated to produce a **test-key**.
- **keyplace**—a form; evaluated initially to produce a **test-key**. Possibly also used later as a place if no types match.
- **test-key**—an object produced by evaluating **keyform** or **keyplace**.
- **type**—a type specifier.
- **forms**—an implicit **progn**.
- **results**—the values returned by the **forms** in the matching **clause**.

**Description:**
These macros allow the conditional execution of a body of **forms** in a **clause** that is selected by matching the **test-key** on the basis of its **type**.

The **keyform** or **keyplace** is evaluated to produce the **test-key**.

Each of the **normal-clauses** is then considered in turn. If the **test-key** is of the type given by the **clause**’s **type**, the **forms** in that **clause** are evaluated as an implicit **progn**, and the **values** it returns are returned as the value of the **typecase**, **ctypecase**, or **etypecase** form.

These macros differ only in their behavior when no **normal-clause** matches; specifically:

**typecase**
If no **normal-clause** matches, and there is an **otherwise-clause**, then that **otherwise-clause** automatically matches; the **forms** in that **clause** are evaluated as an implicit **progn**, and the **values** it returns are returned as the value of the **typecase**.

If there is no **otherwise-clause**, **typecase** returns nil.

**ctypecase**
If no **normal-clause** matches, a correctable error of type **type-error** is signaled. The offending datum is the **test-key** and the expected type is **type equivalent** to (or **type1 type2 ...**). The **store-value restart** can be used to correct the error.

If the **store-value restart** is invoked, its argument becomes the new **test-key**, and is stored in **keyplace** as if by (setf **keyplace** **test-key**). Then **ctypecase** starts over, considering each **clause** anew.
**typecase, ctypecase, etypecase**

If the `store-value` restart is invoked interactively, the user is prompted for a new `test-key` to use.

The subforms of `keyplace` might be evaluated again if none of the cases holds.

**etypecase**

If no `normal-clause` matches, a non-correctable error of type `type-error` is signaled. The offending datum is the `test-key` and the expected type is `type equivalent` to `(or type1 type2 ...)`. Note that in contrast with `ctypecase`, the caller of `etypecase` may rely on the fact that `etypecase` does not return if a `normal-clause` does not match.

In all three cases, is permissible for more than one clause to specify a matching type, particularly if one is a `subtype` of another; the earliest applicable clause is chosen.

**Examples:**

```lisp
;;; (Note that the parts of this example which use TYPE-OF
;;;; are implementation-dependent.)
(defun what-is-it (x)
  (format t "~&~S is ~A.~%
          X (typecase X
               (float "a float")
               (null "a symbol, boolean false, or the empty list")
               (list "a list")
               (t (format nil "a(n) ~(~A~)" (type-of X))))))
→ WHAT-IS-IT
(map 'nil #what-is-it '(nil (a b) 7.0 7 box))
▷ NIL is a symbol, boolean false, or the empty list.
▷ (A B) is a list.
▷ 7.0 is a float.
▷ 7 is a(n) integer.
▷ BOX is a(n) symbol.
▷ NIL
(setq x 1/3)
▷ 1/3
(ctypecase X
  (integer (+ x 4))
  (symbol (symbol-value x)))
▷ Error: The value of X, 1/3, is neither an integer nor a symbol.
▷ To continue, type :CONTINUE followed by an option number:
▷ 1: Specify a value to use instead.
▷ 2: Return to Lisp Toplevel.
▷ Debug> :CONTINUE 1
▷ Use value: 3.7
```

5–78   Programming Language—Common Lisp
Error: The value of X, 3.7, is neither an integer nor a symbol.
To continue, type :CONTINUE followed by an option number:

1: Specify a value to use instead.
2: Return to Lisp Toplevel.

Debug> :CONTINUE 1
Use value: 12
→ 48
x → 12

Affected By:
ctypecase and etypecase, since they might signal an error, are potentially affected by existing handlers and *debug-io*.

Exceptional Situations:
ctypecase and etypecase signal an error of type type-error if no normal-clause matches.
The compiler may choose to issue a warning of type style-warning if a clause will never be selected because it is completely shadowed by earlier clauses.

See Also:
case, cond, setf, Section 5.1 (Generalized Reference)

Notes:

\[
\text{ctypecase test-key}
\{(\text{type } \{\text{form}\}^*)^*\}
\equiv
\begin{align*}
\text{let } ((\#1=\#:g0001 \text{ test-key})) \\
\text{(cond } \{(\text{typep } \#1 \ 'type) \{\text{form}\}^*)^*\})
\end{align*}
\]

The specific error message used by etypecase and ctypecase can vary between implementations. In situations where control of the specific wording of the error message is important, it is better to use typecase with an otherwise-clause that explicitly signals an error with an appropriate message.

\begin{center}
\textbf{multiple-value-bind} \quad \textit{Macro}
\end{center}

Syntax:

\begin{align*}
\text{multiple-value-bind } \&(\{\text{var}\}^*) \text{ values-form } \{(\text{declaration}\}^* \{\text{form}\}^* \\
→ \{\text{result}\}^*
\end{align*}

Arguments and Values:
\textit{var}—a symbol naming a variable; not evaluated.
values-form—a form; evaluated.

declaration—a declare expression; not evaluated.

forms—an implicit progn.

results—the values returned by the forms.

Description:

Creates new variable bindings for the vars and executes a series of forms that use these bindings.

The variable bindings created are lexical unless special declarations are specified.

Values-form is evaluated, and each of the vars is bound to the respective value returned by that form. If there are more vars than values returned, extra values of nil are given to the remaining vars. If there are more values than vars, the excess values are discarded. The vars are bound to the values over the execution of the forms, which make up an implicit progn. The consequences are unspecified if a type declaration is specified for a var, but the value to which that var is bound is not consistent with the type declaration.

The scopes of the name binding and declarations do not include the values-form.

Examples:

(multiple-value-bind (f r)
  (floor 130 11)
  (list f r)) → (11 9)

See Also:

let, multiple-value-call

Notes:

(multiple-value-bind ({var}*) values-form {form}*)

≡ (multiple-value-call #'(lambda (&optional {var}* rest #1=#:ignore)
  (declare (ignore #1#))
  {form}*)
  values-form)
multiple-value-call

Special Operator

Syntax:

\[
\text{multiple-value-call} \quad \text{function-form} \quad \text{form}* \quad \rightarrow \quad \{\text{result}\}*
\]

Arguments and Values:

- **function-form** — a form; evaluated to produce function.
- **function** — a function designator resulting from the evaluation of function-form.
- **form** — a form.
- **results** — the values returned by the function.

Description:

Applies function to a list of the objects collected from groups of multiple values.

**multiple-value-call** first evaluates the function-form to obtain function, and then evaluates each form. All the values of each form are gathered together (not just one value from each) and given as arguments to the function.

Examples:

- \[(\text{multiple-value-call} \quad #'\text{list} \quad 1 \quad '/ \quad (\text{values} \quad 2 \quad 3) \quad '/ \quad (\text{values}) \quad '/ \quad (\text{floor} \quad 2.5))\]
  \(\rightarrow\) \((1 \quad 2 \quad 3 \quad / \quad 2 \quad 0.5)\)
- \((+ \quad (\text{floor} \quad 5 \quad 3) \quad (\text{floor} \quad 19 \quad 4))\) \(\equiv\) \((+ \quad 1 \quad 4)\)
  \(\rightarrow\) \(5\)
- \[(\text{multiple-value-call} \quad #'+ \quad (\text{floor} \quad 5 \quad 3) \quad (\text{floor} \quad 19 \quad 4))\) \(\equiv\) \((+ \quad 1 \quad 2 \quad 4 \quad 3)\)
  \(\rightarrow\) \(10\)

See Also:

- multiple-value-list, multiple-value-bind

multiple-value-list

Macro

Syntax:

\[
\text{multiple-value-list} \quad \text{form} \quad \rightarrow \quad \text{list}
\]

Arguments and Values:

- **form** — a form; evaluated as described below.
- **list** — a list of the values returned by form.
Description:
multiple-value-list evaluates form and creates a list of the multiple values it returns.

Examples:
(multiple-value-list (floor -3 4)) → (-1 1)

See Also:
values-list, multiple-value-call

Notes:
multiple-value-list and values-list are inverses of each other.
(multiple-value-list form) ≡ (multiple-value-call #'list form)

multiple-value-prog1
Special Operator

Syntax:
multiple-value-prog1 first-form {form}* → first-form-results

Arguments and Values:
  first-form—a form; evaluated as described below.
  form—a form; evaluated as described below.
  first-form-results—the values resulting from the evaluation of first-form.

Description:
multiple-value-prog1 evaluates first-form and saves all the values produced by that form. It then evaluates each form from left to right, discarding their values.

Examples:
(setq temp '(1 2 3)) → (1 2 3)
multiple-value-prog1
(values-list temp)
(setq temp nil)
(values-list temp)) → 1, 2, 3

See Also:
prog1
multiple-value-setq

Syntax:

\texttt{multiple-value-setq \textit{vars} \textit{form} \rightarrow \textit{result}}

Arguments and Values:

\textit{vars}—a list of symbols that are either variable names or names of symbol macros.

\textit{form}—a form.

\textit{result}—The primary value returned by the \textit{form}.

Description:

\texttt{multiple-value-setq} assigns values to \textit{vars}.

The \textit{form} is evaluated, and each \textit{var} is assigned to the corresponding \textit{value} returned by that \textit{form}. If there are more \textit{vars} than \textit{values} returned, \texttt{nil} is assigned to the extra \textit{vars}. If there are more \textit{values} than \textit{vars}, the extra \textit{values} are discarded.

If any \textit{var} is the \textit{name} of a \textit{symbol macro}, then it is assigned as if by \texttt{setf}. Specifically,

\begin{verbatim}
(multiple-value-setq (symbol1 ... symbol_n) value-producing-form)

defined to always behave in the same way as

(values (setf (values symbol1 ... symbol_n) value-producing-form))
\end{verbatim}

in order that the rules for order of evaluation and side-effects be consistent with those used by \texttt{setf}. See Section 5.1.2.3 (VALUES Forms as Places).

Examples:

\begin{verbatim}
(multiple-value-setq (quotient remainder) (truncate 3.2 2)) \rightarrow 1
quotient \rightarrow 1
remainder \rightarrow 1.2
(multiple-value-setq (a b c) (values 1 2)) \rightarrow 1
a \rightarrow 1
b \rightarrow 2
c \rightarrow NIL
(multiple-value-setq (a b) (values 4 5 6)) \rightarrow 4
a \rightarrow 4
b \rightarrow 5
\end{verbatim}

See Also:

\texttt{setq, symbol-macrolet}
values

Accessor

Syntax:
values &rest object  → {object}*

(setf (values &rest place) new-values)

Arguments and Values:
object—an object.
place—a place.
new-value—an object.

Description:
values returns the objects as multiple values.

setf of values is used to store the multiple values new-values into the places. See Section 5.1.2.3 (VALUES Forms as Places).

Examples:

(values) → (no values)
(values 1) → 1
(values 1 2) → 1, 2
(values 1 2 3) → 1, 2, 3
(values (values 1 2 3) 4 5) → 1, 4, 5
(defun polar (x y)
  (values (sqrt (+ (* x x) (* y y))) (atan y x))) → POLAR
(multiple-value-bind (r theta) (polar 3.0 4.0)
  (vector r theta))
  → #(5.0 0.927295)

Sometimes it is desirable to indicate explicitly that a function returns exactly one value. For example, the function

(defun foo (x y)
  (floor (+ x y) y)) → FOO

returns two values because floor returns two values. It may be that the second value makes no sense, or that for efficiency reasons it is desired not to compute the second value. values is the standard idiom for indicating that only one value is to be returned:
(defun foo (x y)
  (values (floor (+ x y) y))) → FOO

This works because values returns exactly one value for each of args; as for any function call, if any of args produces more than one value, all but the first are discarded.

See Also:
values-list, multiple-value-bind, multiple-values-limit, Section 3.1 (Evaluation)

Notes:
Since values is a function, not a macro or special form, it receives as arguments only the primary values of its argument forms.

values-list

Function

Syntax:
values-list list → {element}*

Arguments and Values:
list—a list.

elements—the elements of the list.

Description:
Returns the elements of the list as multiple values.

Examples:
(values-list nil) → ⟨no values⟩
(values-list '(1)) → 1
(values-list '(1 2)) → 1, 2
(values-list '(1 2 3)) → 1, 2, 3

Exceptional Situations:
Should signal type-error if its argument is not a proper list.

See Also:
multiple-value-bind, multiple-value-list, multiple-values-limit, values

Notes:
(values-list list) ≡ (apply #'values list)

(equal x (multiple-value-list (values-list x))) returns true for all lists x.
multiple-values-limit

Constant Variable

Constant Value:
An integer not smaller than 20, the exact magnitude of which is implementation-dependent.

Description:
The upper exclusive bound on the number of values that may be returned from a function, bound or assigned by multiple-value-bind or multiple-value-setq, or passed as a first argument to nth-value. (If these individual limits might differ, the minimum value is used.)

See Also: lambda-parameters-limit, call-arguments-limit

Notes:
Implementors are encouraged to make this limit as large as possible.

nth-value

Macro

Syntax:

nth-value n form → object

Arguments and Values:

n—a non-negative integer; evaluated.
form—a form; evaluated as described below.
object—an object.

Description:
Evaluates n and then form, returning as its only value the nth value yielded by form, or nil if n is greater than or equal to the number of values returned by form. (The first returned value is numbered 0.)

Examples:

(nth-value 0 (values 'a 'b)) → A
(nth-value 1 (values 'a 'b)) → B
(nth-value 2 (values 'a 'b)) → NIL
(let* ((x 83927472397238947423879243432432432)       (y 32423489732))

5–86 Programming Language—Common Lisp
(a (nth-value 1 (floor x y)))
(b (mod x y)))
(values a b (= a b))
→ 3332987528, 3332987528, true

See Also:
multiple-value-list, nth

Notes:
Operationally, the following relationship is true, although nth-value might be more efficient in some implementations because, for example, some consing might be avoided.

(nth-value n form)  ≡  (nth n (multiple-value-list form))

**prog, prog**

Macro

Syntax:

\[
\text{prog} \left( \{\text{var} \mid (\text{var} \ \text{init-form})\}^{*}\right) \{\text{declaration}\}^{*} \{\text{tag} \mid \text{statement}\}^{*}
\]

→ \{result\}*

\[
\text{prog*} \left( \{\text{var} \mid (\text{var} \ \text{init-form})\}^{*}\right) \{\text{declaration}\}^{*} \{\text{tag} \mid \text{statement}\}^{*}
\]

→ \{result\}*

Arguments and Values:

var—variable name.

init-form—a form.

declaratio—a declare expression; not evaluated.

tag—a go tag; not evaluated.

statement—a compound form; evaluated as described below.

results—nil if a normal return occurs, or else, if an explicit return occurs, the values that were transferred.

Description:

Three distinct operations are performed by **prog** and **prog***: they bind local variables, they permit use of the return statement, and they permit use of the go statement. A typical **prog** looks like this:

\[
\text{prog} \left( \text{var1} \ \text{var2} \ (\text{var3} \ \text{init-form-3}) \ \text{var4} \ (\text{var5} \ \text{init-form-5})\right) \\
\{\text{declaration}\}^{*}
\]
**prog, prog**

```
statement1
tag1
statement2
statement3
statement4
tag2
statement5
...
)
```

For **prog**, *init-forms* are evaluated first, in the order in which they are supplied. The *vars* are then bound to the corresponding values in parallel. If no *init-form* is supplied for a given *var*, that *var* is bound to nil.

The body of **prog** is executed as if it were a *tagbody* form; the *go* statement can be used to transfer control to a *tag*. *Tags* label *statements*.

**prog** implicitly establishes a *block* named nil around the entire **prog** form, so that *return* can be used at any time to exit from the **prog** form.

The difference between **prog** and **prog** is that in **prog** the binding and initialization of the *vars* is done sequentially, so that the *init-form* for each one can use the values of previous ones.

**Examples:**

```
(prog* ((y z) (x (car y)))
  (return x))
```

returns the *car* of the value of z.

```
(setq a 1) → 1
(prog ((a 2) (b a)) (return (if (= a b) '=' '/=)) → /=
(prog* ((a 2) (b a)) (return (if (= a b) '=' '/=)) → =
(prog () 'no-return-value) → NIL
```

```
(defun king-of-confusion (w)
  ;"Take a cons of two lists and make a list of conses. Think of this function as being like a zipper."
  (prog (x y z) ;Initialize x, y, z to NIL
    (setq y (car w) z (cdr w))
    loop
      (cond ((null y) (return x))
             ((null z) (go err)))
    rejoin
      (setq x (cons (cons (car y) (car z)) x))
      (setq y (cdr y) z (cdr z))
      (go loop)
  err
```

5–88  Programming Language—Common Lisp
(cerror "Will self-pair extraneous items"
 "Mismatch - gleep! "S" y)
(setq z y)
(go rejoin))) → KING-OF-CONFUSION

This can be accomplished more perspicuously as follows:

(defun prince-of-clarity (w)
 "Take a cons of two lists and make a list of conses. Think
 of this function as being like a zipper."
 (do ((y (car w) (cdr y))
       (z (cdr w) (cdr z))
       (x '() (cons (cons (car y) (car z)) x)))
    ((null y) x)
    (when (null z)
      (cerror "Will self-pair extraneous items"
              "Mismatch - gleep! "S" y)
      (setq z y)))) → PRINCE-OF-CLARITY

See Also:
block, let, tagbody, go, return, Section 3.1 (Evaluation)

Notes:
prog can be explained in terms of block, let, and tagbody as follows:

(prog variable-list declaration . body)
≡ (block nil (let variable-list declaration (tagbody . body)))

prog1, prog2
Macro

Syntax:
prog1 first-form {form}* → result-1
prog2 first-form second-form {form}* → result-2

Arguments and Values:
first-form—a form; evaluated as described below.
second-form—a form; evaluated as described below.
forms—an implicit prog; evaluated as described below.
result-1—the primary value resulting from the evaluation of first-form.
prog1, prog2

result-2—the primary value resulting from the evaluation of second-form.

Description:
prog1 evaluates first-form and then forms, yielding as its only value the primary value yielded by first-form.

prog2 evaluates first-form, then second-form, and then forms, yielding as its only value the primary value yielded by first-form.

Examples:

```lisp
(setq temp 1) → 1
(prog1 temp (print temp) (incf temp) (print temp))
▷ 1
▷ 2
→ 1
(prog1 temp (setq temp nil)) → 2
temp → NIL
(prog1 (values 1 2 3) 4) → 1
(setq temp (list 'a 'b 'c))
(prog1 (car temp) (setf (car temp) 'alpha)) → A
temp → (ALPHA B C)
(flet ((swap-symbol-values (x y)
             (setf (symbol-value x)
                (prog1 (symbol-value y))
                (setf (symbol-value y) (symbol-value x))))))
    (let ((*foo* 1) (*bar* 2))
      (declare (special *foo* *bar*))
      (swap-symbol-values '*foo* '*bar*)
      (values *foo* *bar*))
→ 2, 1
(setq temp 1) → 1
(prog2 (incf temp) (incf temp) (incf temp)) → 3
temp → 4
(prog2 1 (values 2 3 4) 5) → 2
```

See Also:
multiple-value-prog1, prog

Notes:
prog1 and prog2 are typically used to evaluate one or more forms with side effects and return a value that must be computed before some or all of the side effects happen.

```lisp
(prog1 {form}*) ≡ (values (multiple-value-prog1 {form}*))
(prog2 forml {form}*) ≡ (let () forml (prog1 {form}*))
```

5–90 Programming Language—Common Lisp
**progn**  

*Special Operator*

**Syntax:**

\[
\text{progn} \{\text{form}\}^* \rightarrow \{\text{result}\}^*
\]

**Arguments and Values:**

*forms*—an implicit progn.

*results*—the values of the *forms*.

**Description:**

*progn* evaluates *forms*, in the order in which they are given. The values of each *form* but the last are discarded. If *progn* appears as a top level form, then all *forms* within that *progn* are considered by the compiler to be top level forms.

**Examples:**

\[
\begin{align*}
\text{(progn)} & \rightarrow \text{NIL} \\
\text{(progn 1 2 3)} & \rightarrow 3 \\
\text{(progn (values 1 2 3))} & \rightarrow 1, 2, 3 \\
\text{(setq a i)} & \rightarrow 1 \\
\text{(if a}
\begin{align*}
\text{\quad (progn (setq a nil) 'here)} \\
\text{\qquad (progn (setq a t) 'there))} & \rightarrow \text{HERE} \\
a & \rightarrow \text{NIL}
\end{align*}
\end{align*}
\]

**See Also:**

progl, prog2, Section 3.1 (Evaluation)

**Notes:**

Many places in Common Lisp involve syntax that uses implicit progs. That is, part of their syntax allows many *forms* to be written that are to be evaluated sequentially, discarding the results of all *forms* but the last and returning the results of the last *form*. Such places include, but are not limited to, the following: the body of a lambda expression; the bodies of various control and conditional *forms* (e.g., case, catch, progn, and when).
define-modify-macro

Syntax:

\[
\text{define-modify-macro} \text{ name lambda-list function [documentation]} \rightarrow \text{name}
\]

Arguments and Values:

- **name** — a symbol.
- **lambda-list** — a define-modify-macro lambda list
- **function** — a symbol.
- **documentation** — a string; not evaluated.

Description:

\text{define-modify-macro} defines a \text{macro} named \text{name} to \text{read} and \text{write} a \text{place}.

The arguments to the new \text{macro} are a \text{place}, followed by the arguments that are supplied in \text{lambda-list}. Macros defined with \text{define-modify-macro} correctly pass the \text{environment parameter} to \text{get-setf-expansion}.

When the \text{macro} is invoked, \text{function} is applied to the old contents of the \text{place} and the \text{lambda-list} arguments to obtain the new value, and the \text{place} is updated to contain the result.

Except for the issue of avoiding multiple evaluation (see below), the expansion of a \text{define-modify-macro} is equivalent to the following:

\[
\text{(defmacro name (reference . lambda-list)}
\text{ documentation}
\text{ '(setf ,reference (function ,reference ,arg1 ,arg2 ...)))}
\]

where \text{arg1}, \text{arg2}, ..., are the parameters appearing in \text{lambda-list}; appropriate provision is made for a \text{rest parameter}.

The \text{subforms} of the macro calls defined by \text{define-modify-macro} are evaluated as specified in Section 5.1.1.1 (Evaluation of Subforms to Places).

\text{Documentation} is attached as a \text{documentation string} to \text{name} (as kind \text{function}) and to the \text{macro function}.

If a \text{define-modify-macro} \text{form} appears as a \text{top level form}, the \text{compiler} must store the \text{macro} definition at compile time, so that occurrences of the \text{macro} later on in the file can be expanded correctly.
Examples:

```
(define-modify-macro appendf (&rest args)
  append "Append onto list") → APPENDF
(setq x '(a b c) y x) → (A B C)
(appendf x '(d e f) '(1 2 3)) → (A B C D E F 1 2 3)
x → (A B C D E F 1 2 3)
y → (A B C)
(define-modify-macro new-incf (optional (delta 1)) +)
(define-modify-macro unionf (other-set &rest keywords) union)
```

Side Effects:

A macro definition is assigned to name.

See Also:

defsetf, define-set-expander, documentation, Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

defsetf

Macro

Syntax:

The “short form”:

```
defsetf access-fn update-fn [documentation]
  → access-fn
```

The “long form”:

```
defsetf access-fn lambda-list [{store-variable}]* [ {declaration}* | documentation ] {form}* 
  → access-fn
```

Arguments and Values:

- **access-fn**—a symbol which names a function or a macro.
- **update-fn**—a symbol naming a function or macro.
- **lambda-list**—a defsetf lambda list.
- **store-variable**—a symbol (a variable name).
- **declaration**—a declare expression; not evaluated.
- **documentation**—a string; not evaluated.
- **form**—a form.
defsetf

Description:

`defsetf` defines how to `setf` a place of the form `(access-fn ...)` for relatively simple cases. (See `define-setf-expander` for more general access to this facility.) It must be the case that the function or macro named by `access-fn` evaluates all of its arguments.

`defsetf` may take one of two forms, called the “short form” and the “long form,” which are distinguished by the type of the second argument.

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

The long form `defsetf` resembles `defmacro`. The `lambda-list` describes the arguments of `access-fn`. The `store-variables` describe the value or values to be stored into the place. The `body` must compute the expansion of a `setf` of a call on `access-fn`. The expansion function is defined in the same lexical environment in which the `defsetf` form appears.

During the evaluation of the forms, the variables in the `lambda-list` and the `store-variables` are bound to names of temporary variables, generated as if by `gensym` or `gentemp`, that will be bound by the expansion of `setf` to the values of those subforms. This binding permits the forms to be written without regard for order-of-evaluation issues. `defsetf` arranges for the temporary variables to be optimized out of the final result in cases where that is possible.

The body code in `defsetf` is implicitly enclosed in a block whose name is `access-fn`.

`defsetf` ensures that subforms of the place are evaluated exactly once.

`Documentation` is attached to `access-fn` as a documentation string of kind `setf`.

If a `defsetf` form appears as a top level form, the compiler must make the `setf` expander available so that it may be used to expand calls to `setf` later on in the file. Users must ensure that the forms, if any, can be evaluated at compile time if the `access-fn` is used in a place later in the same file. The compiler must make these `setf` expanders available to compile-time calls to `get-setf-expansion` when its `environment` argument is a value received as the `environment parameter` of a macro.

Examples:

The effect of

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

is built into the Common Lisp system. This causes the form `(setf (symbol-value foo) fu)` to expand into `(set foo fu)`.

Note that

```lisp
(defsetf car rplaca)
```
would be incorrect because `rplaca` does not return its last argument.

(defun middleguy (x) (nth (truncate (1- (list-length x)) 2) x)) → MIDDLEGUY

(defun set-middleguy (x v)
  (unless (null x)
    (rplaca (nthcdr (truncate (1- (list-length x)) 2) x) v))
  v) → SET-MIDDLEGUY

(defun middleguy (sequence start &optional end) (new-sequence)
  '(progn (replace ,sequence ,new-sequence
    :start1 ,start :end1 ,end)
    ,new-sequence)) → SUBSEQ

(defun xy (&key ((x x) 0) ((y y) 0)) (aref *xy* x y)) → XY

(defun set-xy (new-value &key ((x x) 0) ((y y) 0))
  (setf (aref *xy* x y) new-value)) → SET-XY

(defun subseq (sequence start &optional end) (new-sequence)
  '(progn (replace ,sequence ,new-sequence
    :start1 ,start :end1 ,end)
    ,new-sequence)) → SUBSEQ

An example of the use of the long form of `defsetf`:

(defun subseq (sequence start &optional end) (new-sequence)
  '(progn (replace ,sequence ,new-sequence
    :start1 ,start :end1 ,end)
    ,new-sequence)) → SUBSEQ

(defun subseq (sequence start &optional end) (new-sequence)
  '(progn (replace ,sequence ,new-sequence
    :start1 ,start :end1 ,end)
    ,new-sequence)) → SUBSEQ

(defun middleguy (sequence start &optional end) (new-sequence)
  '(progn (replace ,sequence ,new-sequence
    :start1 ,start :end1 ,end)
    ,new-sequence)) → SUBSEQ

(defun middleguy (sequence start &optional end) (new-sequence)
  '(progn (replace ,sequence ,new-sequence
    :start1 ,start :end1 ,end)
    ,new-sequence)) → SUBSEQ
(xy 'x 1 'y 2) → 3

See Also:
documentation, setf, define-setf-expander, get-setf-expansion, Section 5.1 (Generalized Reference), Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

Notes:
forms must include provision for returning the correct value (the value or values of store-variable). This is handled by forms rather than by defsetf because in many cases this value can be returned at no extra cost, by calling a function that simultaneously stores into the place and returns the correct value.

A setf of a call on access-fn also evaluates all of access-fn’s arguments; it cannot treat any of them specially. This means that defsetf cannot be used to describe how to store into a generalized reference to a byte, such as (ldb field reference). define-setf-expander is used to handle situations that do not fit the restrictions imposed by defsetf and gives the user additional control.

define-setf-expander

Macro

Syntax:

```lisp
define-setf-expander access-fn lambda-list
\[ [\{declaration\}^* | documentation] \{form\}^* \]
\→ access-fn```

Arguments and Values:

- **access-fn**—a symbol that names a function or macro.
- **lambda-list**—macro lambda list.
- **declaration**—a declare expression; not evaluated.
- **documentation**—a string; not evaluated.
- **forms**—an implicit progn.

Description:

define-setf-expander specifies the means by which setf updates a place that is referenced by access-fn.

When setf is given a place that is specified in terms of access-fn and a new value for the place, it is expanded into a form that performs the appropriate update.

The lambda-list supports destructuring. See Section 3.4.4 (Macro Lambda Lists).
define-setf-expander

Documentation is attached to access-fn as a documentation string of kind setf.

Forms constitute the body of the setf expander definition and must compute the setf expansion for a call on setf that references the place by means of the given access-fn. The setf expander function is defined in the same lexical environment in which the define-setf-expander form appears. While forms are being executed, the variables in lambda-list are bound to parts of the place form. The body forms (but not the lambda-list) in a define-setf-expander form are implicitly enclosed in a block whose name is access-fn.

The evaluation of forms must result in the five values described in Section 5.1.1.2 (Setf Expansions).

If a define-setf-expander form appears as a top level form, the compiler must make the setf expander available so that it may be used to expand calls to setf later on in the file. Programmers must ensure that the forms can be evaluated at compile time if the access-fn is used in a place later in the same file. The compiler must make these setf expanders available to compile-time calls to get-setf-expansion when its environment argument is a value received as the environment parameter of a macro.

Examples:

```
(defun lastguy (x) (car (last x))) → LASTGUY
(define-setf-expander lastguy (x &environment env)
   "Set the last element in a list to the given value."
   (multiple-value-bind (dummies vals newval setter getter)
       (get-setf-expansion x env)
       (let ((store (gensym)))
         (values dummies vals '(',store) 'progn (rplaca (last ,getter) ,store) ,store) 'lastguy ,getter))))
→ LASTGUY
(setq a (list 'a 'b 'c 'd)
     b (list 'x)
     c (list 1 2 3 (list 4 5 6))) → (1 2 3 (4 5 6))
(setf (lastguy a) 3) → 3
(setf (lastguy b) 7) → 7
(setf (lastguy (lastguy c)) 'lastguy-symbol) → LASTGUY-SYMBOL
a → (A B C 3)
b → (7)
c → (1 2 3 (4 5 LASTGUY-SYMBOL))
```

;;;; Setf expander for the form (LDB bytespec int).
;;;; Recall that the int form must itself be suitable for SETF.
(define-setf-expander ldb (bytespec int &environment env)
   (multiple-value-bind (temps vals stores
```
store-form access-form)
  (get-setf-expansion int env); Get setf expansion for int.
  (let ((btemp (gensym)) ; Temp var for byte specifier.
    (store (gensym)) ; Temp var for byte to store.
    (stemp (first stores))) ; Temp var for int to store.
  (if (cdr stores) (error "Can't expand this.
))))

;;; Return the setf expansion for LDB as five values.
(values (cons btemp temps) ; Temporary variables.
  (cons bytespec vals) ; Value forms.
  (list store) ; Store variables.
  '(let ((stemp (dpb ,store ,btemp ,access-form)))
    ,store-form
    ,store) ; Storing form.
  '(ldb ,btemp ,access-form) ; Accessing form.
))))

See Also:
setf, defsetf, documentation, get-setf-expansion, Section 3.4.11 (Syntactic Interaction of
Documentation Strings and Declarations)

Notes:
define-setf-expander differs from the long form of defsetf in that while the body is being exe-
cuted the variables in lambda-list are bound to parts of the place form, not to temporary vari-
ables that will be bound to the values of such parts. In addition, define-setf-expander does not
have defsetf’s restriction that access-fn must be a function or a function-like macro; an arbitrary
defmacro destructuring pattern is permitted in lambda-list.

get-setf-expansion

Function

Syntax:

get-setf-expansion place &optional environment
  → vars, vals, store-vars, writer-form, reader-form

Arguments and Values:

place— a place.

environment— an environment object.

vars, vals, store-vars, writer-form, reader-form— a setf expansion.
Description:
Determines five values constituting the setf expansion for place in environment; see Section 5.1.1.2 (Setf Expansions).

If environment is not supplied or nil, the environment is the null lexical environment.

Examples:

\[
\text{(get-setf-expansion 'x)}
\rightarrow \text{NIL, NIL, (#:G0001), (SETQ X #:G0001), X}
\]

;;; This macro is like POP

\[
\text{(defmacro xpop (place &environment env)}
\rightarrow \text{(multiple-value-bind (dummies vals new setter getter)}
\rightarrow \text{get-setf-expansion place env)}
\rightarrow \text{'(let* ,,@(mapcar #'list dummies vals) ,(car new) ,getter)}
\rightarrow \text{if (cdr new) (error "Can't expand this.")}
\rightarrow \text{prog1 (car ,(car new))}
\rightarrow \text{setq ,(car new) (cdr ,(car new))}
\rightarrow ,setter))))
\]

\[
\text{(defsetf frob (x) (value)}
\rightarrow \text{'(setf (car ,x) ,value))} \rightarrow \text{FROB}
\]

;;; The following is an error; an error might be signaled at macro expansion time

\[
\text{(flet ((frob (x) (cdr x))) ;Invalid}
\rightarrow \text{xpop (frob z)))}
\]

See Also:
defsetf, define-setf-expander, setf

Notes:
Any compound form is a valid place, since any compound form whose operator f has no setf expander are expanded into a call to (setf f).

\[
\text{setf, psetf}
\]

Macro

Syntax:
\[
\text{setf \{pair\}* } \rightarrow \text{\{result\}*}
\]

Data and Control Flow 5–99
**setf, psetf**

\[
psetf \{\text{pair}\}^* \rightarrow \text{nil}
\]

\[
pair ::= \text{place newvalue}
\]

**Arguments and Values:**
- **place**—a place.
- **newvalue**—a form.
- **results**—the multiple values returned by the storing form for the last place, or nil if there are no pairs.

**Description:**
`setf` changes the value of place to be newvalue.

`(setf place newvalue)` expands into an update form that stores the result of evaluating `newvalue` into the location referred to by `place`. Some `place` forms involve uses of accessors that take optional arguments. Whether those optional arguments are permitted by `setf`, or what their use is, is up to the `setf` expander function and is not under the control of `setf`. The documentation for any function that accepts `&optional`, `&rest`, or `&key` arguments and that claims to be usable with `setf` must specify how those arguments are treated.

If more than one `pair` is supplied, the `pairs` are processed sequentially; that is,

\[
\begin{align*}
\text{setf place-1} & \text{ newvalue-1} \\
\text{place-2} & \text{ newvalue-2} \\
& \text{...} \\
\text{place-N} & \text{ newvalue-N}
\end{align*}
\]

is precisely equivalent to

\[
\text{progn} \ (\text{setf place-1} \ \text{newvalue-1}) \\
\text{setf place-2} \ \text{newvalue-2} \\
& \text{...} \\
\text{setf place-N} \ \text{newvalue-N})
\]

For `psetf`, if more than one `pair` is supplied then the assignments of new values to places are done in parallel. More precisely, all subforms (in both the `place` and `newvalue` forms) that are to be evaluated are evaluated from left to right; after all evaluations have been performed, all of the assignments are performed in an unpredictable order.

For detailed treatment of the expansion of `setf` and `psetf`, see Section 5.1.2 (Kinds of Places).

**Examples:**

\[
\begin{align*}
\text{(setq x (cons 'a 'b) y (list 1 2 3))} & \rightarrow (1 2 3) \\
\text{(setf (car x) 'x (cadr y) (car x) (cdr x) y)} & \rightarrow (1 X 3) \\
\text{x} & \rightarrow (X 1 X 3)
\end{align*}
\]

5–100  Programming Language—Common Lisp
Version 15.17R, X3J13/94-101R.
Fri 12-Aug-1994 6:35pm EDT

\[
y \rightarrow (1 \ X \ 3) \\
(\text{setq} \ x \ (\text{cons} \ 'a \ 'b) \ y \ (\text{list} \ 1 \ 2 \ 3)) \rightarrow (1 \ 2 \ 3) \\
(p\text{setf} \ (\text{car} \ x) \ 'x \ (\text{cadr} \ y) \ (\text{car} \ x) \ (\text{cdr} \ x) \ y) \rightarrow \text{NIL} \\
x \rightarrow (X \ 1 \ A \ 3) \\
y \rightarrow (1 \ A \ 3)
\]

Affected By:
define-setf-expander, defsetf, *macroexpand-hook*

See Also:
define-setf-expander, defsetf, macroexpand-1, rotatef, shiftf, Section 5.1 (Generalized Reference)

---

\textbf{shiftf} \hspace{1cm} \textit{Macro}

\textbf{Syntax:}
\[
\text{shiftf} \ \{\text{place}\}^+ \ \text{newvalue} \rightarrow \text{old-value-1}
\]

\textbf{Arguments and Values:}

\textit{place}—a \textit{place}.

\textit{newvalue}—a \textit{form}; evaluated.

\textit{old-value-1}—an \textit{object} (the old value of the first \textit{place}).

\textbf{Description:}

\textit{shiftf} modifies the values of each \textit{place} by storing \textit{newvalue} into the last \textit{place}, and shifting the values of the second through the last \textit{place} into the remaining \textit{places}.

If \textit{newvalue} produces more values than there are store variables, the extra values are ignored. If \textit{newvalue} produces fewer values than there are store variables, the missing values are set to \texttt{nil}.

In the form \texttt{(shiftf place1 place2 \ldots place\textit{n} \ newvalue)}, the values in \textit{place1} through \textit{place\textit{n}} are \textit{read} and saved, and \textit{newvalue} is evaluated, for a total of \textit{n}+1 values in all. Values 2 through \textit{n}+1 are then stored into \textit{place1} through \textit{place\textit{n}}, respectively. It is as if all the \textit{places} form a shift register; the \textit{newvalue} is shifted in from the right, all values shift over to the left one place, and the value shifted out of \textit{place1} is returned.

For information about the \textit{evaluation} of \textit{subforms} of \textit{places}, see Section 5.1.1.1 (Evaluation of Subforms to Places).

\textbf{Examples:}

\[
(\text{setq} \ x \ (\text{list} \ 1 \ 2 \ 3) \ y \ '\text{trash}) \rightarrow \text{TRASH}
\]
(shiftf y x (cdr x) '(hi there)) → TRASH
x → (2 3)
y → (1 HI THERE)

(setq x (list 'a 'b 'c)) → (A B C)
(shiftf (cadr x) 'z) → B
x → (A Z C)
(shiftf (cadr x) (cddr x) 'q) → Z
x → (A (C) Q)
(setq n 0) → 0
(setq x (list 'a 'b 'c 'd)) → (A B C D)
(shiftf (nth (setq n (+ n 1)) x) 'z) → B
x → (A Z C D)

Affected By:
define-setf-expander, defsetf, *macroexpand-hook*

See Also:
setf, rotatef, Section 5.1 (Generalized Reference)

Notes:
The effect of (shiftf place1 place2 ... placen newvalue) is roughly equivalent to

(let ((var1 place1)
       (var2 place2)
       ...
       (varn placen)
       (var0 newvalue))
  (setq placen var2)
  (setf place2 var3)
  ...
  (setf placen var0)
  var1)

except that the latter would evaluate any subforms of each place twice, whereas shiftf evaluates
them once. For example,

(setq n 0) → 0
(setq x (list 'a 'b 'c 'd)) → (A B C D)
(progl1 (nth (setq n (+ n 1)) x)
     (setq (nth (setq n (+ n 1)) x) 'z)) → B
x → (A B Z D)
**rotatef**

**Macro**

**Syntax:**

\[
\text{rotatef}\{\text{place}\}* \rightarrow \text{nil}
\]

**Arguments and Values:**

\(\text{place}\) — a place.

**Description:**

\textbf{rotatef} modifies the values of each \textit{place} by rotating values from one \textit{place} into another.

If a \textit{place} produces more values than there are store variables, the extra values are ignored. If a \textit{place} produces fewer values than there are store variables, the missing values are set to \textit{nil}.

In the form \((\text{rotatef} \ \textit{place1} \ \textit{place2} \ldots \ \textit{placen})\), the values in \textit{place1} through \textit{placen} are read and written. Values 2 through \(n\) and value 1 are then stored into \textit{place1} through \textit{placen}. It is as if all the places form an end-around shift register that is rotated one place to the left, with the value of \textit{place1} being shifted around the end to \textit{placen}.

For information about the evaluation of subforms of \textit{places}, see Section 5.1.1.1 (Evaluation of Subforms to Places).

**Examples:**

\[
(\text{let} \ ((n \ 0) \\
\quad \quad \ (x \ \text{(list} \ 'a \ 'b \ 'c \ 'd \ 'e \ 'f \ 'g))) \\
\quad \quad \ (\text{rotatef} \ \text{(nth} \ (\text{incf} \ n) \ x) \\
\quad \quad \quad \quad \text{(nth} \ (\text{incf} \ n) \ x) \\
\quad \quad \quad \quad \text{(nth} \ (\text{incf} \ n) \ x) \\
\quad \quad \quad \quad x) \ \rightarrow \ (A \ C \ D \ B \ E \ F \ G)
\]

**See Also:**

\texttt{define-setf-expander}, \texttt{defsetf}, \texttt{setf}, \texttt{shiftf}, \texttt{*macroexpand-hook*}, Section 5.1 (Generalized Reference)

**Notes:**

The effect of \((\text{rotatef} \ \textit{place1} \ \textit{place2} \ldots \ \textit{placen})\) is roughly equivalent to

\[
(\text{psetf} \ \textit{place1} \ \textit{place2} \\
\quad \quad \ \textit{place2} \ \textit{place3} \\
\quad \quad \quad \ldots \\
\quad \quad \ \textit{placen} \ \textit{place1})
\]

except that the latter would evaluate any \textit{subforms} of each \textit{place} twice, whereas \textbf{rotatef} evaluates them once.

---

Data and Control Flow 5–103
control-error

Class Precedence List:
control-error, error, serious-condition, condition, t

Description:
The type control-error consists of error conditions that result from invalid dynamic transfers of control in a program. The errors that result from giving `throw` a tag that is not active or from giving `go` or `return-from` a tag that is no longer dynamically available are of type control-error.

program-error

Class Precedence List:
program-error, error, serious-condition, condition, t

Description:
The type program-error consists of error conditions related to incorrect program syntax. The errors that result from naming a `go` tag or a `block` tag that is not lexically apparent are of type program-error.

undefined-function

Class Precedence List:
undefined-function, cell-error, error, serious-condition, condition, t

Description:
The type undefined-function consists of error conditions that represent attempts to read the definition of an undefined function.

The name of the cell (see cell-error) is the function name which was `funbound`.

See Also:
cell-error-name
Programming Language—Common Lisp

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 transferring 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; 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 stepping₁.

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 stepping₁. Otherwise, the initialization and stepping₁ 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 nconcinc) 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 specifier`.

6–6  Programming Language—Common Lisp
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)
```

Iteration 6–7
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 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 occur sequentially by default. The and construct can be used to connect two or more iteration
clauses when *sequential binding* and *stepping* 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 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 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 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 | downto | 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* begins, as supplied by
*form1*. Stepping is incremental by default. If decremental *stepping* is desired, the
preposition *downto* or *above* must be used with *form2*. For incremental *stepping*, 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 stepping1 supplied in form2. Stepping1 is incremental by default. If decremental stepping1 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 stepping1, 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 stepping1, 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

6–10 Programming Language—Common Lisp
\[\begin{align*}
&\triangleright 2 \\
&\triangleright 3 \\
&\rightarrow \text{NIL} \\
\end{align*}\]

;; Print every third number.
(\text{loop for } i \text{ from } 10 \text{ downto } 1 \text{ by } 3 \\
 \text{do (print } i))
\begin{align*}
&\triangleright 10 \\
&\triangleright 7 \\
&\triangleright 4 \\
&\triangleright 1 \\
&\rightarrow \text{NIL}
\end{align*}

;; Step incrementally from the default starting value.
(\text{loop for } i \text{ below } 3 \\
 \text{do (print } i))
\begin{align*}
&\triangleright 0 \\
&\triangleright 1 \\
&\triangleright 2 \\
&\rightarrow \text{NIL}
\end{align*}

6.1.2.1.2 The for-as-in-list subclause

In the \textit{for-as-in-list} subclause, the \textit{for} or \textit{as} construct iterates over the contents of a \textit{list}. It checks for the end of the \textit{list} as if by using \texttt{endp}. The variable \textit{var} is bound to the successive elements of the \textit{list} in \textit{form1} before each iteration. At the end of each iteration, the function \textit{step-fun} is applied to the \textit{list}; the default value for \textit{step-fun} is \texttt{cdr}. The \textit{loop keywords} \texttt{in} and \texttt{by} serve as valid prepositions in this syntax. The \textit{for} or \textit{as} construct causes termination when the end of the \textit{list} is reached.

6.1.2.1.2.1 Examples of for-as-in-list subclause

;; Print every item in a list.
(\text{loop for } item \text{ in } '(1 2 3) \text{ do (print } item))
\begin{align*}
&\triangleright 1 \\
&\triangleright 2 \\
&\triangleright 3 \\
&\rightarrow \text{NIL}
\end{align*}

;; Print every other item in a list.
(\text{loop for } item \text{ in } '(1 2 3 4 5) \text{ by #'cddr} \\
 \text{do (print } item))
\begin{align*}
&\triangleright 1 \\
&\triangleright 3 \\
&\triangleright 5 \\
&\rightarrow \text{NIL}
\end{align*}

Iteration 6–11
;; 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.
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.
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 desirable 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:

\[
\text{(loop with } \text{a } = 1 \text{ with } \text{b } = (+ \text{ a } 2) \text{ with } \text{c } = (+ \text{ b } 3) \text{ return } (\text{list a b c}) )
\]

\[
\rightarrow (1 \hspace{1pt} 3 \hspace{1pt} 6)
\]

The execution of the above loop is equivalent to the execution of the following code:

\[
\text{(block nil (let* ((a 1) (b (+ a 2)) (c (+ b 3))) (tagbody (next-loop (return (list a b c)) (go next-loop) end-loop))) )}
\]

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:

\[
\text{(loop with } \text{a } = 1 \text{ and } \text{b } = 2 \text{ and } \text{c } = 3 \text{ return } (\text{list a b c}) )
\]

\[
\rightarrow (1 \hspace{1pt} 2 \hspace{1pt} 3)
\]

The execution of the above loop is equivalent to the execution of the following code:

\[
\text{(block nil (let ((a 1) (b 2) (c 3)) (tagbody (next-loop (return (list a b c)) (go next-loop) end-loop))) )}
\]

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

```lisp
(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`.  

---

*Iteration 6–21*
• 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

```lisp
(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 (print '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.

6–24  Programming Language—Common Lisp
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: " 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: " 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))

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 232 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

Iteration 6–29
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

\( (\text{loop for } \text{internal-variable} \text{ downfrom } (- n 1) \text{ to } 0 \ldots) \)

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.
do, do*  

Macro

Syntax:

\[
\text{do} \ ((\text{var } | \ (\text{var} \ \text{init-form} \ [\text{step-form}])))^* \\
(\text{end-test-form} \ {\text{result-form}}^*) \\
{\text{declaration}}^* \ {\text{tag} \ | \ \text{statement}}^* \\
\rightarrow \ \{\text{result}\}^*
\]

\[
\text{do}^* \ ((\text{var } | \ (\text{var} \ \text{init-form} \ [\text{step-form}])))^* \\
(\text{end-test-form} \ {\text{result-form}}^*) \\
{\text{declaration}}^* \ {\text{tag} \ | \ \text{statement}}^* \\
\rightarrow \ \{\text{result}\}^*
\]

Arguments and Values:

\text{var}—a symbol.

\text{init-form}—a form.

\text{step-form}—a form.

\text{end-test-form}—a form.

\text{result-forms}—an implicit progn.

\text{declaration}—a declare expression; not evaluated.

\text{tag}—a go tag; not evaluated.

\text{statement}—a compound form; evaluated as described below.

\text{results}—if a return or return-from form is executed, the values passed from that form; otherwise, the values returned by the result-forms.

Description:

do iterates over a group of statements while a test condition holds. do accepts an arbitrary number of iteration vars which are bound within the iteration and stepped in parallel. An initial value may be supplied for each iteration variable by use of an init-form. Step-forms may be used to specify how the vars should be updated on succeeding iterations through the loop. Step-forms may be used both to generate successive values or to accumulate results. If the end-test-form condition is met prior to an execution of the body, the iteration terminates. Tags label statements.

do* is exactly like do except that the bindings and steppings of the vars are performed sequentially rather than in parallel.
do, do*

Before the first iteration, all the init-forms are evaluated, and each var is bound to the value of its respective init-form, if supplied. This is a binding, not an assignment; when the loop terminates, the old values of those variables will be restored. For do, all of the init-forms are evaluated before any var is bound. The init-forms can refer to the bindings of the vars visible before beginning execution of do. For do*, the first init-form is evaluated, then the first var is bound to that value, then the second init-form is evaluated, then the second var is bound, and so on; in general, the kth init-form can refer to the new binding of the jth var if j < k, and otherwise to the old binding of the jth var.

At the beginning of each iteration, after processing the variables, the end-test-form is evaluated. If the result is false, execution proceeds with the body of the do (or do*) form. If the result is true, the result-forms are evaluated in order as an implicit prog, and then do or do* returns.

At the beginning of each iteration other than the first, vars are updated as follows. All the step-forms, if supplied, are evaluated, from left to right, and the resulting values are assigned to the respective vars. Any var that has no associated step-form is not assigned to. For do, all the step-forms are evaluated before any var is updated; the assignment of values to vars is done in parallel, as if by psetq. Because all of the step-forms are evaluated before any of the vars are altered, a step-form when evaluated always has access to the old values of all the vars, even if other step-forms precede it. For do*, the first step-form is evaluated, then the value is assigned to the first var, then the second step-form is evaluated, then the value is assigned to the second var, and so on; the assignment of values to variables is done sequentially, as if by setq. For either do or do*, after the vars have been updated, the end-test-form is evaluated as described above, and the iteration continues.

The remainder of the do (or do*) form constitutes an implicit tagbody. Tags may appear within the body of a do loop for use by go statements appearing in the body (but such go statements may not appear in the variable specifiers, the end-test-form, or the result-forms). When the end of a do body is reached, the next iteration cycle (beginning with the evaluation of step-forms) occurs.

An implicit block named nil surrounds the entire do (or do*) form. A return statement may be used at any point to exit the loop immediately.

Init-form is an initial value for the var with which it is associated. If init-form is omitted, the initial value of var is nil. If a declaration is supplied for a var, init-form must be consistent with the declaration.

Declarations can appear at the beginning of a do (or do*) body. They apply to code in the do (or do*) body, to the bindings of the do (or do*) vars, to the step-forms, to the end-test-form, and to the result-forms.

Examples:

```lisp
(do (((temp-one 1 (1+ temp-one))
      (temp-two 0 (1- temp-two)))
     ((> (- temp-one temp-two) 5) temp-one)) → 4
```

6–32 Programming Language—Common Lisp
do, do*

\[\begin{align*}
\text{(do \ ((temp-one 1 (1+ temp-one))} \\
\text{\quad (temp-two 0 (1+ temp-one)))} \\
\text{\quad ((= 3 temp-two) temp-one)) \rightarrow 3}
\end{align*}\]

(\[\begin{align*}
\text{(do* \ ((temp-one 1 (1+ temp-one))} \\
\text{\quad (temp-two 0 (1+ temp-one)))} \\
\text{\quad ((= 3 temp-two) temp-one)) \rightarrow 2}
\end{align*}\]

\[\begin{align*}
\text{(do \ ((j 0 (+ j 1)))} \\
\text{\quad (nil)) ;Do forever.} \\
\text{\quad (format t \ "\%Input \ "D: \ j")} \\
\text{\quad (let ((item (read)))} \\
\text{\quad\quad (if (null item) (return) ;Process items until NIL seen.} \\
\text{\quad\quad \quad (format t \ "\&Output \ "D: \ "S" \ j item))}}
\end{align*}\]

\[\begin{align*}
\text{\texttt{Input 0: banana}} \\
\text{\texttt{Output 0: BANANA}} \\
\text{\texttt{Input 1: (57 boxes)}} \\
\text{\texttt{Output 1: (57 BOXES)}} \\
\text{\texttt{Input 2: NIL}} \\
\text{\texttt{→ NIL}}
\end{align*}\]

(\[\begin{align*}
\text{(setq a-vector (vector 1 nil 3 nil))} \\
\text{(do \ ((i 0 (+ i 1)) ;Sets every null element of a-vector to zero.} \\
\text{\quad (n (array-dimension a-vector 0)))} \\
\text{\quad ((= i n))} \\
\text{\quad (when (null (aref a-vector i))} \\
\text{\quad\quad (setf (aref a-vector i)))} \\
\text{\quad a-vector \rightarrow \#(1 0 3 0)}
\end{align*}\]

\[\begin{align*}
\text{(do \ ((x e (cdr x))} \\
\text{\quad (oldx x x))} \\
\text{\quad ((null x))} \\
\text{\quad body)}
\end{align*}\]

is an example of parallel assignment to index variables. On the first iteration, the value of oldx is whatever value x had before the do was entered. On succeeding iterations, oldx contains the value that x had on the previous iteration.

\[\begin{align*}
\text{(do \ ((x foo (cdr x))} \\
\text{\quad (y bar (cdr y))} \\
\text{\quad (z '() (cons (f (car x) (car y)) z)))} \\
\text{\quad ((or (null x) (null y))} \\
\text{\quad (nreverse z))}
\end{align*}\]

\[\begin{align*}
\text{\texttt{Itereteion \ 6–33}}
\end{align*}\]

does the same thing as (mapcar #'f foo bar). The step computation for z is an example of the
do, do*

fact that variables are stepped in parallel. Also, the body of the loop is empty.

(defun list-reverse (list)
  (do ((x list (cdr x))
       (y '() (cons (car x) y))
       ((endp x) y)))

As an example of nested iterations, consider a data structure that is a list of conses. The car of each cons is a list of symbols, and the cdr of each cons is a list of equal length containing corresponding values. Such a data structure is similar to an association list, but is divided into “frames”; the overall structure resembles a rib-cage. A lookup function on such a data structure might be:

(defun ribcage-lookup (sym ribcage)
  (do ((r ribcage (cdr r)))
      ((null r) nil)
    (do ((s (caar r) (cdr s))
         (v (cdar r) (cdr v)))
        ((null s))
      (when (eq (car s) sym)
        (return-from ribcage-lookup (car v))))))

See Also:

other iteration functions (dolist, dotimes, and loop) and more primitive functionality (tagbody, go, block, return, let, andsetq)

Notes:

If end-test-form is nil, the test will never succeed. This provides an idiom for “do forever”: the body of the do or do* is executed repeatedly. The infinite loop can be terminated by the use of return, return-from, go to an outer level, or throw.

A do form may be explained in terms of the more primitive forms block, return, let, loop, tagbody, and psetq as follows:

(block nil
  (let (((var1 init1)
          (var2 init2)
          ...)
        (varn initn))
    declarations
      (loop (when end-test (return (progn . result))))
        (tagbody . tagbody)
        (psetq var1 step1
               var2 step2
               ...
               varn stepn))))

6–34  Programming Language—Common Lisp
do* is similar, except that let* and setq replace the let and psetq, respectively.

**dotimes**

*Macro*

**Syntax:**

\[
\text{dotimes} \ (\text{var \ count-form \ [result-form]} \ \{\text{declaration}\}^{*} \ \{\text{tag} \mid \text{statement}\}^{*} \\
\rightarrow \ \{\text{result}\}^{*}
\]

**Arguments and Values:**

- **var**—a symbol.
- **count-form**—a form.
- **result-form**—a form.
- **declaration**—a declare expression; not evaluated.
- **tag**—a go tag; not evaluated.
- **statement**—a compound form; evaluated as described below.

**results**—if a return or return-from form is executed, the values passed from that form; otherwise, the values returned by the result-form or nil if there is no result-form.

**Description:**

**dotimes** iterates over a series of integers.

**dotimes** evaluates **count-form**, which should produce an integer. If **count-form** is zero or negative, the body is not executed. **dotimes** then executes the body once for each integer from 0 up to but not including the value of **count-form**, in the order in which the tags and statements occur, with **var** bound to each integer. Then **result-form** is evaluated. At the time **result-form** is processed, **var** is bound to the number of times the body was executed. **Tags** label **statements**.

An implicit block named nil surrounds **dotimes**. return may be used to terminate the loop immediately without performing any further iterations, returning zero or more values.

The body of the loop is an implicit tagbody; it may contain tags to serve as the targets of go statements. Declarations may appear before the body of the loop.

The scope of the binding of **var** does not include the **count-form**, but the **result-form** is included.

It is implementation-dependent whether **dotimes** establishes a new binding of **var** on each iteration or whether it establishes a binding for **var** once at the beginning and then assigns it on any subsequent iterations.
Examples:

(dotimes (temp-one 10 temp-one)) → 10
(setq temp-two 0) → 0
(dotimes (temp-one 10 t) (incf temp-two)) → T
temp-two → 10

Here is an example of the use of `dotimes` in processing strings:

;;; True if the specified subsequence of the string is a
;;; palindrome (reads the same forwards and backwards).
(defun palindromep (string &optional
  (start 0)
  (end (length string)))
  (dotimes (k (floor (- end start) 2) t)
    (unless (char-equal (char string (+ start k))
                         (char string (- end k 1)))
      (return nil))))
(palindromep "Able was I ere I saw Elba") → T
(palindromep "A man, a plan, a canal--Panama!") → NIL
(remove-if-not #'alpha-char-p ;Remove punctuation.
  "A man, a plan, a canal--Panama!") → "AmanaplanacanalPanama"
(palindromep
  (remove-if-not #'alpha-char-p
    "A man, a plan, a canal--Panama!") → T
(palindromep
  (remove-if-not #
    "Unremarkable was I ere I saw Elba Kramer, nu?") → T
(palindromep
  (remove-if-not #
    "A man, a plan, a cat, a ham, a yak,
        a yam, a hat, a canal--Panama!") → T

See Also:
do, dolist, tagbody

Notes:
go may be used within the body of `dotimes` to transfer control to a statement labeled by a `tag`. 
dolist

Macro

dolist

Syntax:

\[
dolist (\text{var list-form [result-form]} \{\text{declaration}\}^{*} \{\text{tag | statement}\}^{*}) \rightarrow \{\text{result}\}^{*}
\]

Arguments and Values:

\text{var}—a symbol.

\text{list-form}—a form.

\text{result-form}—a form.

\text{declaration}—a declare expression; not evaluated.

\text{tag}—a go tag; not evaluated.

\text{statement}—a compound form; evaluated as described below.

\text{results}—if a return or return-from form is executed, the values passed from that form; otherwise, the values returned by the result-form or nil if there is no result-form.

Description:

dolist iterates over the elements of a list. The body of dolist is like a tagbody. It consists of a series of tags and statements.

dolist evaluates list-form, which should produce a list. It then executes the body once for each element in the list, in the order in which the tags and statements occur, with var bound to the element. Then result-form is evaluated. tags label statements.

At the time result-form is processed, var is bound to nil.

An implicit block named nil surrounds dolist. return may be used to terminate the loop immediately without performing any further iterations, returning zero or more values.

The scope of the binding of var does not include the list-form, but the result-form is included.

It is implementation-dependent whether dolist establishes a new binding of var on each iteration or whether it establishes a binding for var once at the beginning and then assigns it on any subsequent iterations.

Examples:

\[
\begin{align*}
\text{(setq temp-two '()) } & \rightarrow \text{NIL} \\
\text{(dolist (temp-one '(1 2 3 4) temp-two) (push temp-one temp-two)) } & \rightarrow (4 3 2 1) \\
\text{(setq temp-two 0) } & \rightarrow 0
\end{align*}
\]
(dolist (temp-one '(1 2 3 4)) (incf temp-two)) → NIL
temp-two → 4
(dolist (x '(a b c d)) (prin1 x) (princ " "))
▷ A B C D
→ NIL

See Also:
do, dotimes, tagbody, Section 3.6 (Traversal Rules and Side Effects)

Notes:
go may be used within the body of dolist to transfer control to a statement labeled by a tag.

---

**loop**

*Macro*

Syntax:
The “simple” **loop** form:

```
loop {compound-form}* → {result}*
```

The “extended” **loop** form:

```
loop [[name-clause] [[variable-clause]* [[main-clause]* → {result}]*
```

name-clause::=named name
variable-clause::=[[with-clause] | [initial-final] | [for-as-clause]
with-clause::=with var1 [type-spec] [= form1] {and var2 [type-spec] [= form2]}*
main-clause::=[[unconditional] | [accumulation] | [conditional] | [termination-test] | [initial-final]
initial-final::=initially {compound-form}+ | finally {compound-form}+
unconditional::=[[do | doing] {compound-form}+ | return {form | it}
accumulation::=[[list-accumulation] | [numeric-accumulation]
list-accumulation::=[collect | collecting | append | appending | nconc | nconcing] {form | it}
<into simple-var>
numeric-accumulation::=[count | counting | sum | summing |
maximize | maximizing | minimize | minimizing] {form | it}
<into simple-var] [type-spec]
loop

conditional ::= { if | when | unless } form | selectable-clause { and | selectable-clause }*
  [ else | selectable-clause { and | selectable-clause }* ]
  [ end ]

selectable-clause ::= unconditional | accumulation | conditional

termination-test ::= while form | until form | repeat form | always form | never form | thereis form

for-as-clause ::= { for | as } | for-as-subclause { and | for-as-subclause }*

for-as-subclause ::= for-as-arithmetic | for-as-in-list | for-as-on-list | for-as-equals-then |
  [ for-as-across | for-as-hash | for-as-package ]

for-as-arithmetic ::= var [ type-spec ] | for-as-arithmetic-subclause

for-as-arithmetic-subclause ::= arithmetic-up | arithmetic-downto | arithmetic-downfrom

arithmetic-up ::= [ { from | upfrom } form1 | { to | upto | below } form2 | by form3 ]^+

arithmetic-downto ::= [ { from form1 }^1 | { { downto | above } form2 }^1 | by form3 ]

arithmetic-downfrom ::= [ { downfrom form1 }^1 | { to | downto | above } form2 | by form3 ]

for-as-in-list ::= var [ type-spec ] in form1 [ by step-fun ]

for-as-on-list ::= var [ type-spec ] on form1 [ by step-fun ]

for-as-equals-then ::= var [ type-spec ] = form1 [ then form2 ]

for-as-across ::= var [ type-spec ] across vector

for-as-hash ::= var [ type-spec ] being { each | the }
  [{ hash-key | hash-keys } { in | of } hash-table
  [ using ( hash-value other-var ) ] |
  [ hash-value | hash-values ] { in | of } hash-table
  [ using ( hash-key other-var ) ]]

for-as-package ::= var [ type-spec ] being { each | the }
  [ symbol | symbols | present-symbol | present-symbols | external-symbol | external-symbols ]
  [ { in | of } package ]
loop

\[
\text{type-spec::=} | \text{simple-type-spec} | \text{destructured-type-spec} \\
\text{simple-type-spec::=} \text{fixnum} | \text{float} | \text{t} | \text{nil} \\
\text{destructured-type-spec::=} \text{of-type} \text{ d-type-spec} \\
\text{d-type-spec::=} \text{type-specifier} | (\text{d-type-spec} . \text{d-type-spec}) \\
\text{var::=} | \text{d-var-spec} \\
\text{var1::=} | \text{d-var-spec} \\
\text{var2::=} | \text{d-var-spec} \\
\text{other-var::=} | \text{d-var-spec} \\
\text{d-var-spec::=} \text{simple-var} | \text{nil} | (\text{d-var-spec} . \text{d-var-spec})
\]

Arguments and Values:
- \text{compound-form}—a \text{compound form}.
- \text{name}—a \text{symbol}.
- \text{simple-var}—a \text{symbol} (a \text{variable name}).
- \text{form, form1, form2, form3}—a \text{form}.
- \text{step-fun}—a \text{form} that evaluates to a \text{function} of one \text{argument}.
- \text{vector}—a \text{form} that evaluates to a \text{vector}.
- \text{hash-table}—a \text{form} that evaluates to a \text{hash table}.
- \text{package}—a \text{form} that evaluates to a \text{package designator}.
- \text{type-specifier}—a \text{type specifier}. This might be either an \text{atomic type specifier} or a \text{compound type specifier}, which introduces some additional complications to proper parsing in the face of destructuring; for further information, see Section 6.1.1.7 (Destructuring).
- \text{result}—an \text{object}.

Description:
For details, see Section 6.1 (The LOOP Facility).

Examples:

\;
\text{;; An example of the simple form of LOOP.} \\
\text{(defun sqrt-advisor ()} \\
\text{ (loop (format t "~&Number: ")} \\
\text{\quad ; Puts the next number on the standard} \\
\text{\quad ; output stream. The values of var1, var2,} \\
\text{\quad ; and other-var are nil because they are not} \\
\text{\quad ; bound to any values.)} \\
\text{)}
(let ((n (parse-integer (read-line) :junk-allowed t)))
  (when (not n) (return))
  (format t "&The square root of ~D is ~D.~" n (sqrt n))))
→ SQRT-ADVISOR
(sqrt-advisor)
⊿ Number: 5←
⊿ The square root of 5 is 2.236068.
⊿ Number: 4←
⊿ The square root of 4 is 2.
⊿ Number: done←
→ NIL

;; An example of the extended form of LOOP.
(defun square-advisor ()
  (loop as n = (progn (format t "&Number: ")
               (parse-integer (read-line) :junk-allowed t))
        while n
        do (format t "&The square of ~D is ~D.~" n (* n n))))
→ SQUARE-ADVISOR
(square-advisor)
⊿ Number: 4←
⊿ The square of 4 is 16.
⊿ Number: 23←
⊿ The square of 23 is 529.
⊿ Number: done←
→ NIL

;; Another example of the extended form of LOOP.
(loop for n from 1 to 10
  when (oddp n)
  collect n)
→ (1 3 5 7 9)

See Also:
do, dolist, dotimes, return, go, throw, Section 6.1.1.7 (Destructuring)

Notes:
Except that loop-finish cannot be used within a simple loop form, a simple loop form is related to an extended loop form in the following way:

(loop {compound-form}*) ≡ (loop do {compound-form}*)
loop-finish

Syntax:

```
loop-finish (no arguments)  -> |
```

Description:

The `loop-finish` macro can be used lexically within an extended loop form to terminate that form "normally." That is, it transfers control to the loop epilogue of the lexically innermost extended loop form. This permits execution of any finally clause (for effect) and the return of any accumulated result.

Examples:

```
;; Terminate the loop, but return the accumulated count.
(loop for i in '(1 2 3 stop-here 4 5 6)
     when (symbolp i) do (loop-finish)
     count i)
  -> 3

;; The preceding loop is equivalent to:
(loop for i in '(1 2 3 stop-here 4 5 6)
     until (symbolp i)
     count i)
  -> 3

;; While LOOP-FINISH can be used can be used in a variety of
;; situations it is really most needed in a situation where a need
;; to exit is detected at other than the loop's 'top level'
;; (where UNTIL or WHEN often work just as well), or where some
;; computation must occur between the point where a need to exit is
;; detected and the point where the exit actually occurs. For example:
(defun tokenize-sentence (string)
  (macrolet ((add-word (wvar svar)
      '(when ,wvar
        (push (coerce (nreverse ,wvar) 'string) ,svar)
        (setq ,wvar nil))))
    (loop with word = '() and sentence = '() and endpos = nil
          for i below (length string)
          do (let ((char (aref string i)))
            (case char
              (#\Space (add-word word sentence))
              (#\. (setq endpos (1+ i)) (loop-finish))
              (otherwise (push char word))))
    finally (add-word word sentence))
```

6–42  Programming Language—Common Lisp
loop-finish

\[(\text{return } (\text{values } (\text{nreverse } \text{sentence}) \text{ endpos}))\)]

\[\rightarrow \text{TOKENIZE-SENTENCE}\]

\[\text{(tokenize-sentence } \text{"this is a sentence. this is another sentence."})\]

\[\rightarrow (\text{"this" } \text{"is" } \text{"a" } \text{"sentence"}), 19\]

\[\text{(tokenize-sentence } \text{"this is a sentence"})\]

\[\rightarrow (\text{"this" } \text{"is" } \text{"a" } \text{"sentence"}), \text{NIL}\]

**Side Effects:**

Transfers control.

**Exceptional Situations:**

Whether or not `loop-finish` is `fbound` in the global environment is implementation-dependent; however, the restrictions on redefinition and shadowing of `loop-finish` are the same as for symbols in the `COMMON-LISP` package which are `fbound` in the global environment. The consequences of attempting to use `loop-finish` outside of `loop` are undefined.

**See Also:**

`loop`, Section 6.1 (The LOOP Facility)

**Notes:**

Iteration 6–43
Programming Language—Common Lisp

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. Defauled 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 \(C_1\) and \(C_2\) supply the values of defaulted initialization arguments for different slots, and suppose that \(C_1\) is more specific than \(C_2\); then the defaulted initialization argument whose value is supplied by \(C_1\) is to the left of the defaulted initialization argument whose value is supplied by \(C_2\) 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:

\[
\begin{align*}
\text{(defclass q () ((x :initarg a)))} \\
\text{(defclass r (q) ((x :initarg b))} \\
\text{ (:default-initargs a 1 b 2))}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Defaulted Form</th>
<th>Initialization Argument List</th>
<th>Contents of Slot X</th>
</tr>
</thead>
<tbody>
<tr>
<td>(make-instance 'r)</td>
<td>(a 1 b 2)</td>
<td>1</td>
</tr>
<tr>
<td>(make-instance 'r 'a 3)</td>
<td>(a 3 b 2)</td>
<td>3</td>
</tr>
<tr>
<td>(make-instance 'r 'b 4)</td>
<td>(b 4 a 1)</td>
<td>4</td>
</tr>
<tr>
<td>(make-instance 'r 'a 1 'a 2)</td>
<td>(a 1 a 2 b 2)</td>
<td>1</td>
</tr>
</tbody>
</table>
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.

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, :default-initarg 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, \( C_{\text{from}} \), to a different class, \( C_{\text{to}} \); it changes the structure of the instance to conform to the definition of the class \( C_{\text{to}} \).

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 \( C_{\text{to}} \), local slots specified by the class \( C_{\text{to}} \) that are not specified by the class \( C_{\text{from}} \) are added, and local slots not specified by the class \( C_{\text{to}} \) that are specified by the class \( C_{\text{from}} \) are discarded.

The values of local slots specified by both the class \( C_{\text{to}} \) and the class \( C_{\text{from}} \) are retained. If such a local slot was unbound, it remains unbound.

The values of slots specified as shared in the class \( C_{\text{from}} \) and as local in the class \( C_{\text{to}} \) 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 \( C_{\text{from}} \); 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 \( C_{\text{to}} \). 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 specifiers 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 \textit{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 \textit{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 \(T_1 \ldots T_n\)) where \(T_1 \ldots T_n\) 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 \textit{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 \(C_1\) defines a slot named \(S\) whose value for the :allocation slot option is :class, that slot is accessible in instances of \(C_1\) and all of its subclasses. However, if \(C_2\) is a subclass of \(C_1\) and also defines a slot named \(S\), \(C_1\)'s slot is not shared by instances of \(C_2\) and its subclasses. When a class \(C_1\) defines a shared slot, any subclass \(C_2\) of \(C_1\) will share this single slot unless the defclass form for \(C_2\) specifies a slot of the same name or there is a superclass of \(C_2\) that precedes \(C_1\) in the class precedence list of \(C_2\) 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, 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 Figure 7–1.

<table>
<thead>
<tr>
<th>defgeneric</th>
<th>defmethod</th>
<th>defclass</th>
</tr>
</thead>
<tbody>
<tr>
<td>define-condition</td>
<td>defstruct</td>
<td></td>
</tr>
</tbody>
</table>

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 evalu-
ated, 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 \((A_1, \ldots, A_n)\) be the required arguments to a generic function in order. Let \((P_1, \ldots, P_n)\) be the parameter specializers corresponding to the required parameters of the method \(M\) in order. The method \(M\) is applicable when each \(A_i\) is of the type specified by the type specifier \(P_i\). 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 $P_{1,1} \ldots P_{1,n}$ and $P_{2,1} \ldots P_{2,n}$.

2. For each $1 \leq i \leq n$, $P_{1,i}$ agrees with $P_{2,i}$. The **parameter specializer** $P_{1,i}$ agrees with $P_{2,i}$ if $P_{1,i}$ and $P_{2,i}$ are the same class or if $P_{1,i} = (\texttt{eql} \ \texttt{object}_1)$, $P_{2,i} = (\texttt{eql} \ \texttt{object}_2)$, and $(\texttt{eql} \ \texttt{object}_1 \ \texttt{object}_2)$. Otherwise $P_{1,i}$ and $P_{2,i}$ 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.

```lisp
(width (make-instance 'character-class :char #\Q)
     :font 'baskerville :pixel-size 10)
```

The evaluation of the following form should signal an error.

```lisp
(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`.

```lisp
(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 Figure 7–2.

<table>
<thead>
<tr>
<th>+</th>
<th>append</th>
<th>max</th>
<th>nconc</th>
<th>progn</th>
<th>standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>list</td>
<td>min</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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

\((\text{generic-function } a_1 \ldots a_n)\)

Let \(M_1, \ldots, M_k\) be the applicable primary methods in order; then the derived Lisp form is

\((\text{operator } \langle M_1 a_1 \ldots a_n \rangle \ldots \langle M_k a_1 \ldots a_n \rangle)\)

If the expression \(\langle M_i a_1 \ldots a_n \rangle\) is evaluated, the method \(M_i\) will be applied to the arguments \(a_1 \ldots a_n\). For example, if operator is or, the expression \(\langle M_i a_1 \ldots a_n \rangle\) is evaluated only if \(\langle M_j a_1 \ldots a_n \rangle\), \(1 \leq j < i\), 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).
**function-keywords**  
*Standard Generic Function*

**Syntax:**

```lisp
function-keywords method → keys, allow-other-keys-p
```

**Method Signatures:**

```
function-keywords (method standard-method)
```

**Arguments and Values:**

- `method` — a method.
- `keys` — a list.
- `allow-other-keys-p` — a generalized boolean.

**Description:**

Returns the keyword parameter specifiers for a method.

Two values are returned: a list of the explicitly named keywords and a generalized boolean that states whether &allow-other-keys had been specified in the method definition.

**Examples:**

```lisp
(defun gf1 ((a integer) &optional (b 2)
        &key (c 3) ((:dee d) 4) e ((eff f)))
  (list a b c d e f))
→ #<STANDARD-METHOD GF1 (INTEGER) 36324653>
(find-method #'gf1 '() (list (find-class 'integer)))
→ #<STANDARD-METHOD GF1 (INTEGER) 36324653>
(function-keywords *)
→ (:C :DEE :E EFF), false
(defun gf2 ((a integer))
  (list a b c d e f))
→ #<STANDARD-METHOD GF2 (INTEGER) 42701775>
(function-keywords (find-method #'gf1 '() (list (find-class 'integer))))
→ (), false
(defun gf3 ((a integer) &key b c d &allow-other-keys)
  (list a b c d e f))
(function-keywords *)
→ (:B :C :D), true
```

**Affected By:**

`defmethod`

---

**7–28** Programming Language—Common Lisp
See Also:
defmethod

ensure-generic-function

Syntax:

```
ensure-generic-function function-name &key argument-precedence-order declare
documentation environment
generic-function-class lambda-list
method-class method-combination
```

→ generic-function

Arguments and Values:

- `function-name` — a function name.

The keyword arguments correspond to the `option` arguments of `defgeneric`, except that the
`:method-class` and `:generic-function-class` arguments can be `class objects` as well as names.


- `Environment` — the same as the `&environment` argument to macro expansion functions and is used
to distinguish between compile-time and run-time environments.

- `generic-function` — a `generic function object`.

Description:

The function `ensure-generic-function` is used to define a globally named `generic function` with
no `methods` or to specify or modify options and declarations that pertain to a globally named
`generic function` as a whole.

If `function-name` is not `fbound` in the `global environment`, a new `generic function` is created. If
`fdefinition function-name`) is an ordinary function, a macro, or a special operator; an error is
signaled.

If `function-name` is a `list`, it must be of the form `(setf symbol)`. If `function-name` specifies a
`generic function` that has a different value for any of the following arguments, the `generic function`
modified to have the new value: `:argument-precedence-order`, `:declare`, `:documentation`, `:method-combination`.

If `function-name` specifies a `generic function` that has a different value for the `:lambda-list`
argument, and the new value is congruent with the `lambda lists` of all existing `methods` or there
are no `methods`, the value is changed; otherwise an error is signaled.
If `function-name` specifies a *generic function* that has a different value for the
`:generic-function-class` argument and if the new generic function class is compatible with
the old, `change-class` is called to change the *class* of the *generic function*; otherwise an error is
signaled.

If `function-name` specifies a *generic function* that has a different value for the :method-class
argument, the value is changed, but any existing *methods* are not changed.

**Affected By:**
Existing function binding of `function-name`.

**Exceptional Situations:**
If `(fdefinition function-name)` is an *ordinary function*, a *macro*, or a *special operator*, an error of
*type error* is signaled.

If `function-name` specifies a *generic function* that has a different value for the :lambda-list
argument, and the new value is not congruent with the *lambda list* of any existing method, an
error of *type error* is signaled.

If `function-name` specifies a *generic function* that has a different value for the
`:generic-function-class` argument and if the new generic function class not is compatible with
the old, an error of *type error* is signaled.

**See Also:**
defgeneric

### allocate-instance

*Standard Generic Function*

**Syntax:**

```
allocate-instance class &rest initargs &key &allow-other-keys → new-instance
```

**Method Signatures:**

```
allocate-instance (class standard-class) &rest initargs
allocate-instance (class structure-class) &rest initargs
```

**Arguments and Values:**

class—a *class*.

initargs—a list of *keyword/value pairs* (initialization argument *names* and *values*).

new-instance—an *object* whose *class* is class.
Description:
The generic function `allocate-instance` creates and returns a new instance of the `class`, without initializing it. When the `class` is a `standard class`, this means that the `slots` are `unbound`; when the `class` is a `structure class`, this means the `slots`' values are unspecified.

The caller of `allocate-instance` is expected to have already checked the initialization arguments.

The generic function `allocate-instance` is called by `make-instance`, as described in Section 7.1 (Object Creation and Initialization).

See Also:
defclass, make-instance, class-of, Section 7.1 (Object Creation and Initialization)

Notes:
The consequences of adding `methods` to `allocate-instance` is unspecified. This capability might be added by the Metaobject Protocol.

---

`reinitialize-instance` Standard Generic Function

Syntax:
```
reinitialize-instance instance &rest initargs &key &allow-other-keys → instance
```

Method Signatures:
```
reinitialize-instance (instance standard-object) &rest initargs
```

Arguments and Values:
- `instance`—an `object`.
- `initargs`—an `initialization argument list`.

Description:
The generic function `reinitialize-instance` can be used to change the values of `local slots` of an `instance` according to `initargs`. This generic function can be called by users.

The system-supplied primary method for `reinitialize-instance` checks the validity of `initargs` and signals an error if an `initarg` is supplied that is not declared as valid. The method then calls the generic function `shared-initialize` with the following arguments: the `instance`, `nil` (which means no `slots` should be initialized according to their `initforms`), and the `initargs` it received.

Side Effects:
The generic function `reinitialize-instance` changes the values of `local slots`.

Objects 7–31
Exceptional Situations:
The system-supplied primary method for \texttt{reinitialize-instance} signals an error if an \texttt{initarg} is supplied that is not declared as valid.

See Also:
\texttt{initialize-instance}, \texttt{shared-initialize}, \texttt{update-instance-for-redefined-class}, \texttt{update-instance-for-different-class}, \texttt{slot-boundp}, \texttt{slot-makunbound}, Section 7.3 (Reinitializing an Instance), Section 7.1.4 (Rules for Initialization Arguments), Section 7.1.2 (Declaring the Validity of Initialization Arguments)

Notes:
\texttt{Initargs} are declared as valid by using the \texttt{:initarg} option to \texttt{defclass}, or by defining \texttt{methods} for \texttt{reinitialize-instance} or \texttt{shared-initialize}. The keyword name of each keyword parameter specifier in the \texttt{lambda list} of any \texttt{method} defined on \texttt{reinitialize-instance} or \texttt{shared-initialize} is declared as a valid initialization argument name for all \texttt{classes} for which that \texttt{method} is applicable.

\begin{verbatim}
shared-initialize  
\end{verbatim}

\textit{Standard Generic Function}

Syntact:
\begin{verbatim}
shared-initialize instance slot-names &rest initargs &key &allow-other-keys → instance
\end{verbatim}

Method Signatures:
\begin{verbatim}
shared-initialize (instance standard-object) slot-names &rest initargs
\end{verbatim}

Arguments and Values:
\begin{itemize}
\item \texttt{instance}—an \texttt{object}.
\item \texttt{slot-names}—a \texttt{list} or \texttt{t}.
\item \texttt{initargs}—a \texttt{list} of \texttt{keyword/value pairs} (of initialization argument \texttt{names} and \texttt{values}).
\end{itemize}

Description:
The generic function \texttt{shared-initialize} is used to fill the \texttt{slots} of an \texttt{instance} using \texttt{initargs} and :\texttt{initform} forms. It is called when an instance is created, when an instance is reinitialized, when an instance is updated to conform to a redefined \texttt{class}, and when an instance is updated to conform to a different \texttt{class}. The generic function \texttt{shared-initialize} is called by the system-supplied primary \texttt{method} for \texttt{initialize-instance}, \texttt{reinitialize-instance}, \texttt{update-instance-for-redefined-class}, and \texttt{update-instance-for-different-class}.

The generic function \texttt{shared-initialize} takes the following arguments: the \texttt{instance} to be initialized, a specification of a set of \texttt{slot-names accessible} in that \texttt{instance}, and any number of \texttt{initargs}. The arguments after the first two must form an \texttt{initialization argument list}. The system-supplied
primary method on shared-initialize initializes the slots with values according to the initargs and supplied :initform forms. Slot-names indicates which slots should be initialized according to their :initform forms if no initargs are provided for those slots.

The system-supplied primary method behaves as follows, regardless of whether the slots are local or shared:

- If an initarg 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.

- Any slots indicated by slot-names 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.

- The rules mentioned in Section 7.1.4 (Rules for Initialization Arguments) are obeyed.

The slots-names argument specifies the slots that are to be initialized according to their :initform forms if no initialization arguments apply. It can be a list of slot names, which specifies the set of those slot names; or it can be the symbol t, which specifies the set of all of the slots.

See Also:
initialize-instance, reinitialize-instance, update-instance-for-redefined-class, update-instance-for-different-class, slot-boundp, slot-makunbound, Section 7.1 (Object Creation and Initialization), Section 7.1.4 (Rules for Initialization Arguments), Section 7.1.2 (Declaring the Validity of Initialization Arguments)

Notes:
initargs are declared as valid by using the :initarg option to defclass, or by defining methods for shared-initialize. The keyword name of each keyword parameter specifier in the lambda list of any method defined on shared-initialize is declared as a valid initarg name for all classes for which that method is applicable.

Implementations are permitted to optimize :initform forms that neither produce nor depend on side effects, by evaluating these forms and storing them into slots before running any initialize-instance methods, rather than by handling them in the primary initialize-instance method. (This optimization might be implemented by having the allocate-instance method copy a prototype instance.)

Implementations are permitted to optimize default initial value forms for initargs associated with slots by not actually creating the complete initialization argument list when the only method that would receive the complete list is the method on standard-object. In this case default initial value forms can be treated like :initform forms. This optimization has no visible effects other than a performance improvement.
update-instance-for-different-class  

Function

Syntax:

update-instance-for-different-class previous current &rest initargs &key &allow-other-keys

→ implementation-dependent

Method Signatures:

update-instance-for-different-class (previous standard-object)
(current standard-object)
&rest initargs

Arguments and Values:

previous—a copy of the original instance.
current—the original instance (altered).
initargs—an initialization argument list.

Description:

The generic function update-instance-for-different-class is not intended to be called by programmers. Programmers may write methods for it. The function update-instance-for-different-class is called only by the function change-class.

The system-supplied primary method on update-instance-for-different-class checks the validity of initargs and signals an error if an initarg is supplied that is not declared as valid. This method then initializes slots with values according to the initargs, and initializes the newly added slots with values according to their :initform forms. It does this by calling the generic function shared-initialize with the following arguments: the instance (current), a list of names of the newly added slots, and the initargs it received. Newly added slots are those local slots for which no slot of the same name exists in the previous class.

Methods for update-instance-for-different-class can 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 therefore will not interfere with the default behavior of update-instance-for-different-class.

Methods on update-instance-for-different-class can be defined to initialize slots differently from change-class. The default behavior of change-class is described in Section 7.2 (Changing the Class of an Instance).
The arguments to **update-instance-for-different-class** are computed by **change-class**. When **change-class** is invoked on an instance, a copy of that instance is made; **change-class** then destructively alters the original instance. The first argument to **update-instance-for-different-class**, *previous*, is that copy; it holds the old slot values temporarily. This argument has dynamic extent within **change-class**; if it is referenced in any way once **update-instance-for-different-class** returns, the results are undefined. The second argument to **update-instance-for-different-class**, *current*, is the altered original instance. The intended use of *previous* is to extract old slot values by using **slot-value** or **with-slots** or by invoking a reader generic function, or to run other methods that were applicable to instances of the original class.

**Examples:**

See the example for the function **change-class**.

**Exceptional Situations:**

The system-supplied primary method on **update-instance-for-different-class** signals an error if an initialization argument is supplied that is not declared as valid.

**See Also:**

**change-class**, **shared-initialize**, Section 7.2 (Changing the Class of an Instance), Section 7.1.4 (Rules for Initialization Arguments), Section 7.1.2 (Declaring the Validity of Initialization Arguments)

**Notes:**

Initargs are declared as valid by using the :initarg option to **defclass**, or by defining methods for **update-instance-for-different-class** or **shared-initialize**. The keyword name of each keyword parameter specifier in the lambda list of any method defined on **update-instance-for-different-class** or **shared-initialize** is declared as a valid initarg name for all classes for which that method is applicable.

The value returned by **update-instance-for-different-class** is ignored by **change-class**.

---

**update-instance-for-redefined-class** **Standard Generic Function**

**Syntax:**

```
(update-instance-for-redefined-class instance added-slots discarded-slots property-list &rest initargs &key &allow-other-keys)
```

→ {result}*

---

Objects 7–35
update-instance-for-redefined-class

Method Signatures:

update-instance-for-redefined-class (instance standard-object)
  added-slots discarded-slots
  property-list
  &rest initargs

Arguments and Values:

instance—an object.
added-slots—a list.
discarded-slots—a list.
property-list—a list.
initargs—an initialization argument list.
result—an object.

Description:
The generic function update-instance-for-redefined-class is not intended to be called by programmers. Programmers may write methods for it. The generic function update-instance-for-redefined-class is called by the mechanism activated by make-instances-obsolete.

The system-supplied primary method on update-instance-for-redefined-class checks the validity of initargs and signals an error if an initarg is supplied that is not declared as valid. This method then initializes slots with values according to the initargs, and initializes the newly added-slots with values according to their :initform forms. It does this by calling the generic function shared-initialize with the following arguments: the instance, a list of names of the newly added-slots to instance, and the initargs it received. Newly added-slots are those local slots for which no slot of the same name exists in the old version of the class.

When make-instances-obsolete is invoked or when a class has been redefined and an instance is being updated, a property-list is created that captures the slot names and values of all the discarded-slots with values in the original instance. The structure of the instance is transformed so that it conforms to the current class definition. The arguments to update-instance-for-redefined-class are this transformed instance, a list of added-slots to the instance, a list discarded-slots from the instance, and the property-list containing the slot names and values for slots that were discarded and had values. Included in this list of discarded slots are slots that were local in the old class and are shared in the new class.

The value returned by update-instance-for-redefined-class is ignored.

Examples:
(defclass position () ()

(defclass x-y-position (position)
  ((x :initform 0 :accessor position-x)
   (y :initform 0 :accessor position-y)))

;;;; It turns out polar coordinates are used more than Cartesian
;;;; coordinates, so the representation is altered and some new
;;;; accessor methods are added.

(defmethod update-instance-for-redefined-class :before
  ((pos x-y-position) added deleted plist &key)
;;;; Transform the x-y coordinates to polar coordinates
;;;; and store into the new slots.
  (let ((x (getf plist 'x))
        (y (getf plist 'y)))
    (setf (position-rho pos) (sqrt (+ (* x x) (* y y))))
    (position-theta pos) (atan y x))))

(defclass x-y-position (position)
  ((rho :initform 0 :accessor position-rho)
   (theta :initform 0 :accessor position-theta)))

;;;; All instances of the old x-y-position class will be updated
;;;; automatically.

;;;; The new representation is given the look and feel of the old one.

(defmethod position-x ((pos x-y-position))
  (with-slots (rho theta) pos (* rho (cos theta))))

(defmethod (setf position-x) (new-x (pos x-y-position))
  (with-slots (rho theta) pos
    (let ((y (position-y pos)))
      (setq rho (sqrt (+ (* new-x new-x) (* y y)))
        theta (atan y new-x)
        new-x))))

(defmethod position-y ((pos x-y-position))
  (with-slots (rho theta) pos (* rho (sin theta))))

(defmethod (setf position-y) (new-y (pos x-y-position))
  (with-slots (rho theta) pos
    (let ((x (position-x pos)))
      (setq rho (sqrt (+ (* x x) (* new-y new-y))))))

Objects 7–37
theta (atan new-y x))
  new-y))

Exceptional Situations:
The system-supplied primary method on update-instance-for-redefined-class signals an error if an initarg is supplied that is not declared as valid.

See Also:
make-instances-obsolete, shared-initialize, Section 4.3.6 (Redefining Classes), Section 7.1.4 (Rules for Initialization Arguments), Section 7.1.2 (Declaring the Validity of Initialization Arguments)

Notes:
Initargs are declared as valid by using the :initarg option to defclass, or by defining methods for update-instance-for-redefined-class or shared-initialize. The keyword name of each keyword parameter specifier in the lambda list of any method defined on update-instance-for-redefined-class or shared-initialize is declared as a valid initarg name for all classes for which that method is applicable.

---

change-class

Standard Generic Function

Syntax:

change-class instance new-class &key &allow-other-keys → instance

Method Signatures:

change-class (instance standard-object) (new-class standard-class) &rest initargs

change-class (instance t) (new-class symbol) &rest initargs

Arguments and Values:

instance—an object.

new-class—a class designator.

initargs—an initialization argument list.

Description:
The generic function change-class changes the class of an instance to new-class. It destructively modifies and returns the instance.
change-class

If in the old class there is any slot of the same name as a local slot in the new-class, the value of that slot is retained. This means that if the slot has a value, the value returned by slot-value after change-class is invoked is eql to the value returned by slot-value before change-class is invoked. Similarly, if the slot was unbound, it remains unbound. The other slots are initialized as described in Section 7.2 (Changing the Class of an Instance).

After completing all other actions, change-class invokes update-instance-for-different-class. The generic function update-instance-for-different-class can be used to assign values to slots in the transformed instance. See Section 7.2.2 (Initializing Newly Added Local Slots).

If the second of the above methods is selected, that method invokes change-class on instance, (find-class new-class), and the initargs.

Examples:

(defun change-class () ()
  (defclass x-y-position (position)
   ((x :initform 0 :initarg :x)
    (y :initform 0 :initarg :y)))

(defun rho-theta-position (position)
  ((rho :initform 0)
   (theta :initform 0)))

(defun update-instance-for-different-class :before ((old x-y-position) 
  (new rho-theta-position) &key)
  ;; Copy the position information from old to new to make new 
  ;; be a rho-theta-position at the same position as old.
  (let ((x (slot-value old 'x)) 
    (y (slot-value old 'y)))
    (setf (slot-value new 'rho) (sqrt (+ (* x x) (* y y)))
          (slot-value new 'theta) (atan y x))))

(defun make-instance ('x-y-position :x 2 :y 0))

(change-class p1 'rho-theta-position)

The result is that the instance bound to p1 is now an instance of 
the class rho-theta-position. The update-instance-for-different-class 
method performed the initialization of the rho and theta slots based
;;; on the value of the x and y slots, which were maintained by
;;; the old instance.

See Also:
update-instance-for-different-class, Section 7.2 (Changing the Class of an Instance)

Notes:
The generic function change-class has several semantic difficulties. First, it performs a destruc-
tive operation that can be invoked within a method on an instance that was used to select that
method. When multiple methods are involved because methods are being combined, the methods
currently executing or about to be executed may no longer be applicable. Second, some imple-
mentations might use compiler optimizations of slot access, and when the class of an instance is
changed the assumptions the compiler made might be violated. This implies that a programmer
must not use change-class inside a method if any methods for that generic function access any
slots, or the results are undefined.

**slot-boundp**

**Function**

Syntax:

```lisp
slot-boundp instance slot-name → generalized-boolean
```

Arguments and Values:

- `instance`—an object.
- `slot-name`—a symbol naming a slot of `instance`.
- `generalized-boolean`—a generalized boolean.

Description:

Returns true if the slot named `slot-name` in `instance` is bound; otherwise, returns false.

Exceptional Situations:

If no slot of the name `slot-name` exists in the `instance`, `slot-missing` is called as follows:

```lisp
(slot-missing (class-of instance)
  instance
  slot-name
  'slot-boundp)
```

(If `slot-missing` is invoked and returns a value, a boolean equivalent to its primary value is
returned by `slot-boundp`.)
The specific behavior depends on instance's metaclass. An error is never signaled if instance has metaclass standard-class. An error is always signaled if instance has metaclass built-in-class. The consequences are undefined if instance has any other metaclass—an error might or might not be signaled in this situation. Note in particular that the behavior for conditions and structures is not specified.

See Also:
slot-makunbound, slot-missing

Notes:
The function slot-boundp allows for writing after methods on initialize-instance in order to initialize only those slots that have not already been bound.

Although no implementation is required to do so, implementors are strongly encouraged to implement the function slot-boundp using the function slot-boundp-using-class described in the Metaobject Protocol.

---

**slot-exists-p**  
*Function*

Syntax:

```
slot-exists-p object slot-name → generalized-boolean
```

Arguments and Values:

- **object**—an object.
- **slot-name**—a symbol.
- **generalized-boolean**—a generalized boolean.

Description:

Returns true if the object has a slot named slot-name.

Affected By:

defclass, defstruct

See Also:

defclass, slot-missing

Notes:

Although no implementation is required to do so, implementors are strongly encouraged to implement the function slot-exists-p using the function slot-exists-p-using-class described in the Metaobject Protocol.
slot-makunbound

Function

Syntax:

slot-makunbound instance slot-name → instance

Arguments and Values:

instance – instance.

Slot-name—A symbol.

Description:

The function slot-makunbound restores a slot of the name slot-name in an instance to the unbound state.

Exceptional Situations:

If no slot of the name slot-name exists in the instance, slot-missing is called as follows:

(slot-missing (class-of instance)
  instance
  slot-name
  'slot-makunbound)

(Any values returned by slot-missing in this case are ignored by slot-makunbound.)

The specific behavior depends on instance’s metaclass. An error is never signaled if instance has metaclass standard-class. An error is always signaled if instance has metaclass built-in-class. The consequences are undefined if instance has any other metaclass—an error might or might not be signaled in this situation. Note in particular that the behavior for conditions and structures is not specified.

See Also:

slot-boundp, slot-missing

Notes:

Although no implementation is required to do so, implementors are strongly encouraged to implement the function slot-makunbound using the function slot-makunbound-using-class described in the Metaobject Protocol.
Standard Generic Function

Syntax:

```
slot-missing class object slot-name operation &optional new-value  → {result}*
```

Method Signatures:

```
slot-missing (class t) object slot-name
  operation &optional new-value
```

Arguments and Values:

- **class**—the class of **object**.
- **object**—an **object**.
- **slot-name**—a symbol (the name of a would-be slot).
- **operation**—one of the symbols setf, slot-boundp, slot-makunbound, or slot-value.
- **new-value**—an **object**.
- **result**—an **object**.

Description:

The generic function **slot-missing** is invoked when an attempt is made to access a slot in an **object** whose metaclass is **standard-class** and the slot of the name **slot-name** is not a name of a slot in that class. The default method signals an error.

The generic function **slot-missing** is not intended to be called by programmers. Programmers may write methods for it.

The generic function **slot-missing** may be called during evaluation of **slot-value**, (setf **slot-value**), **slot-boundp**, and **slot-makunbound**. For each of these operations the corresponding symbol for the **operation** argument is **slot-value**, setf, **slot-boundp**, and **slot-makunbound** respectively.

The optional **new-value** argument to **slot-missing** is used when the operation is attempting to set the value of the slot.

If **slot-missing** returns, its values will be treated as follows:

- If the **operation** is setf or **slot-makunbound**, any values will be ignored by the caller.
- If the **operation** is **slot-value**, only the primary value will be used by the caller, and all other values will be ignored.
• If the operation is `slot-boundp`, any boolean equivalent of the primary value of the method might be is used, and all other values will be ignored.

Exceptional Situations:
The default method on `slot-missing` signals an error of type `error`.

See Also:
defclass, slot-exists-p, slot-value

Notes:
The set of arguments (including the `class` of the instance) facilitates defining methods on the metaclass for `slot-missing`.

**slot-unbound**  
*Standard Generic Function*

Syntax:
```
slot-unbound class instance slot-name → {result}*
```

Method Signatures:
```
slot-unbound (class t) instance slot-name
```

Arguments and Values:
- `class`—the `class` of the `instance`.
- `instance`—the `instance` in which an attempt was made to read the unbound slot.
- `slot-name`—the name of the unbound slot.
- `result`—an object.

Description:
The generic function `slot-unbound` is called when an unbound slot is read in an `instance` whose metaclass is `standard-class`. The default method signals an error of type `unbound-slot`. The name slot of the `unbound-slot condition` is initialized to the name of the offending variable, and the instance slot of the `unbound-slot condition` is initialized to the offending instance.

The generic function `slot-unbound` is not intended to be called by programmers. Programmers may write methods for it. The function `slot-unbound` is called only indirectly by `slot-value`.

If `slot-unbound` returns, only the primary value will be used by the caller, and all other values will be ignored.
Exceptional Situations:
The default method on slot-unbound signals an error of type unbound-slot.

See Also:
slot-makunbound

Notes:
An unbound slot may occur if no :initform form was specified for the slot and the slot value has not been set, or if slot-makunbound has been called on the slot.

**slot-value**  Function

**Syntax:**

slot-value object slot-name → value

**Arguments and Values:**

object—an object.

name—a symbol.

value—an object.

**Description:**

The function slot-value returns the value of the slot named slot-name in the object. If there is no slot named slot-name, slot-missing is called. If the slot is unbound, slot-unbound is called.

The macro setf can be used with slot-value to change the value of a slot.

**Examples:**

```lisp
(defun foo ()
  ((a :accessor foo-a :initarg :a :initform 1)
   (b :accessor foo-b :initarg :b)
   (c :accessor foo-c :initform 3))

(slot-value foo1 'a) → ONE
(slot-value foo1 'b) → TWO
(slot-value foo1 'c) → 3
(setf (slot-value foo1 'a) 'uno) → UNO
(slot-value foo1 'a) → UNO
(defmethod foo-method ((x foo))
```

Objects 7–45
(slot-value x 'a))
⇒ #<STANDARD-METHOD FOO-METHOD (FOO) 42720573>
(foo-method foo1) ⇒ UNO

Exceptional Situations:

If an attempt is made to read a slot and no slot of the name slot-name exists in the object, slot-missing is called as follows:

(slot-missing (class-of instance)
  instance
  slot-name
  'slot-value)

(If slot-missing is invoked, its primary value is returned by slot-value.)

If an attempt is made to write a slot and no slot of the name slot-name exists in the object, slot-missing is called as follows:

(slot-missing (class-of instance)
  instance
  slot-name
  'setf
  new-value)

(If slot-missing returns in this case, any values are ignored.)

The specific behavior depends on object’s metaclass. An error is never signaled if object has metaclass standard-class. An error is always signaled if object has metaclass built-in-class. The consequences are unspecified if object has any other metaclass—an error might or might not be signaled in this situation. Note in particular that the behavior for conditions and structures is not specified.

See Also:

slot-missing, slot-unbound, with-slots

Notes:

Although no implementation is required to do so, implementors are strongly encouraged to implement the function slot-value using the function slot-value-using-class described in the Metaobject Protocol.

Implementations may optimize slot-value by compiling it inline.
method-qualifiers

Method Signatures:
  method-qualifiers (method standard-method)

Arguments and Values:
  method—a method.
  qualifiers—a proper list.

Description:
  Returns a list of the qualifiers of the method.

Examples:
  (defmethod some-gf :before ((a integer)) a)
  → #<STANDARD-METHOD SOME-GF (:BEFORE) (INTEGER) 42736540>
  (method-qualifiers *) → (:BEFORE)

See Also:
  define-method-combination

no-applicable-method

Method Signatures:
  no-applicable-method (generic-function t)
     &rest function-arguments

Arguments and Values:
  generic-function—a generic function on which no applicable method was found.
  function-arguments—arguments to the generic-function.
  result—an object.
Description:
The generic function `no-applicable-method` is called when a generic function is invoked and no method on that generic function is applicable. The default method signals an error.

The generic function `no-applicable-method` is not intended to be called by programmers. Programmers may write methods for it.

Exceptional Situations:
The default method signals an error of type `error`.

See Also:

---

### no-next-method

*Standard Generic Function*

Syntax:

```
no-next-method generic-function method &rest args → {result}*
```

Method Signatures:

```
no-next-method (generic-function standard-generic-function)
   (method standard-method)
   &rest args
```

Arguments and Values:

- `generic-function` – generic function to which `method` belongs.
- `method` – method that contained the call to `call-next-method` for which there is no next method.
- `args` – arguments to `call-next-method`.
- `result` – an object.

Description:
The generic function `no-next-method` is called by `call-next-method` when there is no next method.

The generic function `no-next-method` is not intended to be called by programmers. Programmers may write methods for it.

Exceptional Situations:
The system-supplied method on `no-next-method` signals an error of type `error`.

See Also:

- `call-next-method`
remove-method

**Standard Generic Function**

**Syntax:**

remove-method generic-function method → generic-function

**Method Signatures:**

remove-method (generic-function standard-generic-function) method

**Arguments and Values:**

generic-function—a generic function.
method—a method.

**Description:**

The generic function remove-method removes a method from generic-function by modifying the generic-function (if necessary).

remove-method must not signal an error if the method is not one of the methods on the generic-function.

**See Also:**

find-method

make-instance

**Standard Generic Function**

**Syntax:**

make-instance class &rest initargs &key &allow-other-keys → instance

**Method Signatures:**

make-instance (class standard-class) &rest initargs
make-instance (class symbol) &rest initargs

**Arguments and Values:**
class—a class, or a symbol that names a class.
initargs—an initialization argument list.
instance—a fresh instance of class class.

Description:

The generic function make-instance creates and returns a new instance of the given class.

If the second of the above methods is selected, that method invokes make-instance on the arguments (find-class class) and initargs.

The initialization arguments are checked within make-instance.

The generic function make-instance may be used as described in Section 7.1 (Object Creation and Initialization).

Exceptional Situations:

If any of the initialization arguments has not been declared as valid, an error of type error is signaled.

See Also:

defclass, class-of, allocate-instance, initialize-instance, Section 7.1 (Object Creation and Initialization)

make-instances-obsolete Standard Generic Function

Syntax:

make-instances-obsolete class → class

Method Signatures:

make-instances-obsolete (class standard-class)

make-instances-obsolete (class symbol)

Arguments and Values:

class—a class designator.

Description:

The function make-instances-obsolete has the effect of initiating the process of updating the instances of the class. During updating, the generic function update-instance-for-redefined-class will be invoked.

The generic function make-instances-obsolete is invoked automatically by the system when defclass has been used to redefine an existing standard class and the set of local slots accessible in an instance is changed or the order of slots in storage is changed. It can also be explicitly invoked by the user.
If the second of the above methods is selected, that method invokes make-instances-obsole

Examples:

See Also:

update-instance-for-redefined-class, Section 4.3.6 (Redefining Classes)

make-load-form

Syntax:

make-load-form object &optional environment → creation-form, initialization-form

Method Signatures:

make-load-form (object standard-object) &optional environment
make-load-form (object structure-object) &optional environment
make-load-form (object condition) &optional environment
make-load-form (object class) &optional environment

Arguments and Values:

object—an object.

environment—an environment object.

creation-form—a form.

initialization-form—a form.

Description:

The generic function make-load-form creates and returns one or two forms, a creation-form and an initialization-form, that enable load to construct an object equivalent to object. Environment is an environment object corresponding to the lexical environment in which the forms will be processed.

The file compiler calls make-load-form to process certain classes of literal objects; see Section 3.2.4.4 (Additional Constraints on Externalizable Objects).

Conforming programs may call make-load-form directly, providing object is a generalized instance of standard-object, structure-object, or condition.

The creation form is a form that, when evaluated at load time, should return an object that is equivalent to object. The exact meaning of equivalent depends on the type of object and is up to
make-load-form

the programmer who defines a method for make-load-form; see Section 3.2.4 (Literal Objects in Compiled Files).

The initialization form is a form that, when evaluated at load time, should perform further initialization of the object. The value returned by the initialization form is ignored. If make-load-form returns only one value, the initialization form is nil, which has no effect. If object appears as a constant in the initialization form, at load time it will be replaced by the equivalent object constructed by the creation form; this is how the further initialization gains access to the object.

Both the creation-form and the initialization-form may contain references to any externalizable object. However, there must not be any circular dependencies in creation forms. An example of a circular dependency is when the creation form for the object X contains a reference to the object Y, and the creation form for the object Y contains a reference to the object X. Initialization forms are not subject to any restriction against circular dependencies, which is the reason that initialization forms exist; see the example of circular data structures below.

The creation form for an object is always evaluated before the initialization form for that object. When either the creation form or the initialization form references other objects that have not been referenced earlier in the file being compiled, the compiler ensures that all of the referenced objects have been created before evaluating the referencing form. When the referenced object is of a type which the file compiler processes using make-load-form, this involves evaluating the creation form returned for it. (This is the reason for the prohibition against circular references among creation forms).

Each initialization form is evaluated as soon as possible after its associated creation form, as determined by data flow. If the initialization form for an object does not reference any other objects not referenced earlier in the file and processed by the file compiler using make-load-form, the initialization form is evaluated immediately after the creation form. If a creation or initialization form F does contain references to such objects, the creation forms for those other objects are evaluated before F, and the initialization forms for those other objects are also evaluated before F whenever they do not depend on the object created or initialized by F. Where these rules do not uniquely determine an order of evaluation between two creation/initialization forms, the order of evaluation is unspecified.

While these creation and initialization forms are being evaluated, the objects are possibly in an uninitialized state, analogous to the state of an object between the time it has been created by allocate-instance and it has been processed fully by initialize-instance. Programmers writing methods for make-load-form must take care in manipulating objects not to depend on slots that have not yet been initialized.

It is implementation-dependent whether load calls eval on the forms or does some other operation that has an equivalent effect. For example, the forms might be translated into different but equivalent forms and then evaluated, they might be compiled and the resulting functions called by load, or they might be interpreted by a special-purpose function different from eval. All that is required is that the effect be equivalent to evaluating the forms.
make-load-form

The method specialized on class returns a creation form using the name of the class if the class has a proper name in environment, signaling an error of type error if it does not have a proper name. Evaluation of the creation form uses the name to find the class with that name, as if by calling find-class. If a class with that name has not been defined, then a class may be computed in an implementation-defined manner. If a class cannot be returned as the result of evaluating the creation form, then an error of type error is signaled.

Both conforming implementations and conforming programs may further specialize make-load-form.

Examples:

```lisp
(defun obj ()
  ((x :initarg :x :reader obj-x)
   (y :initarg :y :reader obj-y)
   (dist :accessor obj-dist))
→ #<STANDARD-CLASS OBJ 250020030>
(defun shared-initialize :after ((self obj) slot-names &rest keys)
  (declare (ignore slot-names keys))
  (unless (slot-boundp self 'dist)
    (setf (obj-dist self)
      (sqrt (+ (expt (obj-x self) 2) (expt (obj-y self) 2))))))
→ #<STANDARD-METHOD SHARED-INITIALIZE (:AFTER) (OBJ T) 26266714>
(defun make-load-form ((self obj) &optional environment)
  (declare (ignore environment))
  ;; Note that this definition only works because X and Y do not
  ;; contain information which refers back to the object itself.
  ;; For a more general solution to this problem, see revised example below.
  '(make-instance ',(class-of self)
    :x ',(obj-x self) :y ',(obj-y self)))
→ #<STANDARD-METHOD MAKE-LOAD-FORM (OBJ) 26267532>
(setq obj1 (make-instance 'obj :x 3.0 :y 4.0)) → #<OBJ 26274136>
(1.0) → 5.0
(make-load-form-saving-slots obj1) → (MAKE-INSTANCE 'OBJ :X '3.0 :Y '4.0)
```

In the above example, an equivalent instance of obj is reconstructed by using the values of two of its slots. The value of the third slot is derived from those two values.

Another way to write the make-load-form method in that example is to use make-load-form-saving-slots. The code it generates might yield a slightly different result from the make-load-form method shown above, but the operational effect will be the same. For example:

```lisp
(defun obj ()
  ((x :initarg :x :reader obj-x)
   (y :initarg :y :reader obj-y)
   (dist :accessor obj-dist))
→ #<STANDARD-CLASS OBJ 250020030>
(defun shared-initialize :after ((self obj) slot-names &rest keys)
  (declare (ignore slot-names keys))
  (unless (slot-boundp self 'dist)
    (setf (obj-dist self)
      (sqrt (+ (expt (obj-x self) 2) (expt (obj-y self) 2))))))
→ #<STANDARD-METHOD SHARED-INITIALIZE (:AFTER) (OBJ T) 26266714>
(defun make-load-form ((self obj) &optional environment)
  (declare (ignore environment))
  ;; Note that this definition only works because X and Y do not
  ;; contain information which refers back to the object itself.
  ;; For a more general solution to this problem, see revised example below.
  '(make-instance ',(class-of self)
    :x ',(obj-x self) :y ',(obj-y self)))
→ #<STANDARD-METHOD MAKE-LOAD-FORM (OBJ) 26267532>
(setq obj1 (make-instance 'obj :x 3.0 :y 4.0)) → #<OBJ 26274136>
(1.0) → 5.0
(make-load-form-saving-slots obj1) → (MAKE-INSTANCE 'OBJ :X '3.0 :Y '4.0)
```

Objects 7–53
make-load-form

:slot-names '(x y)
:environment environment))

→ #<STANDARD-METHOD MAKE-LOAD-FORM (OBJ) 42755655>

;; Try MAKE-LOAD-FORM on object created above.
(make-load-form obj1)

→ (ALLOCATE-INSTANCE '#<STANDARD-CLASS OBJ 250020030>),
(PROGN
  (SETF (SLOT-VALUE '#<OBJ 26274136> 'X) '3.0)
  (SETF (SLOT-VALUE '#<OBJ 26274136> 'Y) '4.0)
  (INITIALIZE-INSTANCE '#<OBJ 26274136>))

In the following example, instances of my-frob are “interned” in some way. An equivalent instance is reconstructed by using the value of the name slot as a key for searching existing objects. In this case the programmer has chosen to create a new object if no existing object is found; alternatively an error could have been signaled in that case.

(defun my-name (self))

(defmethod make-load-form ((self my-frob) &optional environment)
  (declare (ignore environment))
  '(find-my-frob ',(my-name self) :if-does-not-exist :create))

In the following example, the data structure to be dumped is circular, because each parent has a list of its children and each child has a reference back to its parent. If make-load-form is called on one object in such a structure, the creation form creates an equivalent object and fills in the children slot, which forces creation of equivalent objects for all of its children, grandchildren, etc. At this point none of the parent slots have been filled in. The initialization form fills in the parent slot, which forces creation of an equivalent object for the parent if it was not already created. Thus the entire tree is recreated at load time. At compile time, make-load-form is called once for each object in the tree. All of the creation forms are evaluated, in implementation-dependent order, and then all of the initialization forms are evaluated, also in implementation-dependent order.

(defun tree-parent (x))

(defmethod make-load-form ((x tree-with-parent) &optional environment)
  (declare (ignore environment))
  (values
    ;; creation form
    '(make-instance ',(class-of x) :children ',(slot-value x 'children))
    ;; initialization form
    '(setf (tree-parent ',x) ',(slot-value x 'parent))))

In the following example, the data structure to be dumped has no special properties and an equivalent structure can be reconstructed simply by reconstructing the slots’ contents.

(defun my-struct (a b c))
(defmethod make-load-form ((s my-struct) &optional environment)
  (make-load-form-saving-slots s :environment environment))

Exceptional Situations:
The methods specialized on standard-object, structure-object, and condition all signal an error of type error.

It is implementation-dependent whether calling make-load-form on a generalized instance of a system class signals an error or returns creation and initialization forms.

See Also:
compile-file, make-load-form-saving-slots, Section 3.2.4.4 (Additional Constraints on Externalizable Objects) Section 3.1 (Evaluation), Section 3.2 (Compilation)

Notes:
The file compiler calls make-load-form in specific circumstances detailed in Section 3.2.4.4 (Additional Constraints on Externalizable Objects).

Some implementations may provide facilities for defining new subclasses of classes which are specified as system classes. (Some likely candidates include generic-function, method, and stream). Such implementations should document how the file compiler processes instances of such classes when encountered as literal objects, and should document any relevant methods for make-load-form.

make-load-form-saving-slots  

Function

Syntax:
make-load-form-saving-slots object &key slot-names environment  
  → creation-form, initialization-form

Arguments and Values:
object—an object.
slot-names—a list.
environment—an environment object.
creation-form—a form.
initialization-form—a form.

Description:
Returns forms that, when evaluated, will construct an object equivalent to object, without executing initialization forms. The slots in the new object that correspond to initialized slots in
object are initialized using the values from object. Uninitialized slots in object are not initialized in the new object. make-load-form-saving-slots works for any instance of standard-object or structure-object.

Slot-names is a list of the names of the slots to preserve. If slot-names is not supplied, its value is all of the local slots.

make-load-form-saving-slots returns two values, thus it can deal with circular structures. Whether the result is useful in an application depends on whether the object's type and slot contents fully capture the application's idea of the object's state.

Environment is the environment in which the forms will be processed.

See Also:
make-load-form, make-instance, setf, slot-value, slot-makunbound

Notes:
make-load-form-saving-slots can be useful in user-written make-load-form methods.

When the object is an instance of standard-object, make-load-form-saving-slots could return a creation form that calls allocate-instance and an initialization form that contains calls to setf of slot-value and slot-makunbound, though other functions of similar effect might actually be used.

### with-accessors

**Macro**

Syntax:

```
with-accessors ((slot-entry)*) instance-form {declaration}* {form}* → {result}*
```

```
slot-entry:=((variable-name accessor-name)
```

Arguments and Values:

- **variable-name**—a variable name; not evaluated.
- **accessor-name**—a function name; not evaluated.
- **instance-form**—a form; evaluated.
- **declaration**—a declare expression; not evaluated.
- **forms**—an implicit progn.
- **results**—the values returned by the forms.
with-accessors

Description:
Creates a lexical environment in which the slots specified by slot-entry are lexically available through their accessors as if they were variables. The macro with-accessors invokes the appropriate accessors to access the slots specified by slot-entry. Both setf and setq can be used to set the value of the slot.

Examples:

(defun thing ()
  ((x :initarg :x :accessor thing-x)
   (y :initarg :y :accessor thing-y)))
  → #<STANDARD-CLASS THING 250020173>
(defun method (setf thing-x) :before (new-x (thing thing))
  (format t "Changing X from ~D to ~D in ~S."
          (thing-x thing) new-x thing))
(setq thing1 (make-instance 'thing :x 1 :y 2)) → #<THING 43135676>
(setq thing2 (make-instance 'thing :x 7 :y 8)) → #<THING 43147374>
(with-accessors ((x1 thing-x) (y1 thing-y))
  thing1
  (with-accessors ((x2 thing-x) (y2 thing-y))
    thing2
    (list (list x1 (thing-x thing1) y1 (thing-y thing1)
           x2 (thing-x thing2) y2 (thing-y thing2))
          (setq x1 (+ y1 x2))
          (list x1 (thing-x thing1) y1 (thing-y thing1)
                x2 (thing-x thing2) y2 (thing-y thing2))
          (setf (thing-x thing2) (list x1))
          (list x1 (thing-x thing1) y1 (thing-y thing1)
                x2 (thing-x thing2) y2 (thing-y thing2)))))
→ Changing X from 1 to 9 in #<THING 43135676>.
→ Changing X from 7 to (9) in #<THING 43147374>.
→ ((1 1 2 2 7 7 8 8)
    (9)
    (9 9 2 2 (9) (9) 8 8))

Affected By:
defclass

Exceptional Situations:
The consequences are undefined if any accessor-name is not the name of an accessor for the instance.
See Also:
with-slots, symbol-macrolet

Notes:
A with-accessors expression of the form:

\[
(with\text{-accessors} \ (slot-entry_1 \ldots slot-entry_n) \ instance-form \ form_1 \ldots form_k)
\]

expands into the equivalent of

\[
(let \ ((in \ instance-form))
  \ (symbol-macrolet \ (Q_1 \ldots Q_n) \ form_1 \ldots form_k))
\]

where \( Q_i \) is

\[
(variable-name_i \ () \ (accessor-name_i \ in))
\]

with-slots

Macro

Syntax:

\[
\text{with-slots} \ \{(\text{slot-entry})^*\} \ \text{instance-form} \ \{(\text{declaration})^*\} \ \{(\text{form})^*\} \\
\rightarrow \ \{(\text{result})^*\}
\]

\[\text{slot-entry}::=\text{slot-name} | (\text{variable-name} \ \text{slot-name})\]

Arguments and Values:

- \text{slot-name}—a \textit{slot name}; not evaluated.
- \text{variable-name}—a \textit{variable name}; not evaluated.
- \text{instance-form}—a \textit{form}; evaluated to produce \textit{instance}.
- \text{instance}—an \textit{object}.
- \text{declaration}—a \textit{declare expression}; not evaluated.
- \text{forms}—an \textit{implicit prog}\_\text{}.
- \text{results}—the \textit{values} returned by the \textit{forms}.
with-slots

Description:

The macro **with-slots** establishes a lexical environment for referring to the slots in the instance named by the given slot-names as though they were variables. Within such a context the value of the slot can be specified by using its slot name, as if it were a lexically bound variable. Both **setf** and **setq** can be used to set the value of the slot.

The macro **with-slots** translates an appearance of the slot name as a variable into a call to **slot-value**.

Examples:

```
(defun defclass (name ()
  ((x :initarg :x :accessor thing-x)
   (y :initarg :y :accessor thing-y)))
  → #<STANDARD-CLASS THING 250020173>
(defmethod (setf thing-x) :before (new-x (thing thing))
  (format t "Changing X from ~D to ~D in ~S."%"
    (thing-x thing) new-x thing))
(setq thing (make-instance 'thing :x 0 :y 1))
  → #<THING 62310540>
(with-slots (x y) thing (incf x) (incf y)) → 2
(values (thing-x thing) (thing-y thing)) → 1, 2
(setq thing1 (make-instance 'thing :x 1 :y 2)) → #<THING 43135676>
(setq thing2 (make-instance 'thing :x 7 :y 8)) → #<THING 43147374>
(with-slots ((x1 x) (y1 y))
  thing1
  (with-slots ((x2 x) (y2 y))
    thing2
    (list (list x1 (thing-x thing1) y1 (thing-y thing1)
      x2 (thing-x thing2) y2 (thing-y thing2))
      (setq x1 (+ y1 x2))
      (list x1 (thing-x thing1) y1 (thing-y thing1)
        x2 (thing-x thing2) y2 (thing-y thing2))
      (setf (thing-x thing2) (list x1))
      (list x1 (thing-x thing1) y1 (thing-y thing1)
        x2 (thing-x thing2) y2 (thing-y thing2)))))
  → Changing X from 7 to (9) in #<THING 43147374>.
  → ((1 1 2 2 7 7 8 8)
    9
    (9 9 2 2 7 7 8 8)
    (9)
    (9 9 2 2 (9) (9) 8 8))
```

Affected By:

**defclass**
Exceptional Situations:
The consequences are undefined if any slot-name is not the name of a slot in the instance.

See Also:
with-accessors, slot-value, symbol-macrolet

Notes:
A with-slots expression of the form:

\[ (\text{with-slots} \ (\text{slot-entry}_1 \ldots \text{slot-entry}_n) \ \text{instance-form} \ \text{form}_1 \ldots \text{form}_k) \]

expands into the equivalent of

\[ (\text{let} \ ((\text{in} \ \text{instance-form})) \ (\text{symbol-macrolet} \ (Q_1 \ldots Q_n) \ \text{form}_1 \ldots \text{form}_k)) \]

where \( Q_i \) is

\[ (\text{slot-entry}_i () \ (\text{slot-value} \ \text{in} \ '\text{slot-entry}_i)) \]

if \( \text{slot-entry}_i \) is a symbol and is

\[ (\text{variable-name}_i () \ (\text{slot-value} \ \text{in} \ '\text{slot-name}_i)) \]

if \( \text{slot-entry}_i \) is of the form

\[ (\text{variable-name}_i \ \text{slot-name}_i) \]

---

**defclass**

Macro

Syntax:

\[ \text{defclass} \ \text{class-name} \ ((\text{superclass-name})*) ((\text{slot-specifier})*) [[\text{class-option}]] \rightarrow \text{new-class} \]

\[ \text{slot-specifier}::= \text{slot-name} | (\text{slot-name} [[\text{slot-option}]]) \]

\[ \text{slot-name}::= \text{symbol} \]

\[ \text{slot-option}::= \{\text{:reader reader-function-name}\}* | \{\text{:writer writer-function-name}\}* \]

7–60  Programming Language—Common Lisp
defclass

::{:accessor reader-function-name}* | 
::{allocation allocation-type} | 
::{initarg initarg-name}* | 
::{initform form} | 
::{type type-specifier} | 
::{documentation string}

function-name::= {symbol | (setf symbol)}
class-option::= (:default-initargs . initarg-list) | 
::{documentation string} | 
::{metaclass class-name}

Arguments and Values:

Class-name—a non-nil symbol.
Superclass-name—a non-nil symbol.
Slot-name—a symbol. The slot-name argument is a symbol that is syntactically valid for use as a variable name.
Reader-function-name—a non-nil symbol. :reader can be supplied more than once for a given slot.
Writer-function-name—a generic function name. :writer can be supplied more than once for a given slot.
Reader-function-name—a non-nil symbol. :accessor can be supplied more than once for a given slot.
Allocation-type—(member :instance :class). :allocation can be supplied once at most for a given slot.
Initarg-name—a symbol. :initarg can be supplied more than once for a given slot.
Form—a form. :init-form can be supplied once at most for a given slot.
Type-specifier—a type specifier. :type can be supplied once at most for a given slot.
Class-option—refers to the class as a whole or to all class slots.
Initarg-list—a list of alternating initialization argument names and default initial value forms. :default-initargs can be supplied at most once.
Class-name—a non-nil symbol. :metaclass can be supplied once at most.
new-class—the new class object.

Description:
The macro defclass defines a new named class. It returns the new class object as its result.
defclass

The syntax of defclass provides options for specifying initialization arguments for slots, for specifying default initialization values for slots, and for requesting that methods on specified generic functions be automatically generated for reading and writing the values of slots. No reader or writer functions are defined by default; their generation must be explicitly requested. However, slots can always be accessed using slot-value.

Defining a new class also causes a type of the same name to be defined. The predicate (typep object class-name) returns true if the class of the given object is the class named by class-name itself or a subclass of the class class-name. A class object can be used as a type specifier. Thus (typep object class) returns true if the class of the object is class itself or a subclass of class.

The class-name argument specifies the proper name of the new class. If a class with the same proper name already exists and that class is an instance of standard-class, and if the defclass form for the definition of the new class specifies a class of class standard-class, the existing class is redefined, and instances of it (and its subclasses) are updated to the new definition at the time that they are next accessed. For details, see Section 4.3.6 (Redefining Classes).

Each superclass-name argument specifies a direct superclass of the new class. If the superclass list is empty, then the superclass defaults depending on the metaclass, with standard-object being the default for standard-class.

The new class will inherit slots and methods from each of its direct superclasses, from their direct superclasses, and so on. For a discussion of how slots and methods are inherited, see Section 4.3.4 (Inheritance).

The following slot options are available:

- The :reader slot option specifies that an unqualified method is to be defined on the generic function named reader-function-name to read the value of the given slot.

- The :writer slot option specifies that an unqualified method is to be defined on the generic function named writer-function-name to write the value of the slot.

- The :accessor slot option specifies that an unqualified method is to be defined on the generic function named reader-function-name to read the value of the given slot and that an unqualified method is to be defined on the generic function named (setf reader-function-name) to be used with setf to modify the value of the slot.

- The :allocation slot option is used to specify where storage is to be allocated for the given slot. Storage for a slot can be located in each instance or in the class object itself. The value of the allocation-type argument can be either the keyword :instance or the keyword :class. If the :allocation slot option is not specified, the effect is the same as specifying :allocation :instance.

  - If allocation-type is :instance, a local slot of the name slot-name is allocated in each instance of the class.
If allocation-type is :class, a shared slot of the given name is allocated in the class object created by this defclass form. The value of the slot is shared by all instances of the class. If a class C1 defines such 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 and that defines a slot of the same name.

- The :initform slot option is used to provide a default initial value form to be used in the initialization of the slot. This form is evaluated every time it is used to initialize the slot. The lexical environment in which this form is evaluated is the lexical environment in which the defclass form was evaluated. Note that the lexical environment refers both to variables and to functions. For local slots, the dynamic environment is the dynamic environment in which make-instance is called; for shared slots, the dynamic environment is the dynamic environment in which the defclass form was evaluated. See Section 7.1 (Object Creation and Initialization).

No implementation is permitted to extend the syntax of defclass to allow (slot-name form) as an abbreviation for (slot-name :initform form).

- The :initarg slot option declares an initialization argument named initarg-name and specifies that this initialization argument initializes the given slot. If the initialization argument has a value in the call to initialize-instance, the value will be stored into the given slot, and the slot’s :initform slot option, if any, is not evaluated. If none of the initialization arguments specified for a given slot has a value, the slot is initialized according to the :initform slot option, if specified.

- The :type slot option specifies that the contents of the slot will always be of the specified data type. It effectively declares the result type of the reader generic function when applied to an object of this class. The consequences of attempting to store in a slot a value that does not satisfy the type of the slot are undefined. The :type slot option is further discussed in Section 7.5.3 (Inheritance of Slots and Slot Options).

- The :documentation slot option provides a documentation string for the slot. :documentation can be supplied once at most for a given slot.

Each class option is an option that refers to the class as a whole. The following class options are available:

- The :default-initargs class option is followed by a list of alternating initialization argument names and default initial value forms. If any of these initialization arguments does not appear in the initialization argument list supplied to make-instance, the corresponding default initial value form is evaluated, and the initialization argument name and the form’s value are added to the end of the initialization argument list before the instance is created; see Section 7.1 (Object Creation and Initialization). The default initial value form is evaluated each time it is used. The lexical environment in which this form is evaluated is the lexical environment in which the defclass form was evaluated. The
defclass

dynamic environment is the dynamic environment in which make-instance was called. If an initialization argument name appears more than once in a :default-initargs class option, an error is signaled.

- The :documentation class option causes a documentation string to be attached with the class object, and attached with kind type to the class-name. :documentation can be supplied once at most.

- The :metaclass class option is used to specify that instances of the class being defined are to have a different metaclass than the default provided by the system (the class standard-class).

Note the following rules of defclass for standard classes:

- It is not required that the superclasses of a class be defined before the defclass form for that class is evaluated.

- All the superclasses of a class must be defined before an instance of the class can be made.

- A class must be defined before it can be used as a parameter specializer in a defmethod form.

The object system can be extended to cover situations where these rules are not obeyed.

Some slot options are inherited by a class from its superclasses, and some can be shadowed or altered by providing a local slot description. No class options except :default-initargs are inherited. For a detailed description of how slots and slot options are inherited, see Section 7.5.3 (Inheritance of Slots and Slot Options).

The options to defclass can be extended. It is required that all implementations signal an error if they observe a class option or a slot option that is not implemented locally.

It is valid to specify more than one reader, writer, accessor, or initialization argument for a slot. No other slot option can appear more than once in a single slot description, or an error is signaled.

If no reader, writer, or accessor is specified for a slot, the slot can only be accessed by the function slot-value.

If a defclass form appears as a top level form, the compiler must make the class name be recognized as a valid type name in subsequent declarations (as for deftype) and be recognized as a valid class name for defmethod parameter specializers and for use as the :metaclass option of a subsequent defclass. The compiler must make the class definition available to be returned by find-class when its environment argument is a value received as the environment parameter of a macro.
Exceptional Situations:

If there are any duplicate slot names, an error of type `program-error` is signaled.

If an initialization argument `name` appears more than once in `:default-initargs` class option, an error of type `program-error` is signaled.

If any of the following slot options appears more than once in a single slot description, an error of type `program-error` is signaled: `:allocation`, `:initform`, `:type`, `:documentation`.

It is required that all implementations signal an error of type `program-error` if they observe a class option or a slot option that is not implemented locally.

See Also:
documentation, initialize-instance, make-instance, slot-value, Section 4.3 (Classes), Section 4.3.4 (Inheritance), Section 4.3.6 (Redefining Classes), Section 4.3.5 (Determining the Class Precedence List), Section 7.1 (Object Creation and Initialization)

defgeneric

Macro

Syntax:

```
defgeneric function-name gf-lambda-list [[ | option | ( | method-description) * ]]
   → new-generic
```

Arguments and Values:

- `function-name`—a function name.
- `generic-function-class`—a non-nil symbol naming a class.
- `gf-declaration`—an optimize declaration specifier; other declaration specifiers are not permitted.
- `gf-documentation`—a string; not evaluated.
- `gf-lambda-list`—a generic function lambda list.
defgeneric

method-class—a non-nil symbol naming a class.

method-combination-argument—an object.

method-combination-name—a symbol naming a method combination type.

method-qualifiers, specialized-lambda-list, declarations, documentation, forms—as per defmethod.

new-generic—the generic function object.

parameter-name—a symbol that names a required parameter in the lambda-list. (If the :argument-precedence-order option is specified, each required parameter in the lambda-list must be used exactly once as a parameter-name.)

Description:
The macro defgeneric is used to define a generic function or to specify options and declarations that pertain to a generic function as a whole.

If function-name is a list it must be of the form (setf symbol). If (fboundp function-name) is false, a new generic function is created. If (fdefinition function-name) is a generic function, that generic function is modified. If function-name names an ordinary function, a macro, or a special operator, an error is signaled.

The effect of the defgeneric macro is as if the following three steps were performed: first, methods defined by previous defgeneric forms are removed; second, ensure-generic-function is called; and finally, methods specified by the current defgeneric form are added to the generic function.

Each method-description defines a method on the generic function. The lambda list of each method must be congruent with the lambda list specified by the gf-lambda-list option. If no method descriptions are specified and a generic function of the same name does not already exist, a generic function with no methods is created.

The gf-lambda-list argument of defgeneric specifies the shape of lambda lists for the methods on this generic function. All methods on the resulting generic function must have lambda lists that are congruent with this shape. If a defgeneric form is evaluated and some methods for that generic function have lambda lists that are not congruent with that given in the defgeneric form, an error is signaled. For further details on method congruence, see Section 7.6.4 (Congruent Lambda-lists for all Methods of a Generic Function).

The generic function passes to the method all the argument values passed to it, and only those; default values are not supported. Note that optional and keyword arguments in method definitions, however, can have default initial value forms and can use supplied-p parameters.

The following options are provided. Except as otherwise noted, a given option may occur only once.

- The :argument-precedence-order option is used to specify the order in which the required arguments in a call to the generic function are tested for specificity when selecting a
particular method. Each required argument, as specified in the \texttt{gf-lambda-list} argument, must be included exactly once as a \texttt{parameter-name} so that the full and unambiguous precedence order is supplied. If this condition is not met, an error is signaled.

- The \texttt{declare} option is used to specify declarations that pertain to the \texttt{generic function}.

An \texttt{optimize} declaration specifier is allowed. It specifies whether method selection should be optimized for speed or space, but it has no effect on \texttt{methods}. To control how a \texttt{method} is optimized, an \texttt{optimize} declaration must be placed directly in the \texttt{defmethod} form or method description. The optimization qualities \texttt{speed} and \texttt{space} are the only qualities this standard requires, but an implementation can extend the object system to recognize other qualities. A simple implementation that has only one method selection technique and ignores \texttt{optimize declaration specifiers} is valid.

The \texttt{special}, \texttt{ftype}, \texttt{function}, \texttt{inline}, \texttt{notinline}, and \texttt{declaration} declarations are not permitted. Individual implementations can extend the \texttt{declare} option to support additional declarations. If an implementation notices a \texttt{declaration specifier} that it does not support and that has not been proclaimed as a non-standard \texttt{declaration identifier} name in a \texttt{declaration proclamation}, it should issue a warning.

The \texttt{declare} option may be specified more than once. The effect is the same as if the lists of \texttt{declaration specifiers} had been appended together into a single list and specified as a single \texttt{declare} option.

- The \texttt{:documentation} argument is a \texttt{documentation string} to be attached to the \texttt{generic function object}, and to be attached with kind \texttt{function} to the \texttt{function-name}.

- The \texttt{:generic-function-class} option may be used to specify that the \texttt{generic function} is to have a different \texttt{class} than the default provided by the system (the \texttt{class standard-generic-function}). The \texttt{class-name} argument is the name of a \texttt{class} that can be the \texttt{class} of a \texttt{generic function}. If \texttt{function-name} specifies an existing \texttt{generic function} that has a different value for the \texttt{:generic-function-class} argument and the new \texttt{generic function} \texttt{class} is compatible with the old, \texttt{change-class} is called to change the \texttt{class} of the \texttt{generic function}; otherwise an error is signaled.

- The \texttt{:method-class} option is used to specify that all \texttt{methods} on this \texttt{generic function} are to have a different \texttt{class} from the default provided by the system (the \texttt{class standard-method}). The \texttt{class-name} argument is the name of a \texttt{class} that is capable of being the \texttt{class} of a \texttt{method}.

- The \texttt{:method-combination} option is followed by a symbol that names a type of method combination. The arguments (if any) that follow that symbol depend on the type of method combination. Note that the standard method combination type does not support any arguments. However, all types of method combination defined by the short form of \texttt{define-method-combination} accept an optional argument named \texttt{order}, defaulting to \texttt{:most-specific-first}, where a value of \texttt{:most-specific-last} reverses the order of the primary \texttt{methods} without affecting the order of the auxiliary \texttt{methods}. 

Objects 7–67
The method-description arguments define methods that will be associated with the generic function. The method-qualifier and specialized-lambda-list arguments in a method description are the same as for defmethod.

The form arguments specify the method body. The body of the method is enclosed in an implicit block. If function-name is a symbol, this block bears the same name as the generic function. If function-name is a list of the form (setf symbol), the name of the block is symbol.

Implementations can extend defgeneric to include other options. It is required that an implementation signal an error if it observes an option that is not implemented locally.

defgeneric is not required to perform any compile-time side effects. In particular, the methods are not installed for invocation during compilation. An implementation may choose to store information about the generic function for the purposes of compile-time error-checking (such as checking the number of arguments on calls, or noting that a definition for the function name has been seen).

Examples:

Exceptional Situations:

If function-name names an ordinary function, a macro, or a special operator, an error of type program-error is signaled.

Each required argument, as specified in the gf-lambda-list argument, must be included exactly once as a parameter-name, or an error of type program-error is signaled.

The lambda list of each method specified by a method-description must be congruent with the lambda list specified by the gf-lambda-list option, or an error of type error is signaled.

If a defgeneric form is evaluated and some methods for that generic function have lambda lists that are not congruent with that given in the defgeneric form, an error of type error is signaled.

A given option may occur only once, or an error of type program-error is signaled.

If function-name specifies an existing generic function that has a different value for the :generic-function-class argument and the new generic function class is compatible with the old, change-class is called to change the class of the generic function; otherwise an error of type error is signaled.

Implementations can extend defgeneric to include other options. It is required that an implementation signal an error of type program-error if it observes an option that is not implemented locally.

See Also:

defmethod, documentation, ensure-generic-function, generic-function, Section 7.6.4 (Congruent Lambda-lists for all Methods of a Generic Function)
**defmethod**

**Macro**

**Syntax:**

```lisp
defmethod function-name {method-qualifier}* specialized-lambda-list
    [{declaration}* | documentation] {form}* → new-method
```

- `function-name` ::= `{symbol | (setq symbol)}`
- `method-qualifier` ::= `non-list`
- `specialized-lambda-list` ::= `{var | (var parameter-specializer-name)}*` 
  - `[&optional {var | (var [initform [supplied-p-parameter] ])}]*` 
  - `[&rest var]` 
  - `[&key {var | (var [initform [supplied-p-parameter] ])}]*` 
  - `[&allow-other-keys]` 
  - `[&aux {var | (var [initform] )}*]`
- `parameter-specializer-name` ::= `symbol | (eql eql-specializer-form)`

**Arguments and Values:**

- `declaration` — a `declare` expression; not evaluated.
- `documentation` — a string; not evaluated.
- `var` — a variable name.
- `eql-specializer-form` — a form.
- `Form` — a form.
- `Initform` — a form.
- `Supplied-p-parameter` — variable name.
- `new-method` — the new method object.

**Description:**

The macro `defmethod` defines a method on a generic function.

If `(fboundp function-name)` is `nil`, a generic function is created with default values for the argument precedence order (each argument is more specific than the arguments to its right in the argument list), for the generic function class (the class `standard-generic-function`), for the method class (the class `standard-method`), and for the method combination type (the standard method combination type). The lambda list of the generic function is congruent with the lambda
defmethod

list of the method being defined; if the defmethod form mentions keyword arguments, the lambda list of the generic function will mention &key (but no keyword arguments). If function-name names an ordinary function, a macro, or a special operator, an error is signaled.

If a generic function is currently named by function-name, the lambda list of the method must be congruent with the lambda list of the generic function. If this condition does not hold, an error is signaled. For a definition of congruence in this context, see Section 7.6.4 (Congruent Lambda-lists for all Methods of a Generic Function).

Each method-qualifier argument is an object that is used by method combination to identify the given method. The method combination type might further restrict what a method qualifier can be. The standard method combination type allows for unqualified methods and methods whose sole qualifier is one of the keywords: before, :after, or :around.

The specialized-lambda-list argument is like an ordinary lambda list except that the names of required parameters can be replaced by specialized parameters. A specialized parameter is a list of the form (var parameter-specializer-name). Only required parameters can be specialized. If parameter-specializer-name is a symbol it names a class; if it is a list, it is of the form (eql eql-specializer-form). The parameter specializer name (eql eql-specializer-form) indicates that the corresponding argument must be eql to the object that is the value of eql-specializer-form for the method to be applicable. The eql-specializer-form is evaluated at the time that the expansion of the defmethod macro is evaluated. If no parameter specializer name is specified for a given required parameter, the parameter specializer defaults to the class t. For further discussion, see Section 7.6.2 (Introduction to Methods).

The form arguments specify the method body. The body of the method is enclosed in an implicit block. If function-name is a symbol, this block bears the same name as the generic function. If function-name is a list of the form (setf symbol), the name of the block is symbol.

The class of the method object that is created is that given by the method class option of the generic function on which the method is defined.

If the generic function already has a method that agrees with the method being defined on parameter specializers and qualifiers, defmethod replaces the existing method with the one now being defined. For a definition of agreement in this context, see Section 7.6.3 (Agreement on Parameter Specializers and Qualifiers).

The parameter specializers are derived from the parameter specializer names as described in Section 7.6.2 (Introduction to Methods).

The expansion of the defmethod macro “refers to” each specialized parameter (see the description of ignore within the description of declare). This includes parameters that have an explicit parameter specializer name of t. This means that a compiler warning does not occur if the body of the method does not refer to a specialized parameter, while a warning might occur if the body of the method does not refer to an unspecialized parameter. For this reason, a parameter that specializes on t is not quite synonymous with an unspecialized parameter in this context.

Declarations at the head of the method body that apply to the method’s lambda variables are
treated as bound declarations whose scope is the same as the corresponding bindings.

Declarations at the head of the method body that apply to the functional bindings of call-next-method or next-method-p apply to references to those functions within the method body forms. Any outer bindings of the function names call-next-method and next-method-p, and declarations associated with such bindings are shadowed within the method body forms.

The scope of free declarations at the head of the method body is the entire method body, which includes any implicit local function definitions but excludes initialization forms for the lambda variables.

defmethod is not required to perform any compile-time side effects. In particular, the methods are not installed for invocation during compilation. An implementation may choose to store information about the generic function for the purposes of compile-time error-checking (such as checking the number of arguments on calls, or noting that a definition for the function name has been seen).

Documentation is attached as a documentation string to the method object.

Affected By:
The definition of the referenced generic function.

Exceptional Situations:
If function-name names an ordinary function, a macro, or a special operator, an error of type error is signaled.

If a generic function is currently named by function-name, the lambda list of the method must be congruent with the lambda list of the generic function, or an error of type error is signaled.

See Also:
defgeneric, documentation, Section 7.6.2 (Introduction to Methods), Section 7.6.4 (Congruent Lambda-lists for all Methods of a Generic Function), Section 7.6.3 (Agreement on Parameter Specializers and Qualifiers), Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

find-class

Syntax:
find-class symbol &optional errorp environment → class
(setf find-class symbol &optional errorp environment) new-class

Accessor

Objects 7–71
Arguments and Values:

symbol—a symbol.
errorp—a generalized boolean. The default is true.
environment—same as the &environment argument to macro expansion functions and is used to distinguish between compile-time and run-time environments. The &environment argument has dynamic extent; the consequences are undefined if the &environment argument is referred to outside the dynamic extent of the macro expansion function.
class—a class object, or nil.

Description:

Returns the class object named by the symbol in the environment. If there is no such class, nil is returned if errorp is false; otherwise, if errorp is true, an error is signaled.

The class associated with a particular symbol can be changed by using setf with find-class; or, if the new class given to setf is nil, the class association is removed (but the class object itself is not affected). The results are undefined if the user attempts to change or remove the class associated with a symbol that is defined as a type specifier in this standard. See Section 4.3.7 (Integrating Types and Classes).

When using setf of find-class, any errorp argument is evaluated for effect, but any values it returns are ignored; the errorp parameter is permitted primarily so that the environment parameter can be used.

The environment might be used to distinguish between a compile-time and a run-time environment.

Exceptional Situations:

If there is no such class and errorp is true, find-class signals an error of type error.

See Also:

defmacro, Section 4.3.7 (Integrating Types and Classes)

---

next-method-p

Local Function

Syntax:

next-method-p (no arguments) → generalized-boolean

Arguments and Values:

generalized-boolean—a generalized boolean.
Description:
The locally defined function `next-method-p` can be used within the body forms (but not the lambda list) defined by a method-defining form to determine whether a next method exists.

The function `next-method-p` has lexical scope and indefinite extent.

Whether or not `next-method-p` is `fbound` in the global environment is implementation-dependent; however, the restrictions on redefinition and shadowing of `next-method-p` are the same as for symbols in the `COMMON-LISP` package which are `fbound` in the global environment. The consequences of attempting to use `next-method-p` outside of a method-defining form are undefined.

See Also:
call-next-method, defmethod, call-method

call-method, make-method

Local Macro

Syntax:
call-method method &optional next-method-list → {result}*
make-method form → method-object

Arguments and Values:
method—a method object, or a list (see below); not evaluated.
method-object—a method object.
ext-method-list—a list of method objects; not evaluated.
results—the values returned by the method invocation.

Description:
The macro `call-method` is used in method combination. It hides the implementation-dependent details of how methods are called. The macro `call-method` has lexical scope and can only be used within an effective method form.

Whether or not `call-method` is `fbound` in the global environment is implementation-dependent; however, the restrictions on redefinition and shadowing of `call-method` are the same as for symbols in the `COMMON-LISP` package which are `fbound` in the global environment. The consequences of attempting to use `call-method` outside of an effective method form are undefined.

The macro `call-method` invokes the specified method, supplying it with arguments and with definitions for `call-next-method` and for `next-method-p`. If the invocation of `call-method` is lexically inside of a `make-method`, the arguments are those that were supplied to that method.
Otherwise the arguments are those that were supplied to the generic function. The definitions of call-next-method and next-method-p rely on the specified next-method-list.

If method is a list, the first element of the list must be the symbol make-method and the second element must be a form. Such a list specifies a method object whose method function has a body that is the given form.

Next-method-list can contain method objects or lists, the first element of which must be the symbol make-method and the second element of which must be a form.

Those are the only two places where make-method can be used. The form used with make-method is evaluated in the null lexical environment augmented with a local macro definition for call-method and with bindings named by symbols not accessible from the COMMON-LISP-USER package.

The call-next-method function available to method will call the first method in next-method-list. The call-next-method function available in that method, in turn, will call the second method in next-method-list, and so on, until the list of next methods is exhausted.

If next-method-list is not supplied, the call-next-method function available to method signals an error of type control-error and the next-method-p function available to method returns nil.

Examples:

See Also:
call-next-method, define-method-combination, next-method-p

call-next-method

Local Function

Syntax:
call-next-method &rest args → {result}* 

Arguments and Values:
arg—an object.

results—the values returned by the method it calls.

Description:
The function call-next-method can be used within the body forms (but not the lambda list) of a method defined by a method-defining form to call the next method.

If there is no next method, the generic function no-next-method is called.

The type of method combination used determines which methods can invoke call-next-method.

The standard method combination type allows call-next-method to be used within primary
methods and around methods. For generic functions using a type of method combination defined by the short form of define-method-combination, call-next-method can be used in around methods only.

When call-next-method is called with no arguments, it passes the current method’s original arguments to the next method. Neither argument defaulting, nor using setq, nor rebinding variables with the same names as parameters of the method affects the values call-next-method passes to the method it calls.

When call-next-method is called with arguments, the next method is called with those arguments.

If call-next-method is called with arguments but omits optional arguments, the next method called defaults those arguments.

The function call-next-method returns any values that are returned by the next method.

The function call-next-method has lexical scope and indefinite extent and can only be used within the body of a method defined by a method-defining form.

Whether or not call-next-method is fbound in the global environment is implementation-dependent; however, the restrictions on redefinition and shadowing of call-next-method are the same as for symbols in the COMMON-LISP package which are fbound in the global environment. The consequences of attempting to use call-next-method outside of a method-defining form are undefined.

Affected By:
defmethod, call-method, define-method-combination.

Exceptional Situations:
When providing arguments to call-next-method, the following rule must be satisfied or an error of type error should be signaled: the ordered set of applicable methods for a 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. Optimizations of the error checking are possible, but they must not change the semantics of call-next-method.

See Also:
define-method-combination, defmethod, next-method-p, no-next-method, call-method, Section 7.6.6 (Method Selection and Combination), Section 7.6.6.2 (Standard Method Combination), Section 7.6.6.4 (Built-in Method Combination Types)
compute-applicable-methods  

*Standard Generic Function*

Syntax:

\[
\text{compute-applicable-methods } \text{generic-function } \text{function-arguments} \rightarrow \text{methods}
\]

Method Signatures:

\[
\text{compute-applicable-methods } (\text{generic-function } \text{standard-generic-function})
\]

Arguments and Values:

- **generic-function**—a *generic function*.
- **function-arguments**—a *list of arguments for the generic-function*.
- **methods**—a *list of method objects*.

Description:

Given a *generic-function* and a set of *function-arguments*, the function *compute-applicable-methods* returns the *set of methods* that are applicable for those arguments sorted according to precedence order. See Section 7.6.6 (Method Selection and Combination).

Affected By:

- defmethod

See Also:

- Section 7.6.6 (Method Selection and Combination)

---

define-method-combination  

*Macro*

Syntax:

\[
\text{define-method-combination } \text{name} \left[ \text{[short-form-option]} \right] \rightarrow \text{name}
\]

\[
\text{define-method-combination } \text{name} \text{ lambda-list}
\]

\[
\left( \{ \text{method-group-specifier} \}^* \right)
\]

\[
\left[ \text{[arguments . args-lambda-list]} \right]
\]

\[
\left[ \text{[generic-function generic-function-symbol]} \right]
\]

\[
\{ \text{declaration} \}^* \text{ [documentation]}
\]

\[
\{ \text{form} \}^*
\]

\[
\rightarrow \text{name}
\]
define-method-combination

\[\text{short-form-option} ::= \text{documentation documentation} | \]
\[\quad \text{:identity-with-one-argument identity-with-one-argument} | \]
\[\quad \text{operator operator} \]

\[\text{method-group-specifier} ::= (\text{name} \{(\text{qualifier-pattern})^+ | \text{predicate} \}[\downarrow \text{long-form-option}])\]

\[\text{long-form-option} ::= \text{description description} | \]
\[\quad \text{:order order} | \]
\[\quad \text{required required-p} \]

Arguments and Values:
- \text{args-lambda-list} — a \text{define-method-combination} arguments lambda list.
- \text{declaration} — a \text{declare expression}; not evaluated.
- \text{description} — a \text{format control}.
- \text{documentation} — a \text{string}; not evaluated.
- \text{forms} — an implicit \text{progn} that must compute and return the form that specifies how the methods are combined, that is, the effective method.
- \text{generic-function-symbol} — a symbol.
- \text{identity-with-one-argument} — a generalized boolean.
- \text{lambda-list} — ordinary lambda list.
- \text{name} — a symbol. Non-keyword, non-nil symbols are usually used.
- \text{operator} — an operator. \text{Name} and \text{operator} are often the same symbol. This is the default, but it is not required.
- \text{order} — \text{:most-specific-first} or \text{:most-specific-last}; evaluated.
- \text{predicate} — a symbol that names a function of one argument that returns a generalized boolean.
- \text{qualifier-pattern} — a list, or the symbol *.
- \text{required-p} — a generalized boolean.

Description:
The macro \text{define-method-combination} is used to define new types of method combination.

There are two forms of \text{define-method-combination}. The short form is a simple facility for the cases that are expected to be most commonly needed. The long form is more powerful but more verbose. It resembles \text{defmacro} in that the body is an expression, usually using backquote, that computes a form. Thus arbitrary control structures can be implemented. The long form also
define-method-combination

allows arbitrary processing of method qualifiers.

Short Form

The short form syntax of define-method-combination is recognized when the second subform is a non-nil symbol or is not present. When the short form is used, name is defined as a type of method combination that produces a Lisp form (operator method-call method-call ...). The operator is a symbol that can be the name of a function, macro, or special operator. The operator can be supplied by a keyword option; it defaults to name.

Keyword options for the short form are the following:

- The :documentation option is used to document the method-combination type; see description of long form below.
- The :identity-with-one-argument option enables an optimization when its value is true (the default is false). If there is exactly one applicable method and it is a primary method, that method serves as the effective method and operator is not called. This optimization avoids the need to create a new effective method and avoids the overhead of a function call. This option is designed to be used with operators such as `progn`, `and`, `+`, and `max`.
- The :operator option specifies the name of the operator. The operator argument is a symbol that can be the name of a function, macro, or special form.

These types of method combination 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.

A method combination procedure defined in this way recognizes two roles for methods. A method whose one qualifier is the symbol naming this type of method combination is defined to be a primary method. At least one primary method must be applicable or an error is signaled. A method with :around as its one qualifier is an auxiliary method that behaves the same as an around method in standard method combination. The function `call-next-method` can only be used in around methods; it cannot be used in primary methods defined by the short form of the define-method-combination macro.

A method combination procedure defined in this way accepts an optional argument named order, which defaults to :most-specific-first. A value of :most-specific-last reverses the order of the primary methods without affecting the order of the auxiliary methods.

The short form automatically includes error checking and support for around methods.

For a discussion of built-in method combination types, see Section 7.6.6.4 (Built-in Method Combination Types).
define-method-combination

Long Form

The long form syntax of define-method-combination is recognized when the second subform is a list.

The lambda-list receives any arguments provided after the name of the method combination type in the :method-combination option to defgeneric.

A list of method group specifiers follows. Each specifier selects a subset of the applicable methods to play a particular role, either by matching their qualifiers against some patterns or by testing their qualifiers with a predicate. These method group specifiers define all method qualifiers that can be used with this type of method combination.

The car of each method-group-specifier is a symbol which names a variable. During the execution of the forms in the body of define-method-combination, this variable is bound to a list of the methods in the method group. The methods in this list occur in the order specified by the :order option.

If qualifier-pattern is a symbol it must be *. A method matches a qualifier-pattern if the method’s list of qualifiers is equal to the qualifier-pattern (except that the symbol * in a qualifier-pattern matches anything). Thus a qualifier-pattern can be one of the following: the empty list, which matches unqualified methods; the symbol *, which matches all methods; a true list, which matches methods with the same number of qualifiers as the length of the list when each qualifier matches the corresponding list element; or a dotted list that ends in the symbol * (the * matches any number of additional qualifiers).

Each applicable method is tested against the qualifier-patterns and predicates in left-to-right order. As soon as a qualifier-pattern matches or a predicate returns true, the method becomes a member of the corresponding method group and no further tests are made. Thus if a method could be a member of more than one method group, it joins only the first such group. If a method group has more than one qualifier-pattern, a method need only satisfy one of the qualifier-patterns to be a member of the group.

The name of a predicate function can appear instead of qualifier-patterns in a method group specifier. The predicate is called for each method that has not been assigned to an earlier method group; it is called with one argument, the method’s qualifier list. The predicate should return true if the method is to be a member of the method group. A predicate can be distinguished from a qualifier-pattern because it is a symbol other than nil or *.

If there is an applicable method that does not fall into any method group, the function invalid-method-error is called.

Method group specifiers can have keyword options following the qualifier patterns or predicate. Keyword options can be distinguished from additional qualifier patterns because they are neither lists nor the symbol *. The keyword options are as follows:

- The :description option is used to provide a description of the role of methods
define-method-combination

in the method group. Programming environment tools use `(apply #'format stream format-control (method-qualifiers method))` to print this description, which is expected to be concise. This keyword option allows the description of a method qualifier to be defined in the same module that defines the meaning of the method qualifier. In most cases, `format-control` will not contain any `format` directives, but they are available for generality. If `:description` is not supplied, a default description is generated based on the variable name and the qualifier patterns and on whether this method group includes the `unqualified methods`.

- The `:order` option specifies the order of methods. The `order` argument is a form that evaluates to `:most-specific-first` or `:most-specific-last`. If it evaluates to any other value, an error is signaled. If `:order` is not supplied, it defaults to `:most-specific-first`.

- The `:required` option specifies whether at least one method in this method group is required. If its value is `true` and the method group is empty (that is, no applicable methods match the `qualifier` patterns or satisfy the predicate), an error is signaled. If `:required` is not supplied, it defaults to `nil`.

The use of method group specifiers provides a convenient syntax to select methods, to divide them among the possible roles, and to perform the necessary error checking. It is possible to perform further filtering of methods in the body forms by using normal list-processing operations and the functions `method-qualifiers` and `invalid-method-error`. It is permissible to use `setq` on the variables named in the method group specifiers and to bind additional variables. It is also possible to bypass the method group specifier mechanism and do everything in the body forms. This is accomplished by writing a single method group with `*` as its only `qualifier-pattern`; the variable is then bound to a `list` of all of the `applicable methods`, in most-specific-first order.

The body forms compute and return the form that specifies how the methods are combined, that is, the effective method. The effective method is evaluated in the `null lexical environment` augmented with a local macro definition for `call-method` and with bindings named by symbols not `accessible` from the `COMMON-LISP-USER` package. Given a method object in one of the lists produced by the method group specifiers and a `list` of next methods, `call-method` will invoke the method such that `call-next-method` has available the next methods.

When an effective method has no effect other than to call a single method, some implementations employ an optimization that uses the single method directly as the effective method, thus avoiding the need to create a new effective method. This optimization is active when the effective method form consists entirely of an invocation of the `call-method` macro whose first `subform` is a method object and whose second `subform` is `nil` or unsupplied. Each `define-method-combination` body is responsible for stripping off redundant invocations of `progn`, `and`, `multiple-value-prog1`, and the like, if this optimization is desired.

The list `(:arguments . lambda-list)` can appear before any declarations or `documentation`
define-method-combination

string. This form is useful when the method combination type performs some specific behavior as part of the combined method and that behavior needs access to the arguments to the generic function. Each parameter variable defined by lambda-list is bound to a form that can be inserted into the effective method. When this form is evaluated during execution of the effective method, its value is the corresponding argument to the generic function; the consequences of using such a form as the place in a setf form are undefined. Argument correspondence is computed by dividing the :arguments lambda-list and the generic function lambda-list into three sections: the required parameters, the optional parameters, and the keyword and rest parameters. The arguments supplied to the generic function for a particular call are also divided into three sections; the required arguments section contains as many arguments as the generic function has required parameters, the optional arguments section contains as many arguments as the generic function has optional parameters, and the keyword/rest arguments section contains the remaining arguments. Each parameter in the required and optional sections of the :arguments lambda-list accesses the argument at the same position in the corresponding section of the arguments. If the section of the :arguments lambda-list is shorter, extra arguments are ignored. If the section of the :arguments lambda-list is longer, excess required parameters are bound to forms that evaluate to nil and excess optional parameters are bound to their initforms. The keyword parameters and rest parameters in the :arguments lambda-list access the keyword/rest section of the arguments. If the :arguments lambda-list contains &key, it behaves as if it also contained &allow-other-keys.

In addition, &whole var can be placed first in the :arguments lambda-list. It causes var to be bound to a form that evaluates to a list of all of the arguments supplied to the generic function. This is different from &rest because it accesses all of the arguments, not just the keyword/rest arguments.

Erroneous conditions detected by the body should be reported with method-combination-error or invalid-method-error; these functions add any necessary contextual information to the error message and will signal the appropriate error.

The body forms are evaluated inside of the bindings created by the lambda list and method group specifiers. Declarations at the head of the body are positioned directly inside of bindings created by the lambda list and outside of the bindings of the method group variables. Thus method group variables cannot be declared in this way. locally may be used around the body, however.

Within the body forms, generic-function-symbol is bound to the generic function object.

Documentation is attached as a documentation string to name (as kind method-combination) and to the method combination object.

Note that two methods with identical specializers, but with different qualifiers, are not ordered by the algorithm described in Step 2 of the method selection and combination process described in Section 7.6.6 (Method Selection and Combination). Normally the two methods play different roles in the effective method because they have different qualifiers, and no matter how they are ordered in the result of Step 2, the
**define-method-combination**

An effective method is the same. If the two methods play the same role and their order matters, an error is signaled. This happens as part of the **qualifier** pattern matching in **define-method-combination**.

If a **define-method-combination** form appears as a top level form, the compiler must make the **method combination name** be recognized as a valid **method combination name** in subsequent **defgeneric** forms. However, the **method combination** is executed no earlier than when the **define-method-combination** form is executed, and possibly as late as the time that generic **functions** that use the **method combination** are executed.

**Examples:**

Most examples of the long form of **define-method-combination** also illustrate the use of the related **functions** that are provided as part of the declarative method combination facility.

;;; Examples of the short form of define-method-combination

```
(define-method-combination and :identity-with-one-argument t)
(defmethod func and ((x class1) y) ...)
```

;;; The equivalent of this example in the long form is:

```
(define-method-combination and
  (optional (order :most-specific-first))
  ((around (:around))
   (primary (and) :order order :required t))
  (let ((form (if (rest primary)
               ['(and ,(mapcar #'(lambda (method)
                              '(call-method ,method)
                              primary))
               '(call-method ,(first primary)))))
     (if around
        ['(call-method ,first around)
         ,(rest around)
         (make-method ,form))
        form)))
```

;;; Examples of the long form of define-method-combination

; The default method-combination technique
```
(define-method-combination standard ()
  ((around (:around))
   (before (:before))
   (primary () :required t)
   (after (:after)))
  (flet ((call-methods (methods)
```
define-method-combination

(mapcar #'(lambda (method)
 '((call-method ,method))
 methods)))
(let ((form (if (or before after (rest primary))
 '((multiple-value-prog1
 (progn ,0(call-methods before)
 (call-method ,(first primary)
 ,(rest primary)))
 ,0(call-methods (reverse after)))
 '((call-method ,(first primary))))
 (if around
 '((call-method ,(first around)
 ,0(rest around)
 (make-method ,form)))
 form))))

; A simple way to try several methods until one returns non-nil
(define-method-combination or ()
 ((methods (or)))
 '((or ,0(mapcar #'(lambda (method)
 '((call-method ,method))
 methods))))

; A more complete version of the preceding
(define-method-combination or
 (&optional (order ':most-specific-first))
 ((around (:around))
 (primary (or)))
 ;; Process the order argument
 (case order
 (:most-specific-first)
 (:most-specific-last (setq primary (reverse primary)))
 (otherwise (method-combination-error "~S is an invalid order."
 :most-specific-first and :most-specific-last are the possible values.
 order)))

;; Must have a primary method
(unless primary
 (method-combination-error "A primary method is required.")
)
;; Construct the form that calls the primary methods
(let ((form (if (rest primary)
 '((or ,0(mapcar #'(lambda (method)
 '((call-method ,method))
 primary)))
 '((call-method ,(first primary))))
 ;; Wrap the around methods around that form

Objects 7–83
define-method-combination

(if around
  '((call-method , (first around))
    ,@((rest around)
        (make-method , form)))
  form))

; The same thing, using the :order and :required keyword options
(define-method-combination or
  (optional (order ':most-specific-first))
  ((around (:around))
    (primary (or) :order order :required t))
  (let ((form (if (rest primary)
                 'or ,@(mapcar #'(lambda (method)
                                  '(call-method , method))
                             primary))
         (call-method , (first primary))))
    (if around
      'call-method , (first around)
      ,@((rest around)
          (make-method , form)))
    form))

; This short-form call is behaviorally identical to the preceding
(define-method-combination or :identity-with-one-argument t)

; Order methods by positive integer qualifiers
; :around methods are disallowed to keep the example small
(define-method-combination example-method-combination ()
  ((methods positive-integer-qualifier-p))
  'progn ,@(mapcar #'(lambda (method)
                        (call-method , method))
                  (stable-sort methods #'<
                    :key #'(lambda (method)
                             (first (method-qualifiers method))))))

(defun positive-integer-qualifier-p (method-qualifiers)
  (and (= (length method-qualifiers) 1)
       (typep (first method-qualifiers) '(integer 0 *)))))

;;; Example of the use of :arguments
(define-method-combination progn-with-lock ()
  ((methods ())))
  (:arguments object)
  '(unwind-protect
      (progn (lock (object-lock , object))

7–84 Programming Language—Common Lisp
Side Effects:
The compiler is not required to perform any compile-time side-effects.

Exceptional Situations:
Method combination types defined with the short form require exactly one qualifier per method. An error of type error is signaled if there are applicable methods with no qualifiers or with qualifiers that are not supported by the method combination type. At least one primary method must be applicable or an error of type error is signaled.

If an applicable method does not fall into any method group, the system signals an error of type error indicating that the method is invalid for the kind of method combination in use.

If the value of the :required option is true and the method group is empty (that is, no applicable methods match the qualifier patterns or satisfy the predicate), an error of type error is signaled.

If the :order option evaluates to a value other than :most-specific-first or :most-specific-last, an error of type error is signaled.

See Also:
call-method, call-next-method, documentation, method-qualifiers, method-combination-error, invalid-method-error, defgeneric, Section 7.6.6 (Method Selection and Combination), Section 7.6.6.4 (Built-in Method Combination Types), Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

Notes:
The :method-combination option of defgeneric is used to specify that a generic function should use a particular method combination type. The first argument to the :method-combination option is the name of a method combination type and the remaining arguments are options for that type.

find-method

Standard Generic Function

Syntax:
find-method generic-function method-qualifiers specializers &optional errorp
→ method
find-method

Method Signatures:

\[
\text{find-method (generic-function standard-generic-function)} \\
\text{method-qualifiers specializers \texttt{optional errorp}}
\]

Arguments and Values:

- \textit{generic-function}—a \textit{generic function}.
- \textit{method-qualifiers}—a list.
- \textit{specializers}—a list.
- \textit{errorp}—a \textit{generalized boolean}. The default is \texttt{true}.
- \textit{method}—a \textit{method object}, or nil.

Description:

The \textit{generic function} \textit{find-method} takes a \textit{generic function} and returns the \textit{method object} that agrees on \textit{qualifiers} and parameter \textit{specializers} with the \textit{method-qualifiers} and \textit{specializers} arguments of \textit{find-method}. \textit{Method-qualifiers} contains the method \textit{qualifiers} for the \textit{method}. The order of the method \textit{qualifiers} is significant. For a definition of agreement in this context, see Section 7.6.3 (Agreement on Parameter Specializers and Qualifiers).

The \textit{specializers} argument contains the parameter specializers for the \textit{method}. It must correspond in length to the number of required arguments of the \textit{generic function}, or an error is signaled. This means that to obtain the default \textit{method} on a given \textit{generic-function}, a list whose elements are the \textit{class} \texttt{t} must be given.

If there is no such \textit{method} and \textit{errorp} is \texttt{true}, \textit{find-method} signals an error. If there is no such \textit{method} and \textit{errorp} is \texttt{false}, \textit{find-method} returns nil.

Examples:

\begin{verbatim}
(defmethod some-operation ((a integer) (b float)) (list a b))
→ #<STANDARD-METHOD SOME-OPERATION (INTEGER FLOAT) 26723357>
(find-method #'some-operation '() (mapcar #'find-class '(integer float)))
→ #<STANDARD-METHOD SOME-OPERATION (INTEGER FLOAT) 26723357>
(find-method #'some-operation '() (mapcar #'find-class '(integer integer)))
▷ Error: No matching method
(find-method #'some-operation '() (mapcar #'find-class '(integer integer)) nil)
→ NIL
\end{verbatim}

Affected By:

- \texttt{add-method}, \texttt{defclass}, \texttt{defgeneric}, \texttt{defmethod}

Exceptional Situations:

If the \textit{specializers} argument does not correspond in length to the number of required arguments of
the generic-function, an an error of type error is signaled.

If there is no such method and errorp is true, find-method signals an error of type error.

See Also:
Section 7.6.3 (Agreement on Parameter Specializers and Qualifiers)

---

add-method

Syntax:

add-method generic-function method  →  generic-function

Method Signatures:

add-method (generic-function standard-generic-function) (method method)

Arguments and Values:

generic-function—a generic function object.

method—a method object.

Description:

The generic function add-method adds a method to a generic function.

If method agrees with an existing method of generic-function on parameter specializers and qualifiers, the existing method is replaced.

Exceptional Situations:

The lambda list of the method function of method must be congruent with the lambda list of generic-function, or an error of type error is signaled.

If method is a method object of another generic function, an error of type error is signaled.

See Also:

defmethod, defgeneric, find-method, remove-method, Section 7.6.3 (Agreement on Parameter Specializers and Qualifiers)
initialize-instance

Syntax:

initialize-instance instance &rest initargs &key &allow-other-keys → instance

Method Signatures:

initialize-instance (instance standard-object) &rest initargs

Arguments and Values:

instance—an object.

initargs—a defaulted initialization argument list.

Description:

Called by make-instance to initialize a newly created instance. The generic function is called with the new instance and the defaulted initialization argument list.

The system-supplied primary method on initialize-instance initializes the slots of the instance with values according to the initargs and the :initform forms of the slots. It does this by calling the generic function shared-initialize with the following arguments: the instance, t (this indicates that all slots for which no initialization arguments are provided should be initialized according to their :initform forms), and the initargs.

Programmers can define methods for initialize-instance to specify actions to be taken when an instance is initialized. If only after methods 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.

See Also:

shared-initialize, make-instance, slot-boundp, slot-makunbound, Section 7.1 (Object Creation and Initialization), Section 7.1.4 (Rules for Initialization Arguments), Section 7.1.2 (Declaring the Validity of Initialization Arguments)

class-name

Syntax:

class-name class → name

Method Signatures:

class-name (class class)
Arguments and Values:

- **class**—a class object.
- **name**—a symbol.

Description:

Returns the name of the given class.

See Also:

- find-class, Section 4.3 (Classes)

Notes:

If \( S \) is a symbol such that \( S = (\text{class-name } C) \) and \( C = (\text{find-class } S) \), then \( S \) is the proper name of \( C \). For further discussion, see Section 4.3 (Classes).

The name of an anonymous class is nil.

---

(setf class-name)  

**Standard Generic Function**

Syntax:

(setf class-name) new-value class  \rightarrow new-value

Method Signatures:

(setf class-name) new-value (class class)

Arguments and Values:

- **new-value**—a symbol.
- **class**—a class.

Description:

The generic function (setf class-name) sets the name of a class object.

See Also:

- find-class, proper name, Section 4.3 (Classes)
class-of

Function

Syntax:

\[
\text{class-of object} \rightarrow \text{class}
\]

Arguments and Values:

- \text{object} — an \text{object}.
- \text{class} — a \text{class} \text{object}.

Description:

Returns the \text{class} of which the \text{object} is a \text{direct instance}.

Examples:

\[
\text{(class-of 'fred)} \rightarrow #\text{<BUILT-IN-CLASS SYMBOL 610327300>}
\]
\[
\text{(class-of 2/3)} \rightarrow #\text{<BUILT-IN-CLASS RATIO 610326642>}
\]
\[
\text{(defclass book () ())} \rightarrow #\text{<STANDARD-CLASS BOOK 33424745>}
\]
\[
\text{(class-of (make-instance 'book))} \rightarrow #\text{<STANDARD-CLASS BOOK 33424745>}
\]
\[
\text{(defclass novel (book) ()}) \rightarrow #\text{<STANDARD-CLASS NOVEL 33424764>}
\]
\[
\text{(class-of (make-instance 'novel))} \rightarrow #\text{<STANDARD-CLASS NOVEL 33424764>}
\]
\[
\text{(defstruct kons kar kdr)} \rightarrow \text{KONS}
\]
\[
\text{(class-of (make-kons :kar 3 :kdr 4))} \rightarrow #\text{<STRUCTURE-CLASS KONS 250020317>}
\]

See Also:

\text{make-instance}, \text{type-of}

unbound-slot

Condition Type

Class Precedence List:

\text{unbound-slot}, \text{cell-error}, \text{error}, \text{serious-condition}, \text{condition}, \text{t}

Description:

The \text{object} having the unbound slot is initialized by the \text{:instance} initialization argument to \text{make-condition}, and is \text{accessed} by the \text{function unbound-slot-instance}.

The name of the cell (see \text{cell-error}) is the name of the slot.
unbound-slot-instance  

*Function*

**Syntax:**

```
unbound-slot-instance condition  →  instance
```

**Arguments and Values:**

- `condition`—a `condition` of type `unbound-slot`.
- `instance`—an `object`.

**Description:**

Returns the instance which had the unbound slot in the *situation* represented by the `condition`.

**See Also:**

- `cell-error-name`, `unbound-slot-object`, Section 9.1 (Condition System Concepts)
Programming Language—Common Lisp

8. Structures
defstruct

Macro

Syntax:

```
defstruct name-and-options [documentation] { [slot-description}* → structure-name

name-and-options::=structure-name | (structure-name [options ])

options::|= conc-name-option |
{ |constructor-option}* |
| copier-option |
| include-option |
| initial-offset-option |
| named-option |
| predicate-option |
| printer-option |
| type-option

conc-name-option::=:conc-name | (:conc-name) | (:conc-name conc-name)

constructor-option::=:constructor |
{ (:constructor) |
{ :constructor constructor-name) |
{ :constructor constructor-name constructor-arglist

copier-option::=:copier | (:copier) | (:copier copier-name)

predicate-option::=:predicate | (:predicate) | (:predicate predicate-name)

include-option::=:include included-structure-name { [slot-description}*]

printer-option::= | print-object-option | print-function-option

print-object-option::=:print-object printer-name | (:print-object)

print-function-option::=:print-function printer-name | (:print-function)

type-option::=:type type

named-option::=:named

initial-offset-option::=:initial-offset initial-offset
```
defstruct

\[ \text{slot-description} ::= \text{slot-name} | (\text{slot-name} \ [\text{slot-initform} \ [\text{slot-option}]]) \]

\[ \text{slot-option} ::= \text{:type} \ \text{slot-type} | \text{:read-only} \ \text{slot-read-only-p} \]

Arguments and Values:
- \text{conc-name}—a string designator.
- \text{constructor-arglist}—a boa lambda list.
- \text{constructor-name}—a symbol.
- \text{copier-name}—a symbol.
- \text{included-structure-name}—an already-defined structure name. Note that a derived type is not permissible, even if it would expand into a structure name.
- \text{initial-offset}—a non-negative integer.
- \text{predicate-name}—a symbol.
- \text{printer-name}—a function name or a lambda expression.
- \text{slot-name}—a symbol.
- \text{slot-initform}—a form.
- \text{slot-read-only-p}—a generalized boolean.
- \text{structure-name}—a symbol.
- \text{type}—one of the type specifiers list, vector, or (vector size), or some other type specifier defined by the implementation to be appropriate.
- \text{documentation}—a string; not evaluated.

Description:
\text{defstruct} defines a structured \text{type}, named \text{structure-type}, with named slots as specified by the \text{slot-options}.

\text{defstruct} defines readers for the slots and arranges for \text{setf} to work properly on such reader functions. Also, unless overridden, it defines a predicate named \text{name-p}, defines a constructor function named \text{make-\text{constructor-name}}, and defines a copier function named \text{copy-\text{constructor-name}}. All names of automatically created functions might automatically be declared \text{inline} (at the discretion of the implementation).

If \text{documentation} is supplied, it is attached to \text{structure-name} as a \text{documentation string} of kind \text{structure}, and unless \text{:type} is used, the \text{documentation} is also attached to \text{structure-name} as a
defstruct
documentation string of kind type and as a documentation string to the class object for the class named structure-name.

defstruct defines a constructor function that is used to create instances of the structure created by defstruct. The default name is make-structure-name. A different name can be supplied by giving the name as the argument to the constructor option. nil indicates that no constructor function will be created.

After a new structure type has been defined, instances of that type normally can be created by using the constructor function for the type. A call to a constructor function is of the following form:

(constructor-function-name
 slot-keyword-1 form-1
 slot-keyword-2 form-2
 . . .
)

The arguments to the constructor function are all keyword arguments. Each slot keyword argument must be a keyword whose name corresponds to the name of a structure slot. All the keywords and forms are evaluated. If a slot is not initialized in this way, it is initialized by evaluating slot-initform in the slot description at the time the constructor function is called. If no slot-initform is supplied, the consequences are undefined if an attempt is later made to read the slot’s value before a value is explicitly assigned.

Each slot-initform supplied for a defstruct component, when used by the constructor function for an otherwise unsupplied component, is re-evaluated on every call to the constructor function. The slot-initform is not evaluated unless it is needed in the creation of a particular structure instance. If it is never needed, there can be no type-mismatch error, even if the type of the slot is specified; no warning should be issued in this case. For example, in the following sequence, only the last call is an error.

(defstruct person (name 007 :type string))
(make-person :name "James")
(make-person)

It is as if the slot-initforms were used as initialization forms for the keyword parameters of the constructor function.

The symbols which name the slots must not be used by the implementation as the names for the lambda variables in the constructor function, since one or more of those symbols might have been proclaimed special or might be defined as the name of a constant variable. The slot default init forms are evaluated in the lexical environment in which the defstruct form itself appears and in the dynamic environment in which the call to the constructor function appears.

For example, if the form (gensym) were used as an initialization form, either in the constructor-function call or as the default initialization form in defstruct, then every call to the constructor function would call gensym once to generate a new symbol.
defstruct

Each slot-description in defstruct can specify zero or more slot-options. A slot-option consists of a pair of a keyword and a value (which is not a form to be evaluated, but the value itself). For example:

(defstruct ship
  (x-position 0.0 :type short-float)
  (y-position 0.0 :type short-float)
  (x-velocity 0.0 :type short-float)
  (y-velocity 0.0 :type short-float)
  (mass *default-ship-mass* :type short-float :read-only t))

This specifies that each slot always contains a short float, and that the last slot cannot be altered once a ship is constructed.

The available slot-options are:

:type type

This specifies that the contents of the slot is always of type type. This is entirely analogous to the declaration of a variable or function; it effectively declares the result type of the reader function. It is implementation-dependent whether the type is checked when initializing a slot or when assigning to it. Type is not evaluated; it must be a valid type specifier.

:read-only x

When x is true, this specifies that this slot cannot be altered; it will always contain the value supplied at construction time. setf will not accept the reader function for this slot. If x is false, this slot-option has no effect. X is not evaluated.

When this option is false or unsupplied, it is implementation-dependent whether the ability to write the slot is implemented by a setf function or a setf expander.

The following keyword options are available for use with defstruct. A defstruct option can be either a keyword or a list of a keyword and arguments for that keyword; specifying the keyword by itself is equivalent to specifying a list consisting of the keyword and no arguments. The syntax for defstruct options differs from the pair syntax used for slot-options. No part of any of these options is evaluated.

:conc-name

This provides for automatic prefixing of names of reader (or access) functions. The default behavior is to begin the names of all the reader functions of a structure with the name of the structure followed by a hyphen.

:conc-name supplies an alternate prefix to be used. If a hyphen is to be used as a separator, it must be supplied as part of the prefix. If :conc-name is nil or no argument is supplied, then no prefix is used; then the names of the reader functions are the same.
as the slot names. If a non-nil prefix is given, the name of the reader function for each slot is constructed by concatenating that prefix and the name of the slot, and interning the resulting symbol in the package that is current at the time the defstruct form is expanded.

Note that no matter what is supplied for :conc-name, slot keywords that match the slot names with no prefix attached are used with a constructor function. The reader function name is used in conjunction with setf. Here is an example:

```lisp
(defstruct (door (:conc-name dr-)) knob-color width material)
→ DOOR
(setq my-door (make-door :knob-color 'red :width 5.0))
→ #S(DOOR :KNOB-COLOR RED :WIDTH 5.0 :MATERIAL NIL)
(dr-width my-door) → 5.0
(setf (dr-width my-door) 43.7) → 43.7
(dr-width my-door) → 43.7
```

Whether or not the :conc-name option is explicitly supplied, the following rule governs name conflicts of generated reader (or accessor) names: For any structure type $S_1$ having a reader function named $R$ for a slot named $X_1$ that is inherited by another structure type $S_2$ that would have a reader function with the same name $R$ for a slot named $X_2$, no definition for $R$ is generated by the definition of $S_2$; instead, the definition of $R$ is inherited from the definition of $S_1$. (In such a case, if $X_1$ and $X_2$ are different slots, the implementation might signal a style warning.)

:constructor

This option takes zero, one, or two arguments. If at least one argument is supplied and the first argument is not nil, then that argument is a symbol which specifies the name of the constructor function. If the argument is not supplied (or if the option itself is not supplied), the name of the constructor is produced by concatenating the string "MAKE-" and the name of the structure, interning the name in whatever package is current at the time defstruct is expanded. If the argument is provided and is nil, no constructor function is defined.

If :constructor is given as (:constructor name arglist), then instead of making a keyword driven constructor function, defstruct defines a “positional” constructor function, taking arguments whose meaning is determined by the argument’s position and possibly by keywords. Arglist is used to describe what the arguments to the constructor will be. In the simplest case something like (:constructor make-foo (a b c)) defines make-foo to be a three-argument constructor function whose arguments are used to initialize the slots named a, b, and c.

Because a constructor of this type operates “By Order of Arguments,” it is sometimes known as a “boa constructor.”

For information on how the arglist for a “boa constructor” is processed, see Section 3.4.6 (Boa Lambda Lists).
defstruct

It is permissible to use the :constructor option more than once, so that you can define several different constructor functions, each taking different parameters.

defstruct creates the default-named keyword constructor function only if no explicit :constructor options are specified, or if the :constructor option is specified without a name argument.

(:constructor nil) is meaningful only when there are no other :constructor options specified. It prevents defstruct from generating any constructors at all.

Otherwise, defstruct creates a constructor function corresponding to each supplied :constructor option. It is permissible to specify multiple keyword constructor functions as well as multiple “boa constructors”.

:copier

This option takes one argument, a symbol, which specifies the name of the copier function. If the argument is not provided or if the option itself is not provided, the name of the copier is produced by concatenating the string “COPY-” and the name of the structure, interning the name in whatever package is current at the time defstruct is expanded. If the argument is provided and is nil, no copier function is defined.

The automatically defined copier function is a function of one argument, which must be of the structure type being defined. The copier function creates a fresh structure that has the same type as its argument, and that has the same component values as the original structure; that is, the component values are not copied recursively. If the defstruct :type option was not used, the following equivalence applies:

\[(\text{copier-name } x) = (\text{copy-structure } (\text{the structure-name } x))\]

:include

This option is used for building a new structure definition as an extension of another structure definition. For example:

(defstruct person name age sex)

To make a new structure to represent an astronaut that has the attributes of name, age, and sex, and functions that operate on person structures, astronaut is defined with :include as follows:

(defstruct (astronaut (:include person)
(:conc-name astro-))
  helmet-size
  (favorite-beverage 'tang))

:include causes the structure being defined to have the same slots as the included structure. This is done in such a way that the reader functions for the included structure also work on the structure being defined. In this example, an astronaut therefore has
five slots: the three defined in person and the two defined in astronaut itself. The reader functions defined by the person structure can be applied to instances of the astronaut structure, and they work correctly. Moreover, astronaut has its own reader functions for components defined by the person structure. The following examples illustrate the use of astronaut structures:

\[
\text{(setq } x \text{ (make-astronaut } \text{:name } \text{'buzz }
\text{ :age } 45.
\text{ :sex } t
\text{ :helmet-size } 17.5))
\]

(person-name \(x\)) \rightarrow BUZZ
(astro-name \(x\)) \rightarrow BUZZ
(astro-favorite-beverage \(x\)) \rightarrow TANG

\[
\text{(reduce } #'+ \text{ astros :key } \text{#'person-age) } \text{; obtains the total of the ages }
\text{; of the possibly empty }
\text{; sequence of astros}
\]

The difference between the reader functions person-name and astro-name is that person-name can be correctly applied to any person, including an astronaut, while astro-name can be correctly applied only to an astronaut. An implementation might check for incorrect use of reader functions.

At most one :include can be supplied in a single defstruct. The argument to :include is required and must be the name of some previously defined structure. If the structure being defined has no :type option, then the included structure must also have had no :type option supplied for it. If the structure being defined has a :type option, then the included structure must have been declared with a :type option specifying the same representation type.

If no :type option is involved, then the structure name of the including structure definition becomes the name of a data type, and therefore a valid type specifier recognizable by typep; it becomes a subtype of the included structure. In the above example, astronaut is a subtype of person; hence

\[
\text{(typep (make-astronaut) } \text{'person) } \rightarrow \text{true}
\]

indicating that all operations on persons also work on astronauts.

The structure using :include can specify default values or slot-options for the included slots different from those the included structure specifies, by giving the :include option as:

\[
\text{(:include included-structure-name } \{\text{slot-description}\}^*)
\]

Each slot-description must have a slot-name that is the same as that of some slot in the included structure. If a slot-description has no slot-initform, then in the new structure the
defstruct

slot has no initial value. Otherwise its initial value form is replaced by the slot-initform in the slot-description. A normally writable slot can be made read-only. If a slot is read-only in the included structure, then it must also be so in the including structure. If a type is supplied for a slot, it must be a subtype of the type specified in the included structure.

For example, if the default age for an astronaut is 45, then

(defstruct (astronaut (:include person (age 45)))
  (helmet-size)
  (favorite-beverage 'tang))

If :include is used with the :type option, then the effect is first to skip over as many representation elements as needed to represent the included structure, then to skip over any additional elements supplied by the :initial-offset option, and then to begin allocation of elements from that point. For example:

(defstruct (binop (:type list) :named (:initial-offset 2))
  (operand 1)
  (operand-2) → BINOP
(defstruct (annotated-binop (:type list) (:initial-offset 3)
                            (:include binop))
  (commutative associative identity) → ANNOTATED-BINOP
(make-annotated-binop :operator '*
                       :operand-1 'x
                       :operand-2 5
                       :commutative t
                       :associative t
                       :identity 1)
→ (NIL NIL BINOP * X 5 NIL NIL NIL T T 1)

The first two nil elements stem from the :initial-offset of 2 in the definition of binop. The next four elements contain the structure name and three slots for binop. The next three nil elements stem from the :initial-offset of 3 in the definition of annotated-binop. The last three list elements contain the additional slots for an annotated-binop.

:initial-offset

:initial-offset instructs defstruct to skip over a certain number of slots before it starts allocating the slots described in the body. This option's argument is the number of slots defstruct should skip. :initial-offset can be used only if :type is also supplied.

:initial-offset allows slots to be allocated beginning at a representational element other than the first. For example, the form

(defstruct (binop (:type list) (:initial-offset 2))

8–8 Programming Language—Common Lisp
defstruct

(operator '?: type symbol)
operand-1
operand-2) → BINOP

would result in the following behavior for make-binop:

(make-binop :operator '+ :operand-1 'x :operand-2 5)
→ (NIL NIL + X 5)
(make-binop :operand-2 4 :operator '*)
→ (NIL NIL * NIL 4)

The selector functions binop-operator, binop-operand-1, and binop-operand-2 would be essentially equivalent to third, fourth, and fifth, respectively. Similarly, the form

(defstruct (binop (:type list) :named (:initial-offset 2))
(operator '? :type symbol)
operand-1
operand-2) → BINOP

would result in the following behavior for make-binop:

(make-binop :operator '+ :operand-1 'x :operand-2 5) → (NIL NIL BINOP + X 5)
(make-binop :operand-2 4 :operator '*) → (NIL NIL BINOP * NIL 4)

The first two nil elements stem from the :initial-offset of 2 in the definition of binop. The next four elements contain the structure name and three slots for binop.

:named

:named specifies that the structure is named. If no :type is supplied, then the structure is always named.

For example:

(defstruct (binop (:type list))
(operator '? :type symbol)
operand-1
operand-2) → BINOP

This defines a constructor function make-binop and three selector functions, namely binop-operator, binop-operand-1, and binop-operand-2. (It does not, however, define a predicate binop-p, for reasons explained below.)

The effect of make-binop is simply to construct a list of length three:

(make-binop :operator '+ :operand-1 'x :operand-2 5) → (+ X 5)
(make-binop :operand-2 4 :operator '*) → (* NIL 4)

It is just like the function list except that it takes keyword arguments and performs slot defaulting appropriate to the binop conceptual data type. Similarly, the selector functions
defstruct

binop-operator, binop-operand-1, and binop-operand-2 are essentially equivalent to car, cadr, and caddr, respectively. They might not be completely equivalent because, for example, an implementation would be justified in adding error-checking code to ensure that the argument to each selector function is a length-3 list.

binop is a conceptual data type in that it is not made a part of the Common Lisp type system. typep does not recognize binop as a type specifier, and type-of returns list when given a binop structure. There is no way to distinguish a data structure constructed by make-binop from any other list that happens to have the correct structure.

There is not any way to recover the structure name binop from a structure created by make-binop. This can only be done if the structure is named. A named structure has the property that, given an instance of the structure, the structure name (that names the type) can be reliably recovered. For structures defined with no :type option, the structure name actually becomes part of the Common Lisp data-type system. type-of, when applied to such a structure, returns the structure name as the type of the object; typep recognizes the structure name as a valid type specifier.

For structures defined with a :type option, type-of returns a type specifier such as list or (vector t), depending on the type supplied to the :type option. The structure name does not become a valid type specifier. However, if the :named option is also supplied, then the first component of the structure (as created by a defstruct constructor function) always contains the structure name. This allows the structure name to be recovered from an instance of the structure and allows a reasonable predicate for the conceptual type to be defined: the automatically defined name-p predicate for the structure operates by first checking that its argument is of the proper type (list, (vector t), or whatever) and then checking whether the first component contains the appropriate type name.

Consider the binop example shown above, modified only to include the :named option:

(defstruct (binop (:type list) :named)  
  (operator '? :type symbol)  
  operand-1  
  operand-2) → BINOP

As before, this defines a constructor function make-binop and three selector functions binop-operator, binop-operand-1, and binop-operand-2. It also defines a predicate binop-p. The effect of make-binop is now to construct a list of length four:

(make-binop :operator '+ :operand-1 'x :operand-2 5) → (BINOP + X 5)  
(make-binop :operand-2 4 :operator '*)) → (BINOP * NIL 4)

The structure has the same layout as before except that the structure name binop is included as the first list element. The selector functions binop-operator, binop-operand-1, and binop-operand-2 are essentially equivalent to cadr, caddr, and cadddr, respectively. The predicate binop-p is more or less equivalent to this definition:

(defun binop-p (x)
defstruct

(and (consp x) (eq (car x) 'binop)) → BINOP-P

The name binop is still not a valid type specifier recognizable to typep, but at least there is a way of distinguishing binop structures from other similarly defined structures.

:predicate

This option takes one argument, which specifies the name of the type predicate. If the argument is not supplied or if the option itself is not supplied, the name of the predicate is made by concatenating the name of the structure to the string "-P", interning the name in whatever package is current at the time defstruct is expanded. If the argument is provided and is nil, no predicate is defined. A predicate can be defined only if the structure is named; if :type is supplied and :named is not supplied, then :predicate must either be unsupplied or have the value nil.

:print-function, :print-object

The :print-function and :print-object options specify that a print-object method for structures of type structure-name should be generated. These options are not synonyms, but do perform a similar service; the choice of which option (:print-function or :print-object) is used affects how the function named printer-name is called. Only one of these options may be used, and these options may be used only if :type is not supplied.

If the :print-function option is used, then when a structure of type structure-name is to be printed, the designated printer function is called on three arguments:

1. the structure to be printed (a generalized instance of structure-name).
2. a stream to print to.
3. an integer indicating the current depth. The magnitude of this integer may vary between implementations; however, it can reliably be compared against *print-level* to determine whether depth abbreviation is appropriate.

Specifying (:print-function printer-name) is approximately equivalent to specifying:

(defun print-object ((object structure-name) stream) (funcall (function printer-name) object stream ⟨⟨current-print-depth⟩⟩))

where the ⟨⟨current-print-depth⟩⟩ represents the printer’s belief of how deep it is currently printing. It is implementation-dependent whether ⟨⟨current-print-depth⟩⟩ is always 0 and *print-level*, if non-nil, is re-bound to successively smaller values as printing descends recursively, or whether current-print-depth varies in value as printing descends recursively and *print-level* remains constant during the same traversal.

If the :print-object option is used, then when a structure of type structure-name is to be printed, the designated printer function is called on two arguments:
defstruct

- the structure to be printed.
- the stream to print to.

Specifying (:print-object printer-name) is equivalent to specifying:

(defmethod print-object ((object structure-name) stream)
  (funcall (function printer-name) object stream))

If no :type option is supplied, and if either a :print-function or a :print-object option is supplied, and if no printer-name is supplied, then a print-object method specialized for structure-name is generated that calls a function that implements the default printing behavior for structures using #S notation; see Section 22.1.3.12 (Printing Structures).

If neither a :print-function nor a :print-object option is supplied, then defstruct does not generate a print-object method specialized for structure-name and some default behavior is inherited either from a structure named in an :include option or from the default behavior for printing structures; see the function print-object and Section 22.1.3.12 (Printing Structures).

When *print-circle* is true, a user-defined print function can print objects to the supplied stream using write, prin1, princ, or format and expect circularities to be detected and printed using the #n# syntax. This applies to methods on print-object in addition to :print-function options. If a user-defined print function prints to a stream other than the one that was supplied, then circularity detection starts over for that stream. See the variable *print-circle*.

:type

:type explicitly specifies the representation to be used for the structure. Its argument must be one of these types:

vector

This produces the same result as specifying (vector t). The structure is represented as a general vector, storing components as vector elements. The first component is vector element 1 if the structure is :named, and element 0 otherwise.

(vector element-type)

The structure is represented as a (possibly specialized) vector, storing components as vector elements. Every component must be of a type that can be stored in a vector of the type specified. The first component is vector element 1 if the structure is :named, and element 0 otherwise. The structure can be :named only if the type symbol is a subtype of the supplied element-type.

list
The structure is represented as a list. The first component is the cadr if the structure is :named, and the car if it is not :named.

Specifying this option has the effect of forcing a specific representation and of forcing the components to be stored in the order specified in defstruct in corresponding successive elements of the specified representation. It also prevents the structure name from becoming a valid type specifier recognizable by typep.

For example:

(defstruct (quux (:type list) :named) x y)

should make a constructor that builds a list exactly like the one that list produces, with quux as its car.

If this type is defined:

(deftype quux () '(satisfies quux-p))

then this form

(typep (make-quux) 'quux)

should return precisely what this one does

(typep (list 'quux nil nil) 'quux)

If :type is not supplied, the structure is represented as an object of type structure-object.

defstruct without a :type option defines a class with the structure name as its name. The metaclass of structure instances is structure-class.

The consequences of redefining a defstruct structure are undefined.

In the case where no defstruct options have been supplied, the following functions are automatically defined to operate on instances of the new structure:

**Predicate**

A predicate with the name structure-name-p is defined to test membership in the structure type. The predicate (structure-name-p object) is true if an object is of this type; otherwise it is false. typep can also be used with the name of the new type to test whether an object belongs to the type. Such a function call has the form (typep object 'structure-name).

**Component reader functions**

Reader functions are defined to read the components of the structure. For each slot name, there is a corresponding reader function with the name structure-name-slot-name.
defstruct

This function reads the contents of that slot. Each reader function takes one argument, which is an instance of the structure type. setf can be used with any of these reader functions to alter the slot contents.

Constructor function

A constructor function with the name make-structure-name is defined. This function creates and returns new instances of the structure type.

Copier function

A copier function with the name copy-structure-name is defined. The copier function takes an object of the structure type and creates a new object of the same type that is a copy of the first. The copier function creates a new structure with the same component entries as the original. Corresponding components of the two structure instances are eql.

If a defstruct form appears as a top level form, the compiler must make the structure type name recognized as a valid type name in subsequent declarations (as for deftype) and make the structure slot readers known to setf. In addition, the compiler must save enough information about the structure type so that further defstruct definitions can use :include in a subsequent defstruct in the same file to refer to the structure type name. The functions which defstruct generates are not defined in the compile time environment, although the compiler may save enough information about the functions to code subsequent calls inline. The #S reader macro might or might not recognize the newly defined structure type name at compile time.

Examples:

An example of a structure definition follows:

(defstruct ship
  x-position
  y-position
  x-velocity
  y-velocity
  mass)

This declares that every ship is an object with five named components. The evaluation of this form does the following:

1. It defines ship-x-position to be a function of one argument, a ship, that returns the x-position of the ship; ship-y-position and the other components are given similar function definitions. These functions are called the access functions, as they are used to access elements of the structure.

2. ship becomes the name of a type of which instances of ships are elements. ship becomes acceptable to typep, for example: (typep x 'ship) is true if x is a ship and false if x is any object other than a ship.
3. A function named ship-p of one argument is defined; it is a predicate that is true if its argument is a ship and is false otherwise.

4. A function called make-ship is defined that, when invoked, creates a data structure with five components, suitable for use with the access functions. Thus executing

```lisp
(setq ship2 (make-ship))
```
sets ship2 to a newly created ship object. One can supply the initial values of any desired component in the call to make-ship by using keyword arguments in this way:

```lisp
(setq ship2 (make-ship :mass *default-ship-mass* :x-position 0 :y-position 0))
```

This constructs a new ship and initializes three of its components. This function is called the “constructor function” because it constructs a new structure.

5. A function called copy-ship of one argument is defined that, when given a ship object, creates a new ship object that is a copy of the given one. This function is called the “copier function.”

setf can be used to alter the components of a ship:

```lisp
(setf (ship-x-position ship2) 100)
```

This alters the x-position of ship2 to be 100. This works because defstruct behaves as if it generates an appropriate defsetf for each access function.

;;; Example 1
;;; define town structure type
;;; area, watertowers, firetrucks, population, elevation are its components
;;;
(defstruct town
  area
  watertowers
  (firetrucks 1 :type fixnum) ; an initialized slot
  population
  (elevation 5128 :read-only t)) ; a slot that can’t be changed
→ TOWN

; create a town instance
(setq town1 (make-town :area 0 :watertowers 0)) → #S(TOWN...)
; town’s predicate recognizes the new instance
(town-p town1) → true
; new town’s area is as specified by make-town
(town-area town1) → 0
; new town’s elevation has initial value
defstruct

(town-elevation town1) → 5128
;setf recognizes reader function
(setf (town-population town1) 99) → 99
(town-population town1) → 99
;copier function makes a copy of town1
(setq town2 (copy-town town1)) → #S(TOWN...)
(= (town-population town1) (town-population town2)) → true
;since elevation is a read-only slot, its value can be set only
;when the structure is created
(setq town3 (make-town :area 0 :watertowers 3 :elevation 1200)) → #S(TOWN...)
;;;;
;;;; Example 2
;;;; define clown structure type
;;;; this structure uses a nonstandard prefix
;;;;
(defstruct (clown (:conc-name bozo-))
     (nose-color 'red)
     frizzy-hair-p polkadots)
(setq funny-clown (make-clown)) → #S(CLOWN)
;use non-default reader name
(bozo-nose-color funny-clown) → RED
(defstruct (klown (:constructor make-up-klown) ;similar def using other
                  (:copier clone-klown) ;customizing keywords
                  (:predicate is-a-bozo-p))
     nose-color frizzy-hair-p polkadots) → klown
;custom constructor now exists
(fbfoundp 'make-up-klown) → true
;;;;
;;;; Example 3
;;;; define a vehicle structure type
;;;; then define a truck structure type that includes
;;;; the vehicle structure
;;;;
(defstruct vehicle name year (diesel t :read-only t)) → VEHICLE
(defstruct (truck (:include vehicle (year 79)))
     load-limit
     (axles 6)) → TRUCK
(setq x (make-truck :name 'mac :diesel t :load-limit 17)) → #S(TRUCK...)
;vehicle readers work on trucks
(vehicle-name x)
→ MAC
;default taken from :include clause

8–16 Programming Language—Common Lisp
(defstruct vehicle-year x)
→ 79

(defstruct (pickup (:include truck)) ; pickup type includes truck
  camper long-bed four-wheel-drive) → PICKUP

(setq x (make-pickup :name 'king :long-bed t)) → #(PICKUP...)

;; include default inherited

(pickup-year x) → 79

;;;; Example 4
;;;; use of BOA constructors

(defun dfs-boa ; BOA constructors
  (:constructor make-dfs-boa (a b c))
  (:constructor create-dfs-boa
    (a &optional b (c 'cc) &rest d &aux e (f 'ff)))
  a b c d e f) → DFS-BOA

(setq x (make-dfs-boa 1 2 3)) → #(DFS-BOA...)

(dfs-boa-a x) → 1

;; a and b set, c and f defaulted

(setq x (create-dfs-boa 1 2)) → #(DFS-BOA...)

(dfs-boa-b x) → 2

(eq (dfs-boa-c x) 'cc) → true

;; a, b, and c set, and the rest are collected into d

(setq x (create-dfs-boa 1 2 3 4 5 6)) → #(DFS-BOA...)

(dfs-boa-d x) → (4 5 6)

Exceptional Situations:
If any two slot names (whether present directly or inherited by the :include option) are the same under string=, defstruct should signal an error of type program-error.

The consequences are undefined if the included-structure-name does not name a structure type.

See Also:
documentation, print-object, setf, subtypep, type-of, typep, Section 3.2 (Compilation)

Notes:
The printer-name should observe the values of such printer-control variables as *print-escape*.

The restriction against issuing a warning for type mismatches between a slot-initform and the corresponding slot’s :type option is necessary because a slot-initform must be specified in order to specify slot options; in some cases, no suitable default may exist.

The mechanism by which defstruct arranges for slot accessors to be usable with setf is implementation-dependent; for example, it may use self functions, self expanders, or some other implementation-dependent mechanism known to that implementation’s code for setf.
copy-structure  

Function

Syntax:

\[
\text{copy-structure} \ structure \rightarrow \ copy
\]

Arguments and Values:

- \text{structure}—a \text{structure}.
- \text{copy}—a copy of the \text{structure}.

Description:

Returns a \text{copy} of the \text{structure}.

Only the \text{structure} itself is copied; not the values of the slots.

See Also:

the \text{:copier} option to \text{defstruct}

Notes:

The \text{copy} is the \text{same} as the given \text{structure} under \text{equalp}, but not under \text{equal}.
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

Figure 9–1 lists the standardized condition types. Additional condition types can be defined by using `define-condition`.

<table>
<thead>
<tr>
<th>condition type</th>
<th>subtypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic-error</td>
<td>floating-point-overflow</td>
</tr>
<tr>
<td>cell-error</td>
<td>floating-point-underflow</td>
</tr>
<tr>
<td>condition</td>
<td>package-error</td>
</tr>
<tr>
<td>control-error</td>
<td>parse-error</td>
</tr>
<tr>
<td>division-by-zero</td>
<td>print-not-readable</td>
</tr>
<tr>
<td>end-of-file</td>
<td>program-error</td>
</tr>
<tr>
<td>error</td>
<td>reader-error</td>
</tr>
<tr>
<td>file-error</td>
<td>serious-condition</td>
</tr>
<tr>
<td>floating-point-inexact</td>
<td>simple-condition</td>
</tr>
<tr>
<td>floating-point-invalid-operation</td>
<td>simple-error</td>
</tr>
<tr>
<td></td>
<td>warning</td>
</tr>
</tbody>
</table>

Figure 9–1. Standardized Condition Types

All condition types are subtypes of `type condition`. That is,

\[
\text{(typep} \, \text{c} \, \text{'condition}) \Rightarrow \text{true}
\]

if and only if \( \text{c} \) 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.

Figure 9–2 shows operators that define condition types and creating conditions.

<table>
<thead>
<tr>
<th>operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>define-condition</code></td>
</tr>
<tr>
<td><code>make-condition</code></td>
</tr>
</tbody>
</table>

Figure 9–2. Operators that define and create conditions.
Figure 9–3 shows operators that read the value of condition slots.

<table>
<thead>
<tr>
<th>arithmetic-error-operands</th>
<th>simple-condition-format-arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic-error-operation</td>
<td>simple-condition-format-control</td>
</tr>
<tr>
<td>cell-error-name</td>
<td>stream-error-stream</td>
</tr>
<tr>
<td>file-error-pathname</td>
<td>type-error-datum</td>
</tr>
<tr>
<td>package-error-package</td>
<td>type-error-expected-type</td>
</tr>
<tr>
<td>print-not-readable-object</td>
<td>unbound-slot-instance</td>
</tr>
</tbody>
</table>

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 defaulted-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 sentence, 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.\nIt 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 $H_1$ is more recent than every handler in a set $H_2$ if the handlers in $H_2$ were active when the handlers in $H_1$ were established.

2. Let $h_1$ and $h_2$ be two applicable active handlers established by the same form. Then $h_1$ is more recent than $h_2$ if $h_1$ was defined to the left of $h_2$ 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.

Figure 9–4 shows operators relating to the handling of conditions.

<table>
<thead>
<tr>
<th>handler-bind</th>
<th>handler-case</th>
<th>ignore-errors</th>
</tr>
</thead>
</table>

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.

Figure 9–5 shows defined names relating to the signaling of conditions.

<table>
<thead>
<tr>
<th><em>break-on-signals</em></th>
<th>error</th>
<th>warn</th>
</tr>
</thead>
<tbody>
<tr>
<td>cerror</td>
<td></td>
<td></td>
</tr>
<tr>
<td>signal</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 $R_1$ is more recent than every restart in a set $R_2$ if the restarts in $R_2$ were active when the restarts in $R_1$ were established.

2. Let $r_1$ and $r_2$ be two active restarts with the same name established by the same form. Then $r_1$ is more recent than $r_2$ if $r_1$ was defined to the left of $r_2$ in the form that established them.

9–8 Programming Language—Common Lisp
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.

Figure 9–6 shows defined names relating to restarts.

<table>
<thead>
<tr>
<th>restart</th>
<th>function</th>
<th>argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>invoke-restart-interactively</td>
<td>store-value</td>
</tr>
<tr>
<td>compute-restarts</td>
<td>muffle-warning</td>
<td>use-value</td>
</tr>
<tr>
<td>continue</td>
<td>restart-bind</td>
<td>with-simple-restart</td>
</tr>
<tr>
<td>find-restart</td>
<td>restart-case</td>
<td></td>
</tr>
<tr>
<td>invoke-restart</td>
<td>restart-name</td>
<td></td>
</tr>
</tbody>
</table>

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 association 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. Figure 9–7 shows operators relating to assertions.

```
assert ccase
check-type ctypecase
ecase 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.
condition

Class Precedence List:
condition, t

Description:
All types of conditions, whether error or non-error, must inherit from this type.

No additional subtype relationships among the specified subtypes of type condition are allowed, except when explicitly mentioned in the text; however implementations are permitted to introduce additional types and one of these types can be a subtype of any number of the subtypes of type condition.

Whether a user-defined condition type has slots that are accessible by with-slots is implementation-dependent. Furthermore, even in an implementation in which user-defined condition types would have slots, it is implementation-dependent whether any condition types defined in this document have such slots or, if they do, what their names might be; only the reader functions documented by this specification may be relied upon by portable code.

Conforming code must observe the following restrictions related to conditions:

- define-condition, not defclass, must be used to define new condition types.
- make-condition, not make-instance, must be used to create condition objects explicitly.
- The :report option of define-condition, not defmethod for print-object, must be used to define a condition reporter.
- slot-value, slot-boundp, slot-makunbound, and with-slots must not be used on condition objects. Instead, the appropriate accessor functions (defined by define-condition) should be used.

warning

Class Precedence List:
warning, condition, t

Description:
The type warning consists of all types of warnings.
See Also:
  style-warning

**style-warning**

*Condition Type*

**Class Precedence List:**
  style-warning, warning, condition, t

**Description:**
The *type* `style-warning` includes those *conditions* that represent *situations* involving *code* that is *conforming code* but that is nevertheless considered to be faulty or substandard.

**See Also:**
muffle-warning

**Notes:**
An *implementation* might signal such a *condition* if it encounters *code* that uses deprecated features or that appears unaesthetic or inefficient.

An ‘unused variable’ warning must be of *type* `style-warning`.

In general, the question of whether *code* is faulty or substandard is a subjective decision to be made by the facility processing that *code*. The intent is that whenever such a facility wishes to complain about *code* on such subjective grounds, it should use this *condition type* so that any clients who wish to redirect or muffle superfluous warnings can do so without risking that they will be redirecting or muffling other, more serious warnings.

**serious-condition**

*Condition Type*

**Class Precedence List:**
  serious-condition, condition, t

**Description:**
All *conditions* serious enough to require interactive intervention if not handled should inherit from the *type* `serious-condition`. This condition type is provided primarily so that it may be included as a *superclass* of other *condition types*; it is not intended to be signaled directly.

**Notes:**
Signaling a *serious condition* does not itself force entry into the debugger. However, except in the unusual situation where the programmer can assure that no harm will come from failing to *handle*
a serious condition, such a condition is usually signaled with error rather than signal in order to assure that the program does not continue without handling the condition. (And conversely, it is conventional to use signal rather than error to signal conditions which are not serious conditions, since normally the failure to handle a non-serious condition is not reason enough for the debugger to be entered.)

---

**error**

*Condition Type*

Class Precedence List:

error, serious-condition, condition, t

Description:
The type error consists of all conditions that represent errors.

---

**cell-error**

*Condition Type*

Class Precedence List:

cell-error, error, serious-condition, condition, t

Description:
The type cell-error consists of error conditions that occur during a location access. The name of the offending cell is initialized by the :name initialization argument to make-condition, and is accessed by the function cell-error-name.

See Also:
cell-error-name
**cell-error-name**

*Function*

**Syntax:**

\[
\text{cell-error-name } \text{condition} \rightarrow \text{name}
\]

**Arguments and Values:**

- *condition*—a condition of type *cell-error*.
- *name*—an object.

**Description:**

Returns the *name* of the offending cell involved in the *situation* represented by *condition*.

The nature of the result depends on the specific *type* of *condition*. For example, if the *condition* is of type *unbound-variable*, the result is the *name* of the *unbound variable* which was being accessed, if the *condition* is of type *undefined-function*, this is the *name* of the undefined function which was being accessed, and if the *condition* is of type *unbound-slot*, this is the *name* of the slot which was being accessed.

**See Also:**

- cell-error, unbound-slot, unbound-variable, undefined-function, [Section 9.1 (Condition System Concepts)]

---

**parse-error**

*Condition Type*

**Class Precedence List:**

- parse-error, error, serious-condition, condition, t

**Description:**

The *type* parse-error consists of error conditions that are related to parsing.

**See Also:**

- parse-namestring, reader-error
storage-condition  

Class Precedence List:
storage-condition, serious-condition, condition, t

Description:
The type storage-condition consists of serious conditions that relate to problems with memory management that are potentially due to implementation-dependent limits rather than semantic errors in conforming programs, and that typically warrant entry to the debugger if not handled. Depending on the details of the implementation, these might include such problems as stack overflow, memory region overflow, and storage exhausted.

Notes:
While some Common Lisp operations might signal storage-condition because they are defined to create objects, it is unspecified whether operations that are not defined to create objects create them anyway and so might also signal storage-condition. Likewise, the evaluator itself might create objects and so might signal storage-condition. (The natural assumption might be that such object creation is naturally inefficient, but even that is implementation-dependent.) In general, the entire question of how storage allocation is done is implementation-dependent, and so any operation might signal storage-condition at any time. Because such a condition is indicative of a limitation of the implementation or of the image rather than an error in a program, objects of type storage-condition are not of type error.

assert  

Syntax:
assert test-form [(place)*] [datum-form (argument-form)*]]  
→ nil

Arguments and Values:
test-form—a form; always evaluated.
place—a place; evaluated if an error is signaled.
datum-form—a form that evaluates to a datum. Evaluated each time an error is to be signaled, or not at all if no error is to be signaled.
argument-form—a form that evaluates to an argument. Evaluated each time an error is to be signaled, or not at all if no error is to be signaled.
datum, arguments—designators for a condition of default type error. (These designators are the result of evaluating datum-form and each of the argument-forms.)
assert

Description:
assert assures that test-form evaluates to true. If test-form evaluates to false, assert signals a correctable error (denoted by datum and arguments). Continuing from this error using the continue restart makes it possible for the user to alter the values of the places before assert evaluates test-form again. If the value of test-form is non-nil, assert returns nil.

The places are generalized references to data upon which test-form depends, whose values can be changed by the user in attempting to correct the error. Subforms of each place are only evaluated if an error is signaled, and might be re-evaluated if the error is re-signalized (after continuing without actually fixing the problem). The order of evaluation of the places is not specified; see Section 5.1.1.1 (Evaluation of Subforms to Places). If a place form is supplied that produces more values than there are store variables, the extra values are ignored. If the supplied form produces fewer values than there are store variables, the missing values are set to nil.

Examples:

```lisp
(setq x (make-array '(3 5) :initial-element 3))
→ #2A((3 3 3 3 3) (3 3 3 3 3) (3 3 3 3 3))
(setq y (make-array '(3 5) :initial-element 7))
→ #2A((7 7 7 7 7) (7 7 7 7 7) (7 7 7 7 7))
(defun matrix-multiply (a b)
  (let ((*print-array* nil))
    (assert (and (= (array-rank a) (array-rank b) 2)
                        (= (array-dimension a 1) (array-dimension b 0)))
      (a b) ; "Cannot multiply "S by "S." a b)
    (really-matrix-multiply a b)))
(matrix-multiply x y)
▷ Correctable error in MATRIX-MULTIPLY:
▷ Cannot multiply #<ARRAY ...> by #<ARRAY ...>.
▷ Restart options:
▷ 1: You will be prompted for one or more new values.
▷ 2: Top level.
▷ Debug> :continue 1
▷ Value for A: x
▷ Value for B: (make-array '(5 3) :initial-element 6)
→ #2A((54 54 54 54 54) (54 54 54 54 54) (54 54 54 54 54) (54 54 54 54 54) (54 54 54 54 54))
(defun double-safely (x) (assert (numberp x) (x)) (+ x x))
(double-safely 4)
→ 8
```

9–16 Programming Language—Common Lisp
(double-safely t)
▷ Correctable error in DOUBLE-SAFELY: The value of (NUMBERP X) must be non-NIL.
▷ Restart options:
▷ 1: You will be prompted for one or more new values.
▷ 2: Top level.
▷ Debug> :continue 1
▷ Value for X: 7
  → 14

Affected By:
  *break-on-signals*

The set of active condition handlers.

See Also:
  check-type, error, Section 5.1 (Generalized Reference)

Notes:
The debugger need not include the test-form in the error message, and the places should not be included in the message, but they should be made available for the user’s perusal. If the user gives the “continue” command, the values of any of the references can be altered. The details of this depend on the implementation’s style of user interface.

---

**error**  
*Function*

Syntax:

```lisp
error datum &rest arguments →
```

Arguments and Values:

datum, arguments—designators for a condition of default type simple-error.

Description:

error effectively invokes signal on the denoted condition.

If the condition is not handled, (invoke-debugger condition) is done. As a consequence of calling invoke-debugger, error cannot directly return; the only exit from error can come by non-local transfer of control in a handler or by use of an interactive debugging command.

Examples:

```lisp
(defun factorial (x)
  (cond ((or (not (typep x 'integer)) (minusp x))
    (error "S is not a valid argument to FACTORIAL." x))
  ((zerop x) 1))
```
Error

(t (* x (factorial (- x 1))))
→ FACTORIAL
(factorial 20)
→ 2432902008176640000
(factorial -1)
▷ Error: -1 is not a valid argument to FACTORIAL.
▷ To continue, type :CONTINUE followed by an option number:
▷ 1: Return to Lisp Toplevel.
▷ Debug>
(setq a 'fred)
→ FRED
(if (numberp a) (1+ a) (error "S is not a number." A))
▷ Error: FRED is not a number.
▷ To continue, type :CONTINUE followed by an option number:
▷ 1: Return to Lisp Toplevel.
▷ Debug> :Continue 1
▷ Return to Lisp Toplevel.

(define-condition not-a-number (error)
  ((argument :reader not-a-number-argument :initarg :argument))
  (:report (lambda (condition stream)
    (format stream "S is not a number." (not-a-number-argument condition)))))
→ NOT-A-NUMBER
(if (numberp a) (1+ a) (error 'not-a-number :argument a))
▷ Error: FRED is not a number.
▷ To continue, type :CONTINUE followed by an option number:
▷ 1: Return to Lisp Toplevel.
▷ Debug> :Continue 1
▷ Return to Lisp Toplevel.

Side Effects:
Handlers for the specified condition, if any, are invoked and might have side effects. Program execution might stop, and the debugger might be entered.

Affected By:
Existing handler bindings.
*break-on-signals*
Signals an error of type type-error if datum and arguments are not designators for a condition.

See Also:
cerror, signal, format, ignore-errors, *break-on-signals*, handler-bind, Section 9.1 (Condition
Some implementations may provide debugger commands for interactively returning from individual stack frames. However, it should be possible for the programmer to feel confident about writing code like:

```lisp
(defun wargames:no-win-scenario ()
  (if (error "pushing the button would be stupid."))
  (push-the-button))
```

In this scenario, there should be no chance that `error` will return and the button will get pushed.

While the meaning of this program is clear and it might be proven 'safe' by a formal theorem prover, such a proof is no guarantee that the program is safe to execute. Compilers have been known to have bugs, computers to have signal glitches, and human beings to manually intervene in ways that are not always possible to predict. Those kinds of errors, while beyond the scope of the condition system to formally model, are not beyond the scope of things that should seriously be considered when writing code that could have the kinds of sweeping effects hinted at by this example.

### cerror

**Function**

**Syntax:**

```
cerror continue-format-control datum &rest arguments  →  nil
```

**Arguments and Values:**

- **Continue-format-control**—a format control.
- **datum, arguments**—designators for a condition of default type `simple-error`.

**Description:**

`cerror` effectively invokes `error` on the condition named by `datum`. As with any function that implicitly calls `error`, if the condition is not handled, `(invoke-debugger condition)` is executed. While signaling is going on, and while in the debugger if it is reached, it is possible to continue code execution (i.e., to return from `cerror`) using the `continue` restart.

If `datum` is a condition, `arguments` can be supplied, but are used only in conjunction with the `continue-format-control`.

**Examples:**

```lisp
(defun real-sqrt (n)
  (when (minusp n)
    (setq n (- n))
```
(cerror "Return sqrt("D) instead." "Tried to take sqrt(¬D)." n))
(sqrt n))

(real-sqrt 4)
to 2.0

(real-sqrt -9)
▷ Correctable error in REAL-SQRT: Tried to take sqrt(-9).
▷ Restart options:
▷ 1: Return sqrt(9) instead.
▷ 2: Top level.
▷ Debug> :continue 1
to 3.0

(define-condition not-a-number (error)
  ((argument :reader not-a-number-argument :initarg :argument))
  (:report (lambda (condition stream)
    (format stream "~S is not a number." (not-a-number-argument condition)))))

(defun assure-number (n)
  (loop (when (numberp n) (return n))
    (cerror "Enter a number."
      'not-a-number :argument n)
    (format t "^Type a number: ")
    (setq n (read))
    (fresh-line)))

(assure-number 'a)
▷ Correctable error in ASSURE-NUMBER: A is not a number.
▷ Restart options:
▷ 1: Enter a number.
▷ 2: Top level.
▷ Debug> :continue 1
▷ Type a number: 1/2
to 1/2

(defun assure-large-number (n)
  (loop (when (and (numberp n) (> n 73)) (return n))
    (cerror "Enter a number~:[~; a bit larger than "D"]:~
      "^"A is not a large number."
      (numberp n) n)
    (format t "^Type a large number: ")
    (setq n (read))
    (fresh-line)))

9–20 Programming Language—Common Lisp
(assure-large-number 10000)
→ 10000

(assure-large-number 'a)
▷ Correctable error in ASSURE-LARGE-NUMBER: A is not a large number.
▷ Restart options:
▷ 1: Enter a number.
▷ 2: Top level.
▷ Debug> :continue 1
▷ Type a large number: 88
→ 88

(assure-large-number 37)
▷ Correctable error in ASSURE-LARGE-NUMBER: 37 is not a large number.
▷ Restart options:
▷ 1: Enter a number a bit larger than 37.
▷ 2: Top level.
▷ Debug> :continue 1
▷ Type a large number: 259
→ 259

(define-condition not-a-large-number (error)
  ((argument :reader not-a-large-number-argument :initarg :argument)))
  (:report (lambda (condition stream)
    (format stream "~S is not a large number." (not-a-large-number-argument condition))))

(defun assure-large-number (n)
  (loop (when (and (numberp n) (> n 73)) (return n))
    (cerror "Enter a number" :3*:; a bit larger than "*"D").
    'not-a-large-number
    :argument n
    :ignore (numberp n)
    :ignore n
    :allow-other-keys t)
  (format t "&Type a large number: ")
  (setq n (read))
  (fresh-line)))

(assure-large-number 'a)
▷ Correctable error in ASSURE-LARGE-NUMBER: A is not a large number.
▷ Restart options:
▷ 1: Enter a number.
Deferred debugger command (version 15.17):

```lisp
Debug> :continue 1
Type a large number: 88
→ 88
(assure-large-number 37)
Correctable error in ASSURE-LARGE-NUMBER: A is not a large number.
Restart options:
1: Enter a number a bit larger than 37.
2: Top level.
Debug> :continue 1
Type a large number: 259
→ 259
```

**Affected By:**

*break-on-signals*.

Existing handler bindings.

**See Also:**

error, format, handler-bind, *break-on-signals*, simple-type-error

**Notes:**

If *datum* is a condition type rather than a string, the format directive `*` may be especially useful in the `continue-format-control` in order to ignore the keywords in the initialization argument list. For example:

```lisp
(cerror "enter a new value to replace "*"s"
    'not-a-number
    :argument a)
```

---

**check-type**

**Macro**

**Syntax:**

```
check-type place typespec [string] → nil
```

**Arguments and Values:**

*place*—a place.

*typespec*—a type specifier.

*string*—a string; evaluated.
check-type

Description:

check-type signals a correctable error of type type-error if the contents of place are not of the type typespec.

check-type can return only if the store-value restart is invoked, either explicitly from a handler or implicitly as one of the options offered by the debugger. If the store-value restart is invoked, check-type stores the new value that is the argument to the restart invocation (or that is prompted for interactively by the debugger) in place and starts over, checking the type of the new value and signaling another error if it is still not of the desired type.

The first time place is evaluated, it is evaluated by normal evaluation rules. It is later evaluated as a place if the type check fails and the store-value restart is used; see Section 5.1.1.1 (Evaluation of Subforms to Places).

string should be an English description of the type, starting with an indefinite article (“a” or “an”). If string is not supplied, it is computed automatically from typespec. The automatically generated message mentions place, its contents, and the desired type. An implementation may choose to generate a somewhat differently worded error message if it recognizes that place is of a particular form, such as one of the arguments to the function that called check-type. string is allowed because some applications of check-type may require a more specific description of what is wanted than can be generated automatically from typespec.

Examples:

```lisp
(setq aardvarks '(sam harry fred))
→ (SAM HARRY FRED)
(check-type aardvarks (array * (3)))
▷ Error: The value of AARDVARKS, (SAM HARRY FRED),
▷ is not a 3-long array.
▷ To continue, type :CONTINUE followed by an option number:
▷ 1: Specify a value to use instead.
▷ 2: Return to Lisp Toplevel.
▷ Debug> :CONTINUE 1
▷ Use Value: #(SAM FRED HARRY)
→ NIL
aardvarks
→ #<ARRAY-T-3 13571>
(map 'list #'identity aardvarks)
→ (SAM FRED HARRY)
(setq aardvark-count 'foo)
→ FOO
(check-type aardvark-count (integer 0 *) "A positive integer")
▷ Error: The value of AARDVARK-COUNT, FOO, is not a positive integer.
▷ To continue, type :CONTINUE followed by an option number:
▷ 1: Specify a value to use instead.
▷ 2: Top level.
```

Conditions 9–23
check-type

(defmacro define-adder (name amount)
  (check-type name (and symbol (not null)) "a name for an adder function")
  (check-type amount integer)
  '(defun ,name (x) (+ x ,amount)))

(macroexpand '(define-adder add3 3))
→ (defun add3 (x) (+ x 3))

(macroexpand '(define-adder 7 7))
➤ Error: The value of NAME, 7, is not a name for an adder function.
➤ To continue, type :CONTINUE followed by an option number:
  1: Specify a value to use instead.
  2: Top level.
➤ Debug> :Continue 1
➤ Specify a value to use instead.
➤ Type a form to be evaluated and used instead: 'ADD7
→ (defun add7 (x) (+ x 7))

(macroexpand '(define-adder add5 something))
➤ Error: The value of AMOUNT, SOMETHING, is not an integer.
➤ To continue, type :CONTINUE followed by an option number:
  1: Specify a value to use instead.
  2: Top level.
➤ Debug> :Continue 1
➤ Type a form to be evaluated and used instead: 5
→ (defun add5 (x) (+ x 5))

Control is transferred to a handler.

Side Effects:
The debugger might be entered.

Affected By:
*break-on-signals*
The implementation.

See Also:
Section 9.1 (Condition System Concepts)

Notes:

(check-type place typespec)

9–24 Programming Language—Common Lisp
≡ (assert (typep place 'typespec) (place)
   'type-error :datum place :expected-type 'typespec)

simple-error

Condition Type

Class Precedence List:
simple-error, simple-condition, error, serious-condition, condition, t

Description:
The type simple-error consists of conditions that are signaled by error or cerror when a format control is supplied as the function’s first argument.

invalid-method-error

Function

Syntax:
invalid-method-error method format-control krest args → implementation-dependent

Arguments and Values:
method—a method.
format-control—a format control.
args—format arguments for the format-control.

Description:
The function invalid-method-error is used to signal an error of type error when there is an applicable method whose qualifiers are not valid for the method combination type. The error message is constructed by using the format-control suitable for format and any args to it. Because an implementation may need to add additional contextual information to the error message, invalid-method-error should be called only within the dynamic extent of a method combination function.

The function invalid-method-error is called automatically when a method fails to satisfy every qualifier pattern and predicate in a define-method-combination form. A method combination function that imposes additional restrictions should call invalid-method-error explicitly if it encounters a method it cannot accept.

Whether invalid-method-error returns to its caller or exits via throw is implementation-dependent.
Side Effects:
The debugger might be entered.

Affected By:
*break-on-signals*

See Also:
define-method-combination

method-combination-error Function

Syntax:
method-combination-error format-control &rest args → implementation-dependent

Arguments and Values:
format-control—a format control.

args—format arguments for format-control.

Description:
The function method-combination-error is used to signal an error in method combination.

The error message is constructed by using a format-control suitable for format and any args
to it. Because an implementation may need to add additional contextual information to the
error message, method-combination-error should be called only within the dynamic extent of a
method combination function.

Whether method-combination-error returns to its caller or exits via throw is implementation-
dependent.

Side Effects:
The debugger might be entered.

Affected By:
*break-on-signals*

See Also:
define-method-combination
signal

Function

Syntax:

signal datum &rest arguments → nil

Arguments and Values:

datum, arguments—designators for a condition of default type simple-condition.

Description:

Signals the condition denoted by the given datum and arguments. If the condition is not handled, signal returns nil.

Examples:

(defun handle-division-conditions (condition)
  (format t "Considering condition for division condition handling")
  (when (and (typep condition 'arithmetic-error)
              (eq '/' (arithmetic-error-operation condition)))
    (invoke-debugger condition)))

HANDLE-DIVISION-CONDITIONS

(defun handle-other-arithmetic-errors (condition)
  (format t "Considering condition for arithmetic condition handling")
  (when (typep condition 'arithmetic-error)
    (abort)))

HANDLE-OTHER-ARITHMETIC-ERRORS

(define-condition a-condition-with-no-handler (condition) ())

A-CONDITION-WITH-NO-HANDLER

(signal 'a-condition-with-no-handler)
NIL

(handler-bind ((condition #'handle-division-conditions)
               (condition #'handle-other-arithmetic-errors))
  Considering condition for division condition handling
  Considering condition for arithmetic condition handling
  NIL

(handler-bind ((arithmetic-error #'handle-division-conditions)
               (arithmetic-error #'handle-other-arithmetic-errors))
  (signal 'arithmetic-error :operation '* :operands '(1.2 b)))

Considering condition for division condition handling
Considering condition for arithmetic condition handling
Back to Lisp Toplevel

Side Effects:

The debugger might be entered due to *break-on-signals*.

Conditions 9–27
Handlers for the condition being signaled might transfer control.

Affected By:
Existing handler bindings.
*bbreak-on-signals*

See Also:
*bbreak-on-signals*, error, simple-condition, Section 9.1.4 (Signaling and Handling Conditions)

Notes:
If (typep datum *break-on-signals*) yields true, the debugger is entered prior to beginning the
signaling process. The continue restart can be used to continue with the signaling process. This
is also true for all other functions and macros that should, might, or must signal conditions.

---

**simple-condition**  
*Condition Type*

**Class Precedence List:**
simple-condition, condition, t

**Description:**
The type simple-condition represents conditions that are signaled by signal whenever a format-
control is supplied as the function’s first argument. The format control and format arguments
are initialized with the initialization arguments named :format-control and :format-arguments
to make-condition, and are accessed by the functions simple-condition-format-control and
simple-condition-format-arguments. If format arguments are not supplied to make-condition,
nil is used as a default.

**See Also:**
simple-condition-format-control, simple-condition-format-arguments
simple-condition-format-control, simple-condition-format-arguments

Function

Syntax:

simple-condition-format-control  condition  →  format-control

simple-condition-format-arguments  condition  →  format-arguments

Arguments and Values:

condition—a  condition  of  type  simple-condition.

format-control—a  format  control.

format-arguments—a  list.

Description:

simple-condition-format-control  returns  the  format  control  needed  to  process  the  condition’s  format  arguments.

simple-condition-format-arguments  returns  a  list  of  format  arguments  needed  to  process  the  condition’s  format  control.

Examples:

(setq foo (make-condition 'simple-condition
:format-control "Hi ~S"
:format-arguments '(ho)))

→  #<SIMPLE-CONDITION 26223553>

(apply #'format nil (simple-condition-format-control foo)
(simple-condition-format-arguments foo))

→  "Hi HO"

See Also:

simple-condition, Section 9.1 (Condition System Concepts)

warn

Function

Syntax:

warn  datum  &rest  arguments  →  nil

Arguments and Values:

datum, arguments—designators  for  a  condition  of  default  type  simple-warning.
**warn**

**Description:**

Signals a condition of type warning. If the condition is not handled, reports the condition to error output.

The precise mechanism for warning is as follows:

The warning condition is signaled

While the warning condition is being signaled, the muffle-warning restart is established for use by a handler. If invoked, this restart bypasses further action by warn, which in turn causes warn to immediately return nil.

If no handler for the warning condition is found

If no handlers for the warning condition are found, or if all such handlers decline, then the condition is reported to error output by warn in an implementation-dependent format.

nil is returned

The value returned by warn if it returns is nil.

**Examples:**

```lisp
(defun foo (x)
  (let ((result (* x 2)))
    (if (not (typep result 'fixnum))
      (warn "You're using very big numbers.")
      result))
  nil)

(foo 3)
⇒ 6

(foo most-positive-fixnum)
▷ Warning: You’re using very big numbers.
⇒ 4294967294

(setq *break-on-signals* t)
⇒ T

(foo most-positive-fixnum)
▷ Break: Caveat emptor.
▷ To continue, type :CONTINUE followed by an option number.
▷ 1: Return from Break.
▷ 2: Abort to Lisp Toplevel.
▷ Debug> :continue 1
```

9–30  Programming Language—Common Lisp
Warning: You’re using very big numbers.
→ 4294967294

Side Effects:
A warning is issued. The debugger might be entered.

Affected By:
Existing handler bindings.

*break-on-signals*, *error-output*.

Exceptional Situations:
If datum is a condition and if the condition is not of type warning, or arguments is non-nil, an error of type type-error is signaled.

If datum is a condition type, the result of (apply #'make-condition datum arguments) must be of type warning or an error of type type-error is signaled.

See Also:
*break-on-signals*, muffle-warning, signal

---

**simple-warning**

Class Precedence List:
simple-warning, simple-condition, warning, condition, t

Description:
The type simple-warning represents conditions that are signaled by warn whenever a format control is supplied as the function’s first argument.
invoke-debugger

Function

Syntax:
invoke-debugger condition → |

Arguments and Values:
condition—a condition object.

Description:
invoke-debugger attempts to enter the debugger with condition.

If *debugger-hook* is not nil, it should be a function (or the name of a function) to be called prior to entry to the standard debugger. The function is called with *debugger-hook* bound to nil, and the function must accept two arguments: the condition and the value of *debugger-hook* prior to binding it to nil. If the function returns normally, the standard debugger is entered.

The standard debugger never directly returns. Return can occur only by a non-local transfer of control, such as the use of a restart function.

Examples:
(ignore-errors ; Normally, this would suppress debugger entry
 (handler-bind ((error #'invoke-debugger)) ; But this forces debugger entry
 (error "Foo.")))

Debug: Foo.
To continue, type :CONTINUE followed by an option number:
1: Return to Lisp Toplevel.
Debug>

Side Effects:
*debugger-hook* is bound to nil, program execution is discontinued, and the debugger is entered.

Affected By:
*debug-io* and *debugger-hook*.

See Also:
error, break
break

Function

Syntax:
break &optional format-control &rest format-arguments → nil

Arguments and Values:
format-control—a format control. The default is implementation-dependent.

format-arguments—format arguments for the format-control.

Description:
break formats format-control and format-arguments and then goes directly into the debugger without allowing any possibility of interception by programmed error-handling facilities.

If the continue restart is used while in the debugger, break immediately returns nil without taking any unusual recovery action.

break binds *debugger-hook* to nil before attempting to enter the debugger.

Examples:

(break "You got here with arguments: ~S. ' (FOO 37 A))
▷ BREAK: You got here with these arguments: FOO, 37, A.
▷ To continue, type :CONTINUE followed by an option number:
▷ 1: Return from BREAK.
▷ 2: Top level.
▷ Debug> :CONTINUE 1
▷ Return from BREAK.
→ NIL

Side Effects:
The debugger is entered.

Affected By:
*debug-in*.

See Also:
error, invoke-debugger.

Notes:
break is used as a way of inserting temporary debugging “breakpoints” in a program, not as a way of signaling errors. For this reason, break does not take the continue-format-control argument that cerror takes. This and the lack of any possibility of interception by condition handling are the only program-visible differences between break and cerror.
The user interface aspects of `break` and `cerror` are permitted to vary more widely, in order to accommodate the interface needs of the implementation. For example, it is permissible for a Lisp read-eval-print loop to be entered by `break` rather than the conventional debugger.

`break` could be defined by:

```lisp
(defun break (&optional (format-control "Break") &rest format-arguments)
  (with-simple-restart (continue "Return from BREAK."
    (let ((*debugger-hook* nil))
      (invoke-debugger
        (make-condition 'simple-condition
          :format-control format-control
          :format-arguments format-arguments)))
    nil))
```

---

### *debugger-hook*

**Variable**

**Value Type:**

A designator for a function of two arguments (a condition and the value of `*debugger-hook*` at the time the debugger was entered), or `nil`.

**Initial Value:**

`nil`.

**Description:**

When the value of `*debugger-hook*` is non-nil, it is called prior to normal entry into the debugger, either due to a call to `invoke-debugger` or due to automatic entry into the debugger from a call to `error` or `cerror` with a condition that is not handled. The function may either handle the condition (transfer control) or return normally (allowing the standard debugger to run). To minimize recursive errors while debugging, `*debugger-hook*` is bound to `nil` by `invoke-debugger` prior to calling the function.

**Examples:**

```lisp
(defun one-of (choices &optional (prompt "Choice"))
  (let ((n (length choices)) (i))
    (do ((c choices (cdr c)) (i 1 (+ i 1)))
        ((null c))
      ((null c))
    (format t "\([D]\) ~A~\^{}%" i (car c)))
    (do () ((typep i '(integer 1 ,n)))
      (format t "\(\&A\)" prompt)
    (setq i (read)))
```

9–34 Programming Language—Common Lisp
(fresh-line))
(nth (- i 1) choices)))

(defun my-debugger (condition me-or-my-encapsulation)
  (format t "Fooey: ~A" condition)
  (let ((restart (one-of (compute-restarts))))
    (if (not restart) (error "My debugger got an error."))
    (let ((\(*debugger-hook* me-or-my-encapsulation))
      (invoke-restart-interactively restart))))

(let (*\(*debugger-hook* #'my-debugger)
  (+ 3 'a))
  ▷ Fooey: The argument to +, A, is not a number.
  ▷ [1] Supply a replacement for A.
  ▷ Choice: 1
  ▷ Form to evaluate and use: (+ 5 'b)
  ▷ Fooey: The argument to +, B, is not a number.
  ▷ [1] Supply a replacement for B.
  ▷ [2] Supply a replacement for A.
  ▷ Choice: 1
  ▷ Form to evaluate and use: 1
    → 9

Affected By:
  invoke-debugger

Notes:
When evaluating code typed in by the user interactively, it is sometimes useful to have the hook function bind \(*debugger-hook* to the function that was its second argument so that recursive errors can be handled using the same interactive facility.

*break-on-signals*  Variable

Value Type:
  a type specifier.

Initial Value:
  nil.

Conditions  9–35
*break-on-signals*

Description:
When `(typep condition *break-on-signals*)` returns `true`, calls to `signal`, and to other `operators` such as `error` that implicitly call `signal`, enter the debugger prior to `signaling` the `condition`.

The `continue` `restart` can be used to continue with the normal `signaling` process when a break occurs process due to `*break-on-signals*`.

Examples:

```
*break-on-signals*  →  NIL
(ignores (error 'simple-error :format-control "Fooey!")
→  NIL, #<SIMPLE-ERROR 32207172>

(let ((*break-on-signals* 'error))
   (ignore-errors (error 'simple-error :format-control "Fooey!")
> Break: Fooey!
> BREAK entered because of *BREAK-ON-SIGNALS*.
> To continue, type :CONTINUE followed by an option number:
> 1: Continue to signal.
> 2: Top level.
> Debug> :CONTINUE 1
> Continue to signal.
> →  NIL, #<SIMPLE-ERROR 32212257>

(let ((*break-on-signals* 'error))
   (error 'simple-error :format-control "Fooey!")
> Break: Fooey!
> BREAK entered because of *BREAK-ON-SIGNALS*.
> To continue, type :CONTINUE followed by an option number:
> 1: Continue to signal.
> 2: Top level.
> Debug> :CONTINUE 1
> Continue to signal.
> Error: Fooey!
> To continue, type :CONTINUE followed by an option number:
> 1: Top level.
> Debug> :CONTINUE 1
> Top level.
```

See Also:
`break`, `signal`, `warn`, `error`, `typep`, Section 9.1 (Condition System Concepts)

Notes:
`*break-on-signals*` is intended primarily for use in debugging code that does signaling. When setting `*break-on-signals*`, the user is encouraged to choose the most restrictive specification.

9–36 Programming Language—Common Lisp
that suffices. Setting *break-on-signals* effectively violates the modular handling of condition signaling. In practice, the complete effect of setting *break-on-signals* might be unpredictable in some cases since the user might not be aware of the variety or number of calls to signal that are used in code called only incidentally.

*break-on-signals* enables an early entry to the debugger but such an entry does not preclude an additional entry to the debugger in the case of operations such as error and cerror.

---

**handler-bind**

*Macro*

**Syntax:**

```
(handler-bind ([binding]*) {form}*) → {result}*
```

**binding ::= (type handler)**

**Arguments and Values:**

- **type**—a type specifier.
- **handler**—a form; evaluated to produce a handler-function.
- **handler-function**—a designator for a function of one argument.
- **forms**—an implicit progn.
- **results**—the values returned by the forms.

**Description:**

Executes forms in a dynamic environment where the indicated handler bindings are in effect.

Each handler should evaluate to a handler-function, which is used to handle conditions of the given type during execution of the forms. This function should take a single argument, the condition being signaled.

If more than one handler binding is supplied, the handler bindings are searched sequentially from top to bottom in search of a match (by visual analogy with typecase). If an appropriate type is found, the associated handler is run in a dynamic environment where none of these handler bindings are visible (to avoid recursive errors). If the handler declines, the search continues for another handler.

If no appropriate handler is found, other handlers are sought from dynamically enclosing contours. If no handler is found outside, then signal returns or error enters the debugger.

**Examples:**

In the following code, if an unbound variable error is signaled in the body (and not handled by an intervening handler), the first function is called.
(handler-bind ((unbound-variable #'(lambda ...))
    (error #'(lambda ...)))
...
)

If any other kind of error is signaled, the second function is called. In either case, neither handler
is active while executing the code in the associated function.

(defun trap-error-handler (condition)
  (format *error-output* "~&~A~&" condition)
  (throw 'trap-errors nil))

(defun trap-errors (&rest forms)
  '(catch 'trap-errors
    (handler-bind ((error #'trap-error-handler))
      ,@forms)))

(list (trap-errors (signal "Foo.")) 1)
(list (trap-errors (error "Bar.")) 2)
(+ 1 2)

> Bar.
→ (1 NIL 3)

Note that “Foo.” is not printed because the condition made by signal is a simple condition, which
is not of type error, so it doesn’t trigger the handler for error set up by trap-errors.

See Also:
handler-case

handler-case

Macro

Syntax:

```
(handler-case expression [{{error-clause}* | no-error-clause}] → {result}* }

clause::=error-clause | no-error-clause
error-clause::=(typespec (var) {declaration}* {form}* }
no-error-clause::=(:no-error lambda-list {declaration}* {form}* )
```

Arguments and Values:

- **expression**—a form.
- **typespec**—a type specifier.
**handler-case**

- **var**—a variable name.
- **lambda-list**—an ordinary lambda list.
- **declaration**—a `declare` expression; not evaluated.
- **form**—a form.

**results**—In the normal situation, the values returned are those that result from the evaluation of `expression`; in the exceptional situation when control is transferred to a `clause`, the value of the last `form` in that `clause` is returned.

**Description:**

`handler-case` executes `expression` in a *dynamic environment* where various handlers are active. Each `error-clause` specifies how to handle a condition matching the indicated `typespec`. A `no-error-clause` allows the specification of a particular action if control returns normally.

If a condition is signaled for which there is an appropriate `error-clause` during the execution of `expression` (i.e., one for which `(typep condition 'typespec)` returns `true`) and if there is no intervening handler for a condition of that type, then control is transferred to the body of the relevant `error-clause`. In this case, the dynamic state is unwound appropriately (so that the handlers established around the `expression` are no longer active), and `var` is bound to the condition that had been signaled. If more than one case is provided, those cases are made accessible in parallel. That is, in

```lisp
(handler-case form
  (typespec1 (var1) form1)
  (typespec2 (var2) form2))
```

if the first `clause` (containing `form1`) has been selected, the handler for the second is no longer visible (or vice versa).

The `clauses` are searched sequentially from top to bottom. If there is type overlap between `typespecs`, the earlier of the `clauses` is selected.

If `var` is not needed, it can be omitted. That is, a `clause` such as:

```lisp
  (typespec (var) (declare (ignore var)) form)
```

can be written `(typespec () form)`.

If there are no `forms` in a selected `clause`, the case, and therefore `handler-case`, returns `nil`. If execution of `expression` returns normally and no `no-error-clause` exists, the values returned by `expression` are returned by `handler-case`. If execution of `expression` returns normally and a `no-error-clause` does exist, the values returned are used as arguments to the function described by constructing `(lambda lambda-list {form}*` from the `no-error-clause`, and the values of that function call are returned by `handler-case`. The handlers which were established around the `expression` are no longer active at the time of this call.
handler-case

Examples:

(defun assess-condition (condition)
  (handler-case (signal condition)
    (warning () "Lots of smoke, but no fire.")
    ((or arithmetic-error control-error cell-error stream-error)
      (condition)
      (format nil "S looks especially bad." condition))
    (serious-condition (condition)
      (format nil "S looks serious." condition))
    (condition () "Hardly worth mentioning.")()))

→ ASSESS-CONDITION
(assess-condition (make-condition 'stream-error :stream *terminal-io*))
→ "STREAM-ERROR 12352256> looks especially bad."
(define-condition random-condition (condition) ()
  (:report (lambda (condition stream)
    (declare (ignore condition))
    (princ "You" stream))))

→ RANDOM-CONDITION
(assess-condition (make-condition 'random-condition))
→ "Hardly worth mentioning."

See Also: handler-bind, ignore-errors, Section 9.1 (Condition System Concepts)

Notes:

(handler-case form
  (type1 (var1) . body1)
  (type2 (var2) . body2) ...)

is approximately equivalent to:

(block #1=#:g0001
  (let ((#2=#:g0002 nil))
    (tagbody
      (handler-bind (((type1 #'(lambda (temp)
                                (setq #1# temp)
                                (go #3=#:g0003)))
                    (type2 #'(lambda (temp)
                                (setq #2# temp)
                                (go #4=#:g0004)))) ...)
      (return-from #1# form)
      #3# (return-from #1# (let ((var1 #2#) . body1))
      #4# (return-from #1# (let ((var2 #2#) . body2) ...)))
    )
  )
)

(handler-case form

9–40 Programming Language—Common Lisp
(type1 (var1) . body1)
...
(no-error (varN-1 varN-2 ...) . bodyN))
is approximately equivalent to:

(block #1=#:error-return
  (multiple-value-call #*(lambda (varN-1 varN-2 ...) . bodyN)
    (block #2=#:normal-return
      (return-from #1#
        (handler-case (return-from #2# form)
          (type1 (var1) . body1) ...))))

ignore-errors  

Macro

Syntax:

ignore-errors {form}* → {result}*

Arguments and Values:
forms—an implicit progn.
results—In the normal situation, the values of the forms are returned; in the exceptional situation, two values are returned: nil and the condition.

Description:
ignore-errors is used to prevent conditions of type error from causing entry into the debugger. Specifically, ignore-errors executes forms in a dynamic environment where a handler for conditions of type error has been established; if invoked, it handles such conditions by returning two values, nil and the condition that was signaled, from the ignore-errors form.

If a normal return from the forms occurs, any values returned are returned by ignore-errors.

Examples:

(defun load-init-file (program)
  (let ((win nil))
    (ignore-errors ;; if this fails, don't enter debugger
      (load (merge-pathnames (make-pathname :name program :type :lisp)
              (user-homedir-pathname)))
      (setq win t))
    (unless win (format t ""\"Init file failed to load."\")
      win))

Conditions  9–41
(load-init-file "no-such-program")
▷ Init file failed to load.
NIL

See Also:
handler-case, Section 9.1 (Condition System Concepts)

Notes:

(ignores-errors . forms)

is equivalent to:

(handler-case (progn . forms)
  (error (condition) (values nil condition)))

Because the second return value is a condition in the exceptional case, it is common (but not required) to arrange for the second return value in the normal case to be missing or nil so that the two situations can be distinguished.

define-condition

Macro

Syntax:

define-condition name ([parent-type]*) ([slot-spec]*) (option)*
→ name

slot-spec::=slot-name | (slot-name slot-option)

slot-option::=[[:reader symbol]* | [:writer (function-name)* | [:accessor symbol]* | [:allocation allocation-type] | [:initarg symbol]* | [:initform form] | [:type type-specifier] ]

option::=[[:default-initargs . initarg-list] | [:documentation string] | [:report report-name] ]

9–42 Programming Language—Common Lisp
define-condition

\[ \text{function-name} ::= \{ \text{symbol} | (\text{setf} \ \text{symbol}) \} \]

\[ \text{allocation-type} ::= \text{:instance} | \text{:class} \]

\[ \text{report-name} ::= \text{string} | \text{symbol} | \text{lambda expression} \]

Arguments and Values:

name—a symbol.

parent-type—a symbol naming a condition type. If no parent-types are supplied, the parent-types default to (condition).

default-initargs—a list of keyword/value pairs.

Slot-spec—the name of a slot or a list consisting of the slot-name followed by zero or more slot-options.

Slot-name—a slot name (a symbol), the list of a slot name, or the list of slot name/slot form pairs.

Option—Any of the following:

:reader

:reader can be supplied more than once for a given slot and cannot be nil.

:writer

:writer can be supplied more than once for a given slot and must name a generic function.

:accessor

:accessor can be supplied more than once for a given slot and cannot be nil.

:allocation

:allocation can be supplied once at most for a given slot. The default if :allocation is not supplied is :instance.

:initarg

:initarg can be supplied more than once for a given slot.

:initform

:initform can be supplied once at most for a given slot.

:type

:type can be supplied once at most for a given slot.
define-condition

:documentation
:documentation can be supplied once at most for a given slot.

:report
:report can be supplied once at most.

Description:
define-condition defines a new condition type called name, which is a subtype of the type or types named by parent-type. Each parent-type argument specifies a direct supertype of the new condition. The new condition inherits slots and methods from each of its direct supertypes, and so on.

If a slot name/slot form pair is supplied, the slot form is a form that can be evaluated by make-condition to produce a default value when an explicit value is not provided. If no slot form is supplied, the contents of the slot is initialized in an implementation-dependent way.

If the type being defined and some other type from which it inherits have a slot by the same name, only one slot is allocated in the condition, but the supplied slot form overrides any slot form that might otherwise have been inherited from a parent-type. If no slot form is supplied, the inherited slot form (if any) is still visible.

Accessors are created according to the same rules as used by defclass.

A description of slot-options follows:

:reader

The :reader slot option specifies that an unqualified method is to be defined on the generic function named by the argument to :reader to read the value of the given slot.

- The :initform slot option is used to provide a default initial value form to be used in the initialization of the slot. This form is evaluated every time it is used to initialize the slot. The lexical environment in which this form is evaluated is the lexical environment in which the define-condition form was evaluated. Note that the lexical environment refers both to variables and to functions. For local slots, the dynamic environment is the dynamic environment in which make-condition was called; for shared slots, the dynamic environment is the dynamic environment in which the define-condition form was evaluated.

No implementation is permitted to extend the syntax of define-condition to allow (slot-name form) as an abbreviation for (slot-name :initform form).

:initarg

The :initarg slot option declares an initialization argument named by its symbol ar-
define-condition

...argument and specifies that this initialization argument initializes the given slot. If the initialization argument has a value in the call to initialize-instance, the value is stored into the given slot, and the slot’s :initform slot option, if any, is not evaluated. If none of the initialization arguments specified for a given slot has a value, the slot is initialized according to the :initform slot option, if specified.

:type

The :type slot option specifies that the contents of the slot is always of the specified type. It effectively declares the result type of the reader generic function when applied to an object of this condition type. The consequences of attempting to store in a slot a value that does not satisfy the type of the slot is undefined.

:default-initargs

This option is treated the same as it would be defclass.

:documentation

The :documentation slot option provides a documentation string for the slot.

:report

Condition reporting is mediated through the print-object method for the condition type in question, with *print-escape* always being nil. Specifying (:report report-name) in the definition of a condition type C is equivalent to:

(defmethod print-object ((x c) stream)
  (if *print-escape* (call-next-method) (report-name x stream)))

If the value supplied by the argument to :report (report-name) is a symbol or a lambda expression, it must be acceptable to function. (function report-name) is evaluated in the current lexical environment. It should return a function of two arguments, a condition and a stream, that prints on the stream a description of the condition. This function is called whenever the condition is printed while *print-escape* is nil.

If report-name is a string, it is a shorthand for

(lambda (condition stream)
  (declare (ignore condition))
  (write-string report-name stream))

This option is processed after the new condition type has been defined, so use of the slot accessors within the :report function is permitted. If this option is not supplied, information about how to report this type of condition is inherited from the parent-type.

The consequences are unspecified if an attempt is made to read a slot that has not been explicitly initialized and that has not been given a default value.
define-condition

The consequences are unspecified if an attempt is made to assign the slots by using setf.

If a define-condition form appears as a top level form, the compiler must make name recognizable as a valid type name, and it must be possible to reference the condition type as the parent-type of another condition type in a subsequent define-condition form in the file being compiled.

Examples:
The following form defines a condition of type peg/hole-mismatch which inherits from a condition type called blocks-world-error:

```
(define-condition peg/hole-mismatch
  (blocks-world-error)
  ((peg-shape :initarg :peg-shape
                :reader peg/hole-mismatch-peg-shape)
   (hole-shape :initarg :hole-shape
                :reader peg/hole-mismatch-hole-shape))
 (:report (lambda (condition stream)
              (format stream "A ~A peg cannot go in a ~A hole."
                       (peg/hole-mismatch-peg-shape condition)
                       (peg/hole-mismatch-hole-shape condition))))
```

The new type has slots peg-shape and hole-shape, so make-condition accepts :peg-shape and :hole-shape keywords. The readers peg/hole-mismatch-peg-shape and peg/hole-mismatch-hole-shape apply to objects of this type, as illustrated in the :report information.

The following form defines a condition type named machine-error which inherits from error:

```
(define-condition machine-error
  (error)
  ((machine-name :initarg :machine-name
                  :reader machine-error-machine-name))
 (:report (lambda (condition stream)
             (format stream "There is a problem with ~A."
                      (machine-error-machine-name condition))))
```

Building on this definition, a new error condition can be defined which is a subtype of machine-error for use when machines are not available:

```
(define-condition machine-not-available-error (machine-error) ()
 (:report (lambda (condition stream)
             (format stream "The machine ~A is not available."
                      (machine-error-machine-name condition))))
```

This defines a still more specific condition, built upon machine-not-available-error, which provides a slot initialization form for machine-name but which does not provide any new slots or report information. It just gives the machine-name slot a default initialization:
(define-condition my-favorite-machine-not-available-error
  (machine-not-available-error)
  ((machine-name :initform "mc.lcs.mit.edu")))

Note that since no :report clause was given, the information inherited from
machine-not-available-error is used to report this type of condition.

(define-condition ate-too-much (error)
  ((person :initarg :person :reader ate-too-much-person)
   (weight :initarg :weight :reader ate-too-much-weight)
   (kind-of-food :initarg :kind-of-food
                :reader :ate-too-much-kind-of-food)))

→ ATE-TOO-MUCH

(define-condition ate-too-much-ice-cream (ate-too-much)
  ((kind-of-food :initform 'ice-cream)
   (flavor :initarg :flavor
           :reader ate-too-much-ice-cream-flavor
           :initform 'vanilla))
  (:report (lambda (condition stream)
           (format stream "A ate too much ~A ice-cream"
                    (ate-too-much-person condition)
                    (ate-too-much-ice-cream-flavor condition)))))

→ ATE-TOO-MUCH-ICE-CREAM

(make-condition 'ate-too-much-ice-cream
    :person 'fred
    :weight 300
    :flavor 'chocolate)

→ #<ATE-TOO-MUCH-ICE-CREAM 32236101>

(format t "~A")

▷ FRED ate too much CHOCOLATE ice-cream
→ NIL

See Also:
make-condition, defclass, Section 9.1 (Condition System Concepts)

---

**make-condition**

*Function*

**Syntax:**

```
make-condition type &rest slot-initializations → condition
```

**Arguments and Values:**

- `type`—a *type specifier* (for a *subtype* of *condition*).
**slot-initializations**—an initialization argument list.

**condition**—a condition.

**Description:**
Constructs and returns a condition of type type using slot-initializations for the initial values of the slots. The newly created condition is returned.

**Examples:**

```lisp
(defun *oops-count* 0)

(setq a (make-condition 'simple-error
  :format-control "This is your ~:R error."
  :format-arguments (list (incf *oops-count*))))
→ #<SIMPLE-ERROR 32245104>

(format t "" ~A"\%" a)
▷ This is your first error.
→ NIL

(error a)
▷ Error: This is your first error.
▷ To continue, type :CONTINUE followed by an option number:
▷ 1: Return to Lisp Toplevel.
▷ Debug>
```

**Affected By:**
The set of defined condition types.

**See Also:**
define-condition, Section 9.1 (Condition System Concepts)

---

**restart**

**System Class**

**Class Precedence List:**
restart, t

**Description:**
An object of type restart represents a function that can be called to perform some form of recovery action, usually a transfer of control to an outer point in the running program.
An implementation is free to implement a restart in whatever manner is most convenient; a restart has only dynamic extent relative to the scope of the binding form which establishes it.

**compute-restarts**

*Function*

**Syntax:**

```
compute-restarts &optional condition → restarts
```

**Arguments and Values:**

- `condition`—a condition object, or nil.
- `restarts`—a list of restarts.

**Description:**

`compute-restarts` uses the dynamic state of the program to compute a list of the restarts which are currently active.

The resulting list is ordered so that the innermost (more-recently established) restarts are nearer the head of the list.

When `condition` is non-nil, only those restarts are considered that are either explicitly associated with that `condition`, or not associated with any `condition`; that is, the excluded restarts are those that are associated with a non-empty set of `condition` objects of which the given `condition` is not an element. If `condition` is nil, all restarts are considered.

`compute-restarts` returns all applicable restarts, including anonymous ones, even if some of them have the same name as others and would therefore not be found by `find-restart` when given a symbol argument.

Implementations are permitted, but not required, to return distinct lists from repeated calls to `compute-restarts` while in the same dynamic environment. The consequences are undefined if the list returned by `compute-restarts` is every modified.

**Examples:**

```
;; One possible way in which an interactive debugger might present
;; restarts to the user.
(defun invoke-a-restart ()
  (let ((restarts (compute-restarts)))
    (do ((i 0 (+ i 1)) (r restarts (cdr r)) ((null r))
      (format t "\%D: ~A~A\n" i (car r)))
    (let ((n nil) (k (length restarts)))
      (loop (when (and (typep n 'integer) (>= n 0) (< n k))))
```

**Conditions** 9–49
(return t))
(format t "Option: ")
(setq n (read))
(fresh-line))
(invoke-restart-interactively (nth n restarts))))

(restart-case (invoke-a-restart)
  (one () 1)
  (two () 2)
  (nil () :report "Who knows?" 'anonymous)
  (one () 'I)
  (two () 'II))
> 0: ONE
> 1: TWO
> 2: Who knows?
> 3: ONE
> 4: TWO
> 5: Return to Lisp Toplevel.
> Option: 4
   → II

;; Note that in addition to user-defined restart points, COMPUTE-RESTARTS
;; also returns information about any system-supplied restarts, such as
;; the "Return to Lisp Toplevel" restart offered above.

Affected By:
Existing restarts.

See Also:
find-restart, invoke-restart, restart-bind

---

**find-restart**

*Function*

**Syntax:**

find-restart *identifier* &optional *condition*

**Arguments and Values:**

- *identifier*—a non-nil symbol, or a restart.
- *condition*—a condition object, or nil.

9–50 Programming Language—Common Lisp
restart—a restart or nil.

Description:

find-restart searches for a particular restart in the current dynamic environment.

When condition is non-nil, only those restarts are considered that are either explicitly associated with that condition, or not associated with any condition; that is, the excluded restarts are those that are associated with a non-empty set of conditions of which the given condition is not an element. If condition is nil, all restarts are considered.

If identifier is a symbol, then the innermost (most recently established) applicable restart with that name is returned. nil is returned if no such restart is found.

If identifier is a currently active restart, then it is returned. Otherwise, nil is returned.

Examples:

(restart-case
  (let ((r (find-restart 'my-restart)))
    (format t "S is named ~S" r (restart-name r)))
  (my-restart () nil))
▷ #<RESTART 32307325> is named MY-RESTART
→ NIL
  (find-restart 'my-restart)
→ NIL

Affected By:

Existing restarts.

restart-case, restart-bind, with-condition-restarts.

See Also:

compute-restarts

Notes:

(find-restart identifier)
≡ (find identifier (compute-restarts) :key :restart-name)

Although anonymous restarts have a name of nil, the consequences are unspecified if nil is given as an identifier. Occasionally, programmers lament that nil is not permissible as an identifier argument. In most such cases, compute-restarts can probably be used to simulate the desired effect.
invoke-restart

**Function**

Syntax:

\[
\text{invoke-restart} \ \text{restart} \ \&\text{rest} \ \text{arguments} \ \rightarrow \ \{\text{result}\}^*
\]

Arguments and Values:

- **restart**—a restart designator.
- **argument**—an object.
- **results**—the values returned by the function associated with **restart**, if that function returns.

Description:

Calls the function associated with **restart**, passing **arguments** to it. **Restart** must be valid in the current dynamic environment.

Examples:

\[
\begin{align*}
\text{(defun add3 (x) (check-type x number) (+ x 3))} \\
\text{(foo 'seven)} \\
\text{⊿ Error: The value SEVEN was not of type NUMBER.} \\
\text{⊿ To continue, type :CONTINUE followed by an option number:} \\
\text{⊿ 1: Specify a different value to use.} \\
\text{⊿ 2: Return to Lisp Toplevel.} \\
\text{⊿ Debug> (invoke-restart 'store-value 7)} \\
\text{→ 10}
\end{align*}
\]

Side Effects:

A non-local transfer of control might be done by the restart.

Affected By:

Existing restarts.

Exceptional Situations:

- If **restart** is not valid, an error of type **control-error** is signaled.

See Also:

- find-restart, restart-bind, restart-case, invoke-restart-interactively

Notes:

The most common use for **invoke-restart** is in a handler. It might be used explicitly, or implicitly through **invoke-restart-interactively** or a **restart function**.
Restart functions call `invoke-restart`, not vice versa. That is, `invoke-restart` provides primitive functionality, and restart functions are non-essential “syntactic sugar.”

### invoke-restart-interactively

**Function**

**Syntax:**

```
invoke-restart-interactively restart → {result}*
```

**Arguments and Values:**

- `restart`—a restart designator.

- `results`—the values returned by the function associated with `restart`, if that function returns.

**Description:**

`invoke-restart-interactively` calls the function associated with `restart`, prompting for any necessary arguments. If `restart` is a name, it must be valid in the current dynamic environment.

`invoke-restart-interactively` prompts for arguments by executing the code provided in the `:interactive` keyword to `restart-case` or `:interactive-function` keyword to `restart-bind`.

If no such options have been supplied in the corresponding `restart-bind` or `restart-case`, then the consequences are undefined if the `restart` takes required arguments. If the arguments are optional, an argument list of `nil` is used.

Once the arguments have been determined, `invoke-restart-interactively` executes the following:

```
(apply #'invoke-restart restart arguments)
```

**Examples:**

```
(defun add3 (x) (check-type x number) (+ x 3))

(add3 'seven)
```

Error: The value SEVEN was not of type NUMBER.

To continue, type :CONTINUE followed by an option number:

- 1: Specify a different value to use.
- 2: Return to Lisp Toplevel.
- Debug> (invoke-restart-interactively 'store-value)

Type a form to evaluate and use: 7

→ 10

**Side Effects:**

If prompting for arguments is necessary, some typeout may occur (on query I/O).

A non-local transfer of control might be done by the restart.
Affected By:

*query-io*, active restarts

Exceptional Situations:

If restart is not valid, an error of type control-error is signaled.

See Also:

find-restart, invoke-restart, restart-case, restart-bind

Notes:

invoke-restart-interactively is used internally by the debugger and may also be useful in implementing other portable, interactive debugging tools.

---

```
restart-bind
```

**Macro**

**Syntax:**

```
restart-bind (((name function (key-val-pair)*)) (form)*
  → (result)*

  key-val-pair::=:interactive-function interactive-function |
  :report-function report-function |
  :test-function test-function
```

**Arguments and Values:**

- **name**—a symbol; not evaluated.
- **function**—a form; evaluated.
- **forms**—an implicit progn.
- **interactive-function**—a form; evaluated.
- **report-function**—a form; evaluated.
- **test-function**—a form; evaluated.
- **results**—the values returned by the forms.

**Description:**

restart-bind executes the body of forms in a dynamic environment where restarts with the given names are in effect.
If a name is nil, it indicates an anonymous restart; if a name is a non-nil symbol, it indicates a named restart.

The function, interactive-function, and report-function are unconditionally evaluated in the current lexical and dynamic environment prior to evaluation of the body. Each of these forms must evaluate to a function.

If invoke-restart is done on that restart, the function which resulted from evaluating function is called, in the dynamic environment of the invoke-restart, with the arguments given to invoke-restart. The function may either perform a non-local transfer of control or may return normally.

If the restart is invoked interactively from the debugger (using invoke-restart-interactively), the arguments are defaulted by calling the function which resulted from evaluating interactive-function. That function may optionally prompt interactively on query I/O, and should return a list of arguments to be used by invoke-restart-interactively when invoking the restart.

If a restart is invoked interactively but no interactive-function is used, then an argument list of nil is used. In that case, the function must be compatible with an empty argument list.

If the restart is presented interactively (e.g., by the debugger), the presentation is done by calling the function which resulted from evaluating report-function. This function must be a function of one argument, a stream. It is expected to print a description of the action that the restart takes to that stream. This function is called any time the restart is printed while *print-escape* is nil.

In the case of interactive invocation, the result is dependent on the value of :interactive-function as follows.

:interactive-function

Value is evaluated in the current lexical environment and should return a function of no arguments which constructs a list of arguments to be used by invoke-restart-interactively when invoking this restart. The function may prompt interactively using query I/O if necessary.

:report-function

Value is evaluated in the current lexical environment and should return a function of one argument, a stream, which prints on the stream a summary of the action that this restart takes. This function is called whenever the restart is reported (printed while *print-escape* is nil). If no :report-function option is provided, the manner in which the restart is reported is implementation-dependent.

:test-function

Value is evaluated in the current lexical environment and should return a function of one argument, a condition, which returns true if the restart is to be considered visible.
Affected By:
*query-io*.

See Also:
restart-case, with-simple-restart

Notes:
restart-bind is primarily intended to be used to implement restart-case and might be useful in implementing other macros. Programmers who are uncertain about whether to use restart-case or restart-bind should prefer restart-case for the cases where it is powerful enough, using restart-bind only in cases where its full generality is really needed.

### restart-case

**Macro**

**Syntax:**
```
restart-case restartable-form {↓ clause} → {result}*
```

**Arguments and Values:**
- **restartable-form**—a form.
- **case-name**—a symbol or nil.
- **lambda-list**—an ordinary lambda list.
- **interactive-expression**—a symbol or a lambda expression.
- **report-expression**—a string, a symbol, or a lambda expression.
- **test-expression**—a symbol or a lambda expression.
- **declaration**—a declare expression; not evaluated.
- **form**—a form.
- **results**—the values resulting from the evaluation of restartable-form, or the values returned by the last form executed in a chosen clause, or nil.
**restart-case**

**Description:**

`restart-case` evaluates `restartable-form` in a *dynamic environment* where the clauses have special meanings as points to which control may be transferred. If `restartable-form` finishes executing and returns any values, all values returned are returned by `restart-case` and processing has completed. While `restartable-form` is executing, any code may transfer control to one of the clauses (see `invoke-restart`). If a transfer occurs, the forms in the body of that clause is evaluated and any values returned by the last such form are returned by `restart-case`. In this case, the dynamic state is unwound appropriately (so that the restarts established around the `restartable-form` are no longer active) prior to execution of the clause.

If there are no forms in a selected clause, `restart-case` returns `nil`.

If `case-name` is a *symbol*, it names this restart.

It is possible to have more than one clause use the same `case-name`. In this case, the first clause with that name is found by `find-restart`. The other clauses are accessible using `compute-restarts`.

Each `arglist` is an *ordinary lambda list* to be bound during the execution of its corresponding forms. These parameters are used by the `restart-case` clause to receive any necessary data from a call to `invoke-restart`.

By default, `invoke-restart-interactively` passes no arguments and all arguments must be optional in order to accommodate interactive restarting. However, the arguments need not be optional if the `:interactive` keyword has been used to inform `invoke-restart-interactively` about how to compute a proper argument list.

**Keyword** options have the following meaning.

`:interactive`

The *value* supplied by `:interactive value` must be a suitable argument to `function`. `(function value)` is evaluated in the current lexical environment. It should return a function of no arguments which returns arguments to be used by `invoke-restart-interactively` when it is invoked. `invoke-restart-interactively` is called in the dynamic environment available prior to any restart attempt, and uses *query I/O* for user interaction.

If a restart is invoked interactively but no `:interactive` option was supplied, the argument list used in the invocation is the empty list.

`:report`

If the *value* supplied by `:report value` is a *lambda expression* or a *symbol*, it must be acceptable to `function`. `(function value)` is evaluated in the current lexical environment. It should return a function of one argument, a *stream*, which prints on the *stream* a description of the restart. This function is called whenever the restart is printed while `*print-escape*` is `nil`.

Conditions 9–57
**restart-case**

If `value` is a *string*, it is a shorthand for

```
(lambda (stream) (write-string value stream))
```

If a named restart is asked to report but no report information has been supplied, the name of the restart is used in generating default report text.

When `*print-escape*` is `nil`, the printer uses the report information for a restart. For example, a debugger might announce the action of typing a “continue” command by:

```
(format t "\&S -- \&A\%" #'continue some-restart)
```

which might then display as something like:

```
:CONTINUE -- Return to command level
```

The consequences are unspecified if an unnamed restart is specified but no `:report` option is provided.

**:test**

The `value` supplied by `:test value` must be a suitable argument to `function`. 

```
(function value)
```

is evaluated in the current lexical environment. It should return a `function` of one `argument`, the `condition`, that returns `true` if the restart is to be considered visible.

The default for this option is equivalent to `(lambda (c) (declare (ignore c)) t)`.

If the `restartable-form` is a `list` whose `car` is any of the symbols `signal`, `error`, `cerror`, or `warn` (or is a `macro form` which macroexpands into such a `list`), then `with-condition-restarts` is used implicitly to associate the indicated `restarts` with the `condition` to be signaled.

**Examples:**

```
(restart-case
 (handler-bind ((error #'(lambda (c)
 (declare (ignore condition))
 (invoke-restart 'my-restart 7))))
 (error "Foo."))
 (my-restart (&optional v) v))
→ 7

(define-condition food-error (error) ())
→ FOOD-ERROR
(define-condition bad-tasting-sundae (food-error)
 ((ice-cream :initarg :ice-cream :reader bad-tasting-sundae-ice-cream)
  (sauce :initarg :sauce :reader bad-tasting-sundae-sauce)
  (topping :initarg :topping :reader bad-tasting-sundae-topping))
 (:report (lambda (condition stream)

```

9–58  Programming Language—Common Lisp
(format stream "Bad tasting sundaes with "S, "S, and "S"
(bad-tasting-sundae-ice-cream condition)
(bad-tasting-sundae-sauce condition)
(bad-tasting-sundae-topping condition))))

→ BAD-TASTING-SUNDAE
(defun all-start-with-same-letter (symbol1 symbol2 symbol3)
  (let ((first-letter (char (symbol-name symbol1) 0)))
    (and (eql first-letter (char (symbol-name symbol2) 0))
         (eql first-letter (char (symbol-name symbol3) 0))))

→ ALL-START-WITH-SAME-LETTER
(defun read-new-value ()
  (format t "Enter a new value: ")
  (multiple-value-list (eval (read))))

→ READ-NEW-VALUE
(defun verify-or-fix-perfect-sundae (ice-cream sauce topping)
  (do ()
    ((all-start-with-same-letter ice-cream sauce topping))
    (restart-case
      (error 'bad-tasting-sundae
        :ice-cream ice-cream
        :sauce sauce
        :topping topping)
      (use-new-ice-cream (new-ice-cream)
        :report "Use a new ice cream."
        :interactive read-new-value
        (setq ice-cream new-ice-cream))
      (use-new-sauce (new-sauce)
        :report "Use a new sauce."
        :interactive read-new-value
        (setq sauce new-sauce))
      (use-new-topping (new-topping)
        :report "Use a new topping."
        :interactive read-new-value
        (setq topping new-topping))))
  (values ice-cream sauce topping))
→ VERIFY-OR-FIX-PERFECT-SUNDAE
(verify-or-fix-perfect-sundae 'vanilla 'caramel 'cherry)
▷ Error: Bad tasting sundae with VANILLA, CARAMEL, and CHERRY.
▷ To continue, type :CONTINUE followed by an option number:
▷ 1: Use a new ice cream.
▷ 2: Use a new sauce.
▷ 3: Use a new topping.
▷ 4: Return to Lisp Toplevel.
▷ Debug> :continue 1
▷ Use a new ice cream.
▷ Enter a new ice cream: 'chocolate
→ CHOCOLATE, CARAMEL, CHERRY

See Also:
restart-bind, with-simple-restart.

Notes:

(restart-case expression
  (name1 arglist1 ...options1... body1)
  (name2 arglist2 ...options2... body2))

is essentially equivalent to
Unnamed restarts are generally only useful interactively and an interactive option which has no
description is of little value. Implementations are encouraged to warn if an unnamed restart is
used and no report information is provided at compilation time. At runtime, this error might be
noticed when entering the debugger. Since signaling an error would probably cause recursive en-
try into the debugger (causing yet another recursive error, etc.) it is suggested that the debugger
print some indication of such problems when they occur but not actually signal errors.

(restart-case (signal fred)
  (a ...)
  (b ...))
≡
(restart-case
  (with-condition-restarts fred
    (list (find-restart 'a)
          (find-restart 'b)))
  (signal fred))
  (a ...)
  (b ...))

restart-name

Function

Syntax:

restart-name restart  → name

Conditions  9–61
Arguments and Values:

restart—a restart.

name—a symbol.

Description:

Returns the name of the restart, or nil if the restart is not named.

Examples:

(restart-case
   (loop for restart in (compute-restarts)
     collect (restart-name restart))
(casel () :report "Return 1." 1)
(nil () :report "Return 2." 2)
(case3 () :report "Return 3." 3)
(casel () :report "Return 4." 4))
→ (CASE1 NIL CASE3 CASE1 ABORT)
;; In the example above the restart named ABORT was not created
;; explicitly, but was implicitly supplied by the system.

See Also:

compute-restarts find-restart

with-condition-restarts  

Macro

Syntax:

with-condition-restarts condition-form restarts-form {form}* → {result}* 

Arguments and Values:

condition-form—a form; evaluated to produce a condition.

condition—a condition object resulting from the evaluation of condition-form.

restart-form—a form; evaluated to produce a restart-list.

restart-list—a list of restart objects resulting from the evaluation of restart-form.

forms—an implicit progn; evaluated.

results—the values returned by forms.
Description:
First, the condition-form and restarts-form are evaluated in normal left-to-right order; the primary values yielded by these evaluations are respectively called the condition and the restart-list.

Next, the forms are evaluated in a dynamic environment in which each restart in restart-list is associated with the condition. See Section 9.1.4.2.4 (Associating a Restart with a Condition).

See Also:
restart-case

Notes:
Usually this macro is not used explicitly in code, since restart-case handles most of the common cases in a way that is syntactically more concise.

with-simple-restart

Syntax:
with-simple-restart (name format-control {format-argument}* ) {form}* → {result}* 

Arguments and Values:
name—a symbol.
format-control—a format control.
format-argument—an object (i.e., a format argument).
forms—an implicit progn.
results—in the normal situation, the values returned by the forms; in the exceptional situation where the restart named name is invoked, two values—nil and t.

Description:
with-simple-restart establishes a restart.

If the restart designated by name is not invoked while executing forms, all values returned by the last of forms are returned. If the restart designated by name is invoked, control is transferred to with-simple-restart, which returns two values, nil and t.

If name is nil, an anonymous restart is established.

The format-control and format-arguments are used report the restart.
with-simple-restart

Examples:

```
(defun read-eval-print-loop (level)
  (with-simple-restart (abort "Exit command level ~D." level)
    (loop
      (with-simple-restart (abort "Return to command level ~D." level)
        (let ((form (prog2 (fresh-line) (read) (fresh-line))))
          (prin1 (eval form)))))))

→ READ-EVAL-PRINT-LOOP
(read-eval-print-loop 1)
(+ 'a 3)
\ Error: The argument, A, to the function + was of the wrong type.
\    The function expected a number.
\ To continue, type :CONTINUE followed by an option number:
\ 1: Specify a value to use this time.
\ 2: Return to command level 1.
\ 3: Exit command level 1.
\ 4: Return to Lisp Toplevel.

(defun compute-fixnum-power-of-2 (x)
  (with-simple-restart (nil "Give up on computing 2^~D." x)
    (let ((result 1))
      (do ((i x result)
        (result (* 2 result))
        (unless (fixnump result)
          (error "Power of 2 is too large."))))))

compute-fixnum-power-of-2
(compute-fixnum-power-of-2 10)
1024
\ Error: Power of 2 is too large.
\ To continue, type :CONTINUE followed by an option number.
\ 1: Give up on computing 2^10000.
\ 2: Return to Lisp Toplevel
\ Debug> :continue 1
\ → SOMETHING-BIG
```

See Also:
restart-case

Notes:

with-simple-restart is shorthand for one of the most common uses of restart-case.
with-simple-restart could be defined by:

```lisp
(defmacro with-simple-restart ((restart-name format-control
    &rest format-arguments)
    &body forms)
  '(restart-case (progn ,@forms)
      (,restart-name ()
        :report (lambda (stream)
          (format stream ,format-control ,@format-arguments))
        (values nil t))))
```

Because the second return value is `t` in the exceptional case, it is common (but not required) to arrange for the second return value in the normal case to be missing or `nil` so that the two situations can be distinguished.

---

**abort**

**Restart**

**Data Arguments Required:**

None.

**Description:**

The intent of the **abort** restart is to allow return to the innermost “command level.” Implementors are encouraged to make sure that there is always a restart named **abort** around any user code so that user code can call **abort** at any time and expect something reasonable to happen; exactly what the reasonable thing is may vary somewhat. Typically, in an interactive listener, the invocation of **abort** returns to the Lisp reader phase of the Lisp read-eval-print loop, though in some batch or multi-processing situations there may be situations in which having it kill the running process is more appropriate.

**See Also:**

Section 9.1.4.2 (Restarts), Section 9.1.4.2.2 (Interfaces to Restarts), invoke-restart, abort (function)
continue

Data Arguments Required:
None.

Description:
The continue restart is generally part of protocols where there is a single “obvious” way to continue, such as in break and cerror. Some user-defined protocols may also wish to incorporate it for similar reasons. In general, however, it is more reliable to design a special purpose restart with a name that more directly suits the particular application.

Examples:

```lisp
(let ((x 3))
  (handler-bind ((error #'(lambda (c)
      (let ((r (find-restart 'continue c)))
        (when r (invoke-restart r))))))
    (cond ((not (floatp x))
       (cerror "Try floating it.","D is not a float." x)
       (float x))
       (t x))) → 3.0
```

See Also:
Section 9.1.4.2 (Restarts), Section 9.1.4.2.2 (Interfaces to Restarts), invoke-restart, continue

muffle-warning

Data Arguments Required:
None.

Description:
This restart is established by warn so that handlers of warning conditions have a way to tell warn that a warning has already been dealt with and that no further action is warranted.

Examples:

```lisp
(defvar *all-quiet* nil) → *ALL-QUIET*
(defvar *saved-warnings* '()) → *SAVED-WARNINGS*
(defun quiet-warning-handler (c)
  (when *all-quiet*
    (let ((r (find-restart 'muffle-warning c))))
```

9–66 Programming Language—Common Lisp
(when r
  (push c *saved-warnings*)
  (invoke-restart r))))
→ CUSTOM-WARNING-HANDLER
(defmacro with-quiet-warnings (&body forms)
  `(let ((*all-quiet* t)
      (*saved-warnings* '()))
    (handler-bind ((warning #'quiet-warning-handler)
                   ,@forms
           *saved-warnings*)))
→ WITH-QUIET-WARNINGS
(setq saved
  (with-quiet-warnings
    (warn "Situation #1.")
    (let ((*all-quiet* nil))
      (warn "Situation #2.")
      (warn "Situation #3.")))
→ Warning: Situation #2.
→ (#<SIMPLE-WARNING 42744421> #<SIMPLE-WARNING 42744365>)
(dolist (s saved) (format t "~&~A~%" s))
→ Situation #3.
→ Situation #1.
→ NIL

See Also:
Section 9.1.4.2 (Restarts), Section 9.1.4.2.2 (Interfaces to Restarts), invoke-restart, muffle-warning (function), warn

---

store-value

**Restart**

Data Arguments Required:
a value to use instead (on an ongoing basis).

**Description:**
The **store-value** restart is generally used by *handlers* trying to recover from errors of *types* such as **cell-error** or **type-error**, which may wish to supply a replacement datum to be stored permanently.

**Examples:**

(defun type-error-auto-coerce (c)
  (when (typep c 'type-error)
    (let ((r (find-restart 'store-value c))))

Conditions  9–67
(handler-case (let ((v (coerce (type-error-datum c)
   (type-error-expected-type c))))
   (invoke-restart r v))
   (error ())))) → TYPE-ERROR-AUTO-COERCE
(let ((x 3))
  (handler-bind ((type-error #'type-error-auto-coerce))
    (check-type x float)
    x)) → 3.0

See Also:
Section 9.1.4.2 (Restarts), Section 9.1.4.2.2 (Interfaces to Restarts), invoke-restart, store-value
(function), ccase, check-type, ctypecase, use-value (function and restart)

use-value

Data Arguments Required:
a value to use instead (once).

Description:
The use-value restart is generally used by handlers trying to recover from errors of types such as
cell-error, where the handler may wish to supply a replacement datum for one-time use.

See Also:
Section 9.1.4.2 (Restarts), Section 9.1.4.2.2 (Interfaces to Restarts), invoke-restart, use-value
(function), store-value (function and restart)

abort, continue, muffle-warning, store-value, use-value

Syntax:
abort &optional condition →|
continue &optional condition → nil
muffle-warning &optional condition →|
store-value value &optional condition → nil
use-value value &optional condition → nil

9–68 Programming Language—Common Lisp
abort, continue, muffle-warning, store-value, use-value

Arguments and Values:

- **value**—an object.
- **condition**—a condition object, or nil.

Description:

Transfers control to the most recently established applicable restart having the same name as the function. That is, the function `abort` searches for an applicable abort restart, the function `continue` searches for an applicable continue restart, and so on.

If no such restart exists, the functions `continue`, `store-value`, and `use-value` return nil, and the functions `abort` and `muffle-warning` signal an error of type `control-error`.

When `condition` is non-nil, only those restarts are considered that are either explicitly associated with that `condition`, or not associated with any `condition`; that is, the excluded restarts are those that are associated with a non-empty set of `conditions` of which the given `condition` is not an element. If `condition` is nil, all restarts are considered.

Examples:

```lisp
;;; Example of the ABORT restart

(defun abort-on-error (&body forms)
  `(handler-bind ((error #'abort))
    ,@forms) → ABORT-ON-ERROR

(abort-on-error (+ 3 5)) → 8
(abort-on-error (error "You lose.
"))
> Returned to Lisp Top Level.

;;; Example of the CONTINUE restart

(defun real-sqrt (n)
  (when (minusp n)
    (setq n (- n))
    (cerror "Return sqrt("D) instead." "Tried to take sqrt(-"D)." n))
  (sqrt n))

(real-sqrt 4) → 2
(real-sqrt -9)
> Error: Tried to take sqrt(-9).
> To continue, type :CONTINUE followed by an option number:
> 1: Return sqrt(9) instead.
> 2: Return to Lisp Toplevel.
Debug> (continue)
> Return sqrt(9) instead.
   → 3
```

Conditions 9–69
**abort, continue, muffle-warning, store-value, use-value**

```
(handler-bind ((error '#(lambda (c) (continue))))
  (real-sqrt -9)) → 3

;;; Example of the MUFFLE-WARNING restart

(defun count-down (x)
  (do ((counter x (1- counter)))
      ((= counter 0) 'done)
    (when (= counter 1)
      (warn "Almost done"))
    (format t "~&D~%" counter)))
→ COUNT-DOWN
(count-down 3)
> 3
> 2
> Warning: Almost done
> 1
→ DONE

(defun ignore-warnings-while-counting (x)
  (handler-bind ((warning #'ignore-warning))
    (count-down x)))
→ IGNORE-WARNINGS-WHILE-COUNTING
(defun ignore-warning (condition)
  (declare (ignore condition))
  (muffle-warning))
→ IGNORE-WARNING
(ignore-warnings-while-counting 3)
> 3
> 2
> 1
→ DONE

;;; Example of the STORE-VALUE and USE-VALUE restarts

(defun careful-symbol-value (symbol)
  (check-type symbol symbol)
  (restart-case (if (boundp symbol)
                   (return-from careful-symbol-value
                     (symbol-value symbol))
                   (error 'unbound-variable
                           :name symbol))
    (use-value (value)
               :report "Specify a value to use this time."
               value))
```
**abort, continue, muffle-warning, store-value, use-value**

```lisp
(defun add-symbols-with-default (default &rest symbols)
  (handler-bind ((sys:unbound-symbol
                  #'(lambda (c)
                      (declare (ignore c))
                      (use-value default))))
    (apply #'+ (mapcar #'careful-symbol-value symbols))))
```

Side Effects:
A transfer of control may occur if an appropriate restart is available, or (in the case of the function abort or the function muffle-warning) execution may be stopped.

Affected By:
Each of these functions can be affected by the presence of a restart having the same name.
Exceptional Situations:

If an appropriate `abort` restart is not available for the function `abort`, or an appropriate `muffle-warning` restart is not available for the function `muffle-warning`, an error of type `control-error` is signaled.

See Also:

invoke-restart, Section 9.1.4.2 (Restarts), Section 9.1.4.2.2 (Interfaces to Restarts), assert, ccase, cerror, check-type, ctypecase, use-value, warn

Notes:

(a) (abort condition) ≡ (invoke-restart 'abort)
(b) (muffle-warning) ≡ (invoke-restart 'muffle-warning)
(c) (continue) ≡ (let ((r (find-restart 'continue))) (if r (invoke-restart r)))
(d) (use-value x) ≡ (let ((r (find-restart 'use-value))) (if r (invoke-restart r x)))
(e) (store-value x) ≡ (let ((r (find-restart 'store-value))) (if r (invoke-restart r x)))

No functions defined in this specification are required to provide a `use-value` restart.
Programming Language—Common Lisp

10. Symbols
10.1 Symbol Concepts

Figure 10–1 lists some defined names that are applicable to the property lists of symbols.

<table>
<thead>
<tr>
<th>get</th>
<th>remprop</th>
<th>symbol-plist</th>
</tr>
</thead>
</table>

Figure 10–1. Property list defined names

Figure 10–2 lists some defined names that are applicable to the creation of and inquiry about symbols.

<table>
<thead>
<tr>
<th>copy-symbol</th>
<th>keywordp</th>
<th>symbol-package</th>
</tr>
</thead>
<tbody>
<tr>
<td>gensym</td>
<td>make-symbol</td>
<td>symbol-value</td>
</tr>
<tr>
<td>gentemp</td>
<td>symbol-name</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10–2. Symbol creation and inquiry defined names
symbol

System Class

Class Precedence List:
symbol, t

Description:
Symbols are used for their object identity to name various entities in Common Lisp, including (but not limited to) linguistic entities such as variables and functions.

Symbols can be collected together into packages. A symbol is said to be interned in a package if it is accessible in that package; the same symbol can be interned in more than one package. If a symbol is not interned in any package, it is called uninterned.

An interned symbol is uniquely identifiable by its name from any package in which it is accessible.

Symbols have the following attributes. For historical reasons, these are sometimes referred to as cells, although the actual internal representation of symbols and their attributes is implementation-dependent.

Name

The name of a symbol is a string used to identify the symbol. Every symbol has a name, and the consequences are undefined if that name is altered. The name is used as part of the external, printed representation of the symbol; see Section 2.1 (Character Syntax).

The function symbol-name returns the name of a given symbol. A symbol may have any character in its name.

Package

The object in this cell is called the home package of the symbol. If the home package is nil, the symbol is sometimes said to have no home package.

When a symbol is first created, it has no home package. When it is first interned, the package in which it is initially interned becomes its home package. The home package of a symbol can be accessed by using the function symbol-package.

If a symbol is uninterned from the package which is its home package, its home package is set to nil. Depending on whether there is another package in which the symbol is interned, the symbol might or might not really be an uninterned symbol. A symbol with no home package is therefore called apparently uninterned.

The consequences are undefined if an attempt is made to alter the home package of a symbol external in the COMMON-LISP package or the KEYWORD package.

Property list

The property list of a symbol provides a mechanism for associating named attributes
with that symbol. The operations for adding and removing entries are destructive to the property list. Common Lisp provides operators both for direct manipulation of property list objects (e.g., see `getf`, `remf`, and `symbol-plist`) and for implicit manipulation of a symbol’s property list by reference to the symbol (e.g., see `get` and `remprop`). The property list associated with a fresh symbol is initially null.

**Value**

If a symbol has a value attribute, it is said to be bound, and that fact can be detected by the function `boundp`. The object contained in the value cell of a bound symbol is the value of the global variable named by that symbol, and can be accessed by the function `symbol-value`. A symbol can be made to be unbound by the function `makunbound`.

The consequences are undefined if an attempt is made to change the value of a symbol that names a constant variable, or to make such a symbol be unbound.

**Function**

If a symbol has a function attribute, it is said to be fbound, and that fact can be detected by the function `fboundp`. If the symbol is the name of a function in the global environment, the function cell contains the function, and can be accessed by the function `symbol-function`. If the symbol is the name of either a macro in the global environment (see `macro-function`) or a special operator (see `special-operator-p`), the symbol is fbound, and can be accessed by the function `symbol-function`, but the object which the function cell contains is of implementation-dependent type and purpose. A symbol can be made to be funbound by the function `fmakunbound`.

The consequences are undefined if an attempt is made to change the functional value of a symbol that names a special form.

Operations on a symbol’s value cell and function cell are sometimes described in terms of their effect on the symbol itself, but the user should keep in mind that there is an intimate relationship between the contents of those cells and the global variable or global function definition, respectively.

Symbols are used as identifiers for lexical variables and lexical function definitions, but in that role, only their object identity is significant. Common Lisp provides no operation on a symbol that can have any effect on a lexical variable or on a lexical function definition.

**See Also:**

Section 2.3.4 (Symbols as Tokens), Section 2.3.1.1 (Potential Numbers as Tokens), Section 22.1.3.3 (Printing Symbols)
keyword

Supertypes:

keyword, symbol, t

Description:

The type keyword includes all symbols interned the KEYWORD package.

Interning a symbol in the KEYWORD package has three automatic effects:

1. It causes the symbol to become bound to itself.
2. It causes the symbol to become an external symbol of the KEYWORD package.
3. It causes the symbol to become a constant variable.

See Also:

keywordp

symbolp

Syntax:

symbolp object → generalized-boolean

Arguments and Values:

object—an object.

generalized-boolean—a generalized boolean.

Description:

Returns true if object is of type symbol; otherwise, returns false.

Examples:

(symbolp 'elephant) → true
(symbolp 12) → false
(symbolp nil) → true
(symbolp '(') → true
(symbolp :test) → true
(symbolp "hello") → false

See Also:

keywordp, symbol, typep
keywordp

Function

Syntax:

\[
\text{keywordp } \text{object} \rightarrow \text{generalized-boolean}
\]

Arguments and Values:

- object—an object.
- generalized-boolean—a generalized boolean.

Description:

Returns true if object is a keyword; otherwise, returns false.

Examples:

- `(keywordp 'elephant) → false`
- `(keywordp 12) → false`
- `(keywordp :test) → true`
- `(keywordp ':test) → true`
- `(keywordp nil) → false`
- `(keywordp :nil) → true`
- `(keywordp '(:test)) → false`
- `(keywordp "hello") → false`
- `(keywordp "hello") → false`
- `(keywordp "optional") → false`

See Also:

- constantp
- keyword
- symbolp
- symbol-package
make-symbol

Function

Syntax:

\[
\text{make-symbol } name \to new-symbol
\]

Arguments and Values:

- \(name\)—a string.
- \(new-symbol\)—a fresh, uninterned symbol.

Description:

\texttt{make-symbol} creates and returns a fresh, uninterned symbol whose name is the given name. The new-symbol is neither bound nor fbound and has a null property list.

It is implementation-dependent whether the string that becomes the new-symbol’s name is the given name or a copy of it. Once a string has been given as the name argument to make-symbol, the consequences are undefined if a subsequent attempt is made to alter that string.

Examples:

\[
\begin{align*}
&(\text{setq temp-string } \text{"temp"}) \rightarrow \text{"temp"} \\
&(\text{setq temp-symbol } (\text{make-symbol temp-string})) \rightarrow \#:\text{temp} \\
&(\text{symbol-name temp-symbol}) \rightarrow \text{"temp"} \\
&(\text{find-symbol } \text{"temp"} \text{ temp-string}) \rightarrow \text{implementation-dependent} \\
&(\text{eq (make-symbol temp-string) temp-string}) \rightarrow \text{NIL, NIL} \\
&(\text{eq (make-symbol temp-string) (make-symbol temp-string)}) \rightarrow \text{false}
\end{align*}
\]

Exceptional Situations:

Should signal an error of type type-error if name is not a string.

See Also:

copy-symbol

Notes:

No attempt is made by \texttt{make-symbol} to convert the case of the name to uppercase. The only case conversion which ever occurs for symbols is done by the Lisp reader. The program interface to symbol creation retains case, and the program interface to interning symbols is case-sensitive.
**copy-symbol**

**Function**

Syntax:

```
copy-symbol symbol &optional copy-properties → new-symbol
```

Arguments and Values:

- `symbol` — a symbol.
- `copy-properties` — a generalized boolean. The default is `false`.
- `new-symbol` — a fresh, uninterned symbol.

Description:

`copy-symbol` returns a fresh, uninterned symbol, the name of which is `string=` to and possibly the same as the name of the given `symbol`.

If `copy-properties` is `false`, the `new-symbol` is neither `bound` nor `fbound` and has a null property list. If `copy-properties` is `true`, then the initial value of `new-symbol` is the value of `symbol`, the initial function definition of `new-symbol` is the functional value of `symbol`, and the property list of `new-symbol` is a copy of the property list of `symbol`.

**Examples:**

```
(setq fred 'fred-smith) → FRED-SMITH
(setf (symbol-value fred) 3) → 3
(setq fred-clone-1a (copy-symbol fred nil)) → #:FRED-SMITH
(setq fred-clone-1b (copy-symbol fred nil)) → #:FRED-SMITH
(setq fred-clone-2a (copy-symbol fred t)) → #:FRED-SMITH
(setq fred-clone-2b (copy-symbol fred t)) → #:FRED-SMITH
(eq fred fred-clone-1a) → false
(eq fred-clone-1a fred-clone-1b) → false
(eq fred-clone-2a fred-clone-2b) → false
(eq fred-clone-1a fred-clone-2a) → false
(symbol-value fred) → 3
(boundp fred-clone-1a) → false
(symbol-value fred-clone-2a) → 3
(setf (symbol-value fred-clone-2a) 4) → 4
(symbol-value fred) → 3
(symbol-value fred-clone-2a) → 4
(symbol-value fred-clone-2b) → 3
(boundp fred-clone-1a) → false
(setf (symbol-function fred) #'(lambda (x) x)) → #<FUNCTION anonymous>
(fbounDP fred) → true
(fbounDP fred-clone-1a) → false
(fbounDP fred-clone-2a) → false
```
Exceptional Situations:
Should signal an error of type type-error if symbol is not a symbol.

See Also:
make-symbol

Notes:
Implementors are encouraged not to copy the string which is the symbol’s name unnecessarily. Unless there is a good reason to do so, the normal implementation strategy is for the new-symbol’s name to be identical to the given symbol’s name.

gensym  Function

Syntax:
gensym &optional x  → new-symbol

Arguments and Values:
x—a string or a non-negative integer. Complicated defaulting behavior; see below.
new-symbol—a fresh, uninterned symbol.

Description:
Creates and returns a fresh, uninterned symbol, as if by calling make-symbol. (The only difference between gensym and make-symbol is in how the new-symbol’s name is determined.)

The name of the new-symbol is the concatenation of a prefix, which defaults to "G", and a suffix, which is the decimal representation of a number that defaults to the value of *gensym-counter*.

If x is supplied, and is a string, then that string is used as a prefix instead of "G" for this call to gensym only.

If x is supplied, and is an integer, then that integer, instead of the value of *gensym-counter*, is used as the suffix for this call to gensym only.

If and only if no explicit suffix is supplied, *gensym-counter* is incremented after it is used.

Examples:
(setq sym1 (gensym))  → #:G3142
(symbol-package sym1) → NIL
(setq sym2 (gensym 100))  → #:G100
(setq sym3 (gensym 100))  → #:G100
(eq sym2 sym3) → false
(find-symbol "G100")  → NIL, NIL
Side Effects:
Might increment *gensym-counter*.

Affected By:
*gensym-counter*

Exceptional Situations:
Should signal an error of type type-error if x is not a string or a non-negative integer.

See Also:
gentemp, *gensym-counter*

Notes:
The ability to pass a numeric argument to gensym has been deprecated; explicitly binding
*gensym-counter* is now stylistically preferred. (The somewhat baroque conventions for the
optional argument are historical in nature, and supported primarily for compatibility with older
dialects of Lisp. In modern code, it is recommended that the only kind of argument used be
a string prefix. In general, though, to obtain more flexible control of the new-symbol’s name,
consider using make-symbol instead.)

*gensym-counter* Variable

Value Type:
a non-negative integer.

Initial Value:
implementation-dependent.

Description:
A number which will be used in constructing the name of the next symbol generated by the
function gensym.

*gensym-counter* can be either assigned or bound at any time, but its value must always be a
non-negative integer.

Affected By:
gensym.

See Also:
gensym
Notes:
The ability to pass a numeric argument to `gensym` has been deprecated; explicitly binding `*gensym-counter*` is now stylistically preferred.

---

gentemp  

Function

Syntax:

`gentemp &optional prefix package  → new-symbol`

Arguments and Values:

- `prefix`—a string. The default is "T".
- `package`—a package designator. The default is the current package.
- `new-symbol`—a fresh, interned symbol.

Description:

`gentemp` creates and returns a fresh symbol, interned in the indicated `package`. The symbol is guaranteed to be one that was not previously accessible in `package`. It is neither `bound` nor `fbound`, and has a null property list.

The `name` of the `new-symbol` is the concatenation of the `prefix` and a suffix, which is taken from an internal counter used only by `gentemp`. (If a symbol by that name is already accessible in `package`, the counter is incremented as many times as is necessary to produce a name that is not already the name of a symbol accessible in `package`.)

Examples:

```
(gentemp) → T1298
(gentemp "FOO") → FOO1299
(find-sym "FOO") → NIL, NIL
(gentemp "FOO") → FOO1300
(find-sym "FOO") → FOO1300, :INTERNAL
(intern "FOO") → FOO1301, :INTERNAL
(gentemp "FOO") → FOO1302
(gentemp) → T1303
```

Side Effects:

Its internal counter is incremented one or more times.

`Interns` the `new-symbol` in `package`.

Affected By:

The current state of its internal counter, and the current state of the `package`.

10–10  Programming Language—Common Lisp
Exceptional Situations:
Should signal an error of type type-error if prefix is not a string. Should signal an error of type type-error if package is not a package designator.

See Also:
gensym

Notes:
The function gentemp is deprecated.
If package is the keyword package, the result is an external symbol of package. Otherwise, the result is an internal symbol of package.
The gentemp internal counter is independent of *gensym-counter*, the counter used by gensym. There is no provision for accessing the gentemp internal counter.
Just because gentemp creates a symbol which did not previously exist does not mean that such a symbol might not be seen in the future (e.g., in a data file—perhaps even created by the same program in another session). As such, this symbol is not truly unique in the same sense as a gensym would be. In particular, programs which do automatic code generation should be careful not to attach global attributes to such generated symbols (e.g., special declarations) and then write them into a file because such global attributes might, in a different session, end up applying to other symbols that were automatically generated on another day for some other purpose.

symbol-function

Syntax:
symbol-function symbol → contents
(setf (symbol-function symbol) new-contents)

Arguments and Values:
symbol—a symbol.
contents—if the symbol is globally defined as a macro or a special operator, an object of implementation-dependent nature and identity is returned. If the symbol is not globally defined as either a macro or a special operator, and if the symbol is fbound, a function object is returned.
new-contents—a function.

Description:
Accesses the symbol's function cell.
symbol-function

Examples:

(symbol-function 'car) → #<FUNCTION CAR>
(symbol-function 'twice) is an error ;because TWICE isn't defined.
(defun twice (n) (* n 2)) → TWICE
(symbol-function 'twice) → #<FUNCTION TWICE>
(list (twice 3)
  (funcall (function twice) 3)
  (funcall (symbol-function 'twice) 3))
→ (6 6 6)
(list (twice x) (list x x))
  (list (twice 3)
    (funcall (function twice) 3)
    (funcall (symbol-function 'twice) 3))
→ ((3 3) (3 3) 6)
(setf (symbol-function 'twice) #'(lambda (x) (list x x)))
  → #<FUNCTION anonymous>
  (list (twice 3)
    (funcall (function twice) 3)
    (funcall (symbol-function 'twice) 3))
→ (((3 3) (3 3) (3 3))
(fboundp 'defun) → true
(symbol-function 'defun)
→ implementation-dependent
(functionp (symbol-function 'defun))
→ implementation-dependent
(defun symbol-function-or-nil (symbol)
  (if (and (fboundp symbol)
    (not (macro-function symbol))
    (not (special-operator-p symbol)))
    (symbol-function symbol)
  nil)) → SYMBOL-FUNCTION-OR-NIL
(symbol-function-or-nil 'car) → #<FUNCTION CAR>
(symbol-function-or-nil 'defun) → NIL

Affected By:
defun

Exceptional Situations:

Should signal an error of type type-error if symbol is not a symbol.

Should signal undefined-function if symbol is not fbound and an attempt is made to read its definition. (No such error is signaled on an attempt to write its definition.)

See Also:

fboundp, fmakunbound, macro-function, special-operator-p
Notes:

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

setf may be used with symbol-function to replace a global function definition when the symbol’s function definition does not represent a special operator.

(symbol-function symbol) ≡ (fdefinition symbol)

However, fdefinition accepts arguments other than just symbols.

symbol-name Function

Syntax:

symbol-name symbol → name

Arguments and Values:

symbol—a symbol.

name—a string.

Description:

symbol-name returns the name of symbol. The consequences are undefined if name is ever modified.

Examples:

(symmetric-name 'temp) → "TEMP"

(symmetric-name :start) → "START"

(symmetric-name (gensym)) → "G1234" ; for example

Exceptional Situations:

Should signal an error of type type-error if symbol is not a symbol.
symbol-package

symbol-package

Function

Syntax:

symbol-package symbol → contents

Arguments and Values:

symbol—a symbol.

contents—a package object or nil.

Description:

Returns the home package of symbol.

Examples:

(in-package "CL-USER") → #<PACKAGE "COMMON-LISP-USER">
(symbol-package 'car) → #<PACKAGE "COMMON-LISP">
(symbol-package 'bus) → #<PACKAGE "COMMON-LISP-USER">
(symbol-package :optional) → #<PACKAGE "KEYWORD">
;; Gensyms are uninterned, so have no home package.
(symbol-package (gensym)) → NIL
(make-package 'pk1) → #<PACKAGE "PK1">
(intern "SAMPLE1" "PK1") → PK1::SAMPLE1, NIL
(export (find-symbol "SAMPLE1" "PK1") "PK1") → T
(make-package 'pk2 :use '(pk1)) → #<PACKAGE "PK2">
(find-symbol "SAMPLE1" "PK2") → PK1::SAMPLE1, :INHERITED
(symbol-package 'pk1::sample1) → #<PACKAGE "PK1">
(symbol-package 'pk2::sample1) → #<PACKAGE "PK1">
(symbol-package 'pk1::sample2) → #<PACKAGE "PK1">
(symbol-package 'pk2::sample2) → #<PACKAGE "PK2">
;; The next several forms create a scenario in which a symbol
;; is not really uninterned, but is "apparently uninterned",
;; and so SYMBOL-PACKAGE still returns NIL.
(setq s3 'pk1::sample3) → PK1::SAMPLE3
(import s3 'pk2) → T
(unintern s3 'pk1) → T
(symbol-package s3) → NIL
(eq s3 'pk2::sample3) → T

Affected By:

import, intern, unintern

Exceptional Situations:

Should signal an error of type type-error if symbol is not a symbol.
symbol-plist

Syntax:

```
symbol-plist symbol  →  plist
```

```
(setf (symbol-plist symbol) new-plist)
```

Arguments and Values:

- `symbol`—a `symbol`.
- `plist`, `new-plist`—a `property list`.

Description:

Accesses the `property list` of `symbol`.

Examples:

```
(setq sym (gensym))  →  #:G9723
(symbol-plist sym)  →  ()
(setf (get sym 'prop1) 'val1)  →  VAL1
(symbol-plist sym)  →  (PROP1 VAL1)
(setf (get sym 'prop2) 'val2)  →  VAL2
(symbol-plist sym)  →  (PROP2 VAL2 PROP1 VAL1)
(setf (symbol-plist sym) (list 'prop3 'val3))  →  (PROP3 VAL3)
(symbol-plist sym)  →  (PROP3 VAL3)
```

Exceptional Situations:

Should signal an error of `type type-error` if `symbol` is not a `symbol`.

See Also:

`get`, `remprop`

Notes:

The use of `setf` should be avoided, since a `symbol’s property list` is a global resource that can contain information established and depended upon by unrelated programs in the same `Lisp image`.

Symbols 10–15
symbol-value

Syntax:

\[
\text{symbol-value } \text{symbol} \rightarrow \text{value}
\]

\[
\text{(setf} \ (\text{symbol-value} \ \text{symbol}) \ \text{new-value})
\]

Arguments and Values:

- **symbol**—a symbol that must have a value.
- **value, new-value**—an object.

Description:

Accesses the symbol’s value cell.

Examples:

\[
\begin{align*}
\text{(setf} \ (\text{symbol-value} \ 'a) \ 1) & \rightarrow 1 \\
(\text{symbol-value} \ 'a) & \rightarrow 1 \\
\text{;; SYMBOL-VALUE cannot see lexical variables.} \\
(\text{let} \ ((a \ 2)) \ (\text{symbol-value} \ 'a)) & \rightarrow 1 \\
(\text{let} \ ((a \ 2)) \ (\text{setq} \ a \ 3) \ (\text{symbol-value} \ 'a)) & \rightarrow 1 \\
\text{;; SYMBOL-VALUE can see dynamic variables.} \\
(\text{let} \ ((a \ 2)) & \\
\ \ \ \ (\text{declare} \ (\text{special} \ a)) \\
\ \ \ \ (\text{symbol-value} \ 'a)) & \rightarrow 2 \\
(\text{let} \ ((a \ 2)) & \\
\ \ \ \ (\text{declare} \ (\text{special} \ a)) \\
\ \ \ \ (\text{setq} \ a \ 3) \\
\ \ \ \ (\text{symbol-value} \ 'a)) & \rightarrow 3 \\
(\text{let} \ ((a \ 2)) & \\
\ \ \ \ (\text{setf} \ (\text{symbol-value} \ 'a) \ 3) \\
\ \ \ \ a) & \rightarrow 2 \\
\ \ \ a & \rightarrow 3 \\
(\text{symbol-value} \ 'a) & \rightarrow 3 \\
(\text{let} \ ((a \ 4)) & \\
\ \ \ \ (\text{declare} \ (\text{special} \ a)) \\
\ \ \ \ (\text{let} \ ((b \ (\text{symbol-value} \ 'a))) & \\
\ \ \ \ \ \ (\text{setf} \ (\text{symbol-value} \ 'a) \ 5) \\
\ \ \ \ \ \ (\text{values} \ a \ b))) & \rightarrow 5, 4 \\
\ \ \ a & \rightarrow 3 \\
(\text{symbol-value} \ :\text{any-keyword}) & \rightarrow :\text{ANY-KEYWORD} \\
(\text{symbol-value} \ \text{nil}) & \rightarrow \text{NIL} \\
(\text{symbol-value} \ '\) & \rightarrow \text{NIL}
\end{align*}
\]

10–16 Programming Language—Common Lisp
 ;; The precision of this next one is implementation-dependent.
(symbol-value 'pi) → 3.141592653589793d0

Affected By:
makunbound, set, setq

Exceptional Situations:
Should signal an error of type type-error if symbol is not a symbol.
Should signal unbound-variable if symbol is unbound and an attempt is made to read its value.
(No such error is signaled on an attempt to write its value.)

See Also:
boundp, makunbound, set, setq

Notes:
symbol-value can be used to get the value of a constant variable. symbol-value cannot access the value of a lexical variable.

get

Syntax:
get symbol indicator &optional default → value
(setf (get symbol indicator &optional default) new-value)

Arguments and Values:
symbol—a symbol.
indicator—an object.
default—an object. The default is nil.
value—if the indicated property exists, the object that is its value; otherwise, the specified default.
new-value—an object.

Description:
get finds a property on the property list2 of symbol whose property indicator is identical to indicator, and returns its corresponding property value. If there are multiple properties1 with that property indicator, get uses the first such property. If there is no property with that property indicator, default is returned.
setf of get may be used to associate a new object with an existing indicator already on the symbol’s property list, or to create a new association if none exists. If there are multiple properties with that property indicator, setf of get associates the new-value with the first such property. When a get form is used as a setf place, any default which is supplied is evaluated according to normal left-to-right evaluation rules, but its value is ignored.

Examples:

(defun make-person (first-name last-name)
  (let ((person (gensym "PERSON")))
    (setf (get person 'first-name) first-name)
    (setf (get person 'last-name) last-name)
    person) → MAKE-PERSON

(defvar *john* (make-person "John" "Dow") → *JOHN*
  *john* → #:PERSON4603

(defvar *sally* (make-person "Sally" "Jones") → *SALLY*
  (get *john* 'first-name) → "John"
  (get *sally* 'last-name) → "Jones"

(defun marry (man woman married-name)
  (setf (get man 'wife) woman)
  (setf (get woman 'husband) man)
  (setf (get man 'last-name) married-name)
  (setf (get woman 'last-name) married-name)
  married-name) → MARRY

(marry *john* *sally* "Dow-Jones") → "Dow-Jones"
  (get *john* 'first-name) → "John"
  (get *sally* 'last-name) → "Sally"
  (symbol-plist *john*)

→ (WIFE #:PERSON4604 LAST-NAME "Dow-Jones" FIRST-NAME "John")

(defmacro age (person &optional (default 'thirty-something))
  ' (get ,person 'age ,default)) → AGE

(age *john*) → THIRTY-SOMETHING
  (age *john* 20) → 20
  (setf (age *john*) 25) → 25
  (age *john*) → 25
  (age *john* 20) → 25

Exceptional Situations:
Should signal an error of type type-error if symbol is not a symbol.

See Also:
getf, symbol-plist, remprop

Notes:

(get x y) ≡ (getf (symbol-plist x) y)
Numbers and characters are not recommended for use as indicators in portable code since get tests with eq rather than eql, and consequently the effect of using such indicators is implementation-dependent.

There is no way using get to distinguish an absent property from one whose value is default. However, see get-properties.

**remprop**  

Function

Syntax:  

\[
\text{remprop} \ symbol \ indicator \rightarrow \text{generalized-boolean}
\]

Arguments and Values:

- **symbol** — a symbol.
- **indicator** — an object.
- **generalized-boolean** — a generalized boolean.

Description:

remprop removes from the property list of symbol a property \(p\) with a property indicator identical to indicator. If there are multiple properties \(p\) with the identical key, remprop only removes the first such property. remprop returns false if no such property was found, or true if a property was found.

The property indicator and the corresponding property value are removed in an undefined order by destructively splicing the property list. The permissible side-effects correspond to those permitted for remf, such that:

\[
(\text{remprop} \ x \ y) \equiv (\text{remf} \ (\text{symbol-plist} \ x) \ y)
\]

Examples:

\[
\begin{align*}
\text{(setq test (make-symbol "PSEUDO-PI"))} & \rightarrow \#:PSEUDO-PI \\
(\text{symbol-plist test}) & \rightarrow () \\
(\text{setf (get test 'constant) t}) & \rightarrow T \\
(\text{setf (get test 'approximation) 3.14}) & \rightarrow 3.14 \\
(\text{setf (get test 'error-range) 'noticeable}) & \rightarrow \text{NOTICEABLE} \\
(\text{symbol-plist test}) & \rightarrow (\text{ERROR-RANGE NOTICEABLE APPROXIMATION 3.14 CONSTANT T}) \\
(\text{setf (get test 'approximation) nil}) & \rightarrow \text{NIL} \\
(\text{symbol-plist test}) & \rightarrow (\text{ERROR-RANGE NOTICEABLE APPROXIMATION NIL CONSTANT T}) \\
(\text{get test 'approximation}) & \rightarrow \text{NIL}
\end{align*}
\]
(remprop test 'approximation) → true
(get test 'approximation) → NIL
(symbol-plist test)
→ (ERROR-RANGE NOTICEABLE CONSTANT T)
(remprop test 'approximation) → NIL
(symbol-plist test)
→ (ERROR-RANGE NOTICEABLE CONSTANT T)
(remprop test 'error-range) → true
(setf (get test 'approximation) 3) → 3
(symbol-plist test)
→ (APPROXIMATION 3 CONSTANT T)

Side Effects:
The *property list* of *symbol* is modified.

Exceptional Situations:
Should signal an error of *type type-error* if *symbol* is not a *symbol*.

See Also:
remf, symbol-plist

Notes:
*Numbers* and *characters* are not recommended for use as *indicators* in portable code since *remprop* tests with *eq* rather than *eql*, and consequently the effect of using such *indicators* is *implementation-dependent*. Of course, if you’ve gotten as far as needing to remove such a *property*, you don’t have much choice—the time to have been thinking about this was when you used *setf* of *get* to establish the *property*.

---

**boundp**

*Function*

Syntax:

`boundp symbol  → generalized-boolean`

Arguments and Values:

`symbol`—a *symbol*.

`generalized-boolean`—a *generalized boolean*.

Description:

Returns *true* if *symbol* is *bound*; otherwise, returns *false*.
Examples:

\[
\begin{align*}
\text{(setq x 1)} & \Rightarrow 1 \\
\text{(boundp 'x)} & \Rightarrow \text{true} \\
\text{(makunbound 'x)} & \Rightarrow \text{X} \\
\text{(boundp 'x)} & \Rightarrow \text{false} \\
\text{(let ((x 2) (boundp 'x)) (false)} & \\
\text{(let ((x 2) (declare (special x))) (boundp 'x))} & \Rightarrow \text{true}
\end{align*}
\]

Exceptional Situations:
Should signal an error of type type-error if symbol is not a symbol.

See Also:
set, setq, symbol-value, makunbound

Notes:
The function bound determines only whether a symbol has a value in the global environment; any lexical bindings are ignored.

makunbound Function

Syntax:

makunbound symbol \rightarrow symbol

Arguments and Values:
symbol—symbol

Description:
Makes the symbol be unbound, regardless of whether it was previously bound.

Examples:

\[
\begin{align*}
\text{(setf (symbol-value 'a) 1)} \\
\text{(boundp 'a)} & \Rightarrow \text{true} \\
\text{a} & \Rightarrow 1 \\
\text{(makunbound 'a)} & \Rightarrow \text{A} \\
\text{(boundp 'a)} & \Rightarrow \text{false}
\end{align*}
\]

Side Effects:
The value cell of symbol is modified.

Exceptional Situations:
Should signal an error of type type-error if symbol is not a symbol.
set

Function

Syntax:

\[
\text{set } symbol \text{ value } \rightarrow value
\]

Arguments and Values:

- symbol—a symbol.
- value—an object.

Description:

set changes the contents of the value cell of symbol to the given value.

\[(\text{set } symbol \text{ value}) \equiv (\text{setf (symbol-value } symbol) \text{ value})\]

Examples:

\[
\begin{align*}
(\text{setf (symbol-value 'n) 1}) & \rightarrow 1 \\
(\text{set 'n 2}) & \rightarrow 2 \\
(\text{symbol-value 'n}) & \rightarrow 2 \\
(\text{let ((n 3))}
  (\text{declare (special n)})
  (\text{setq n (+ n 1)})
  (\text{setf (symbol-value 'n) (* n 10)})
  (\text{set 'n (+ (symbol-value 'n) n)})
  n) & \rightarrow 80 \\
  n & \rightarrow 2 \\
(\text{let ((n 3))}
  (\text{setq n (+ n 1)})
  (\text{setf (symbol-value 'n) (* n 10)})
  (\text{set 'n (+ (symbol-value 'n) n)})
  n) & \rightarrow 4 \\
  n & \rightarrow 44 \\
(\text{defvar *n* 2})
(\text{let ((*n* 3))}
  (\text{setq *n* (+ *n* 1)})
  (\text{setf (symbol-value 'n*) (* n* 10)})
  (\text{set 'n* (+ (symbol-value 'n*) n*)})
  *n* & \rightarrow 80 \\
  *n* & \rightarrow 2
\end{align*}
\]
(defvar *even-count* 0) → *EVEN-COUNT*
(defvar *odd-count* 0) → *ODD-COUNT*
(defun tally-list (list)
  (dolist (element list)
    (set (if (evenp element) '*even-count* '*odd-count*)
         (+ element (if (evenp element) *even-count* *odd-count*)))))))
(tally-list '(1 9 4 3 2 7)) → NIL
*even-count* → 6
*odd-count* → 20

Side Effects:
The value of symbol is changed.

See Also:
setq, progv, symbol-value

Notes:
The function set is deprecated.
set cannot change the value of a lexical variable.

unbound-variable

Class Precedence List:
unbound-variable, cell-error, error, serious-condition, condition, t

Description:
The type unbound-variable consists of error conditions that represent attempts to read the value
of an unbound variable.

The name of the cell (see cell-error) is the name of the variable that was unbound.

See Also:
cell-error-name
Programming Language—Common Lisp

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.

Figure 11–1 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

Packages 11–1
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 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.

11–2 Programming Language—Common Lisp
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 Figure 11–2.

<table>
<thead>
<tr>
<th>Name</th>
<th>Nicknames</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMON-LISP</td>
<td>CL</td>
</tr>
<tr>
<td>COMMON-LISP-USER</td>
<td>CL-USER</td>
</tr>
<tr>
<td>KEYWORD</td>
<td>none</td>
</tr>
</tbody>
</table>

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 stan-
dardized 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 stan-
dardized 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, that are typically used as special markers in programs and their associated data expressions.

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 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 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.)
**package**

*System Class*

**Class Precedence List:**
package, t

**Description:**
A package is a namespace that maps symbol names to symbols; see Section 11.1 (Package Concepts).

**See Also:**
Section 11.1 (Package Concepts), Section 22.1.3.13 (Printing Other Objects), Section 2.3.4 (Symbols as Tokens)

---

**export**

*Function*

**Syntax:**
`export symbols &optional package` → t

**Arguments and Values:**
- `symbols`—a designator for a list of symbols.
- `package`—a package designator. The default is the current package.

**Description:**
`export` makes one or more symbols that are accessible in package (whether directly or by inheritance) be external symbols of that package.

If any of the symbols is already accessible as an external symbol of package, `export` has no effect on that symbol. If the symbol is present in package as an internal symbol, it is simply changed to external status. If it is accessible as an internal symbol via `use-package`, it is first imported into package, then exported. (The symbol is then present in the package whether or not package continues to use the package through which the symbol was originally inherited.)

`export` makes each symbol accessible to all the packages that use package. All of these packages are checked for name conflicts: (export s p) does (find-symbol (symbol-name s) q) for each package q in (package-used-by-list p). Note that in the usual case of an `export` during the initial definition of a package, the result of `package-used-by-list` is nil and the name-conflict checking takes negligible time. When multiple changes are to be made, for example when `export` is given a list of symbols, it is permissible for the implementation to process each change separately, so that aborting from a name conflict caused by any but the first symbol in the list does not unexport the first symbol in the list. However, aborting from a name-conflict error caused by `export` of one of
symbols does not leave that symbol accessible to some packages and inaccessible to others; with respect to each of symbols processed, export behaves as if it were as an atomic operation.

A name conflict in export between one of symbols being exported and a symbol already present in a package that would inherit the newly-exported symbol may be resolved in favor of the exported symbol by uninterning the other one, or in favor of the already-present symbol by making it a shadowing symbol.

Examples:

(make-package 'temp :use nil) → #<PACKAGE "TEMP">
(use-package 'temp) → T
(intern "TEMP-SYM" 'temp) → TEMP::TEMP-SYM, NIL
(find-symbol "TEMP-SYM") → NIL, NIL
(export (find-symbol "TEMP-SYM" 'temp) 'temp) → T
(find-symbol "TEMP-SYM") → TEMP-SYM, :INHERITED

Side Effects:
The package system is modified.

Affected By:
Accessible symbols.

Exceptional Situations:
If any of the symbols is not accessible at all in package, an error of type package-error is signaled that is correctable by permitting the user to interactively specify whether that symbol should be imported.

See Also:
import, unexport, Section 11.1 (Package Concepts)

---

**find-symbol**

*Syntax:*  
```
find-symbol string &optional package  → symbol, status
```

*Arguments and Values:*  
- `string`—a string.
- `package`—a package designator. The default is the current package.
- `symbol`—a symbol accessible in the package, or nil.
- `status`—one of :inherited, :external, :internal, or nil.
find-symbol

Description:

find-symbol locates a symbol whose name is string in a package. If a symbol named string is found in package, directly or by inheritance, the symbol found is returned as the first value; the second value is as follows:

:internal

If the symbol is present in package as an internal symbol.

:external

If the symbol is present in package as an external symbol.

:inherited

If the symbol is inherited by package through use-package, but is not present in package.

If no such symbol is accessible in package, both values are nil.

Examples:

(find-symbol "NEVER-BEFORE-USED") → NIL, NIL
(find-symbol "NEVER-BEFORE-USED") → NIL, NIL
(intern "NEVER-BEFORE-USED") → NEVER-BEFORE-USED, NIL
(intern "NEVER-BEFORE-USED") → NEVER-BEFORE-USED, :INTERNAL
(find-symbol "NEVER-BEFORE-USED") → NEVER-BEFORE-USED, :INTERNAL
(find-symbol "NEVER-BEFORE-USED") → NIL, NIL
(find-symbol "CAR" 'common-lisp-user) → CAR, :INHERITED
(find-symbol "CAR" 'common-lisp) → CAR, :EXTERNAL
(find-symbol "NIL" 'common-lisp-user) → NIL, :INHERITED
(find-symbol "NIL" 'common-lisp) → NIL, :EXTERNAL
(find-symbol "NIL" (prog1 (make-package "JUST-TESTING" :use '())
                        (intern "NIL" "JUST-TESTING")))
  → JUST-TESTING::NIL, :INTERNAL
(export 'just-testing::nil 'just-testing)
(find-symbol "NIL" 'just-testing) → JUST-TESTING:NIL, :EXTERNAL
(find-symbol "NIL" "KEYWORD")
  → NIL, NIL
  or :NIL, :EXTERNAL
(find-symbol (symbol-name :nil) "KEYWORD") → :NIL, :EXTERNAL

Affected By:

intern, import, export, use-package, unintern, unexport, unuse-package

See Also:

intern, find-all-symbols

11–10 Programming Language—Common Lisp
Notes:

find-symbol is operationally equivalent to intern, except that it never creates a new symbol.

---

**find-package**

**Function**

**Syntax:**

\[
\text{find-package} \ name \ \to\ \text{package}
\]

**Arguments and Values:**

- **name** — a string designator or a package object.
- **package** — a package object or nil.

**Description:**

If `name` is a string designator, `find-package` locates and returns the package whose name or nickname is `name`. This search is case sensitive. If there is no such package, `find-package` returns nil.

If `name` is a package object, that package object is returned.

**Examples:**

\[
\begin{align*}
\text{(find-package 'common-lisp)} & \to \text{#<PACKAGE "COMMON-LISP">} \\
\text{(find-package "COMMON-LISP-USER"}) & \to \text{#<PACKAGE "COMMON-LISP-USER"}> \\
\text{(find-package 'not-there)} & \to \text{NIL}
\end{align*}
\]

**Affected By:**

The set of packages created by the implementation.

defpackage, delete-package, make-package, rename-package

**See Also:**

make-package
find-all-symbols

Syntax:

\[\text{find-all-symbols} \ string \ \rightarrow \ symbols\]

Arguments and Values:

- \(\text{string}\) — a string designator.
- \(\text{symbols}\) — a list of symbols.

Description:

\text{find-all-symbols} searches every registered package for symbols that have a name that is the same (under \text{string} =) as \text{string}. A list of all such symbols is returned. Whether or how the list is ordered is implementation-dependent.

Examples:

\[
\begin{align*}
\text{(find-all-symbols} \ '\text{car}) &= \text{(CAR)} \\text{or} \ (\text{CAR VEHICLES:CAR}) \\
&= \text{(VEHICLES:CAR CAR)} \\
\text{(intern} \ "\text{CAR}" \ \text{(make-package} \ '\text{temp :use nil})) &= \text{TEMP::CAR, NIL} \\
\text{(find-all-symbols} \ '\text{car}) &= \text{(TEMP::CAR CAR)} \\text{or} \ (\text{CAR TEMP::CAR}) \\
&= \text{(TEMP::CAR CAR VEHICLES:CAR)} \\
&= \text{(CAR TEMP::CAR VEHICLES:CAR)}
\end{align*}
\]

See Also:

find-symbol

import

Syntax:

\[\text{import} \ symbols \ \&\text{optional package} \ \rightarrow \ t\]

Arguments and Values:

- \(\text{symbols}\) — a designator for a list of symbols.
- \(\text{package}\) — a package designator. The default is the current package.
Description:

`import` adds `symbol` or `symbols` to the internals of `package`, checking for name conflicts with existing `symbols` either `present` in `package` or `accessible` to it. Once the `symbols` have been `imported`, they may be referenced in the `importing package` without the use of a `package prefix` when using the `Lisp reader`.

A name conflict in `import` between the `symbol` being imported and a symbol inherited from some other `package` can be resolved in favor of the `symbol` being `imported` by making it a shadowing symbol, or in favor of the `symbol` already `accessible` by not doing the `import`. A name conflict in `import` with a `symbol` already `present` in the `package` may be resolved by uninterning that `symbol`, or by not doing the `import`.

The imported `symbol` is not automatically exported from the `current package`, but if it is already `present` and external, then the fact that it is external is not changed. If any `symbol` to be `imported` has no home package (i.e., `(symbol-package symbol) → nil`), `import` sets the home `package` of the `symbol` to `package`.

If the `symbol` is already `present` in the importing `package`, `import` has no effect.

Examples:

```lisp
(import 'common-lisp::car (make-package 'temp :use nil)) → T
(find-symbol "CAR" 'temp) → CAR, :INTERNAL
(find-symbol "CDR" 'temp) → NIL, NIL
```

The form `(import 'editor:buffer)` takes the external symbol named `buffer` in the `EDITOR package` (this symbol was located when the form was read by the `Lisp reader`) and adds it to the `current package` as an internal symbol. The symbol `buffer` is then `present` in the `current package`.

Side Effects:

The package system is modified.

Affected By:

Current state of the package system.

Exceptional Situations:

`import` signals a correctable error of type `package-error` if any of the `symbols` to be `imported` has the same name (under `string=`) as some distinct `symbol` (under `eql`) already `accessible` in the `package`, even if the conflict is with a shadowing `symbol` of the `package`.

See Also:

shadow, export
**list-all-packages**

*Function*

**Syntax:**

```lisp
list-all-packages [no arguments] → packages
```

**Arguments and Values:**

- `packages`—a list of `package` objects.

**Description:**

`list-all-packages` returns a fresh list of all registered packages.

**Examples:**

```
(let ((before (list-all-packages)))
  (make-package 'temp)
  (set-difference (list-all-packages) before)) → (#<PACKAGE "TEMP">
```

**Affected By:**

defpackage, delete-package, make-package

---

**rename-package**

*Function*

**Syntax:**

```lisp
rename-package package new-name &optional new-nicknames → package-object
```

**Arguments and Values:**

- `package`—a `package` designator.
- `new-name`—a `package` designator.
- `new-nicknames`—a list of string designators. The default is the empty list.
- `package-object`—the renamed `package` object.

**Description:**

Replaces the name and nicknames of `package`. The old name and all of the old nicknames of `package` are eliminated and are replaced by `new-name` and `new-nicknames`.

The consequences are undefined if `new-name` or any `new-nickname` conflicts with any existing package names.

---

11–14 Programming Language—Common Lisp
Examples:

(make-package 'temporary :nicknames '("TEMP")) → #<PACKAGE "TEMPORARY">
(rename-package 'temp 'ephemeral) → #<PACKAGE "EPHEMERAL">
(package-nicknames (find-package 'ephemeral)) → ()
(find-package 'temporary) → NIL
(rename-package 'ephemeral 'temporary '(temp fleeting))
→ #<PACKAGE "TEMPORARY">
(package-nicknames (find-package 'temp)) → ("TEMP" "FLEETING")

See Also:

make-package

shadow

Function

Syntax:

shadow symbol-names &optional package  → t

Arguments and Values:

symbol-names—a designator for a list of string designators.

package—a package designator. The default is the current package.

Description:

shadow assures that symbols with names given by symbol-names are present in the package.

Specifically, package is searched for symbols with the names supplied by symbol-names. For each such name, if a corresponding symbol is not present in package (directly, not by inheritance), then a corresponding symbol is created with that name, and inserted into package as an internal symbol. The corresponding symbol, whether pre-existing or newly created, is then added, if not already present, to the shadowing symbols list of package.

Examples:

(package-shadowing-symbols (make-package 'temp)) → NIL
(find-symbol 'car 'temp) → CAR, :INHERITED
(shadow 'car 'temp) → T
(find-symbol 'car 'temp) → TEMP::CAR, :INTERNAL
(package-shadowing-symbols 'temp) → (TEMP::CAR)

(make-package 'test-1) → #<PACKAGE "TEST-1">
(intern "TEST" (find-package 'test-1)) → TEST-1::TEST, NIL
(shadow 'test-1::test (find-package 'test-1)) → T
(shadow 'TEST (find-package 'test-1)) → T
(assert (not (null (member 'test-1:test (package-shadowing-symbols
    (find-package 'test-1))))))

(make-package 'test-2) → #<PACKAGE "TEST-2">
(intern "TEST" (find-package 'test-2)) → TEST-2::TEST, NIL
(export 'test-2::test (find-package 'test-2)) → T
(use-package 'test-2 (find-package 'test-1)) ;should not error

Side Effects:
shadow changes the state of the package system in such a way that the package consistency rules
do not hold across the change.

Affected By:
Current state of the package system.

See Also:
package-shadowing-symbols, Section 11.1 (Package Concepts)

Notes:
If a symbol with a name in symbol-names already exists in package, but by inheritance, the
inherited symbol becomes shadowed3 by a newly created internal symbol.

shadowing-import

Function

Syntax:
shadowing-import symbols &optional package → t

Arguments and Values:
symbols—a designator for a list of symbols.
package—a package designator. The default is the current package.

Description:
shadowing-import is like import, but it does not signal an error even if the importation of a
symbol would shadow some symbol already accessible in package.

shadowing-import inserts each of symbols into package as an internal symbol, regardless of
whether another symbol of the same name is shadowed by this action. If a different symbol of the
same name is already present in package, that symbol is first uninterned from package. The new
symbol is added to package's shadowing-symbols list.
shadowing-import does name-conflict checking to the extent that it checks whether a distinct existing symbol with the same name is accessible; if so, it is shadowed by the new symbol, which implies that it must be uninterned if it was present in package.

Examples:

```
(in-package "COMMON-LISP-USER") → #<PACKAGE "COMMON-LISP-USER">
(setq sym (intern "CONFLICT")) → CONFLICT
(intern "CONFLICT" (make-package 'temp)) → TEMP::CONFLICT, NIL
(package-shadowing-symbols 'temp) → NIL
(shadowing-import sym 'temp) → T
(package-shadowing-symbols 'temp) → (CONFLICT)
```

Side Effects:

shadowing-import changes the state of the package system in such a way that the consistency rules do not hold across the change.

package's shadowing-symbols list is modified.

Affected By:

Current state of the package system.

See Also:

import, unintern, package-shadowing-symbols

delete-package

Function

Syntax:

```
delete-package package → generalized-boolean
```

Arguments and Values:

- `package`—a package designator.
- `generalized-boolean`—a generalized boolean.

Description:

delete-package deletes package from all package system data structures. If the operation is successful, delete-package returns true, otherwise nil. The effect of delete-package is that the name and nicknames of package cease to be recognized package names. The package object is still a package (i.e., packagep is true of it) but package-name returns nil. The consequences of deleting the COMMON-LISP package or the KEYWORD package are undefined. The consequences of invoking any other package operation on package once it has been deleted are unspecified. In particular, the consequences of invoking find-symbol, intern and other functions that look for a
**delete-package**

Symbol name in a package are unspecified if they are called with *package* bound to the deleted package or with the deleted package as an argument.

If *package* is a package object that has already been deleted, delete-package immediately returns nil.

After this operation completes, the home package of any symbol whose home package had previously been package is implementation-dependent. Except for this, symbols accessible in package are not modified in any other way; symbols whose home package is not package remain unchanged.

**Examples:**

```lisp
(setq *foo-package* (make-package "FOO" :use nil))
(setq *foo-symbol* (intern "FOO" *foo-package*))
(export *foo-symbol* *foo-package*)

(setq *bar-package* (make-package "BAR" :use '("FOO")))
(setq *bar-symbol* (intern "BAR" *bar-package*))
(export *foo-symbol* *bar-package*)
(export *bar-symbol* *bar-package*)

(setq *baz-package* (make-package "BAZ" :use '("BAR")))

(symbol-package *foo-symbol*) → #<PACKAGE "FOO">
(symbol-package *bar-symbol*) → #<PACKAGE "BAR">

(prin1-to-string *foo-symbol*) → "FOO:FOO"
(prin1-to-string *bar-symbol*) → "BAR:BAR"

(find-symbol "FOO" *bar-package*) → FOO:FOO, :EXTERNAL
(find-symbol "FOO" *baz-package*) → FOO:FOO, :INHERITED
(find-symbol "BAR" *baz-package*) → BAR:BAR, :INHERITED

(packagep *foo-package*) → true
(packagep *bar-package*) → true
(packagep *baz-package*) → true

(package-name *foo-package*) → "FOO"
(package-name *bar-package*) → "BAR"
(package-name *baz-package*) → "BAZ"

(package-use-list *foo-package*) → ()
(package-use-list *bar-package*) → (#<PACKAGE "FOO">)
(package-use-list *baz-package*) → (#<PACKAGE "BAR">)
```
delete-package

(package-used-by-list *foo-package*) → (#<PACKAGE "BAR">
(package-used-by-list *bar-package*) → (#<PACKAGE "BAZ">
(package-used-by-list *baz-package*) → ()

(delete-package *bar-package*)
▷ Error: Package BAZ uses package BAR.
▷ If continued, BAZ will be made to unuse-package BAR,
▷ and then BAR will be deleted.
▷ Type :CONTINUE to continue.
▷ Debug> :CONTINUE
→ T

(symbol-package *foo-symbol*) → #<PACKAGE "FOO">
(symbol-package *bar-symbol*) is unspecified

(prin1-to-string *foo-symbol*) → "FOO:FOO"
(prin1-to-string *bar-symbol*) is unspecified

(find-symbol "FOO" *bar-package*) is unspecified
(find-symbol "FOO" *baz-package*) → NIL, NIL
(find-symbol "BAR" *baz-package*) → NIL, NIL

(packagep *foo-package*) → T
(packagep *bar-package*) → T
(packagep *baz-package*) → T

(package-name *foo-package*) → "FOO"
(package-name *bar-package*) → NIL
(package-name *baz-package*) → "BAZ"

(package-use-list *foo-package*) → ()
(package-use-list *bar-package*) is unspecified
(package-use-list *baz-package*) → ()

(package-used-by-list *foo-package*) → ()
(package-used-by-list *bar-package*) is unspecified
(package-used-by-list *baz-package*) → ()

Exceptional Situations:
If the package designator is a name that does not currently name a package, a correctable error of type package-error is signaled. If correction is attempted, no deletion action is attempted; instead, delete-package immediately returns nil.
If \textit{package} is used by other \textit{packages}, a \textit{correctable} error of type \textit{package-error} is signaled. If correction is attempted, \texttt{unuse-package} is effectively called to remove any dependencies, causing \textit{package}'s \textit{external symbols} to cease being \textit{accessible} to those \textit{packages} that use \textit{package}. \texttt{delete-package} then deletes \textit{package} just as it would have had there been no \textit{packages} that used it.

\textbf{See Also:}

\texttt{unuse-package}

---

\textbf{make-package} \hspace{\textwidth}

\textit{Function}

\textbf{Syntax:}

\[ \text{make-package package-name \&key nicknames use} \rightarrow \text{package} \]

\textbf{Arguments and Values:}

- \textit{package-name}—a \textit{string} designator.
- \textit{nicknames}—a list of \textit{string} designators. The default is the \textit{empty list}.
- \textit{use}—a list of \textit{package} designators. The default is \textit{implementation-defined}.
- \textit{package}—a \textit{package}.

\textbf{Description:}

Creates a new \textit{package} with the name \textit{package-name}.

\textit{Nicknames} are additional \textit{names} which may be used to refer to the new \textit{package}.

\textit{use} specifies zero or more \textit{packages} the \textit{external symbols} of which are to be inherited by the new \textit{package}. See the function \texttt{use-package}.

\textbf{Examples:}

\begin{itemize}
  \item [(make-package 'temporary :nicknames ('"TEMP" "temp")) \rightarrow #<\textit{PACKAGE} "TEMPORARY">]
  \item [(make-package "OWNER" :use ('"temp")) \rightarrow #<\textit{PACKAGE} "OWNER">]
  \item [(package-used-by-list 'temp) \rightarrow (#<\textit{PACKAGE} "OWNER">)]
  \item [(package-use-list 'owner) \rightarrow (#<\textit{PACKAGE} "TEMPORARY">)]
\end{itemize}

\textbf{Affected By:}

The existence of other \textit{packages} in the system.

\textbf{Exceptional Situations:}

The consequences are unspecified if \textit{packages} denoted by \textit{use} do not exist.
A correctable error is signaled if the package-name or any of the nicknames is already the name or nickname of an existing package.

See Also:
defpackage, use-package

Notes:
In situations where the packages to be used contain symbols which would conflict, it is necessary to first create the package with \( \text{use '()} \), then to use shadow or shadowing-import to address the conflicts, and then after that to use use-package once the conflicts have been addressed.

When packages are being created as part of the static definition of a program rather than dynamically by the program, it is generally considered more stylistically appropriate to use defpackage rather than make-package.

### with-package-iterator

**Macro**

**Syntax:**

```lisp
(with-package-iterator (name package-list-form &rest symbol-types) {declaration}* {form}* → {result}*)
```

**Arguments and Values:**

- **name**—a symbol.
- **package-list-form**—a form; evaluated once to produce a package-list.
- **package-list**—a designator for a list of package designators.
- **symbol-type**—one of the symbols :internal, :external, or :inherited.
- **declaration**—a declare expression; not evaluated.
- **forms**—an implicit progns.
- **results**—the values of the forms.

**Description:**

Within the lexical scope of the body forms, the name is defined via macrolet such that successive invocations of (name) will return the symbols, one by one, from the packages in package-list.

It is unspecified whether symbols inherited from multiple packages are returned more than once. The order of symbols returned does not necessarily reflect the order of packages in package-list.

When package-list has more than one element, it is unspecified whether duplicate symbols are returned once or more than once.
Symbol-types controls which symbols that are accessible in a package are returned as follows:

- **:internal**
  The symbols that are present in the package, but that are not exported.

- **:external**
  The symbols that are present in the package and are exported.

- **:inherited**
  The symbols that are exported by used packages and that are not shadowed.

When more than one argument is supplied for symbol-types, a symbol is returned if its accessibility matches any one of the symbol-types supplied. Implementations may extend this syntax by recognizing additional symbol accessibility types.

An invocation of (name) returns four values as follows:

1. A flag that indicates whether a symbol is returned (true means that a symbol is returned).
2. A symbol that is accessible in one the indicated packages.
3. The accessibility type for that symbol; i.e., one of the symbols :internal, :external, or :inherited.
4. The package from which the symbol was obtained. The package is one of the packages present or named in package-list.

After all symbols have been returned by successive invocations of (name), then only one value is returned, namely nil.

The meaning of the second, third, and fourth values is that the returned symbol is accessible in the returned package in the way indicated by the second return value as follows:

- **:internal**
  Means present and not exported.

- **:external**
  Means present and exported.

- **:inherited**
  Means not present (thus not shadowed) but inherited from some used package.

It is unspecified what happens if any of the implicit interior state of an iteration is returned outside the dynamic extent of the with-package-iterator form such as by returning some closure over the invocation form.
with-package-iterator

Any number of invocations of with-package-iterator can be nested, and the body of the innermost one can invoke all of the locally established macros, provided all those macros have distinct names.

Examples:

The following function should return t on any package, and signal an error if the usage of with-package-iterator does not agree with the corresponding usage of do-symbols.

```lisp
(defun test-package-iterator (package)
  (unless (packagep package)
    (setq package (find-package package)))
  (let ((all-entries '())
        (generated-entries '()))
    (do-symbols (x package)
      (multiple-value-bind (symbol accessibility)
          (find-symbol (symbol-name x) package)
        (push (list symbol accessibility) all-entries)))
    (with-package-iterator (generator-fn package :internal :external :inherited)
      (loop
        (multiple-value-bind (more? symbol accessibility pkg)
            (generator-fn)
          (unless more? (return))
          (let ((l (multiple-value-list (find-symbol (symbol-name symbol)
                                              package))))
            (unless (equal l (list symbol accessibility))
              (error "Symbol ~S not found as ~S in package ~A [~S]" symbol accessibility (package-name package) l)))
          (push l generated-entries))))
    (unless (and (subsetp all-entries generated-entries :test #'equal)
                  (subsetp generated-entries all-entries :test #'equal))
      (error "Generated entries and Do-Symbols entries don't correspond"))
  t))
```

The following function prints out every present symbol (possibly more than once):

```lisp
(defun print-all-symbols ()
  (with-package-iterator (next-symbol (list-all-packages)
                        :internal :external)
    (loop
      (multiple-value-bind (more? symbol) (next-symbol)
        (if more?
            (print symbol)
            (return))))))
```

Packages 11–23
Exceptional Situations:

with-package-iterator signals an error of type program-error if no symbol-types are supplied or if a symbol-type is not recognized by the implementation is supplied.

The consequences are undefined if the local function named name established by with-package-iterator is called after it has returned false as its primary value.

See Also:
Section 3.6 (Traversal Rules and Side Effects)

unexport Function

Syntax:

unexport symbols &optional package → t

Arguments and Values:

symbols—a designator for a list of symbols.

package—a package designator. The default is the current package.

Description:

unexport reverts external symbols in package to internal status; it undoes the effect of export.

unexport works only on symbols present in package, switching them back to internal status. If unexport is given a symbol that is already accessible as an internal symbol in package, it does nothing.

Examples:

(in-package "COMMON-LISP-USER") → #<PACKAGE "COMMON-LISP-USER">
(export (intern "CONTRABAND" (make-package 'temp)) 'temp) → T
(find-symbol "CONTRABAND") → NIL, NIL
(use-package 'temp) → T
(find-symbol "CONTRABAND") → CONTRABAND, :INHERITED
(unexport 'contraband 'temp) → T
(find-symbol "CONTRABAND") → NIL, NIL

Side Effects:

Package system is modified.

Affected By:

Current state of the package system.
Exceptional Situations:

If \texttt{unexport} is given a \textit{symbol} not \textit{accessible} in \textit{package} at all, an error of type \texttt{package-error} is signaled.

The consequences are undefined if \textit{package} is the \texttt{KEYWORD} \textit{package} or the \texttt{COMMON-LISP} \textit{package}.

See Also:

\texttt{export}, Section 11.1 (Package Concepts)

\begin{itemize}
\item \texttt{unintern} \hspace{1cm} \textit{Function}
\end{itemize}

\textbf{Syntax:}

\texttt{unintern symbol \&optional package} \hspace{1cm} \rightarrow generalized-boolean

\textbf{Arguments and Values:}

\textit{symbol}—a \textit{symbol}.

\textit{package}—a \textit{package designator}. The default is the \textit{current package}.

\textit{generalized-boolean}—a \textit{generalized boolean}.

\textbf{Description:}

\texttt{unintern} removes \textit{symbol} from \textit{package}. If \textit{symbol} is \textit{present} in \textit{package}, it is removed from \textit{package} and also from \textit{package}'s \textit{shadowing symbols list} if it is present there. If \textit{package} is the \textit{home package} for \textit{symbol}, \textit{symbol} is made to have no \textit{home package}. \textit{Symbol} may continue to be \textit{accessible} in \textit{package} by inheritance.

Use of \texttt{unintern} can result in a \textit{symbol} that has no recorded \textit{home package}, but that in fact is \textit{accessible} in some \textit{package}. Common Lisp does not check for this pathological case, and such \textit{symbols} are always printed preceded by \#:.

\texttt{unintern} returns \texttt{true} if it removes \textit{symbol}, and \texttt{nil} otherwise.

\textbf{Examples:}

\begin{verbatim}
(in-package "COMMON-LISP-USER") \rightarrow #<PACKAGE "COMMON-LISP-USER">
(setq temps-unpack (intern "UNPACK" (make-package 'temp))) \rightarrow TEMP::UNPACK
(unintern temps-unpack 'temp) \rightarrow T
(find-symbol "UNPACK" 'temp) \rightarrow NIL, NIL
temps-unpack \rightarrow #:UNPACK
\end{verbatim}

\textbf{Side Effects:}

\texttt{unintern} changes the state of the package system in such a way that the consistency rules do not hold across the change.
Affected By:
Current state of the package system.

Exceptional Situations:
Giving a shadowing symbol to `unintern` can uncover a name conflict that had previously been resolved by the shadowing. If package A uses packages B and C, A contains a shadowing symbol `x`, and B and C each contain external symbols named `x`, then removing the shadowing symbol `x` from A will reveal a name conflict between `b::x` and `c::x` if those two `symbols` are distinct. In this case `unintern` will signal an error.

See Also:
Section 11.1 (Package Concepts)

---

**in-package**

Macro

Syntax:
```
in-package name  →  package
```

Arguments and Values:
- `name`—a string designator; not evaluated.
- `package`—the package named by `name`.

Description:
Causes the the `package` named by `name` to become the current package—that is, the value of `*package*`. If no such `package` already exists, an error of type `package-error` is signaled.

Everything `in-package` does is also performed at compile time if the call appears as a top level form.

Side Effects:
The `variable` `*package*` is assigned. If the `in-package form` is a top level form, this assignment also occurs at compile time.

Exceptional Situations:
An error of type `package-error` is signaled if the specified `package` does not exist.

See Also:
- `*package*`
unuse-package  

Function

Syntax:

unuse-package packages-to-unuse &optional package  → t

Arguments and Values:

packages-to-unuse—a designator for a list of package designators.

package—a package designator. The default is the current package.

Description:

unuse-package causes package to cease inheriting all the external symbols of packages-to-unuse; unuse-package undoes the effects of use-package. The packages-to-unuse are removed from the use list of package.

Any symbols that have been imported into package continue to be present in package.

Examples:

(in-package "COMMON-LISP-USER") → #<PACKAGE "COMMON-LISP-USER">
(export (intern "SHOES" (make-package 'temp)) 'temp) → T
(find-symbol "SHOES") → NIL, NIL
(use-package 'temp) → T
(find-symbol "SHOES") → SHOES, :INHERITED
(find (find-package 'temp) (package-use-list 'common-lisp-user)) → #<PACKAGE "TEMP">(unuse-package 'temp) → T
(find-symbol "SHOES") → NIL, NIL

Side Effects:

The use list of package is modified.

Affected By:

Current state of the package system.

See Also:

use-package, package-use-list
use-package

Function

Syntax:

use-package packages-to-use &optional package → t

Arguments and Values:

packages-to-use—a designator for a list of package designators. The `KEYWORD` package may not be supplied.

package—a package designator. The default is the current package. The `package` cannot be the `KEYWORD` package.

Description:

use-package causes `package` to inherit all the external symbols of `packages-to-use`. The inherited symbols become accessible as internal symbols of `package`.

Packages-to-use are added to the use list of `package` if they are not there already. All external symbols in packages-to-use become accessible in `package` as internal symbols. use-package does not cause any new symbols to be present in `package` but only makes them accessible by inheritance.

use-package checks for name conflicts between the newly imported symbols and those already accessible in `package`. A name conflict in use-package between two external symbols inherited by `package` from packages-to-use may be resolved in favor of either symbol by importing one of them into `package` and making it a shadowing symbol.

Examples:

(export (intern "LAND-FILL" (make-package 'trash)) 'trash) → T
(find-symbol "LAND-FILL" (make-package 'temp)) → NIL, NIL
(package-use-list 'temp) → (#<PACKAGE "TEMP">)
(use-package 'trash 'temp) → T
(package-use-list 'temp) → (#<PACKAGE "TEMP"> #<PACKAGE "TRASH">
(find-symbol "LAND-FILL" 'temp) → TRASH:LAND-FILL, :INHERITED

Side Effects:

The use list of `package` may be modified.

See Also:

unuse-package, package-use-list, Section 11.1 (Package Concepts)

Notes:

It is permissible for a `package` P₁ to use a `package` P₂ even if P₂ already uses P₁. The using of packages is not transitive, so no problem results from the apparent circularity.

11–28 Programming Language—Common Lisp
**defpackage**

**Macro**

**Syntax:**

```
defpackage  defined-package-name [ | option ]  →  package
```

**Arguments and Values:**

- `defined-package-name`—a string designator.
- `package-name`—a package designator.
- `nickname`—a string designator.
- `symbol-name`—a string designator.
- `package`—the package named `package-name`.

**Description:**

`defpackage` creates a package as specified and returns the package.

If `defined-package-name` already refers to an existing package, the name-to-package mapping for that name is not changed. If the new definition is at variance with the current state of that package, the consequences are undefined; an implementation might choose to modify the existing package to reflect the new definition. If `defined-package-name` is a symbol, its name is used.

The standard options are described below.

- `:nicknames`

  The arguments to `:nicknames` set the package’s nicknames to the supplied names.

- `:documentation`

  The argument to `:documentation` specifies a documentation string; it is attached as a
defpackage

documentation string to the package. At most one documentation option can appear in a single defpackage form.

:use

The arguments to :use set the packages that the package named by package-name will inherit from. If :use is not supplied, it defaults to the same implementation-dependent value as the :use argument to make-package.

:shadow

The arguments to :shadow, symbol-names, name symbols that are to be created in the package being defined. These symbols are added to the list of shadowing symbols effectively as if by shadow.

:shadowing-import-from

The symbols named by the argument symbol-names are found (involving a lookup as if by find-symbol) in the specified package-name. The resulting symbols are imported into the package being defined, and placed on the shadowing symbols list as if by shadowing-import. In no case are symbols created in any package other than the one being defined.

:import-from

The symbols named by the argument symbol-names are found in the package named by package-name and they are imported into the package being defined. In no case are symbols created in any package other than the one being defined.

:export

The symbols named by the argument symbol-names are found or created in the package being defined and exported. The :export option interacts with the :use option, since inherited symbols can be used rather than new ones created. The :export option interacts with the :import-from and :shadowing-import-from options, since imported symbols can be used rather than new ones created. If an argument to the :export option is accessible as an (inherited) internal symbol via use-package, that the symbol named by symbol-name is first imported into the package being defined, and is then exported from that package.

:intern

The symbols named by the argument symbol-names are found or created in the package being defined. The :intern option interacts with the :use option, since inherited symbols can be used rather than new ones created.

:size
defpackage

The argument to the :size option declares the approximate number of symbols expected in the package. This is an efficiency hint only and might be ignored by an implementation.

The order in which the options appear in a defpackage form is irrelevant. The order in which they are executed is as follows:

1. :shadow and :shadowing-import-from.
2. :use.
3. :import-from and :intern.
4. :export.

Shadows are established first, since they might be necessary to block spurious name conflicts when the :use option is processed. The :use option is executed next so that :intern and :export options can refer to normally inherited symbols. The :export option is executed last so that it can refer to symbols created by any of the other options; in particular, shadowing symbols and imported symbols can be made external.

If a defpackage form appears as a top level form, all of the actions normally performed by this macro at load time must also be performed at compile time.

Examples:

```lisp
(defpackage "MY-PACKAGE"
 (:nicknames "MYPKG" "MY-PKG")
 (:use "COMMON-LISP")
 (:shadow "CAR" "CDR")
 (:shadowing-import-from "VENDOR-COMMON-LISP" "CONS")
 (:import-from "VENDOR-COMMON-LISP" "GC")
 (:export "EQ" "CONS" "FROBOLA")
)
```

```lisp
(defpackage my-package
 (:nicknames mypkg :MY-PKG) ; remember Common Lisp conventions for case
 (:use common-lisp) ; conversion on symbols
 (:shadow CAR :cdr #:cons)
 (:export "CONS") ; this is the shadowed one.
)
```

Affected By:
Existing packages.

Exceptional Situations:
If one of the supplied :nicknames already refers to an existing package, an error of type package-error is signaled.
defpackage

An error of type program-error should be signaled if :size or :documentation appears more than once.

Since implementations might allow extended options an error of type program-error should be signaled if an option is present that is not actually supported in the host implementation.

The collection of symbol-name arguments given to the options :shadow, :intern, :import-from, and :shadowing-import-from must all be disjoint; additionally, the symbol-name arguments given to :export and :intern must be disjoint. Disjoint in this context is defined as no two of the symbol-names being string= with each other. If either condition is violated, an error of type program-error should be signaled.

For the :shadowing-import-from and :import-from options, a correctable error of type package-error is signaled if no symbol is accessible in the package named by package-name for one of the argument symbol-names.

Name conflict errors are handled by the underlying calls to make-package, use-package, import, and export. See Section 11.1 (Package Concepts).

See Also:
documentation, Section 11.1 (Package Concepts), Section 3.2 (Compilation)

Notes:
The :intern option is useful if an :import-from or a :shadowing-import-from option in a subsequent call to defpackage (for some other package) expects to find these symbols accessible but not necessarily external.

It is recommended that the entire package definition is put in a single place, and that all the package definitions of a program are in a single file. This file can be loaded before loading or compiling anything else that depends on those packages. Such a file can be read in the COMMON-LISP-USER package, avoiding any initial state issues.

defpackage cannot be used to create two “mutually recursive” packages, such as:

(defpackage my-package
  (:use common-lisp your-package) ;requires your-package to exist first
  (:export "MY-FUN"))
(defpackage your-package
  (:use common-lisp)
  (:import-from my-package "MY-FUN") ;requires my-package to exist first
  (:export "MY-FUN"))

However, nothing prevents the user from using the package-affecting functions such as use-package, import, and export to establish such links after a more standard use of defpackage.

The macroexpansion of defpackage could usefully canonicalize the names into strings, so that even if a source file has random symbols in the defpackage form, the compiled file would only contain strings.
Frequently additional implementation-dependent options take the form of a keyword standing by itself as an abbreviation for a list (keyword T); this syntax should be properly reported as an unrecognized option in implementations that do not support it.

**do-symbols, do-external-symbols, do-all-symbols**

*Macro*

**Syntax:**

```lisp
do-symbols (var [package [result-form]])
{declaration}* {tag | statement}*
→ {result}*  
do-external-symbols (var [package [result-form]])
{declaration}* {tag | statement}*
→ {result}*  
do-all-symbols (var [result-form])
{declaration}* {tag | statement}*
→ {result}*
```

**Arguments and Values:**

- **var**—a variable name; not evaluated.
- **package**—a package designator; evaluated. The default in do-symbols and do-external-symbols is the current package.
- **result-form**—a form; evaluated as described below. The default is nil.
- **declaration**—a declare expression; not evaluated.
- **tag**—a go tag; not evaluated.
- **statement**—a compound form; evaluated as described below.
- **results**—the values returned by the result-form if a normal return occurs, or else, if an explicit return occurs, the values that were transferred.

**Description:**

do-symbols, do-external-symbols, and do-all-symbols iterate over the symbols of packages. For each symbol in the set of packages chosen, the var is bound to the symbol, and the statements in the body are executed. When all the symbols have been processed, result-form is evaluated and returned as the value of the macro.
do-symbols, do-external-symbols, do-all-symbols

**do-symbols** iterates over the symbols accessible in `package`. Statements may execute more than once for symbols that are inherited from multiple packages.

**do-all-symbols** iterates on every registered package. **do-all-symbols** will not process every symbol whatsoever, because a symbol not accessible in any registered package will not be processed. **do-all-symbols** may cause a symbol that is present in several packages to be processed more than once.

**do-external-symbols** iterates on the external symbols of `package`.

When `result-form` is evaluated, `var` is bound and has the value `nil`.

An *implicit block* named `nil` surrounds the entire **do-symbols**, **do-external-symbols**, or **do-all-symbols** form. `return` or `return-from` may be used to terminate the iteration prematurely.

If execution of the body affects which symbols are contained in the set of packages over which iteration is occurring, other than to remove the symbol currently the value of `var` by using `unintern`, the consequences are undefined.

For each of these macros, the scope of the name binding does not include any initial value form, but the optional result forms are included.

Any `tag` in the body is treated as with `tagbody`.

**Examples:**

```lisp
(make-package 'temp :use nil) → #<PACKAGE "TEMP">
(intern "SHY" 'temp) → TEMP::SHY, NIL ;SHY will be an internal symbol
    ;in the package TEMP
(export (intern "BOLD" 'temp) 'temp) → T ;BOLD will be external
(let ((lst ()))
    (do-symbols (s (find-package 'temp)) (push s lst))
    lst))
→ (TEMP::SHY TEMP:BOLD)
or
→ (TEMP:BOLD TEMP::SHY)
(let ((lst ()))
    (do-external-symbols (s (find-package 'temp) lst) (push s lst))
    lst))
→ (TEMP:BOLD)
(let ((lst ()))
    (do-all-symbols (s lst)
        (when (eq (find-package 'temp) (symbol-package s)) (push s lst)))
    lst))
→ (TEMP::SHY TEMP:BOLD)
or
→ (TEMP:BOLD TEMP::SHY)
```

11–34  Programming Language—Common Lisp
See Also:
  intern, export, Section 3.6 (Traversal Rules and Side Effects)

**intern**

*Function*

**Syntax:**

```lisp
intern string &optional package → symbol, status
```

**Arguments and Values:**

- `string`—a string.
- `package`—a package designator. The default is the current package.
- `symbol`—a symbol.
- `status`—one of :inherited, :external, :internal, or nil.

**Description:**

The `intern` function enters a symbol named `string` into `package`. If a symbol whose name is the same as `string` is already accessible in `package`, it is returned. If no such symbol is accessible in `package`, a new symbol with the given name is created and entered into `package` as an internal symbol, or as an external symbol if the `package` is the keyword `package`; `package` becomes the home package of the created symbol.

The first value returned by `intern`, `symbol`, is the symbol that was found or created. The meaning of the secondary value, `status`, is as follows:

- **:internal**
  
  The symbol was found and is present in `package` as an internal symbol.

- **:external**
  
  The symbol was found and is present as an external symbol.

- **:inherited**
  
  The symbol was found and is inherited via use-package (which implies that the symbol is internal).

- **nil**
  
  No pre-existing symbol was found, so one was created.
It is *implementation-dependent* whether the *string* that becomes the new *symbol*’s name is the given *string* or a copy of it. Once a *string* has been given as the *string* argument to *intern* in this situation where a new *symbol* is created, the consequences are undefined if a subsequent attempt is made to alter that *string*.

**Examples:**

```
(in-package "COMMON-LISP-USER") → #<PACKAGE "COMMON-LISP-USER">
(intern "Never-Before") → |Never-Before|, NIL
(intern "Never-Before") → |Never-Before|, :INTERNAL
(intern "NEVER-BEFORE" "KEYWORD") → :NEVER-BEFORE, NIL
(intern "NEVER-BEFORE" "KEYWORD") → :NEVER-BEFORE, :EXTERNAL
```

**See Also:**

find-symbol, read, symbol, unintern, Section 2.3.4 (Symbols as Tokens)

**Notes:**

*intern* does not need to do any name conflict checking because it never creates a new *symbol* if there is already an accessible *symbol* with the name given.

---

**package-name**

*Function*

**Syntax:**

```
package-name package  → name
```

**Arguments and Values:**

- package—a *package designator*.
- name—a *string* or nil.

**Description:**

*package-name* returns the *string* that names *package*, or nil if the *package designator* is a *package object* that has no name (see the function delete-package).

**Examples:**

```
(in-package "COMMON-LISP-USER") → #<PACKAGE "COMMON-LISP-USER">
(package-name *package*) → "COMMON-LISP-USER"
(package-name (symbol-package :test)) → "KEYWORD"
(package-name (find-package 'common-lisp)) → "COMMON-LISP"

(defvar *foo-package* (make-package "FOO"))
```
(rename-package "FOO" "FO00")
(package-name *foo-package*) → "FO00"

Exceptional Situations:
Should signal an error of type type-error if package is not a package designator.

package-nicknames Function

Syntax:
    package-nicknames package → nicknames

Arguments and Values:
    package—a package designator.
    nicknames—a list of strings.

Description:
Returns the list of nickname strings for package, not including the name of package.

Examples:
    (package-nicknames (make-package 'temporary
      :nicknames '("TEMP" "temp"))
    → ("temp" "TEMP")

Exceptional Situations:
Should signal an error of type type-error if package is not a package designator.

package-shadowing-symbols Function

Syntax:
    package-shadowing-symbols package → symbols

Arguments and Values:
    package—a package designator.
    symbols—a list of symbols.
Description:

Returns a list of symbols that have been declared as shadowing symbols in package by shadow or shadowing-import (or the equivalent defpackage options). All symbols on this list are present in package.

Examples:

\[
\begin{align*}
\text{(package-shadowing-symbols (make-package 'temp))} & \rightarrow () \\
\text{(shadow 'cdr 'temp)} & \rightarrow \text{T} \\
\text{(package-shadowing-symbols 'temp)} & \rightarrow (\text{TEMP::CDR}) \\
\text{(intern "PILL" 'temp)} & \rightarrow \text{TEMP::PILL, NIL} \\
\text{(shadowing-import 'pill 'temp)} & \rightarrow \text{T} \\
\text{(package-shadowing-symbols 'temp)} & \rightarrow (\text{PILL TEMP::CDR})
\end{align*}
\]

Exceptional Situations:

Should signal an error of type type-error if package is not a package designator.

See Also:

shadow, shadowing-import

Notes:

Whether the list of symbols is fresh is implementation-dependent.

---

**package-use-list**

*Function*

Syntax:

`package-use-list package → use-list`

Arguments and Values:

- `package`—a package designator.
- `use-list`—a list of package objects.

Description:

Returns a list of other packages used by `package`.

Examples:

\[
\begin{align*}
\text{(package-use-list (make-package 'temp))} & \rightarrow (\text{#<PACKAGE "COMMON-LISP">}) \\
\text{(use-package 'common-lisp-user 'temp)} & \rightarrow \text{T} \\
\text{(package-use-list 'temp)} & \rightarrow (\text{#<PACKAGE "COMMON-LISP">} \text{#<PACKAGE "COMMON-LISP-USER">})
\end{align*}
\]
Exceptional Situations:
Should signal an error of type type-error if package is not a package designator.

See Also:
use-package, unuse-package

### package-used-by-list

**Function**

**Syntax:**

\[
\text{package-used-by-list} \quad \text{package} \rightarrow \text{used-by-list}
\]

**Arguments and Values:**

- **package**—a package designator.
- **used-by-list**—a list of package objects.

**Description:**

package-used-by-list returns a list of other packages that use package.

**Examples:**

\[
\begin{align*}
\text{(package-used-by-list (make-package 'temp))} & \rightarrow () \\
\text{(make-package 'trash :use '(temp))} & \rightarrow \text{ #<PACKAGE "TRASH" >} \\
\text{(package-used-by-list 'temp)} & \rightarrow \text{(#<PACKAGE "TRASH" >)}
\end{align*}
\]

**Exceptional Situations:**

Should signal an error of type type-error if package is not a package.

**See Also:**

use-package, unuse-package
packagep  

Function

Syntax:

\[ \text{packagep object} \rightarrow \text{generalized-boolean} \]

Arguments and Values:

- **object**—an object.
- **generalized-boolean**—a generalized boolean.

Description:

Returns `true` if `object` is of type `package`; otherwise, returns `false`.

Examples:

- `(packagep *package*) \rightarrow true`
- `(packagep 'common-lisp) \rightarrow false`
- `(packagep (find-package 'common-lisp)) \rightarrow true`

Notes:

\((\text{packagep object}) \equiv (\text{typep object 'package})\)

*package*

Variable

Value Type:

a package object.

Initial Value:

the `COMMON-LISP-USER` package.

Description:

Whatever `package` object is currently the value of `*package*` is referred to as the current package.

Examples:

- `(in-package "COMMON-LISP-USER") \rightarrow #<PACKAGE "COMMON-LISP-USER">`
- `*package* \rightarrow #<PACKAGE "COMMON-LISP-USER">`
- `(make-package "SAMPLE-PACKAGE" :use '("COMMON-LISP")) \rightarrow #<PACKAGE "SAMPLE-PACKAGE">`
- `(list

11–40 Programming Language—Common Lisp
(symbol-package
  (let ((*package* (find-package 'sample-package)))
    (setq *some-symbol* (read-from-string "just-testing")))
  *package*)
→ (#<PACKAGE "SAMPLE-PACKAGE"> #<PACKAGE "COMMON-LISP-USER">)
(list (symbol-package (read-from-string "just-testing"))
  *package*)
→ (#<PACKAGE "COMMON-LISP-USER"> #<PACKAGE "COMMON-LISP-USER">)
(eq 'foo (intern "FOO")) → true
(eq 'foo (let ((*package* (find-package 'sample-package)))
            (intern "FOO"))
→ false

Affected By:
  load, compile-file, in-package

See Also:
  compile-file, in-package, load, package

package-error Condition Type

Class Precedence List:
  package-error, error, serious-condition, condition, t

Description:
The type package-error consists of error conditions related to operations on packages. The offending package (or package name) is initialized by the :package initialization argument to make-condition, and is accessed by the function package-error-package.

See Also:
  package-error-package, Chapter 9 (Conditions)
package-error-package

Syntax:
    package-error-package condition → package

Arguments and Values:
    condition—a condition of type package-error.
    package—a package designator.

Description:
    Returns a designator for the offending package in the situation represented by the condition.

Examples:

    (package-error-package
      (make-condition 'package-error
          :package (find-package "COMMON-LISP")))
    → #<Package "COMMON-LISP">

See Also:
    package-error
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.

Figure 12–1 shows operators relating to arithmetic operations.

| *         | 1+        | gcd       |
| +         | 1-        | incf      |
| -         | conjugate | lcm       |
| /         | decf      |           |

Figure 12–1. Operators relating to Arithmetic.

Figure 12–2 shows defined names relating to exponential, logarithmic, and trigonometric operations.

| abs        | cos        | signum    |
| acos       | cosh       | sin       |
| acosh      | exp        | sinh      |
| asin       | expit      | sqrt      |
| asinh      | isqrt      | tan       |
| atan       | log        | tanh      |
| atanh      | phase      |           |
| cis        | pi         |           |

Figure 12–2. Defined names relating to Exponentials, Logarithms, and Trigonometry.

Figure 12–3 shows operators relating to numeric comparison and predication.

| /=         | >=         | oddp      |
| <          | evenp      | plusp     |
| <=         | max        | zerop     |
| =          | min        |           |
| >          | minusp     |           |

Figure 12–3. Operators for numeric comparison and predication.
Figure 12–4 shows *defined names* relating to numeric type manipulation and coercion.

<table>
<thead>
<tr>
<th>ceiling</th>
<th>float-radix</th>
<th>rational</th>
</tr>
</thead>
<tbody>
<tr>
<td>complex</td>
<td>float-sign</td>
<td>rationalize</td>
</tr>
<tr>
<td>decode-float</td>
<td>floor</td>
<td>realpart</td>
</tr>
<tr>
<td>denominator</td>
<td>fround</td>
<td>rem</td>
</tr>
<tr>
<td>fceiling</td>
<td>ftruncate</td>
<td>round</td>
</tr>
<tr>
<td>floor</td>
<td>imagpart</td>
<td>scale-float</td>
</tr>
<tr>
<td>float</td>
<td>integer-decode-float</td>
<td>truncate</td>
</tr>
<tr>
<td>float-digits</td>
<td>mod</td>
<td></td>
</tr>
<tr>
<td>float-precision</td>
<td>numerator</td>
<td></td>
</tr>
</tbody>
</table>

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.

Figure 12–5 shows *defined names* relating to logical operations on numbers.

<table>
<thead>
<tr>
<th>ash</th>
<th>boole-ior</th>
<th>logbitp</th>
</tr>
</thead>
<tbody>
<tr>
<td>boole</td>
<td>boole-nand</td>
<td>logcount</td>
</tr>
<tr>
<td>boole-1</td>
<td>boole-nor</td>
<td>logeqv</td>
</tr>
<tr>
<td>boole-2</td>
<td>boole-orc1</td>
<td>logior</td>
</tr>
<tr>
<td>boole-and</td>
<td>boole-orc2</td>
<td>lognand</td>
</tr>
<tr>
<td>boole-andc1</td>
<td>boole-set</td>
<td>lognor</td>
</tr>
<tr>
<td>boole-andc2</td>
<td>boole-xor</td>
<td>lognot</td>
</tr>
<tr>
<td>boole-c1</td>
<td>integer-length</td>
<td>logorc1</td>
</tr>
<tr>
<td>boole-c2</td>
<td>logand</td>
<td>logorc2</td>
</tr>
<tr>
<td>boole-clr</td>
<td>logandc1</td>
<td>logtest</td>
</tr>
<tr>
<td>boole-eqv</td>
<td>logandc2</td>
<td>logxor</td>
</tr>
</tbody>
</table>

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.

Figure 12–6 shows defined names relating to manipulating bytes of numbers.

<table>
<thead>
<tr>
<th>byte</th>
<th>byte-position</th>
<th>byte-size</th>
</tr>
</thead>
<tbody>
<tr>
<td>deposit-field</td>
<td>dpb</td>
<td>ldb</td>
</tr>
<tr>
<td>ldb-test</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 12–6. Defined names relating to byte manipulation.

12.1.2 Implementation-Dependent Numeric Constants

Figure 12–7 shows defined names relating to implementation-dependent details about numbers.

<table>
<thead>
<tr>
<th>double-float-epsilon</th>
<th>most-negative-fixnum</th>
</tr>
</thead>
<tbody>
<tr>
<td>double-float-negative-epsilon</td>
<td>most-negative-long-float</td>
</tr>
<tr>
<td>least-negative-double-float</td>
<td>most-negative-short-float</td>
</tr>
<tr>
<td>least-negative-long-float</td>
<td>most-negative-single-float</td>
</tr>
<tr>
<td>least-negative-single-float</td>
<td>most-positive-double-float</td>
</tr>
<tr>
<td>least-positive-double-float</td>
<td>most-positive-fixnum</td>
</tr>
<tr>
<td>least-positive-long-float</td>
<td>most-positive-long-float</td>
</tr>
<tr>
<td>least-positive-single-float</td>
<td>most-positive-short-float</td>
</tr>
<tr>
<td>long-float-epsilon</td>
<td>short-float-epsilon</td>
</tr>
<tr>
<td>long-float-negative-epsilon</td>
<td>short-float-negative-epsilon</td>
</tr>
<tr>
<td>most-negative-double-float</td>
<td>single-float-epsilon</td>
</tr>
</tbody>
</table>

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)).
### Table 12–8. Functions Affected by Rule of Float Substitutability

<table>
<thead>
<tr>
<th>Function</th>
<th>Sample Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>((\text{abs #c(3 4)}) \to 5 \text{ or } 5.0)</td>
</tr>
<tr>
<td>acos</td>
<td>((\text{acos 1}) \to 0 \text{ or } 0.0)</td>
</tr>
<tr>
<td>acosh</td>
<td>((\text{acosh 1}) \to 0 \text{ or } 0.0)</td>
</tr>
<tr>
<td>asin</td>
<td>((\text{asin 0}) \to 0 \text{ or } 0.0)</td>
</tr>
<tr>
<td>asinh</td>
<td>((\text{asinh 0}) \to 0 \text{ or } 0.0)</td>
</tr>
<tr>
<td>atan</td>
<td>((\text{atan 0}) \to 0 \text{ or } 0.0)</td>
</tr>
<tr>
<td>atanh</td>
<td>((\text{atanh 0}) \to 0 \text{ or } 0.0)</td>
</tr>
<tr>
<td>cis</td>
<td>((\text{cis 0}) \to 1 \text{ or } #c(1.0 \ 0.0))</td>
</tr>
<tr>
<td>cos</td>
<td>((\text{cos 0}) \to 1 \text{ or } 1.0)</td>
</tr>
<tr>
<td>cosh</td>
<td>((\text{cosh 0}) \to 1 \text{ or } 1.0)</td>
</tr>
<tr>
<td>exp</td>
<td>((\text{exp 0}) \to 1 \text{ or } 1.0)</td>
</tr>
<tr>
<td>expt</td>
<td>((\text{expt 8 1/3}) \to 2 \text{ or } 2.0)</td>
</tr>
<tr>
<td>log</td>
<td>((\text{log 1}) \to 0 \text{ or } 0.0)</td>
</tr>
<tr>
<td></td>
<td>((\text{log 8 2}) \to 3 \text{ or } 3.0)</td>
</tr>
<tr>
<td>phase</td>
<td>((\text{phase 7}) \to 0 \text{ or } 0.0)</td>
</tr>
<tr>
<td>signum</td>
<td>((\text{signum #c(3 4)}) \to #c(3/5 \ 4/5) \text{ or } #c(0.6 \ 0.8))</td>
</tr>
<tr>
<td>sin</td>
<td>((\text{sin 0}) \to 0 \text{ or } 0.0)</td>
</tr>
<tr>
<td>sinh</td>
<td>((\text{sinh 0}) \to 0 \text{ or } 0.0)</td>
</tr>
<tr>
<td>sqrt</td>
<td>((\sqrt{4}) \to 2 \text{ or } 2.0)</td>
</tr>
<tr>
<td></td>
<td>((\sqrt{9/16}) \to 3/4 \text{ or } 0.75)</td>
</tr>
<tr>
<td>tan</td>
<td>((\text{tan 0}) \to 0 \text{ or } 0.0)</td>
</tr>
<tr>
<td>tanh</td>
<td>((\text{tanh 0}) \to 0 \text{ or } 0.0)</td>
</tr>
</tbody>
</table>

### 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

12.1.4.2 Rule of Float Approximation

Computations with floats are only approximate, although they are described as if the results were mathematically accurate. Two mathematically identical expressions may be computationally different because of errors inherent in the floating-point approximation process. The precision of a float is not necessarily correlated with the accuracy of that number. For instance, 3.142857142857142857 is a more precise approximation to π than 3.14159, but the latter is more accurate. The precision refers to the number of bits retained in the representation. When an operation combines a short float with a long float, the result will be a long float. Common Lisp functions assume that the accuracy of arguments to them does not exceed their precision. Therefore when two small floats are combined, the result is a small float. Common Lisp functions never convert automatically from a larger size to a smaller one.

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

When a real and a complex are both part of a computation, the real is first converted to a complex by providing an imaginary part of 0.

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.

Figure 12–9 lists the identities that are obeyed throughout the applicable portion of the complex domain, even on the branch cuts:

<table>
<thead>
<tr>
<th>Function</th>
<th>Identity</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin i z</td>
<td>i sinh z</td>
</tr>
<tr>
<td>sinh i z</td>
<td>i sin z</td>
</tr>
<tr>
<td>cos i z</td>
<td>cosh z</td>
</tr>
<tr>
<td>cosh i z</td>
<td>cos z</td>
</tr>
<tr>
<td>tan i z</td>
<td>i tanh z</td>
</tr>
<tr>
<td>tanh i z</td>
<td>arcsin i z = i arcsinh z</td>
</tr>
<tr>
<td>arcsin i z</td>
<td>arctanh i z</td>
</tr>
<tr>
<td>arcsinh i z</td>
<td>arctan i z = i arctan z</td>
</tr>
</tbody>
</table>

Figure 12–9. Trigonometric Identities for Complex Domain

The quadrant numbers referred to in the discussions of branch cuts are as illustrated in Figure 12–10.

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 $T$. 

Numbers 12–9
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

Figure 12–11 lists some *defined names* that are applicable to *random states*.

<table>
<thead>
<tr>
<th><em>random-state</em></th>
<th>random</th>
</tr>
</thead>
<tbody>
<tr>
<td>make-random-state</td>
<td>random-state-p</td>
</tr>
</tbody>
</table>

Figure 12–11. Random-state defined names
**number**  
*System Class*

**Class Precedence List:**
number, t

**Description:**
The *type number* contains objects which represent mathematical numbers. The *types real* and *complex* are disjoint subtypes of *number*.

The function `=` tests for numerical equality. The function `eql`, when its arguments are both `numbers`, tests that they have both the same *type* and numerical value. Two *numbers* that are the same under `eql` or `=` are not necessarily the same under `eq`.

**Notes:**
Common Lisp differs from mathematics on some naming issues. In mathematics, the set of real numbers is traditionally described as a subset of the complex numbers, but in Common Lisp, the *type real* and the *type complex* are disjoint. The Common Lisp type which includes all mathematical complex numbers is called *number*. The reasons for these differences include historical precedent, compatibility with most other popular computer languages, and various issues of time and space efficiency.

**complex**  
*System Class*

**Class Precedence List:**
complex, number, t

**Description:**
The *type complex* includes all mathematical complex numbers other than those included in the *type rational*. Complexes are expressed in Cartesian form with a real part and an imaginary part, each of which is a *real*. The real part and imaginary part are either both *rational* or both of the same *float* type. The imaginary part can be a *float* zero, but can never be a *rational* zero, for such a number is always represented by Common Lisp as a *rational* rather than a *complex*.

**Compound Type Specifier Kind:**
Specializing.

**Compound Type Specifier Syntax:**

```
(complex [typespec | *])
```
Compound Type Specifier Arguments:

`typespec`—a type specifier that denotes a subtype of type `real`.

Compound Type Specifier Description:

Every element of this type is a `complex` whose real part and imaginary part are each of type `(upgraded-complex-part-type typespec)`. This type encompasses those `complexes` that can result by giving numbers of type `typespec` to `complex`.

`(complex type-specifier)` refers to all `complexes` that can result from giving numbers of type `type-specifier` to the function `complex`, plus all other `complexes` of the same specialized representation.

See Also:
Section 12.1.5.3 (Rule of Canonical Representation for Complex Rationals), Section 2.3.2 (Constructing Numbers from Tokens), Section 22.1.3.1.4 (Printing Complexes)

Notes:

The input syntax for a `complex` with real part `r` and imaginary part `i` is `#C(r i)`. For further details, see Section 2.4 (Standard Macro Characters).

For every `float`, `n`, there is a `complex` which represents the same mathematical number and which can be obtained by `(COERCE n 'COMPLEX)`. 

---

**real**

*System Class*

Class Precedence List:

real, number, t

Description:

The type `real` includes all `numbers` that represent mathematical real numbers, though there are mathematical real numbers (e.g., irrational numbers) that do not have an exact representation in Common Lisp. Only `reals` can be ordered using the `<`, `>`, `<=`, and `>=` functions.

The types `rational` and `float` are disjoint subtypes of type `real`.

Compound Type Specifier Kind:

Abbreviating.

Compound Type Specifier Syntax:

`(real [lower-limit [upper-limit]])`
Compound Type Specifier Arguments:

lower-limit, upper-limit—interval designators for type real. The defaults for each of lower-limit and upper-limit is the symbol *.

Compound Type Specifier Description:

This denotes the reals on the interval described by lower-limit and upper-limit.

---

float

System Class

Class Precedence List:

float, real, number, t

Description:

A float is a mathematical rational (but not a Common Lisp rational) of the form \( s \cdot f \cdot b^e \), where \( s \) is +1 or -1, the sign; \( b \) is an integer greater than 1, the base or radix of the representation; \( p \) is a positive integer, the precision (in base-b digits) of the float; \( f \) is a positive integer between \( b^{p-1} \) and \( b^p - 1 \) (inclusive), the significand; and \( e \) is an integer, the exponent. The value of \( p \) and the range of \( e \) depends on the implementation and on the type of float within that implementation.

In addition, there is a floating-point zero; depending on the implementation, there can also be a “minus zero”. If there is no minus zero, then 0.0 and -0.0 are both interpreted as simply a floating-point zero. (= 0.0 -0.0) is always true. If there is a minus zero, (=eq -0.0 0.0) is false, otherwise it is true.

The types short-float, single-float, double-float, and long-float are subtypes of type float. Any two of them must be either disjoint types or the same type; if the same type, then any other types between them in the above ordering must also be the same type. For example, if the type single-float and the type long-float are the same type, then the type double-float must be the same type also.

Compound Type Specifier Kind:

Abbreviating.

Compound Type Specifier Syntax:

(float [lower-limit [upper-limit]])

Compound Type Specifier Arguments:

lower-limit, upper-limit—interval designators for type float. The defaults for each of lower-limit and upper-limit is the symbol *.

Compound Type Specifier Description:

This denotes the floats on the interval described by lower-limit and upper-limit.
See Also:
Figure 2–9, Section 2.3.2 (Constructing Numbers from Tokens), Section 22.1.3.1.3 (Printing Floats)

Notes:
Note that all mathematical integers are representable not only as Common Lisp reals, but also as complex floats. For example, possible representations of the mathematical number 1 include the integer 1, the float 1.0, or the complex #C(1.0 0.0).

**short-float, single-float, double-float, long-float**  
*Type*

**Supertypes:**
- short-float: short-float, float, real, number, t
- single-float: single-float, float, real, number, t
- double-float: double-float, float, real, number, t
- long-float: long-float, float, real, number, t

**Description:**
For the four defined subtypes of type float, it is true that intermediate between the type short-float and the type long-float are the type single-float and the type double-float. The precise definition of these categories is implementation-defined. The precision (measured in “bits”, computed as \( p \log_2 b \)) and the exponent size (also measured in “bits,” computed as \( \log_2(n + 1) \), where \( n \) is the maximum exponent value) is recommended to be at least as great as the values in Figure 12–12. Each of the defined subtypes of type float might or might not have a minus zero.

<table>
<thead>
<tr>
<th>Format</th>
<th>Minimum Precision</th>
<th>Minimum Exponent Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short</td>
<td>13 bits</td>
<td>5 bits</td>
</tr>
<tr>
<td>Single</td>
<td>24 bits</td>
<td>8 bits</td>
</tr>
<tr>
<td>Double</td>
<td>50 bits</td>
<td>8 bits</td>
</tr>
<tr>
<td>Long</td>
<td>50 bits</td>
<td>8 bits</td>
</tr>
</tbody>
</table>

Figure 12–12. Recommended Minimum Floating-Point Precision and Exponent Size

There can be fewer than four internal representations for floats. If there are fewer distinct representations, the following rules apply:

- If there is only one, it is the type single-float. In this representation, an object is simultaneously of types single-float, double-float, short-float, and long-float.
short-float, single-float, double-float, long-float

- Two internal representations can be arranged in either of the following ways:
  - Two types are provided: single-float and short-float. An object is simultaneously of types single-float, double-float, and long-float.
  - Two types are provided: single-float and double-float. An object is simultaneously of types single-float and short-float, or double-float and long-float.

- Three internal representations can be arranged in either of the following ways:
  - Three types are provided: short-float, single-float, and double-float. An object can simultaneously be of type double-float and long-float.
  - Three types are provided: single-float, double-float, and long-float. An object can simultaneously be of types single-float and short-float.

Compound TypeSpecifier Kind:
Abbreviating.

Compound TypeSpecifier Syntax:
( short-float [short-lower-limit [short-upper-limit]])

( single-float [single-lower-limit [single-upper-limit]])

( double-float [double-lower-limit [double-upper-limit]])

( long-float [long-lower-limit [long-upper-limit]])

Compound TypeSpecifier Arguments:
short-lower-limit, short-upper-limit—interval designators for type short-float. The defaults for each of lower-limit and upper-limit is the symbol *.

single-lower-limit, single-upper-limit—interval designators for type single-float. The defaults for each of lower-limit and upper-limit is the symbol *.

double-lower-limit, double-upper-limit—interval designators for type double-float. The defaults for each of lower-limit and upper-limit is the symbol *.

long-lower-limit, long-upper-limit—interval designators for type long-float. The defaults for each of lower-limit and upper-limit is the symbol *.

Compound TypeSpecifier Description:
Each of these denotes the set of floats of the indicated type that are on the interval specified by the interval designators.
rational  

**System Class**

**Class Precedence List:**
- rational, real, number, t

**Description:**
The canonical representation of a *rational* is as an *integer* if its value is integral, and otherwise as a *ratio*.

The types *integer* and *ratio* are disjoint subtypes of type *rational*.

**Compound TypeSpecifier Kind:**
Abbreviating.

**Compound TypeSpecifier Syntax:**
(rational [lower-limit [upper-limit]])

**Compound TypeSpecifier Arguments:**
- lower-limit, upper-limit—interval designators for type *rational*. The defaults for each of lower-limit and upper-limit is the symbol `*`.

**Compound TypeSpecifier Description:**
This denotes the rationals on the interval described by lower-limit and upper-limit.

---

ratio  

**System Class**

**Class Precedence List:**
- ratio, rational, real, number, t

**Description:**
A *ratio* is a *number* representing the mathematical ratio of two non-zero integers, the numerator and denominator, whose greatest common divisor is one, and of which the denominator is positive and greater than one.

**See Also:**
- Figure 2–9, Section 2.3.2 (Constructing Numbers from Tokens), Section 22.1.3.1.2 (Printing Ratios)

---

12–16 Programming Language—Common Lisp
### integer

**System Class**

**Class Precedence List:**
integer, rational, real, number, t

**Description:**
An integer is a mathematical integer. There is no limit on the magnitude of an integer.

The types fixnum and bignum form an exhaustive partition of type integer.

**Compound Type Specifier Kind:**
Abbreviating.

**Compound Type Specifier Syntax:**
(integer [lower-limit [upper-limit]])

**Compound Type Specifier Arguments:**
lower-limit, upper-limit — interval designators for type integer. The defaults for each of lower-limit and upper-limit is the symbol *.

**Compound Type Specifier Description:**
This denotes the integers on the interval described by lower-limit and upper-limit.

**See Also:**
Figure 2–9, Section 2.3.2 (Constructing Numbers from Tokens), Section 22.1.3.1.1 (Printing Integers)

**Notes:**
The type (integer lower upper), where lower and upper are most-negative-fixnum and most-positive-fixnum, respectively, is also called fixnum.

The type (integer 0 1) is also called bit. The type (integer 0 *) is also called unsigned-byte.

### signed-byte

**Type**

**Supertypes:**
signed-byte, integer, rational, real, number, t

**Description:**
The atomic type specifier signed-byte denotes the same type as is denoted by the type specifier integer; however, the list forms of these two type specifiers have different semantics.
Compound Type Specifier Kind: Abbreviating.

Compound Type Specifier Syntax: 
(signed-byte [s | *])

Compound Type Specifier Arguments: 
- s—a positive integer.

Compound Type Specifier Description: 
This denotes the set of integers that can be represented in two’s-complement form in a byte of s bits. This is equivalent to (integer \( -2^{s-1} \cdot 2^{s-1} - 1 \)). The type signed-byte or the type (signed-byte *) is the same as the type integer.

---

unsigned-byte

Supertypes: 
unsigned-byte, signed-byte, integer, rational, real, number, t

Description:
The atomic type specifier unsigned-byte denotes the same type as is denoted by the type specifier (integer 0 *).

Compound Type Specifier Kind: Abbreviating.

Compound Type Specifier Syntax: 
(unsigned-byte [s | *])

Compound Type Specifier Arguments: 
- s—a positive integer.

Compound Type Specifier Description: 
This denotes the set of non-negative integers that can be represented in a byte of size s (bits). This is equivalent to \( (\text{mod} \ m) \) for \( m = 2^s \), or to \( (\text{integer} \ 0 \ n) \) for \( n = 2^s - 1 \). The type unsigned-byte or the type (unsigned-byte *) is the same as the type (integer 0 *), the set of non-negative integers.

Notes: 
The type (unsigned-byte 1) is also called bit.
**mod**

*Type Specifier*

**Compound Type Specifier Kind:**
Abbreviating.

**Compound Type Specifier Syntax:**
(mod n)

**Compound Type Specifier Arguments:**

n—a positive integer.

**Compound Type Specifier Description:**

This denotes the set of non-negative integers less than n. This is equivalent to (integer 0 (n)) or to (integer 0 m), where m = n - 1.

The argument is required, and cannot be *.

The symbol mod is not valid as a *type specifier*.

---

**bit**

*Type*

**Supertypes:**

bit, unsigned-byte, signed-byte, integer, rational, real, number, t

**Description:**

The type bit is equivalent to the type (integer 0 1) and (unsigned-byte 1).
fixnum

Supertypes:
fixnum, integer, rational, real, number, t

Description:
A fixnum is an integer whose value is between most-negative-fixnum and most-positive-fixnum inclusive. Exactly which integers are fixnums is implementation-defined. The type fixnum is required to be a supertype of (signed-byte 16).

bignum

Supertypes:
bignum, integer, rational, real, number, t

Description:
The type bignum is defined to be exactly (and integer (not fixnum)).

==, /=, <, >, <=, >=

Syntax:
== krest numbers+ → generalized-boolean
/= krest numbers+ → generalized-boolean
< krest numbers+ → generalized-boolean
> krest numbers+ → generalized-boolean
<= krest numbers+ → generalized-boolean
>= krest numbers+ → generalized-boolean

Arguments and Values:
number—for <, >, <=, >=: a real; for ==, /=: a number.
generalized-boolean—a generalized boolean.
Description:

\(=, /=, \lt, \gt, \leq, \geq\) perform arithmetic comparisons on their arguments as follows:

\(=\)

The value of \(=\) is \textit{true} if all \textit{numbers} are the same in value; otherwise it is \textit{false}. Two \textit{complexes} are considered equal by \(=\) if their real and imaginary parts are equal according to \(=\).

\(/=\)

The value of \(/=\) is \textit{true} if no two \textit{numbers} are the same in value; otherwise it is \textit{false}.

\(<\)

The value of \(<\) is \textit{true} if the \textit{numbers} are in monotonically increasing order; otherwise it is \textit{false}.

\(\gt\)

The value of \(\gt\) is \textit{true} if the \textit{numbers} are in monotonically decreasing order; otherwise it is \textit{false}.

\(\leq\)

The value of \(\leq\) is \textit{true} if the \textit{numbers} are in monotonically nondecreasing order; otherwise it is \textit{false}.

\(\geq\)

The value of \(\geq\) is \textit{true} if the \textit{numbers} are in monotonically nonincreasing order; otherwise it is \textit{false}.

\(=, /=, \lt, \gt, \leq, \geq\) perform necessary type conversions.

Examples:

The uses of these functions are illustrated in Figure 12–13.
Figure 12–13. Uses of /=, =, <, >, <=, and >=

Exceptional Situations:
Might signal type-error if some argument is not a real. Might signal arithmetic-error if otherwise unable to fulfill its contract.

Notes:
= differs from eql in that (= 0.0 -0.0) is always true, because = compares the mathematical values of its operands, whereas eql compares the representational values, so to speak.

max, min

Function

Syntax:
max &rest reals+ → max-real

12–22 Programming Language—Common Lisp
**max, min**

\[ \text{min \ \&rest \ reals^+ \rightarrow min\text{-real}} \]

**Arguments and Values:**

- \( \text{real} \) — a \text{real}.
- \( \text{max-\text{real, min-\text{real}} - a \text{real}} \).

**Description:**

**max** returns the \text{real} that is greatest (closest to positive infinity). \text{min} returns the \text{real} that is least (closest to negative infinity).

For \text{max}, the implementation has the choice of returning the largest argument as is or applying the rules of floating-point contagion, taking all the arguments into consideration for contagion purposes. Also, if one or more of the arguments are \( = \), then any one of them may be chosen as the value to return. For example, if the \text{reals} are a mixture of \text{rationals} and \text{floats}, and the largest argument is a \text{rational}, then the implementation is free to produce either that \text{rational} or its \text{float} approximation; if the largest argument is a \text{float} of a smaller format than the largest format of any \text{float} argument, then the implementation is free to return the argument in its given format or expanded to the larger format. Similar remarks apply to \text{min} (replacing “largest argument” by “smallest argument”).

**Examples:**

- \((\text{max } 3) \rightarrow 3\)
- \((\text{min } 3) \rightarrow 3\)
- \((\text{max } 6 \ 12) \rightarrow 12\)
- \((\text{min } 6 \ 12) \rightarrow 6\)
- \((\text{max } -6 \ -12) \rightarrow -6\)
- \((\text{min } -6 \ -12) \rightarrow -12\)
- \((\text{max } 1 \ 3 \ 2 \ -7) \rightarrow 3\)
- \((\text{min } 1 \ 3 \ 2 \ -7) \rightarrow -7\)
- \((\text{max } -2 \ 3 \ 0 \ 7) \rightarrow 7\)
- \((\text{min } -2 \ 3 \ 0 \ 7) \rightarrow -2\)
- \((\text{max } 5.0 \ 2) \rightarrow 5.0\)
- \((\text{min } 5.0 \ 2) \rightarrow 2\)
- \(\rightarrow 2.0\)
- \((\text{max } 3.0 \ 7 \ 1) \rightarrow 7\)
- \(\rightarrow 7.0\)
- \((\text{min } 3.0 \ 7 \ 1) \rightarrow 1\)
- \(\rightarrow 1.0\)
- \((\text{max } 1.0s0 \ 7.0d0) \rightarrow 7.0d0\)
(min 1.0s0 7.0d0)
→ 1.0s0
or
→ 1.0d0
(max 3 1.0s0 1.0d0)
→ 3
or
→ 3.0d0
(min 3 1.0s0 1.0d0)
→ 1
or
→ 1.0s0
or
→ 1.0d0

Exceptional Situations:
Should signal an error of type type-error if any number is not a real.

minusp, plusp

Function

Syntax:
minusp real → generalized-boolean
plusp real → generalized-boolean

Arguments and Values:
real—a real.
generalized-boolean—a generalized boolean.

Description:
minusp returns true if real is less than zero; otherwise, returns false.
plusp returns true if real is greater than zero; otherwise, returns false.

Regardless of whether an implementation provides distinct representations for positive and negative float zeros, (minusp -0.0) always returns false.

Examples:
(minusp -1) → true
(plusp 0) → false
(plusp least-positive-single-float) → true

Exceptional Situations:
Should signal an error of type type-error if real is not a real.
zerop

Function

Syntax:

zerop number → generalized-boolean

Pronunciation:

[ˈzɛr(ə)p]ē]

Arguments and Values:

number—a number.

generalized-boolean—a generalized boolean.

Description:

Returns true if number is zero (integer, float, or complex); otherwise, returns false.

Regardless of whether an implementation provides distinct representations for positive and negative floating-point zeros, (zerop -0.0) always returns true.

Examples:

(zerop 0) → true
(zerop 1) → false
(zerop -0.0) → true
(zerop 0/100) → true
(zerop #c(0 0.0)) → true

Exceptional Situations:

Should signal an error of type type-error if number is not a number.

Notes:

(zerop number) ≡ (= number 0)

floor, ffloor, ceiling, fceiling, truncate, ftruncate, round, fround

Function

Syntax:

floor number &optional divisor → quotient, remainder

ffloor number &optional divisor → quotient, remainder

ceiling number &optional divisor → quotient, remainder

Numbers 12–25
floor, ffloor, ceiling, fceiling, truncate, ftruncate, ...

\[ \text{fceiling} \text{ number} \&\text{optional divisor} \rightarrow \text{quotient, remainder} \]
\[ \text{truncate} \text{ number} \&\text{optional divisor} \rightarrow \text{quotient, remainder} \]
\[ \text{ftruncate} \text{ number} \&\text{optional divisor} \rightarrow \text{quotient, remainder} \]
\[ \text{round} \text{ number} \&\text{optional divisor} \rightarrow \text{quotient, remainder} \]
\[ \text{fround} \text{ number} \&\text{optional divisor} \rightarrow \text{quotient, remainder} \]

Arguments and Values:
- number—a real.
- divisor—a non-zero real. The default is the integer 1.
- quotient—for floor, ceiling, truncate, and round: an integer; for ffloor, fceiling, ftruncate, and fround: a float.
- remainder—a real.

Description:
These functions divide number by divisor, returning a quotient and remainder, such that
\[ \text{quotient} \cdot \text{divisor} + \text{remainder} = \text{number} \]
The quotient always represents a mathematical integer. When more than one mathematical integer might be possible (i.e., when the remainder is not zero), the kind of rounding or truncation depends on the operator:

- **floor, ffloor**
  - floor and ffloor produce a quotient that has been truncated toward negative infinity; that is, the quotient represents the largest mathematical integer that is not larger than the mathematical quotient.

- **ceiling, fceiling**
  - ceiling and fceiling produce a quotient that has been truncated toward positive infinity; that is, the quotient represents the smallest mathematical integer that is not smaller than the mathematical result.

- **truncate, ftruncate**
  - truncate and ftruncate produce a quotient that has been truncated towards zero; that is, the quotient represents the mathematical integer of the same sign as the mathematical quotient, and that has the greatest integral magnitude not greater than that of the mathematical quotient.

- **round, fround**
  - round and fround produce a quotient that has been rounded to the nearest mathematical integer; if the mathematical quotient is exactly halfway between two integers, (that is,
floor, ffloor, ceiling, fceiling, truncate, ftruncate, ...

it has the form integer\(+\frac{1}{2}\), then the quotient has been rounded to the even (divisible by two) integer.

All of these functions perform type conversion operations on numbers.
The remainder is an integer if both \(x\) and \(y\) are integers, is a rational if both \(x\) and \(y\) are rationals, and is a float if either \(x\) or \(y\) is a float.

ffloor, fceiling, ftruncate, and fround handle arguments of different types in the following way: If number is a float, and divisor is not a float of longer format, then the first result is a float of the same type as number. Otherwise, the first result is of the type determined by contagion rules; see Section 12.1.1.2 (Contagion in Numeric Operations).

Examples:

(floor 3/2) \rightarrow 1, 1/2
(ceiling 3 2) \rightarrow 2, -1
(ffloor 3 2) \rightarrow 1.0, 1
(ffloor -4.7) \rightarrow -5.0, 0.3
(ffloor 3.5d0) \rightarrow 3.0d0, 0.5d0
(fceiling 3/2) \rightarrow 2.0, -1/2
(truncate 1) \rightarrow 1, 0
(truncate .5) \rightarrow 0, 0.5
(round .5) \rightarrow 0, 0.5
(ftruncate -7 2) \rightarrow -3.0, -1
(fround -7 2) \rightarrow -4.0, 1

(dolist (n '(2.6 2.5 2.4 0.7 0.3 -0.3 -0.7 -2.4 -2.5 -2.6))
      (format t "~&~4,1@F ~2,' D ~2,' D ~2,' D ~2,' D"
              n (floor n) (ceiling n) (truncate n) (round n)))
> +2.6  2  3  2  3
> +2.5  2  3  2  2
> +2.4  2  3  2  2
> +0.7  0  1  0  1
> +0.3  0  1  0  0
> -0.3  -1  0  0  0
> -0.7  -1  0  0  -1
> -2.4  -3  -2  -2  -2
> -2.5  -3  -2  -2  -2
> -2.6  -3  -2  -2  -3
→ NIL

Notes:
When only number is given, the two results are exact; the mathematical sum of the two results is always equal to the mathematical value of number.

(function number divisor) and (function (\(/\) number divisor)) (where function is any of one of floor, ceiling, ffloor, fceiling, truncate, round, ftruncate, and fround) return the same first
value, but they return different remainders as the second value. For example:

(fLOOR 5 2) → 2, 1
(fLOOR (/ 5 2)) → 2, 1/2

If an effect is desired that is similar to round, but that always rounds up or down (rather than toward the nearest even integer) if the mathematical quotient is exactly halfway between two integers, the programmer should consider a construction such as (FLOOR (+ X 1/2)) or (CEILING (- X 1/2)).

**sin, cos, tan**

*Function*

**Syntax:**

\[
\begin{align*}
\text{sin} & \quad \text{radians} & \rightarrow & \text{number} \\
\text{cos} & \quad \text{radians} & \rightarrow & \text{number} \\
\text{tan} & \quad \text{radians} & \rightarrow & \text{number}
\end{align*}
\]

**Arguments and Values:**

- **radians**—a number given in radians.
- **number**—a number.

**Description:**

\(\text{sin, cos, and tan}\) return the sine, cosine, and tangent, respectively, of \(\text{radians}\).

**Examples:**

\[
\begin{align*}
\text{(sin 0)} & \rightarrow 0.0 \\
\text{(cos 0.7853982)} & \rightarrow 0.707107 \\
\text{(tan #C(0 1))} & \rightarrow #C(0.0 0.761594)
\end{align*}
\]

**Exceptional Situations:**

Should signal an error of type type-error if \(\text{radians}\) is not a number. Might signal arithmetic-error.

**See Also:**

asin, acos, atan, Section 12.1.3.3 (Rule of Float Substitutability)
**asin, acos, atan**

**Syntax:**

- `asin number → radians`
- `acos number → radians`
- `atan number1 &optional number2 → radians`

**Arguments and Values:**

- `number`—a `number`.
- `number1`—a `number` if `number2` is not supplied, or a `real` if `number2` is supplied.
- `number2`—a `real`.
- `radians`—a `number` (of radians).

**Description:**

- `asin`, `acos`, and `atan` compute the arc sine, arc cosine, and arc tangent respectively.

The arc sine, arc cosine, and arc tangent (with only `number1` supplied) functions can be defined mathematically for `number` or `number1` specified as $x$ as in Figure 12–14.

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arc sine</td>
<td>$-i \log (ix + \sqrt{1 - x^2})$</td>
</tr>
<tr>
<td>Arc cosine</td>
<td>$(\pi/2) - \arcsin x$</td>
</tr>
<tr>
<td>Arc tangent</td>
<td>$-i \log ((1 + ix) \sqrt{1/(1 + x^2)})$</td>
</tr>
</tbody>
</table>

Figure 12–14. Mathematical definition of arc sine, arc cosine, and arc tangent

These formulae are mathematically correct, assuming completely accurate computation. They are not necessarily the simplest ones for real-valued computations.

If both `number1` and `number2` are supplied for `atan`, the result is the arc tangent of `number1/number2`. The value of `atan` is always between $-\pi$ (exclusive) and $\pi$ (inclusive) when minus zero is not supported. The range of the two-argument arc tangent when minus zero is supported includes $-\pi$.

For a `real` `number1`, the result is a `real` and lies between $-\pi/2$ and $\pi/2$ (both exclusive). `number1` can be a `complex` if `number2` is not supplied. If both are supplied, `number2` can be zero provided `number1` is not zero.

The following definition for arc sine determines the range and branch cuts:
asin, acos, atan

\[ \text{arcsin } z = -i \log \left( iz + \sqrt{1 - z^2} \right) \]

The branch cut for the arc sine function is in two pieces: one along the negative real axis to the left of \(-1\) (inclusive), continuous with quadrant II, and one along the positive real axis to the right of 1 (inclusive), continuous with quadrant IV. The range is that strip of the complex plane containing numbers whose real part is between \(-\pi/2\) and \(\pi/2\). A number with real part equal to \(-\pi/2\) is in the range if and only if its imaginary part is non-negative; a number with real part equal to \(\pi/2\) is in the range if and only if its imaginary part is non-positive.

The following definition for arc cosine determines the range and branch cuts:

\[ \text{arccos } z = \frac{\pi}{2} - \text{arcsin } z \]

or, which are equivalent,

\[ \text{arccos } z = -i \log \left( z + i \sqrt{1 - z^2} \right) \]

\[ \text{arccos } z = \frac{2 \log \left( \sqrt{(1+z)/2} + i \sqrt{(1-z)/2} \right)}{i} \]

The branch cut for the arc cosine function is in two pieces: one along the negative real axis to the left of \(-1\) (inclusive), continuous with quadrant II, and one along the positive real axis to the right of 1 (inclusive), continuous with quadrant IV. This is the same branch cut as for arc sine. The range is that strip of the complex plane containing numbers whose real part is between 0 and \(\pi\). A number with real part equal to 0 is in the range if and only if its imaginary part is non-negative; a number with real part equal to \(\pi\) is in the range if and only if its imaginary part is non-positive.

The following definition for (one-argument) arc tangent determines the range and branch cuts:

\[ \text{arctan } z = \frac{\log (1 + iz) - \log (1 - iz)}{2i} \]

Beware of simplifying this formula; “obvious” simplifications are likely to alter the branch cuts or the values on the branch cuts incorrectly. The branch cut for the arc tangent function is in two pieces: one along the positive imaginary axis above \(i\) (exclusive), continuous with quadrant II, and one along the negative imaginary axis below \(-i\) (exclusive), continuous with quadrant IV. The points \(i\) and \(-i\) are excluded from the domain. The range is that strip of the complex plane containing numbers whose real part is between \(-\pi/2\) and \(\pi/2\). A number with real part equal to \(-\pi/2\) is in the range if and only if its imaginary part is strictly positive; a number with real part equal to \(\pi/2\) is in the range if and only if its imaginary part is strictly negative. Thus the range of arc tangent is identical to that of arc sine with the points \(-\pi/2\) and \(\pi/2\) excluded.
For `atan`, the signs of `number1` (indicated as `x`) and `number2` (indicated as `y`) are used to derive quadrant information. Figure 12–15 details various special cases. The asterisk (*) indicates that the entry in the figure applies to implementations that support minus zero.

<table>
<thead>
<tr>
<th>y Condition</th>
<th>x Condition</th>
<th>Cartesian locus</th>
<th>Range of result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>y = 0</code></td>
<td><code>x &gt; 0</code></td>
<td>Positive x-axis</td>
<td><code>0</code></td>
</tr>
<tr>
<td>* <code>y = +0</code></td>
<td><code>x &gt; 0</code></td>
<td>Positive x-axis</td>
<td><code>+0</code></td>
</tr>
<tr>
<td>* <code>y = -0</code></td>
<td><code>x &gt; 0</code></td>
<td>Positive x-axis</td>
<td><code>−0</code></td>
</tr>
<tr>
<td><code>y &gt; 0</code></td>
<td><code>x &gt; 0</code></td>
<td>Quadrant I</td>
<td><code>0 &lt; result &lt; π/2</code></td>
</tr>
<tr>
<td><code>y &gt; 0</code></td>
<td><code>x = 0</code></td>
<td>Positive y-axis</td>
<td><code>π/2</code></td>
</tr>
<tr>
<td><code>y &gt; 0</code></td>
<td><code>x &lt; 0</code></td>
<td>Quadrant II</td>
<td><code>π/2 &lt; result &lt; π</code></td>
</tr>
<tr>
<td><code>y = 0</code></td>
<td><code>x &lt; 0</code></td>
<td>Negative x-axis</td>
<td><code>π</code></td>
</tr>
<tr>
<td>* <code>y = +0</code></td>
<td><code>x &lt; 0</code></td>
<td>Negative x-axis</td>
<td><code>+π</code></td>
</tr>
<tr>
<td>* <code>y = -0</code></td>
<td><code>x &lt; 0</code></td>
<td>Negative x-axis</td>
<td><code>−π</code></td>
</tr>
<tr>
<td><code>y &lt; 0</code></td>
<td><code>x &lt; 0</code></td>
<td>Quadrant III</td>
<td><code>−π &lt; result &lt; −π/2</code></td>
</tr>
<tr>
<td><code>y &lt; 0</code></td>
<td><code>x = 0</code></td>
<td>Negative y-axis</td>
<td><code>−π/2</code></td>
</tr>
<tr>
<td><code>y &lt; 0</code></td>
<td><code>x &gt; 0</code></td>
<td>Quadrant IV</td>
<td><code>−π/2 &lt; result &lt; 0</code></td>
</tr>
<tr>
<td><code>y = 0</code></td>
<td><code>x = 0</code></td>
<td>Origin</td>
<td>undefined consequences</td>
</tr>
<tr>
<td>* <code>y = +0</code></td>
<td><code>x = +0</code></td>
<td>Origin</td>
<td><code>+0</code></td>
</tr>
<tr>
<td>* <code>y = -0</code></td>
<td><code>x = +0</code></td>
<td>Origin</td>
<td><code>−0</code></td>
</tr>
<tr>
<td>* <code>y = +0</code></td>
<td><code>x = -0</code></td>
<td>Origin</td>
<td><code>+π</code></td>
</tr>
<tr>
<td>* <code>y = -0</code></td>
<td><code>x = -0</code></td>
<td>Origin</td>
<td><code>−π</code></td>
</tr>
</tbody>
</table>

Figure 12–15. Quadrant information for arc tangent

Examples:

```
(asin 0) → 0.0
(acos #c(0 1)) → #C(1.5707963267948966 -0.8813735870195432)
(/ (atan 1 (sqrt 3)) 6) → 0.087266
(atan #c(0 2)) → #C(-1.5707964 0.54930615)
```

Exceptional Situations:

`acos` and `asin` should signal an error of type `type-error` if `number` is not a `number`. `atan` should signal `type-error` if one argument is supplied and that argument is not a `number`, or if two arguments are supplied and both of those arguments are not `reals`.

`acos`, `asin`, and `atan` might signal `arithmetic-error`.

See Also:

`log`, `sqrt`, Section 12.1.3.3 (Rule of Float Substitutability)
Notes:
The result of either \texttt{asin} or \texttt{acos} can be a complex even if \texttt{number} is not a complex; this occurs when the absolute value of \texttt{number} is greater than one.

\begin{tabular}{ll}
\textbf{pi} & \textit{Constant Variable} \\
\end{tabular}

Value:
an implementation-dependent long float.

Description:
The best long float approximation to the mathematical constant $\pi$.

Examples:

\begin{verbatim}
;; In each of the following computations, the precision depends
;; on the implementation. Also, if 'long float' is treated by
;; the implementation as equivalent to some other float format
;; (e.g., 'double float') the exponent marker might be the marker
;; for that equivalent (e.g., 'D' instead of 'L').
pi \to 3.141592653589793L0
(cos pi) \to -1.0L0

(defun sin-of-degrees (degrees)
  (let ((x (if (floatp degrees) degrees (float degrees pi))))
    (sin (* x (/ (float pi x) 180)))))
\end{verbatim}

Notes:
An approximation to $\pi$ in some other precision can be obtained by writing \texttt{(float pi x)}, where \texttt{x} is a float of the desired precision, or by writing \texttt{(coerce pi type)}, where \texttt{type} is the desired type, such as short-float.
sinh, cosh, tanh, asinh, acosh, atanh

Function

Syntax:
- sinh number → result
- cosh number → result
- tanh number → result
- asinh number → result
- acosh number → result
- atanh number → result

Arguments and Values:
- number—a number.
- result—a number.

Description:
These functions compute the hyperbolic sine, cosine, tangent, arc sine, arc cosine, and arc tangent functions, which are mathematically defined for an argument $x$ as given in Figure 12–16.

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hyperbolic sine</td>
<td>$(e^x - e^{-x})/2$</td>
</tr>
<tr>
<td>Hyperbolic cosine</td>
<td>$(e^x + e^{-x})/2$</td>
</tr>
<tr>
<td>Hyperbolic tangent</td>
<td>$(e^x - e^{-x})/(e^x + e^{-x})$</td>
</tr>
<tr>
<td>Hyperbolic arc sine</td>
<td>$\log (x + \sqrt{1 + x^2})$</td>
</tr>
<tr>
<td>Hyperbolic arc cosine</td>
<td>$2 \log \left( \sqrt{(x + 1)/2} + \sqrt{(x - 1)/2} \right)$</td>
</tr>
<tr>
<td>Hyperbolic arc tangent</td>
<td>$(\log (1 + x) - \log (1 - x))/2$</td>
</tr>
</tbody>
</table>

Figure 12–16. Mathematical definitions for hyperbolic functions

The following definition for the inverse hyperbolic cosine determines the range and branch cuts:

$$ \text{arccosh } z = 2 \log \left( \sqrt{(z + 1)/2} + \sqrt{(z - 1)/2} \right). $$

The branch cut for the inverse hyperbolic cosine function lies along the real axis to the left of 1 (inclusive), extending indefinitely along the negative real axis, continuous with quadrant II and (between 0 and 1) with quadrant I. The range is that half-strip of the complex plane containing numbers whose real part is non-negative and whose imaginary part is between $-\pi$ (exclusive) and $\pi$ (inclusive). A number with real part zero is in the range if its imaginary part is between zero (inclusive) and $\pi$ (inclusive).
The following definition for the inverse hyperbolic sine determines the range and branch cuts:

\[ \text{arcsinh } z = \log \left( z + \sqrt{1 + z^2} \right). \]

The branch cut for the inverse hyperbolic sine function is in two pieces: one along the positive imaginary axis above \( i \) (inclusive), continuous with quadrant I, and one along the negative imaginary axis below \( -i \) (inclusive), continuous with quadrant III. The range is that strip of the complex plane containing numbers whose imaginary part is between \( -\pi/2 \) and \( \pi/2 \). A number with imaginary part equal to \( -\pi/2 \) is in the range if and only if its real part is non-positive; a number with imaginary part equal to \( \pi/2 \) is in the range if and only if its imaginary part is non-negative.

The following definition for the inverse hyperbolic tangent determines the range and branch cuts:

\[ \text{arctanh } z = \frac{\log (1 + z) - \log (1 - z)}{2}. \]

Note that:

\[ i \text{ arctan } z = \text{ arctanh } iz. \]

The branch cut for the inverse hyperbolic tangent function is in two pieces: one along the negative real axis to the left of \( -1 \) (inclusive), continuous with quadrant III, and one along the positive real axis to the right of \( 1 \) (inclusive), continuous with quadrant I. The points \( -1 \) and \( 1 \) are excluded from the domain. The range is that strip of the complex plane containing numbers whose imaginary part is between \( -\pi/2 \) and \( \pi/2 \). A number with imaginary part equal to \( -\pi/2 \) is in the range if and only if its real part is strictly negative; a number with imaginary part equal to \( \pi/2 \) is in the range if and only if its imaginary part is strictly positive. Thus the range of the inverse hyperbolic tangent function is identical to that of the inverse hyperbolic sine function with the points \( -\pi i/2 \) and \( \pi i/2 \) excluded.

**Examples:**

- \((\sinh 0) \rightarrow 0.0\)
- \((\cosh (\text{complex } 0 -1)) \rightarrow \#C(0.540302 -0.0)\)

**Exceptional Situations:**

Should signal an error of `type type-error` if `number` is not a `number`. Might signal `arithmetic-error`.

**See Also:**

`log`, `sqrt`, Section 12.1.3.3 (Rule of Float Substitutability)
Notes:
The result of acosh may be a complex even if number is not a complex; this occurs when number is less than one. Also, the result of atanh may be a complex even if number is not a complex; this occurs when the absolute value of number is greater than one.

The branch cut formulae are mathematically correct, assuming completely accurate computation. Implementors should consult a good text on numerical analysis. The formulae given above are not necessarily the simplest ones for real-valued computations; they are chosen to define the branch cuts in desirable ways for the complex case.

Syntax:
* &rest numbers → product

Arguments and Values:
number—a number.
product—a number.

Description:
Returns the product of numbers, performing any necessary type conversions in the process. If no numbers are supplied, 1 is returned.

Examples:

(*) → 1
(* 3 5) → 15
(* 1.0 #c(22 33) 55/98) → #C(12.346938775510203 18.520408163265305)

Exceptional Situations:
Might signal type-error if some argument is not a number. Might signal arithmetic-error.

See Also:
Section 12.1.1 (Numeric Operations), Section 12.1.3 (Rational Computations), Section 12.1.4 (Floating-point Computations), Section 12.1.5 (Complex Computations)
+ Function

Syntax:

+ &rest numbers → sum

Arguments and Values:
number—a number.

sum—a number.

Description:
Returns the sum of numbers, performing any necessary type conversions in the process. If no numbers are supplied, 0 is returned.

Examples:

(+) → 0
(+ 1) → 1
(+ 31/100 69/100) → 1
(+ 1/5 0.8) → 1.0

Exceptional Situations:
Might signal type-error if some argument is not a number. Might signal arithmetic-error.

See Also:
Section 12.1.1 (Numeric Operations), Section 12.1.3 (Rational Computations), Section 12.1.4 (Floating-point Computations), Section 12.1.5 (Complex Computations)

− Function

Syntax:

− number → negation

− minuend &rest subtrahends → difference

Arguments and Values:
number, minuend, subtrahend—a number.

negation, difference—a number.

Description:
The function - performs arithmetic subtraction and negation.

12–36 Programming Language—Common Lisp
If only one `number` is supplied, the negation of that `number` is returned.

If more than one `argument` is given, it subtracts all of the `subtrahends` from the `minuend` and returns the result.

The `function -` performs necessary type conversions.

**Examples:**

```
(- 55.55) → -55.55
(- #c(3 -5)) → #C(-3 5)
(- 0) → 0
(eql (- 0.0) -0.0) → true
(- #c(100 45) #c(0 45)) → 100
(- 10 1 2 3 4) → 0
```

**Exceptional Situations:**

Might signal `type-error` if some `argument` is not a `number`. Might signal `arithmetic-error`.

**See Also:**

Section 12.1.1 (Numeric Operations), Section 12.1.3 (Rational Computations), Section 12.1.4 (Floating-point Computations), Section 12.1.5 (Complex Computations)

---

```
/ Function

Syntax:

/ number → reciprocal

/ numerator &rest denominators† → quotient

Arguments and Values:

- `number, denominator`—a non-zero `number`.
- `numerator, quotient, reciprocal`—a `number`.

Description:

The `function /` performs division or reciprocation.

If no `denominators` are supplied, the `function /` returns the reciprocal of `number`.

If at least one `denominator` is supplied, the `function /` divides the `numerator` by all of the `denominators` and returns the resulting `quotient`.

If each `argument` is either an `integer` or a `ratio`, and the result is not an `integer`, then it is a `ratio`.  

Numbers 12–37
The function / performs necessary type conversions.

If any argument is a float then the rules of floating-point contagion apply; see Section 12.1.4 (Floating-point Computations).

Examples:

```
(/ 12 4) → 3
(/ 13 4) → 13/4
(/ -8) → -1/8
(/ 3 4 5) → 3/20
(/ 0.5) → 2.0
(/ 20 5) → 4
(/ 5 20) → 1/4
(/ 60 -2 3 5.0) → -2.0
(/ 2 #c(2 2)) → #C(1/2 -1/2)
```

Exceptional Situations:

The consequences are unspecified if any argument other than the first is zero. If there is only one argument, the consequences are unspecified if it is zero.

Might signal type-error if some argument is not a number. Might signal division-by-zero if division by zero is attempted. Might signal arithmetic-error.

See Also:

floor, ceiling, truncate, round

### 1+, 1− Function

**Syntax:**

```
1+ number → successor
1- number → predecessor
```

**Arguments and Values:**

- `number`—a number.
- `successor`, `predecessor`—a number.

**Description:**

1+ returns a number that is one more than its argument number. 1- returns a number that is one less than its argument number.
Examples:

(1+ 99) → 100
(1- 100) → 99
(1+ (complex 0.0)) → #C(1.0 0.0)
(1- 5/3) → 2/3

Exceptional Situations:
Might signal type-error if its argument is not a number. Might signal arithmetic-error.

See Also:
incf, decf

Notes:

(1+ number) ≡ (+ number 1)
(1- number) ≡ (- number 1)

Implementors are encouraged to make the performance of both the previous expressions be the same.

abs Function

Syntax:
abs number → absolute-value

Arguments and Values:
number—a number.
absolute-value—a non-negative real.

Description:
abs returns the absolute value of number.

If number is a real, the result is of the same type as number.

If number is a complex, the result is a positive real with the same magnitude as number. The result can be a float even if number’s components are rationals and an exact rational result would have been possible. Thus the result of (abs #c(3 4)) can be either 5 or 5.0, depending on the implementation.

Examples:

(abs 0) → 0
(abs 12/13) → 12/13
(abs -1.09) → 1.09
(abs #c(5.0 -5.0)) → 7.071068
(abs #c(5 5)) → 7.071068
(abs #c(3/5 4/5)) → 1 or approximately 1.0
(eql (abs -0.0) -0.0) → true

See Also:
Section 12.1.3.3 (Rule of Float Substitutability)

Notes:
If number is a complex, the result is equivalent to the following:

(sqrt (+ (expt (realpart number) 2) (expt (imagpart number) 2)))

An implementation should not use this formula directly for all complexes but should handle very large or very small components specially to avoid intermediate overflow or underflow.

evenp, oddp

Function

Syntax:

evenp integer → generalized-boolean
oddp integer → generalized-boolean

Arguments and Values:

integer—an integer.

generalized-boolean—a generalized boolean.

Description:

evenp returns true if integer is even (divisible by two); otherwise, returns false.

oddp returns true if integer is odd (not divisible by two); otherwise, returns false.

Examples:

(evenp 0) → true
(oddp 10000000000000000000) → false
(oddp -1) → true

Exceptional Situations:

Should signal an error of type type-error if integer is not an integer.
Notes:

\[
\begin{align*}
\text{(evenp integer)} & \equiv (\text{not (oddp integer)}) \\
\text{(oddp integer)} & \equiv (\text{not (evenp integer)})
\end{align*}
\]

exp, expt  

Function

Syntax:

\[
\begin{align*}
\text{exp } & \text{ number } \rightarrow \text{ result} \\
\text{expt } & \text{ base-number power-number } \rightarrow \text{ result}
\end{align*}
\]

Arguments and Values:

number—a number.
base-number—a number.
power-number—a number.
result—a number.

Description:

exp and expt perform exponentiation.

exp returns \( e \) raised to the power \( \text{number} \), where \( e \) is the base of the natural logarithms. exp has no branch cut.

expt returns base-number raised to the power power-number. If the base-number is a rational and power-number is an integer, the calculation is exact and the result will be of type rational; otherwise a floating-point approximation might result. For expt of a complex rational to an integer power, the calculation must be exact and the result is of type (or rational (complex rational)).

The result of expt can be a complex, even when neither argument is a complex, if base-number is negative and power-number is not an integer. The result is always the principal complex value. For example, (expt -8 1/3) is not permitted to return -2, even though -2 is one of the cube roots of -8. The principal cube root is a complex approximately equal to #C(1.0 1.73205), not -2.

expt is defined as \( b^x = e^{x \log b} \). This defines the principal values precisely. The range of expt is the entire complex plane. Regarded as a function of \( x \), with \( b \) fixed, there is no branch cut. Regarded as a function of \( b \), with \( x \) fixed, there is in general a branch cut along the negative real axis, continuous with quadrant II. The domain excludes the origin. By definition, \( 0^0=1 \). If \( b=0 \) and the real part of \( x \) is strictly positive, then \( b^x=0 \). For all other values of \( x \), \( 0^x \) is an error.
When power-number is an integer 0, then the result is always the value one in the type of base-number, even if the base-number is zero (of any type). That is:

\[(\text{expt } x \ 0) \equiv (\text{coerce } 1 \ \text{(type-of } x))\]

If power-number is a zero of any other type, then the result is also the value one, in the type of the arguments after the application of the contagion rules in Section 12.1.1.2 (Contagion in Numeric Operations), with one exception: the consequences are undefined if base-number is zero when power-number is zero and not of type integer.

Examples:

\[
\begin{align*}
\text{(exp 0)} & \rightarrow 1.0 \\
\text{(exp 1)} & \rightarrow 2.718282 \\
\text{(exp (log 5))} & \rightarrow 5.0 \\
\text{(expt 2 8)} & \rightarrow 256 \\
\text{(expt 4 .5)} & \rightarrow 2.0 \\
\text{(expt #c(0 1) 2)} & \rightarrow -1 \\
\text{(expt #c(2 2) 3)} & \rightarrow #C(-16 16) \\
\text{(expt #c(2 2) 4)} & \rightarrow -64
\end{align*}
\]

See Also:

log, Section 12.1.3.3 (Rule of Float Substitutability)

Notes:

Implementations of \texttt{expt} are permitted to use different algorithms for the cases of a power-number of type rational and a power-number of type float.

Note that by the following logic, (\texttt{sqrt (expt x 3)}) is not equivalent to (\texttt{expt x 3/2}).

\[
\begin{align*}
\text{(setq x (exp (/ (* 2 pi #c(0 1)) 3)))} & \quad ;\text{exp(2.pi.1/3)} \\
\text{(expt x 3)} & \rightarrow 1 \ ;\text{except for round-off error} \\
\text{(sqrt (expt x 3))} & \rightarrow 1 \ ;\text{except for round-off error} \\
\text{(expt x 3/2)} & \rightarrow -1 \ ;\text{except for round-off error}
\end{align*}
\]

\[
\text{gcd}
\]

\textbf{Function}

**Syntax:**

\texttt{gcd \&rest integers} \rightarrow greatest-common-denominator

**Arguments and Values:**

\texttt{integer}—an integer.

12–42 Programming Language—Common Lisp
**greatest-common-denominator**—a non-negative integer.

**Description:**
Returns the greatest common divisor of integers. If only one integer is supplied, its absolute value is returned. If no integers are given, \( \text{gcd} \) returns 0, which is an identity for this operation.

**Examples:**
- \((\text{gcd}) \rightarrow 0\)
- \((\text{gcd} \ 60 \ 42) \rightarrow 6\)
- \((\text{gcd} \ 3333 \ -33 \ 101) \rightarrow 1\)
- \((\text{gcd} \ 3333 \ -33 \ 1002001) \rightarrow 11\)
- \((\text{gcd} \ 91 \ -49) \rightarrow 7\)
- \((\text{gcd} \ 63 \ -42 \ 35) \rightarrow 7\)
- \((\text{gcd} \ 5) \rightarrow 5\)
- \((\text{gcd} \ -4) \rightarrow 4\)

**Exceptional Situations:**
Should signal an error of type \text{type-error} if any integer is not an integer.

**See Also:**
lcm

**Notes:**
For three or more arguments,

\((\text{gcd} \ b \ c \ldots \ z) \equiv (\text{gcd} \ (\text{gcd} \ a \ b) \ c \ldots \ z)\)

---

**incf, decf**

*Macro*

**Syntax:**
- \(\text{incf} \ \text{place} \ [\text{delta-form}] \ → \text{new-value}\)
- \(\text{decf} \ \text{place} \ [\text{delta-form}] \ → \text{new-value}\)

**Arguments and Values:**
- \(\text{place}\)—a \text{place}.
- \(\text{delta-form}\)—a \text{form}; evaluated to produce a \text{delta}. The default is 1.
- \(\text{delta}\)—a \text{number}.
- \(\text{new-value}\)—a \text{number}.

Numbers 12–43
Description:

`incf` and `decf` are used for incrementing and decrementing the *value* of `place`, respectively.

The *delta* is added to (in the case of `incf`) or subtracted from (in the case of `decf`) the number in `place` and the result is stored in `place`.

Any necessary type conversions are performed automatically.

For information about the *evaluation* of subforms of `places`, see Section 5.1.1.1 (Evaluation of Subforms to Places).

Examples:

```lisp
(setq n 0)
(incf n) → 1
n → 1
(decf n 3) → -2
n → -2
(decf n -5) → 3
(decf n) → 2
(incf n 0.5) → 2.5
(decf n) → 1.5
n → 1.5
```

Side Effects:

`Place` is modified.

See Also:

`+`, `-`, `1+`, `1-`, `setf`

---

**lcm**

*Function*

Syntax:

```
lcm &rest integers  → least-common-multiple
```

Arguments and Values:

- `integer`—an `integer`.
  - *least-common-multiple*—a non-negative `integer`.

Description:

`lcm` returns the least common multiple of the `integers`.

If no `integer` is supplied, the `integer` 1 is returned.
If only one integer is supplied, the absolute value of that integer is returned.

For two arguments that are not both zero,

\[(\text{lcm} \ a \ b) \equiv \left(\frac{\text{abs}(a \times b)}{\text{gcd} \ a \ b}\right)\]

If one or both arguments are zero,

\[(\text{lcm} \ a \ 0) \equiv \left(\text{lcm} \ 0 \ a\right) \equiv 0\]

For three or more arguments,

\[(\text{lcm} \ a \ b \ c \ldots \ z) \equiv \left(\text{lcm} \ (\text{lcm} \ a \ b) \ c \ldots \ z\right)\]

**Examples:**

\[
\begin{align*}
(\text{lcm} \ 10) & \rightarrow 10 \\
(\text{lcm} \ 25 \ 30) & \rightarrow 150 \\
(\text{lcm} \ -24 \ 18 \ 10) & \rightarrow 360 \\
(\text{lcm} \ 14 \ 35) & \rightarrow 70 \\
(\text{lcm} \ 0 \ 5) & \rightarrow 0 \\
(\text{lcm} \ 1 \ 2 \ 3 \ 4 \ 5 \ 6) & \rightarrow 60
\end{align*}
\]

**Exceptional Situations:**
Should signal type-error if any argument is not an integer.

See Also:
gcd

---

\textbf{log}

\textbf{Function}

**Syntax:**

\[
\text{log} \ \text{number} \ \&\text{optional} \ \text{base} \ \rightarrow \ \text{logarithm}
\]

**Arguments and Values:**

- \text{number}—a non-zero number.
- \text{base}—a number.
- \text{logarithm}—a number.

**Description:**

\text{log} returns the logarithm of \text{number} in base \text{base}. If \text{base} is not supplied its value is \(e\), the base of the natural logarithms.
**log**

log may return a complex when given a real negative number.

\[(\log -1.0) \equiv (\text{complex } 0.0 (\text{float pi } 0.0))\]

If base is zero, log returns zero.

The result of \((\log 8 2)\) may be either 3 or 3.0, depending on the implementation. An implementation can use floating-point calculations even if an exact integer result is possible.

The branch cut for the logarithm function of one argument (natural logarithm) lies along the negative real axis, continuous with quadrant II. The domain excludes the origin.

The mathematical definition of a complex logarithm is as follows, whether or not minus zero is supported by the implementation:

\[(\log x) \equiv (\text{complex } (\log (\text{abs } x)) (\text{phase } x))\]

Therefore the range of the one-argument logarithm function is that strip of the complex plane containing numbers with imaginary parts between \(-\pi\) (exclusive) and \(\pi\) (inclusive) if minus zero is not supported, or \(-\pi\) (inclusive) and \(\pi\) (inclusive) if minus zero is supported.

The two-argument logarithm function is defined as

\[(\log \text{ base number}) \equiv (/ (\log \text{ number}) (\log \text{ base}))\]

This defines the principal values precisely. The range of the two-argument logarithm function is the entire complex plane.

**Examples:**

\[\begin{align*}
\log 100 10 & \to 2.0 \\
& \to 2 \\
\log 100.0 10 & \to 2.0 \\
\log #c(0 1) #c(0 -1)) & \to #c(-1.0 0.0) \\
& \to #c(-1 0) \\
\log 8.0 2 & \to 3.0 \\
\log #c(-16 16) #c(2 2)) & \to 3 \text{ or approximately } #c(3.0 0.0) \\
& \text{or approximately } 3.0 \text{ (unlikely)}
\end{align*}\]

**Affected By:**

The implementation.

12-46 Programming Language—Common Lisp
**See Also:**

*exp, exp*, Section 12.1.3.3 (Rule of Float Substitutability)

---

**mod, rem**

*Function*

**Syntax:**

- `mod number divisor → modulus`
- `rem number divisor → remainder`

**Arguments and Values:**

- `number` — a *real*.
- `divisor` — a *real*.
- `modulus`, `remainder` — a *real*.

**Description:**

`mod` and `rem` are generalizations of the modulus and remainder functions respectively.

- `mod` performs the operation `floor` on `number` and `divisor` and returns the remainder of the `floor` operation.
- `rem` performs the operation `truncate` on `number` and `divisor` and returns the remainder of the `truncate` operation.

`mod` and `rem` are the modulus and remainder functions when `number` and `divisor` are *integers*.

**Examples:**

- `(rem -1 5) → -1`
- `(mod -1 5) → 4`
- `(mod 13 4) → 1`
- `(rem 13 4) → 1`
- `(mod -13 4) → 3`
- `(rem -13 4) → -1`
- `(mod 13 -4) → -3`
- `(rem 13 -4) → -1`
- `(mod -13 -4) → -1`
- `(rem -13 -4) → -1`
- `(mod 13.4 1) → 0.4`
- `(rem 13.4 1) → 0.4`
- `(mod -13.4 1) → 0.6`
- `(rem -13.4 1) → -0.4`
See Also:

floor, truncate

Notes:

The result of mod is either zero or a real with the same sign as divisor.

**signum**  
*Function*

**Syntax:**

\[ \text{signum} \ number \rightarrow \text{signed-prototype} \]

**Arguments and Values:**

- **number**—a number.
- **signed-prototype**—a number.

**Description:**

**signum** determines a numerical value that indicates whether **number** is negative, zero, or positive.

For a rational, **signum** returns one of -1, 0, or 1 according to whether **number** is negative, zero, or positive. For a float, the result is a float of the same format whose value is minus one, zero, or one. For a complex number \( z \), \((\text{signum} \ z)\) is a complex number of the same phase but with unit magnitude, unless \( z \) is a complex zero, in which case the result is \( z \).

For rational arguments, **signum** is a rational function, but it may be irrational for complex arguments.

If **number** is a float, the result is a float. If **number** is a rational, the result is a rational. If **number** is a complex float, the result is a complex float. If **number** is a complex rational, the result is a complex, but it is implementation-dependent whether that result is a complex rational or a complex float.

**Examples:**

```
(signum 0)  →  0
(signum 99)  →  1
(signum 4/5)  →  1
(signum -99/100)  →  -1
(signum 0.0)  →  0.0
(signum #c(0.33))  →  #C(0.0 1.0)
(signum #c(7.5 10.0))  →  #C(0.6 0.8)
(signum #c(0.0 -14.7))  →  #C(0.0 -1.0)
(eql (signum -0.0) -0.0)  →  true
```
See Also:
Section 12.1.3.3 (Rule of Float Substitutability)

Notes:

(signum x) ≡ (if (zerop x) x (/ x (abs x)))

sqrt, isqrt

Syntax:

sqrt number → root
isqrt natural → natural-root

Arguments and Values:

number, root—a number.
natural, natural-root—a non-negative integer.

Description:

sqrt and isqrt compute square roots.

sqrt returns the principal square root of number. If the number is not a complex but is negative, then the result is a complex.

isqrt returns the greatest integer less than or equal to the exact positive square root of natural.

If number is a positive rational, it is implementation-dependent whether root is a rational or a float. If number is a negative rational, it is implementation-dependent whether root is a complex rational or a complex float.

The mathematical definition of complex square root (whether or not minus zero is supported) follows:

(sqrt x) = (exp (/ (log x) 2))

The branch cut for square root lies along the negative real axis, continuous with quadrant II. The range consists of the right half-plane, including the non-negative imaginary axis and excluding the negative imaginary axis.

Examples:

(sqrt 9.0) → 3.0
(sqrt -9.0) → #C(0.0 3.0)
(isqrt 9) → 3  
(sqrt 12) → 3.4641016  
(isqrt 12) → 3  
(isqrt 300) → 17  
(isqrt 325) → 18  
(sqrt 25)  
→ 5  
≈ 5.0  
(isqrt 25) → 5  
(sqrt -1) → #C(0.0 1.0)  
(sqrt #c(0 2)) → #C(1.0 1.0)

Exceptional Situations:
The function sqrt should signal type-error if its argument is not a number.
The function isqrt should signal type-error if its argument is not a non-negative integer.
The functions sqrt and isqrt might signal arithmetic-error.

See Also:
exp, log, Section 12.1.3.3 (Rule of Float Substitutability)

Notes:
(isqrt x) ≡ (values (floor (sqrt x)))
but it is potentially more efficient.

random-state System Class

Class Precedence List:
random-state, t

Description:
A random state object contains state information used by the pseudo-random number generator. The nature of a random state object is implementation-dependent. It can be printed out and successfully read back in by the same implementation, but might not function correctly as a random state in another implementation.
Implementations are required to provide a read syntax for objects of type random-state, but the specific nature of that syntax is implementation-dependent.

See Also:
*random-state*, random, Section 22.1.3.10 (Printing Random States)

12–50 Programming Language—Common Lisp
Syntax:
make-random-state &optional state → new-state

Arguments and Values:
state—a random state, or nil, or t. The default is nil.
new-state—a random state object.

Description:
Creates a fresh object of type random-state suitable for use as the value of *random-state*.
If state is a random state object, the new-state is a copy of that object. If state is nil, the new-state is a copy of the current random state. If state is t, the new-state is a fresh random state object that has been randomly initialized by some means.

Examples:
(let* ((rs1 (make-random-state nil))
       (rs2 (make-random-state t))
       (rs3 (make-random-state rs2))
       (rs4 nil))
  (list (loop for i from 1 to 10
              collect (random 100)
              when (= i 5)
              do (setq rs4 (make-random-state)))
        (loop for i from 1 to 10 collect (random 100 rs1))
        (loop for i from 1 to 10 collect (random 100 rs2))
        (loop for i from 1 to 10 collect (random 100 rs3))
        (loop for i from 1 to 10 collect (random 100 rs4))))
→ ((29 25 72 57 55 68 24 35 54 65)
    (29 25 72 57 55 68 24 35 54 65)
    (93 85 53 99 58 62 2 23 23 59)
    (93 85 53 99 58 62 2 23 23 59)
    (68 24 35 54 65 54 55 50 59 49))

Exceptional Situations:
Should signal an error of type type-error if state is not a random state, or nil, or t.

See Also:
random, *random-state*
Notes:
One important use of `make-random-state` is to allow the same series of pseudo-random numbers to be generated many times within a single program.

---

### random

**Function**

**Syntax:**

```
random limit &optional random-state → random-number
```

**Arguments and Values:**
- `limit`—a positive integer, or a positive float.
- `random-state`—a random state. The default is the current random state.
- `random-number`—a non-negative number less than `limit` and of the same type as `limit`.

**Description:**

Returns a pseudo-random number that is a non-negative number less than `limit` and of the same type as `limit`.

The `random-state`, which is modified by this function, encodes the internal state maintained by the random number generator.

An approximately uniform choice distribution is used. If `limit` is an integer, each of the possible results occurs with (approximate) probability `1/limit`.

**Examples:**

```
(<= 0 (random 1000) 1000) → true
(let ((state1 (make-random-state))
     (state2 (make-random-state)))
  (= (random 1000 state1) (random 1000 state2))) → true
```

**Side Effects:**

The `random-state` is modified.

**Exceptional Situations:**

Should signal an error of type `type-error` if `limit` is not a positive integer or a positive real.

**See Also:**

- `make-random-state`, `*random-state*`

**Notes:**

`See Common Lisp: The Language` for information about generating random numbers.
random-state-p

Function

Syntax:

random-state-p object  \rightarrow generalized-boolean

Arguments and Values:

object—an object.

generalized-boolean—a generalized boolean.

Description:

Returns true if object is of type random-state; otherwise, returns false.

Examples:

(random-state-p *random-state*) \rightarrow true
(random-state-p (make-random-state)) \rightarrow true
(random-state-p 'test-function) \rightarrow false

See Also:

make-random-state, *random-state*

Notes:

(random-state-p object) \equiv (typep object 'random-state)

*random-state*

Variable

Value Type:

a random state.

Initial Value:

implementation-dependent.

Description:

The current random state, which is used, for example, by the function random when a random state is not explicitly supplied.
Examples:

(random-state-p *random-state*) → true
(setq snap-shot (make-random-state))
  ;; The series from any given point is random,
  ;; but if you backtrack to that point, you get the same series.
(list (loop for i from 1 to 10 collect (random))
  (let ((*random-state* snap-shot))
    (loop for i from 1 to 10 collect (random)))
  (loop for i from 1 to 10 collect (random))
  (let ((*random-state* snap-shot))
    (loop for i from 1 to 10 collect (random)))
) → ((19 16 44 19 96 15 76 96 13 61)
  (19 16 44 19 96 15 76 96 13 61)
  (16 67 0 43 70 79 58 5 63 50)
  (16 67 0 43 70 79 58 5 63 50))

Affected By:
The implementation.
random.

See Also:
  make-random-state, random, random-state

Notes:
  Binding *random-state* to a different random state object correctly saves and restores the old random state object.

---

**numberp**

**Function**

Syntax:

numberp object → generalized-boolean

Arguments and Values:

  object—an object.

  generalized-boolean—a generalized boolean.

Description:

  Returns true if object is of type number; otherwise, returns false.
Examples:

\[
\begin{align*}
\text{(numberp 12)} & \rightarrow \text{true} \\
\text{(numberp (expt 2 130))} & \rightarrow \text{true} \\
\text{(numberp #c(5/3 7.2))} & \rightarrow \text{true} \\
\text{(numberp nil)} & \rightarrow \text{false} \\
\text{(numberp (cons 1 2))} & \rightarrow \text{false}
\end{align*}
\]

Notes:

\[
\text{(numberp object)} \equiv \text{(typep object 'number)}
\]

---

**cis**

Function

Syntax:

\[
\text{cis radians} \rightarrow \text{number}
\]

Arguments and Values:

- *radians*—a real.
- *number*—a complex.

Description:

\[
\text{cis returns the value of } e^{i \cdot \text{radians}}, \text{ which is a complex in which the real part is equal to the cosine of } \text{radians}, \text{ and the imaginary part is equal to the sine of } \text{radians}.
\]

Examples:

\[
\text{(cis 0)} \rightarrow \#C(1.0 0.0)
\]

See Also:

Section 12.1.3.3 (Rule of Float Substitutability)
complex

Function

Syntax:

\[
\text{complex } \text{realpart} \, \&\text{optional imagpart} \to \text{complex}
\]

Arguments and Values:

- \text{realpart} — a \text{real}.
- \text{imagpart} — a \text{real}.
- \text{complex} — a \text{rational} or a \text{complex}.

Description:

\text{complex} returns a \text{number} whose real part is \text{realpart} and whose imaginary part is \text{imagpart}.

If \text{realpart} is a \text{rational} and \text{imagpart} is the \text{rational} number zero, the result of \text{complex} is \text{realpart}, a \text{rational}. Otherwise, the result is a \text{complex}.

If either \text{realpart} or \text{imagpart} is a \text{float}, the non-\text{float} is converted to a \text{float} before the \text{complex} is created. If \text{imagpart} is not supplied, the imaginary part is a zero of the same \text{type} as \text{realpart}; i.e.,

\[
\text{(coerce 0 (type-of realpart))}
\]

is effectively used.

Type upgrading implies a movement upwards in the type hierarchy lattice. In the case of \text{complexes}, the \text{type-specifier} must be a subtype of \text{(upgraded-complex-part-type type-specifier)}.

If \text{type-specifier1} is a subtype of \text{type-specifier2}, then \text{(upgraded-complex-element-type type-specifier1)} must also be a subtype of \text{(upgraded-complex-element-type type-specifier2)}. Two disjoint types can be upgraded into the same thing.

Examples:

- \text{(complex 0)} \to 0
- \text{(complex 0.0)} \to #C(0.0 0.0)
- \text{(complex 1 1/2)} \to #C(1 1/2)
- \text{(complex 1.99)} \to #C(1.99 0.99)
- \text{(complex 3/2 0.0)} \to #C(1.5 0.0)

See Also:

realpart, imagpart, Section 2.4.8.11 (Sharpsign C)
**complexp**

*Function*

**Syntax:**

```
complexp object  →  generalized-boolean
```

**Arguments and Values:**

- `object`—an `object`.
- `generalized-boolean`—a `generalized boolean`.

**Description:**

Returns `true` if `object` is of type `complex`; otherwise, returns `false`.

**Examples:**

```
(complexp 1.2d2)  →  false
(complexp #c(5/3 7.2))  →  true
```

**See Also:**

`complex (function and type)`, `typep`

**Notes:**

```
(complexp object)  ≡  (typep object 'complex)
```

---

**conjugate**

*Function*

**Syntax:**

```
conjugate number  →  conjugate
```

**Arguments and Values:**

- `number`—a `number`.
- `conjugate`—a `number`.

**Description:**

Returns the complex conjugate of `number`. The conjugate of a `real` number is itself.

**Examples:**

```
(conjugate #c(0 -1))  →  #C(0 1)
```
(conjugate #c(1 1)) → #c(1 -1)
(conjugate 1.5) → 1.5
(conjugate #c(3/5 4/5)) → #c(3/5 -4/5)
(conjugate #c(0.0D0 -1.0D0)) → #c(0.0D0 1.0D0)
(conjugate 3.7) → 3.7

Notes:
For a complex number z,

(conjugate z) ≡ (complex (realpart z) (- (imagpart z)))

### phase

**Function**

**Syntax:**

```lisp
phase number → phase
```

**Arguments and Values:**

- `number`—a number.
- `phase`—a number.

**Description:**

The `phase` function returns the phase of `number` (the angle part of its polar representation) in radians, in the range $-\pi$ (exclusive) if minus zero is not supported, or $-\pi$ (inclusive) if minus zero is supported, to $\pi$ (inclusive). The phase of a positive real number is zero; that of a negative real number is $\pi$. The phase of zero is defined to be zero.

If `number` is a complex float, the result is a float of the same type as the components of `number`. If `number` is a float, the result is a float of the same type. If `number` is a rational or a complex rational, the result is a single float.

The branch cut for `phase` lies along the negative real axis, continuous with quadrant II. The range consists of that portion of the real axis between $-\pi$ (exclusive) and $\pi$ (inclusive).

The mathematical definition of `phase` is as follows:

```lisp
(phase x) = (atan (imagpart x) (realpart x))
```

**Examples:**

- `(phase 1) → 0.0s0`
- `(phase 0) → 0.0s0`
- `(phase (cis 30)) → -1.4159266`
- `(phase #c(0 1)) → 1.5707964`
Exceptional Situations:
Should signal type-error if its argument is not a number. Might signal arithmetic-error.

See Also:
Section 12.1.3.3 (Rule of Float Substitutability)

realpart, imagpart

Function

Syntax:

realpart number → real
imagpart number → real

Arguments and Values:

number—a number.
real—a real.

Description:
realpart and imagpart return the real and imaginary parts of number respectively. If number
is real, then realpart returns number and imagpart returns (* 0 number), which has the effect
that the imaginary part of a rational is 0 and that of a float is a floating-point zero of the same
format.

Examples:

(realpart #c(23 41)) → 23
(imagpart #c(23 41.0)) → 41.0
(realpart #c(23 41.0)) → 23.0
(imagpart 23.0) → 0.0

Exceptional Situations:
Should signal an error of type type-error if number is not a number.

See Also:

complex

Numbers 12–59
upgraded-complex-part-type  Function

Syntax:

upgraded-complex-part-type typespec &optional environment → upgraded-typespec

Arguments and Values:

typespec—*a type specifier*.

environment—an environment object. The default is nil, denoting the null lexical environment and the and current global environment.

upgraded-typespec—*a type specifier*.

Description:

upgraded-complex-part-type returns the part type of the most specialized complex number representation that can hold parts of type typespec.

The typespec is a subtype of (and possibly type equivalent to) the upgraded-typespec.

The purpose of upgraded-complex-part-type is to reveal how an implementation does its upgrading.

See Also:

complex (function and type)

Notes:

realp  Function

Syntax:

realp object → generalized-boolean

Arguments and Values:

object—an object.

generalized-boolean—a generalized boolean.

Description:

Returns true if object is of type real; otherwise, returns false.

Examples:

(realp 12) → true
(realp #c(5/3 7.2)) → false
(realp nil) → false
(REALP (CONS 1 2)) → false

Notes:

(realp object) ≡ (typep object 'real)

numererator, denominator

Function

Syntax:

numerator rational → numerator
denominator rational → denominator

Arguments and Values:

rational—A rational.

numerator—an integer.
denominator—a positive integer.

Description:

numerator and denominator reduce rational to canonical form and compute the numerator or denominator of that number.

numerator and denominator return the numerator or denominator of the canonical form of rational.

If rational is an integer, numerator returns rational and denominator returns 1.

Examples:

(numerator 1/2) → 1
(denominator 12/36) → 3
(numerator -1) → -1
(denominator (/ -33)) → 33
(numerator (/ 8 -6)) → -4
(denominator (/ 8 -6)) → 3

See Also:

/
Notes:

\[(\text{gcd \ (numerator \ x) \ (denominator \ x)}) \to 1\]

---

**rational, rationalize**

*Function*

**Syntax:**

- rational *number* → rational
- rationalize *number* → rational

**Arguments and Values:**

- *number*—a real.
- rational—a rational.

**Description:**

rational and rationalize convert reals to rationals.

If *number* is already rational, it is returned.

If *number* is a float, rational returns a rational that is mathematically equal in value to the float. rationalize returns a rational that approximates the float to the accuracy of the underlying floating-point representation.

rational assumes that the float is completely accurate.

rationalize assumes that the float is accurate only to the precision of the floating-point representation.

**Examples:**

- (rational 0) → 0
- (rationalize -11/100) → -11/100
- (rational .1) → 13421773/134217728 ; implementation-dependent
- (rationalize .1) → 1/10

**Affected By:**

The implementation.

**Exceptional Situations:**

Should signal an error of type type-error if *number* is not a real. Might signal arithmetic-error.
Notes:

It is always the case that

\[(\text{float (rational } x) \ x) \equiv x\]

and

\[(\text{float (rationalize } x) \ x) \equiv x\]

That is, rationalizing a \textit{float} by either method and then converting it back to a \textit{float} of the same format produces the original \textit{number}.

---

**rationalp**

*Function*

Syntax:

\[
\text{rationalp object} \to \text{generalized-boolean}
\]

Arguments and Values:

- \textit{object}—an \textit{object}.
- \textit{generalized-boolean}—a \textit{generalized boolean}.

Description:

Returns \texttt{true} if \textit{object} is of type \texttt{rational}; otherwise, returns \texttt{false}.

Examples:

- \((\text{rationalp 12}) \to \text{true}\)
- \((\text{rationalp 6/5}) \to \text{true}\)
- \((\text{rationalp 1.212}) \to \text{false}\)

See Also:

- \texttt{rational}

Notes:

\[(\text{rationalp object}) \equiv (\text{typep object 'rational})\]
ash

Function

Syntax:
ash integer count → shifted-integer

Arguments and Values:
integer—an integer.
count—an integer.
shifted-integer—an integer.

Description:
ash performs the arithmetic shift operation on the binary representation of integer, which is treated as if it were binary.

ash shifts integer arithmetically left by count bit positions if count is positive, or right count bit positions if count is negative. The shifted value of the same sign as integer is returned.

Mathematically speaking, ash performs the computation floor(integer \(2^{\text{count}}\)). Logically, ash moves all of the bits in integer to the left, adding zero-bits at the right, or moves them to the right, discarding bits.

ash is defined to behave as if integer were represented in two's complement form, regardless of how integers are represented internally.

Examples:
(ash 16 1) → 32
(ash 16 0) → 16
(ash 16 -1) → 8
(ash -100000000000000000000000000000000 -100) → -79

Exceptional Situations:
Should signal an error of type type-error if integer is not an integer. Should signal an error of type type-error if count is not an integer. Might signal arithmetic-error.

Notes:
(logbitp j (ash n k))
≡ (and (>= j k) (logbitp (- j k) n))
integer-length

Syntax:
   integer-length integer → number-of-bits

Arguments and Values:
   integer—an integer.
   number-of-bits—a non-negative integer.

Description:
   Returns the number of bits needed to represent integer in binary two's-complement format.

Examples:
   (integer-length 0) → 0
   (integer-length 1) → 1
   (integer-length 3) → 2
   (integer-length 4) → 3
   (integer-length 7) → 3
   (integer-length -1) → 0
   (integer-length -4) → 2
   (integer-length -7) → 3
   (integer-length -8) → 3
   (integer-length (expt 2 9)) → 10
   (integer-length (1- (expt 2 9))) → 9
   (integer-length (- (expt 2 9))) → 9
   (integer-length (- (1+ (expt 2 9)))) → 10

Exceptional Situations:
   Should signal an error of type type-error if integer is not an integer.

Notes:
   This function could have been defined by:

   (defun integer-length (integer)
      (ceiling (log (if (minusp integer)
                      (- integer)
                      (1+ integer))
       2)))

   If integer is non-negative, then its value can be represented in unsigned binary form in a field
   whose width in bits is no smaller than (integer-length integer). Regardless of the sign of integer,
   its value can be represented in signed binary two's-complement form in a field whose width in bits
   is no smaller than (+ (integer-length integer) 1).
### integerp

**Function**

**Syntax:**

integerp object  →  generalized-boolean

**Arguments and Values:**

object—an object.

generalized-boolean—a generalized boolean.

**Description:**

Returns true if object is of type integer; otherwise, returns false.

**Examples:**

- (integerp 1) → true
- (integerp (expt 2 130)) → true
- (integerp 6/5) → false
- (integerp nil) → false

**Notes:**

(integerp object) ≡ (typep object 'integer)

### parse-integer

**Function**

**Syntax:**

parse-integer string &key start end radix junk-allowed  →  integer, pos

**Arguments and Values:**

string—a string.

start, end—bounding index designators of string. The defaults for start and end are 0 and nil, respectively.

radix—a radix. The default is 10.

junk-allowed—a generalized boolean. The default is false.
integer—an integer or false.
pos—a bounding index of string.

Description:
parse-integer parses an integer in the specified radix from the substring of string delimited by start and end.

parse-integer expects an optional sign (+ or -) followed by a non-empty sequence of digits to be interpreted in the specified radix. Optional leading and trailing whitespace is ignored.

parse-integer does not recognize the syntactic radix-specifier prefixes #O, #B, #X, and #nR, nor does it recognize a trailing decimal point.

If junk-allowed is false, an error of type parse-error is signaled if substring does not consist entirely of the representation of a signed integer, possibly surrounded on either side by whitespace characters.

The first value returned is either the integer that was parsed, or else nil if no syntactically correct integer was seen but junk-allowed was true.

The second value is either the index into the string of the delimiter that terminated the parse, or the upper bounding index of the substring if the parse terminated at the end of the substring (as is always the case if junk-allowed is false).

Examples:

(parse-integer "123") → 123, 3
(parse-integer "123" :start 1 :radix 5) → 13, 3
(parse-integer "no-integer" :junk-allowed t) → NIL, 0

Exceptional Situations:
If junk-allowed is false, an error is signaled if substring does not consist entirely of the representation of an integer, possibly surrounded on either side by whitespace characters.

boole Function

Syntax:
boole op integer-1 integer-2 → result-integer

Arguments and Values:
Op—a bit-wise logical operation specifier.
integer-1—an integer.
**boole**

integer-2—an integer.

result-integer—an integer.

**Description:**

boole performs bit-wise logical operations on integer-1 and integer-2, which are treated as if they were binary and in two’s complement representation.

The operation to be performed and the return value are determined by op.

boole returns the values specified for any op in Figure 12–17.

<table>
<thead>
<tr>
<th>Op</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>boole-1</td>
<td>integer-1</td>
</tr>
<tr>
<td>boole-2</td>
<td>integer-2</td>
</tr>
<tr>
<td>boole-andc1</td>
<td>and complement of integer-1 with integer-2</td>
</tr>
<tr>
<td>boole-andc2</td>
<td>and integer-1 with complement of integer-2</td>
</tr>
<tr>
<td>boole-and</td>
<td>and</td>
</tr>
<tr>
<td>boole-c1</td>
<td>complement of integer-1</td>
</tr>
<tr>
<td>boole-c2</td>
<td>complement of integer-2</td>
</tr>
<tr>
<td>boole-clr</td>
<td>always 0 (all zero bits)</td>
</tr>
<tr>
<td>boole-equiv</td>
<td>equivalence (exclusive nor)</td>
</tr>
<tr>
<td>boole-ior</td>
<td>inclusive or</td>
</tr>
<tr>
<td>boole-nand</td>
<td>not-and</td>
</tr>
<tr>
<td>boole-nor</td>
<td>not-or</td>
</tr>
<tr>
<td>boole-orc1</td>
<td>or complement of integer-1 with integer-2</td>
</tr>
<tr>
<td>boole-orc2</td>
<td>or integer-1 with complement of integer-2</td>
</tr>
<tr>
<td>boole-set</td>
<td>always -1 (all one bits)</td>
</tr>
<tr>
<td>boole-xor</td>
<td>exclusive or</td>
</tr>
</tbody>
</table>

Figure 12–17. Bit-Wise Logical Operations

**Examples:**

```
(boole boole-ior 1 16) → 17
(boole boole-and -2 5) → 4
(boole boole-equiv 17 15) → -31
```

;; These examples illustrate the result of applying BOOLE and each
;; of the possible values of OP to each possible combination of bits.

```
(progn
  (format t "Results of (BOOLE <op> #b0011 #b0101) ...
           "
           "---Op-------Decimal-----Binary----Bits---%")
  (dolist (symbol '(boole-1 boole-2 boole-and boole-andc1
                 boole-andc2 boole-c1 boole-c2 boole-clr
                 boole-equiv boole-ior boole-nand boole-nor
                 boole-orc1 boole-orc2 boole-set boole-xor))
    (format t (symbol) symbol))
```

12–68 Programming Language—Common Lisp
boole

(let ((result (boole (symbol-value symbol) #b0011 #b0101)))
  (format t "& " "A"13T"3, ' D"23T":*5,' B"31T .."4,'0B"%" 
            symbol result (logand result #b1111))))

▷ Results of (BOOLE <op> #b0011 #b0101) ...
▷ ---Op------Decimal------Binary-----Bits---
▷ BOOLE-1 3 11 ...0011
▷ BOOLE-2 5 101 ...0101
▷ BOOLE-AND 1 1 ...0001
▷ BOOLE-ANDC1 4 100 ...0100
▷ BOOLE-ANDC2 2 10 ...0010
▷ BOOLE-C1 -4 -100 ...1100
▷ BOOLE-C2 -6 -110 ...1010
▷ BOOLE-CLR 0 0 ...0000
▷ BOOLE-EQV -7 -111 ...1001
▷ BOOLE-IDR 7 111 ...0111
▷ BOOLE-NAND -2 -10 ...1110
▷ BOOLE-NOR -8 -1000 ...1000
▷ BOOLE-ORC1 -3 -11 ...1100
▷ BOOLE-ORC2 -5 -101 ...1011
▷ BOOLE-SET -1 -1 ...1111
▷ BOOLE-XOR 6 110 ...0110

⇒ NIL

Exceptional Situations:
Should signal type-error if its first argument is not a bit-wise logical operation specifier or if any subsequent argument is not an integer.

See Also:
logand

Notes:
In general,

(boole boole-and x y) ≡ (logand x y)

Programmers who would prefer to use numeric indices rather than bit-wise logical operation specifiers can get an equivalent effect by a technique such as the following:

;; The order of the values in this 'table' are such that
;; (logand (boole (elt boole-n-vector n) #b0101 #b0011) #b1111) => n
(defconstant boole-n-vector
  (vector boole-clr boole-and boole-andc1 boole-2
    boole-andc2 boole-1 boole-xor boole-ior
    boole-nor boole-eqv boole-c1 boole-orc1

Numbers  12–69
boole-c2, boole-orc2, boole-nand, boole-set)

→ BOOLE-N-VECTOR
(proclaim '(inline boole-n))
→ implementation-dependent
(defun boole-n (n integer &rest more-integers)
  (apply #'boole (elt boole-n-vector n) integer more-integers))
→ BOOLE-N
(boole-n #b0111 5 3) → 7
(boole-n #b0001 5 3) → 1
(boole-n #b1101 5 3) → -3
(loop for n from #b0000 to #b1111 collect (boole-n n 5 3))
→ (0 1 2 3 4 5 6 7 -8 -7 -6 -5 -4 -3 -2 -1)

boole-1, boole-2, boole-and, boole-andc1, boole-andc2, boole-c1, boole-c2, boole-clr, boole-eqv, boole-ior, boole-nand, boole-nor, boole-orc1, boole-orc2, boole-set, boole-xor

Constant Variable

Constant Value:
The identity and nature of the values of each of these variables is implementation-dependent, except that it must be distinct from each of the values of the others, and it must be a valid first argument to the function boole.

Description:
Each of these constants has a value which is one of the sixteen possible bit-wise logical operation specifiers.

Examples:

 (boole boole-ior 1 16) → 17
 (boole boole-and -2 5) → 4
 (boole boole-eqv 17 15) → -31

See Also:
boole

12–70 Programming Language—Common Lisp
logand, logandc1, logandc2, logeqv, logior, lognand, ...

logand, logandc1, logandc2, logeqv, logior, lognand, lognor, lognot, logorc1, logorc2, logxor

Syntax:

logand &rest integers → result-integer
logandc1 integer-1 integer-2 → result-integer
logandc2 integer-1 integer-2 → result-integer
logeqv &rest integers → result-integer
logior &rest integers → result-integer
lognand integer-1 integer-2 → result-integer
lognor integer-1 integer-2 → result-integer
lognot integer → result-integer
logorc1 integer-1 integer-2 → result-integer
logorc2 integer-1 integer-2 → result-integer
logxor &rest integers → result-integer

Arguments and Values:

integers—integers.
integer—an integer.
integer-1—an integer.
integer-2—an integer.
result-integer—an integer.

Description:

The functions logandc1, logandc2, logand, logeqv, logior, lognand, lognor, lognot, logorc1, logorc2, and logxor perform bit-wise logical operations on their arguments, that are treated as if they were binary.

Figure 12–18 lists the meaning of each of the functions. Where an ‘identity’ is shown, it indicates the value yielded by the function when no arguments are supplied.
### Function Identity Operation performed

<table>
<thead>
<tr>
<th>Function</th>
<th>Identity</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>logandc1</td>
<td>—</td>
<td>and complement of integer-1 with integer-2</td>
</tr>
<tr>
<td>logandc2</td>
<td>—</td>
<td>integer-1 with complement of integer-2</td>
</tr>
<tr>
<td>logand</td>
<td>-1</td>
<td>and</td>
</tr>
<tr>
<td>logeqv</td>
<td>-1</td>
<td>equivalence (exclusive nor)</td>
</tr>
<tr>
<td>logior</td>
<td>0</td>
<td>inclusive or</td>
</tr>
<tr>
<td>lognand</td>
<td>—</td>
<td>complement of integer-1 and integer-2</td>
</tr>
<tr>
<td>lognor</td>
<td>—</td>
<td>complement of integer-1 or integer-2</td>
</tr>
<tr>
<td>lognot</td>
<td>—</td>
<td>complement</td>
</tr>
<tr>
<td>logorc1</td>
<td>—</td>
<td>or complement of integer-1 with integer-2</td>
</tr>
<tr>
<td>logorc2</td>
<td>—</td>
<td>or integer-1 with complement of integer-2</td>
</tr>
<tr>
<td>logxor</td>
<td>0</td>
<td>exclusive or</td>
</tr>
</tbody>
</table>

#### Figure 12–18. Bit-wise Logical Operations on Integers

Negative integers are treated as if they were in two’s-complement notation.

**Examples:**

```lisp
(logior 1 2 4 8) → 15
(logxor 1 3 7 15) → 10
(logeqv) → -1
(logand 16 31) → 16
(lognot 0) → -1
(lognot 1) → -2
(lognot -1) → 0
(lognot (1+ (lognot 1000))) → 999

;;; In the following example, m is a mask. For each bit in
;;; the mask that is a 1, the corresponding bits in x and y are
;;; exchanged. For each bit in the mask that is a 0, the
;;; corresponding bits of x and y are left unchanged.
(let ((show (m x y)
    (format t "#m = #o~6,'0O~%x = #o~6,'0O~%y = #o~6,'0O~%m x y))

(let ((m #o007750)
    (x #o452576)
    (y #o317407))
    (show m x y)
    (let ((z (logand (logxor x y) m)))
        (setq x (logxor z x))
        (setq y (logxor z y))
        (show m x y))))
```

> m = #o007750

**12–72** Programming Language—Common Lisp
Exceptional Situations:
Should signal type-error if any argument is not an integer.

See Also:
boole

Notes:
(logbitp k -1) returns true for all values of k.

Because the following functions are not associative, they take exactly two arguments rather than
any number of arguments.

(logand n1 n2) ≡ (lognot (logand n1 n2))
(lognor n1 n2) ≡ (lognot (logior n1 n2))
(logandc1 n1 n2) ≡ (logand (lognot n1) n2)
(logandc2 n1 n2) ≡ (logand n1 (lognot n2))
(logiorc1 n1 n2) ≡ (logior (lognot n1) n2)
(logiorc2 n1 n2) ≡ (logior n1 (lognot n2))
(logbitp j (lognot x)) ≡ (not (logbitp j x))

logbitp

Function

Syntax:
logbitp index integer  → generalized-boolean

Arguments and Values:
index—a non-negative integer.
integer—an integer.
generalized-boolean—a generalized boolean.
Description:

logbitp is used to test the value of a particular bit in integer, that is treated as if it were binary. The value of logbitp is true if the bit in integer whose index is index (that is, its weight is $2^{\text{index}}$) is a one-bit; otherwise it is false.

Negative integers are treated as if they were in two's-complement notation.

Examples:

- (logbitp 1 1) → false
- (logbitp 0 1) → true
- (logbitp 3 10) → true
- (logbitp 1000000 -1) → true
- (logbitp 2 6) → true
- (logbitp 0 6) → false

Exceptional Situations:

Should signal an error of type type-error if index is not a non-negative integer. Should signal an error of type type-error if integer is not an integer.

Notes:

- (logbitp k n) ≡ (ldb-test (byte 1 k) n)

logcount

Function

Syntax:

logcount integer → number-of-on-bits

Arguments and Values:

integer—an integer.

number-of-on-bits—a non-negative integer.

Description:

Computes and returns the number of bits in the two's-complement binary representation of integer that are ‘on’ or ‘set’. If integer is negative, the 0 bits are counted; otherwise, the 1 bits are counted.

Examples:

- (logcount 0) → 0
- (logcount -1) → 0

12–74 Programming Language—Common Lisp
(logcount 7) → 3
(logcount 13) → 3 ;Two’s-complement binary: ...0001101
(logcount -13) → 2 ;Two’s-complement binary: ...1110011
(logcount 30) → 4 ;Two’s-complement binary: ...0011110
(logcount -30) → 4 ;Two’s-complement binary: ...1100010
(logcount (expt 2 100)) → 1
(logcount (¬ (expt 2 100))) → 100
(logcount (¬ (1+ (expt 2 100)))) → 1

Exceptional Situations:
Should signal type-error if its argument is not an integer.

Notes:
Even if the implementation does not represent integers internally in two’s complement binary, logcount behaves as if it did.

The following identity always holds:

(logcount x) ≡ (logcount (- (+ x 1)))
≡ (logcount (lognot x))

logtest

Function

Syntax:

logtest integer-1 integer-2 → generalized-boolean

Arguments and Values:

integer-1—an integer.

integer-2—an integer.

generalized-boolean—a generalized boolean.

Description:

Returns true if any of the bits designated by the 1’s in integer-1 is 1 in integer-2; otherwise it is false. integer-1 and integer-2 are treated as if they were binary.

Negative integer-1 and integer-2 are treated as if they were represented in two’s-complement binary.
Examples:

```lisp
(logtest 1 7) → true
(logtest 1 2) → false
(logtest -2 -1) → true
(logtest 0 -1) → false
```

Exceptional Situations:
Should signal an error of type `type-error` if `integer-1` is not an `integer`. Should signal an error of `type type-error` if `integer-2` is not an `integer`.

Notes:

```lisp
(logtest x y) ≡ (not (zerop (logand x y)))
```

---

**byte, byte-size, byte-position**

*Function*

Syntax:

- `byte size position` → `bytespec`
- `byte-size bytespec` → `size`
- `byte-position bytespec` → `position`

Arguments and Values:
- `size, position`—a non-negative `integer`.
- `bytespec`—a `byte specifier`.

Description:

- `byte` returns a `byte specifier` that indicates a `byte` of width `size` and whose bits have weights $2^{position+size-1}$ through $2^{position}$, and whose representation is `implementation-dependent`.
- `byte-size` returns the number of bits specified by `bytespec`.
- `byte-position` returns the position specified by `bytespec`.

Examples:

```lisp
(setq b (byte 100 200)) → #<BYTE-SPECIFIER size 100 position 200>
(byte-size b) → 100
(byte-position b) → 200
```
See Also:
ldb, dpb

Notes:

\[
\begin{align*}
\text{byte-size (byte } j \ k) & \equiv j \\
\text{byte-position (byte } j \ k) & \equiv k
\end{align*}
\]

A byte of size of 0 is permissible; it refers to a byte of width zero. For example,

\[
\begin{align*}
(\text{ldb} (\text{byte} 0 3) \ #07777) & \rightarrow 0 \\
(\text{dpb} \ #07777 \ (\text{byte} 0 3) \ 0) & \rightarrow 0
\end{align*}
\]

### deposit-field

**Function**

Syntax:

\[\text{deposit-field} \ \text{newbyte} \ \text{bytespec} \ \text{integer} \ \rightarrow \ \text{result-integer}\]

Arguments and Values:

- **newbyte**—an integer.
- **bytespec**—a byte specifier.
- **integer**—an integer.
- **result-integer**—an integer.

Description:

Replaces a field of bits within integer; specifically, returns an integer that contains the bits of newbyte within the byte specified by bytespec, and elsewhere contains the bits of integer.

Examples:

\[
\begin{align*}
(\text{deposit-field} \ 7 \ (\text{byte} \ 2 \ 1) \ 0) & \rightarrow 6 \\
(\text{deposit-field} \ -1 \ (\text{byte} \ 4 \ 0) \ 0) & \rightarrow 15 \\
(\text{deposit-field} \ 0 \ (\text{byte} \ 2 \ 1) \ -3) & \rightarrow -7
\end{align*}
\]

See Also:
byte, dpb

Notes:

\[
(\text{logbitp} \ j \ (\text{deposit-field} \ m \ (\text{byte} \ s \ p) \ n)) \equiv (\text{if} \ (\text{and} \ (\geq j \ p) \ (< j \ (+ \ p \ s)))
\]

Numbers 12–77
(logbitp j m)
(logbitp j n))

deposit-field is to mask-field as dpb is to ldb.

dpb

Function

Syntax:

dpb newbyte bytespec integer → result-integer

Pronunciation:

[ˌdə'pib] or [ˌdə'pəb] or [ˈdē′pē′bē]

Arguments and Values:

c newbyte—an integer.

bytespec—a byte specifier.

integer—an integer.

result-integer—an integer.

Description:

dpb (deposit byte) is used to replace a field of bits within integer. dpb returns an integer that is the same as integer except in the bits specified by bytespec.

Let s be the size specified by bytespec; then the low s bits of newbyte appear in the result in the byte specified by bytespec. Newbyte is interpreted as being right-justified, as if it were the result of ldb.

Examples:

(dpb 1 (byte 1 10) 0) → 1024
(dpb -2 (byte 2 10) 0) → 2048
(dpb 1 (byte 2 10) 2048) → 1024

See Also:

byte, deposit-field, ldb

Notes:

(logbitp j (dpb m (byte s p) n))
≡ (if (and (≥ j p) (< j (+ p s)))
   (logbitp (- j p) m)

12–78 Programming Language—Common Lisp
In general, 

\[(dpb \ x \ (byte \ 0 \ y) \ z) \rightarrow z\]

for all valid values of \(x\), \(y\), and \(z\).

Historically, the name “dpb” comes from a DEC PDP-10 assembly language instruction meaning “deposit byte.”

---

**ldb**

*Accessor*

**Syntax:**

\[
\text{ldb} \ \text{bytespec} \ \text{integer} \ \rightarrow \ \text{byte}
\]

\[
\text{(setq} \ \text{(ldb} \ \text{bytespec} \ \text{place}) \ \text{new-byte})
\]

**Pronunciation:**

\[\text{"lidib"} \ \text{or} \ \text{"lidib"} \ \text{or} \ \text{"el\,d\,e\,b"}\]

**Arguments and Values:**

- \text{bytespec}—a byte specifier.
- \text{integer}—an integer.
- \text{byte}, \text{new-byte}—a non-negative integer.

**Description:**

\text{ldb} extracts and returns the byte of \text{integer} specified by \text{bytespec}.

\text{ldb} returns an integer in which the bits with weights \(2^{(s-1)}\) through \(2^0\) are the same as those in \text{integer} with weights \(2^{(p+s-1)}\) through \(2^p\), and all other bits zero; \(s\) is \text{(byte-size bytespec)} and \(p\) is \text{(byte-position bytespec)}.

\text{setf} may be used with \text{ldb} to modify a byte within the \text{integer} that is stored in a given \text{place}. The order of evaluation, when an \text{ldb} form is supplied to \text{setf}, is exactly left-to-right. The effect is to perform a \text{dpb} operation and then store the result back into the \text{place}.

**Examples:**

\[
(\text{ldb} \ (\text{byte} \ 2 \ 1) \ 10) \rightarrow 1
\]
\[
(\text{setq} \ \text{a} \ \text{(list} \ 8)) \rightarrow (8)
\]
\[
(\text{setf} \ \text{(ldb} \ \text{(byte} \ 2 \ 1) \ \text{(car} \ \text{a})) \ 1) \rightarrow 1
\]
\[ a \rightarrow (10) \]

**See Also:**
- byte, byte-position, byte-size, dpb

**Notes:**

\[
(\logbitp j (ldb (byte s p) n)) \\
\equiv (\text{and} (< s) (\logbitp (+ j p) n))
\]

In general,

\[
(ldb (byte 0 x) y) \rightarrow 0
\]

for all valid values of \( x \) and \( y \).

Historically, the name “ldb” comes from a DEC PDP-10 assembly language instruction meaning “load byte.”

---

### ldb-test

**Function**

**Syntax:**

\[
\text{ldb-test} \ \text{bytespec} \ \text{integer} \rightarrow \text{generalized-boolean}
\]

**Arguments and Values:**

- `bytespec`—a byte specifier.
- `integer`—an integer.
- `generalized-boolean`—a generalized boolean.

**Description:**

Returns `true` if any of the bits of the byte in `integer` specified by `bytespec` is non-zero; otherwise returns `false`.

**Examples:**

\[
(\text{ldb-test} \ (\text{byte} \ 4 \ 1) \ 16) \rightarrow \text{true} \\
(\text{ldb-test} \ (\text{byte} \ 3 \ 1) \ 16) \rightarrow \text{false} \\
(\text{ldb-test} \ (\text{byte} \ 3 \ 2) \ 16) \rightarrow \text{true}
\]

**See Also:**
- byte, ldb, zerop

---

Programming Language—Common Lisp
mask-field

Syntax:

\[
\text{mask-field bytespec integer } \rightarrow \text{masked-integer}
\]

\[
\text{(setf (mask-field bytespec place) new-masked-integer)}
\]

Arguments and Values:

- **bytespec**—a byte specifier.
- **integer**—an integer.
- **masked-integer, new-masked-integer**—a non-negative integer.

Description:

The **mask-field** performs a “mask” operation on **integer**. It returns an integer that has the same bits as **integer** in the byte specified by **bytespec**, but that has zero-bits everywhere else.

**setf** may be used with **mask-field** to modify a byte within the **integer** that is stored in a given **place**. The effect is to perform a **deposit-field** operation and then store the result back into the **place**.

Examples:

\[
\begin{align*}
\text{(mask-field (byte 1 5) -1)} & \rightarrow 32 \\
\text{(setq a 15)} & \rightarrow 15 \\
\text{(mask-field (byte 2 0) a)} & \rightarrow 3 \\
\text{a} & \rightarrow 15 \\
\text{(setf (mask-field (byte 2 0) a) 1)} & \rightarrow 1 \\
\text{a} & \rightarrow 13
\end{align*}
\]

See Also:

- byte
- ldb

Notes:

\[
\begin{align*}
\text{(ldb-test bytespec n)} & \equiv \\
\text{(not (zerop (ldb bytespec n)))} & \equiv \\
\text{(logtest (ldb bytespec -1) n)}
\end{align*}
\]
Notes:

\[(\text{ldb} \ bs \ (\text{mask-field} \ bs \ n)) \equiv (\text{ldb} \ bs \ n)\]
\[(\text{logbitp} \ j \ (\text{mask-field} \ (\text{byte} \ s \ p) \ n))\]
\[\equiv \ (\text{and} \ (> j \ p) \ (< j \ s) \ (\text{logbitp} \ j \ n))\]
\[(\text{mask-field} \ bs \ n) \equiv (\text{logand} \ n \ (\text{dpb} \ -1 \ bs \ 0))\]

**most-positive-fixnum, most-negative-fixnum** Constant

**Variable**

**Constant Value:**

*implementation-dependent.*

**Description:**

- **most-positive-fixnum** is that *fixnum* closest in value to positive infinity provided by the implementation, and greater than or equal to both \(2^{15} - 1\) and *array-dimension-limit*.
- **most-negative-fixnum** is that *fixnum* closest in value to negative infinity provided by the implementation, and less than or equal to \(-2^{15}\).

**decode-float, scale-float, float-radix, float-sign, float-digits, float-precision, integer-decode-float** Function

**Syntax:**

- decode-float float → significand, exponent, sign
- scale-float float integer → scaled-float
- float-radix float → float-radix
- float-sign float-1 &optional float-2 → signed-float
- float-digits float → digits1
- float-precision float → digits2
- integer-decode-float float → significand, exponent, integer-sign
decode-float, scale-float, float-radix, float-sign, ...

Arguments and Values:
- digits1—a non-negative integer.
- digits2—a non-negative integer.
- exponent—an integer.
- float—a float.
- float-1—a float.
- float-2—a float.
- float-radix—an integer.
- integer—a non-negative integer.
- integer-sign—the integer -1, or the integer 1.
- scaled-float—a float.
- sign—A float of the same type as float but numerically equal to 1.0 or -1.0.
- signed-float—a float.
- significand—a float.

Description:

decode-float computes three values that characterize float. The first value is of the same type as float and represents the significand. The second value represents the exponent to which the radix (notated in this description by b) must be raised to obtain the value that, when multiplied with the first result, produces the absolute value of float. If float is zero, any integer value may be returned, provided that the identity shown for scale-float holds. The third value is of the same type as float and is 1.0 if float is greater than or equal to zero or -1.0 otherwise.

decode-float divides float by an integral power of b so as to bring its value between 1/b (inclusive) and 1 (exclusive), and returns the quotient as the first value. If float is zero, however, the result equals the absolute value of float (that is, if there is a negative zero, its significand is considered to be a positive zero).

scale-float returns (* float (expt (float b float) integer)), where b is the radix of the floating-point representation. float is not necessarily between 1/b and 1.

float-radix returns the radix of float.

float-sign returns a number z such that z and float-1 have the same sign and also such that z and float-2 have the same absolute value. If float-2 is not supplied, its value is (float 1 float-1). If an implementation has distinct representations for negative zero and positive zero, then (float-sign -0.0) → -1.0.
**decode-float, scale-float, float-radix, float-sign, ...**

- **float-digits** returns the number of radix \( b \) digits used in the representation of \( \text{float} \) (including any implicit digits, such as a “hidden bit”).

- **float-precision** returns the number of significant radix \( b \) digits present in \( \text{float} \); if \( \text{float} \) is a \( \text{float} \) zero, then the result is an integer zero.

For normalized floats, the results of **float-digits** and **float-precision** are the same, but the precision is less than the number of representation digits for a denormalized or zero number.

**integer-decode-float** computes three values that characterize \( \text{float} \) - the significand scaled so as to be an integer, and the same last two values that are returned by **decode-float**. If \( \text{float} \) is zero, **integer-decode-float** returns zero as the first value. The second value bears the same relationship to the first value as for **decode-float**:

\[
\begin{align*}
\text{(multiple-value-bind (signif expon sign)}).
\text{(integer-decode-float f)}
\text{≡ (abs f)}
\end{align*}
\]

**Examples:**

;; Note that since the purpose of this functionality is to expose
;; details of the implementation, all of these examples are necessarily
;; very implementation-dependent. Results may vary widely.
;; Values shown here are chosen consistently from one particular implementation.

(d) (decode-float .5) \(→ 0.5, 0, 1.0 \)
(d) (decode-float 1.0) \(→ 0.5, 1, 1.0 \)
(s) (scale-float 1.0 1) \(→ 2.0 \)
(s) (scale-float 10.01 -2) \(→ 2.5025 \)
(s) (scale-float 23.0 0) \(→ 23.0 \)
(r) (float-radix 1.0) \(→ 2 \)
(s) (float-sign 5.0) \(→ 1.0 \)
(s) (float-sign -5.0) \(→ -1.0 \)
(s) (float-sign 0.0) \(→ 1.0 \)
(s) (float-sign 1.0 0.0) \(→ 0.0 \)
(s) (float-sign 1.0 -10.0) \(→ 10.0 \)
(s) (float-sign -1.0 10.0) \(→ -10.0 \)
(s) (float-sign 1.0) \(→ 24 \)
(r) (float-radix 1.0) \(→ 24 \)
(r) (float-sign 1.0) \(→ 1 \)
(r) (integer-decode-float 1.0) \(→ 8388608, -23, 1 \)

**Affected By:**

The implementation’s representation for \( \text{floats} \).

**Exceptional Situations:**

- The functions **decode-float**, **float-radix**, **float-sign**, **float-digits**, **float-precision**, and **integer-decode-float** should signal an error if their only argument is not a float.

12–84 Programming Language—Common Lisp
The function scale-float should signal an error if its first argument is not a float or if its second argument is not an integer.

The function float-sign should signal an error if its first argument is not a float or if its second argument is supplied but is not a float.

Notes:

The product of the first result of decode-float or integer-decode-float, of the radix raised to the power of the second result, and of the third result is exactly equal to the value of float.

\[
\begin{align*}
\text{(multiple-value-bind (signif expon sign) (decode-float f))} \\
\text{≡ (abs f)}
\end{align*}
\]

and

\[
\begin{align*}
\text{(multiple-value-bind (signif expon sign) (decode-float f))} \\
\text{(* (scale-float signif expon) sign))} \\
\text{≡ f}
\end{align*}
\]

float

Function

Syntax:

float number koptional prototype → float

Arguments and Values:

number—a real.

prototype—a float.

float—a float.

Description:

float converts a real number to a float.

If a prototype is supplied, a float is returned that is mathematically equal to number but has the same format as prototype.

If prototype is not supplied, then if the number is already a float, it is returned; otherwise, a float is returned that is mathematically equal to number but is a single float.
Examples:

(float 0) → 0.0
(float 1.5) → 1.0
(float 1.0) → 1.0
(float 1/2) → 0.5
    → 1.0d0
as 1.0
(eql (float 1.0 1.0d0) 1.0d0) → true

See Also:
    coerce

---

**floatp**

*Function*

Syntax:

```lisp
floatp object
```

generalized-boolean

Arguments and Values:

- **object**—an *object*.
- **generalized-boolean**—a *generalized boolean*.

Description:

Returns true if *object* is of type float; otherwise, returns false.

Examples:

(floatp 1.2d2) → true
(floatp 1.21d2) → true
(floatp 1.2s2) → true
(floatp (expt 2 130)) → false

Notes:

(floatp object) ≡ (typep object 'float)
most-positive-short-float, least-positive-short-float, ...


Constant Variable

Constant Value:
implementation-dependent.

Description:
These constant variables provide a way for programs to examine the implementation-defined limits for the various float formats.

Of these variables, each which has “-normalized” in its name must have a value which is a normalized float, and each which does not have “-normalized” in its name may have a value which is either a normalized float or a denormalized float, as appropriate.

Of these variables, each which has “short-float” in its name must have a value which is a short float, each which has “single-float” in its name must have a value which is a single float, each which has “double-float” in its name must have a value which is a double float, and each which has “long-float” in its name must have a value which is a long float.

• most-positive-short-float, most-positive-single-float, most-positive-double-float, most-positive-long-float

Each of these constant variables has as its value the positive float of the largest magni-
tude (closest in value to, but not equal to, positive infinity) for the float format implied by its name.


Each of these constant variables has as its value the smallest positive (nonzero) float for the float format implied by its name.

- least-negative-short-float, least-negative-normalized-short-float, least-negative-single-float, least-negative-normalized-single-float, least-negative-double-float, least-negative-normalized-double-float, least-negative-long-float, least-negative-normalized-long-float

Each of these constant variables has as its value the negative (nonzero) float of the smallest magnitude for the float format implied by its name. (If an implementation supports minus zero as a different object from positive zero, this value must not be minus zero.)

- most-negative-short-float, most-negative-single-float, most-negative-double-float, most-negative-long-float

Each of these constant variables has as its value the negative float of the largest magnitude (closest in value to, but not equal to, negative infinity) for the float format implied by its name.

Notes:

short-float-epsilon, short-float-negative-epsilon, single-float-epsilon, single-float-negative-epsilon, double-float-epsilon, double-float-negative-epsilon, long-float-epsilon, long-float-negative-epsilon

Constant Variable

Constant Value:
implementation-dependent.

Description:
The value of each of the constants short-float-epsilon, single-float-epsilon, double-float-epsilon, and long-float-epsilon is the smallest positive float $\epsilon$ of the given format, such that the following expression is true when evaluated:
(not (= (float 1 \(\epsilon\)) (+ (float 1 \(\epsilon\)) \(\epsilon\))))

The value of each of the constants \texttt{short-float-negative-epsilon}, \texttt{single-float-negative-epsilon}, \texttt{double-float-negative-epsilon}, and \texttt{long-float-negative-epsilon} is the smallest positive float \(\epsilon\) of the given format, such that the following expression is true when evaluated:

(not (= (float 1 \(\epsilon\)) (- (float 1 \(\epsilon\)) \(\epsilon\))))

\textbf{arithmetic-error} 

\textit{Condition Type}

\textbf{Class Precedence List:}
\texttt{arithmetic-error}, \texttt{error}, \texttt{serious-condition}, \texttt{condition}, \texttt{t}

\textbf{Description:}
The type \texttt{arithmetic-error} consists of error conditions that occur during arithmetic operations. The operation and operands are initialized with the initialization arguments named \texttt{:operation} and \texttt{:operands} to \texttt{make-condition}, and are accessed by the functions \texttt{arithmetic-error-operation} and \texttt{arithmetic-error-operands}.

\textbf{See Also:}
\texttt{arithmetic-error-operation}, \texttt{arithmetic-error-operands}

\textbf{arithmetic-error-operands, arithmetic-error-operation} 

\textit{Function}

\textbf{Syntax:}
\begin{align*}
\texttt{arithmetic-error-operands} & \texttt{condition} \rightarrow \texttt{operands} \\
\texttt{arithmetic-error-operation} & \texttt{condition} \rightarrow \texttt{operation}
\end{align*}

\textbf{Arguments and Values:}
\begin{itemize}
\item \texttt{condition}—a \texttt{condition} of type \texttt{arithmetic-error}.
\item \texttt{operands}—a \texttt{list}.
\item \texttt{operation}—a \texttt{function designator}.
\end{itemize}

\textbf{Description:}
\texttt{arithmetic-error-operands} returns a \texttt{list} of the operands which were used in the offending call to the operation that signaled the \texttt{condition}.
arithmetic-error-operation returns a list of the offending operation in the offending call that signaled the condition.

See Also:
arithmetic-error, Chapter 9 (Conditions)

Notes:

division-by-zero

Class Precedence List:
division-by-zero, arithmetic-error, error, serious-condition, condition, t

Description:
The type division-by-zero consists of error conditions that occur because of division by zero.

floating-point-invalid-operation

Class Precedence List:
floating-point-invalid-operation, arithmetic-error, error, serious-condition, condition, t

Description:
The type floating-point-invalid-operation consists of error conditions that occur because of certain floating point traps.

It is implementation-dependent whether floating point traps occur, and whether or how they may be enabled or disabled. Therefore, conforming code may establish handlers for this condition, but must not depend on its being signaled.
**floating-point-inexact**

*Condition Type*

Class Precedence List:

floating-point-inexact, arithmetic-error, error, serious-condition, condition, t

Description:

The type `floating-point-inexact` consists of error conditions that occur because of certain floating point traps.

It is *implementation-dependent* whether floating point traps occur, and whether or how they may be enabled or disabled. Therefore, conforming code may establish handlers for this condition, but must not depend on its being *signaled*.

---

**floating-point-overflow**

*Condition Type*

Class Precedence List:

floating-point-overflow, arithmetic-error, error, serious-condition, condition, t

Description:

The type `floating-point-overflow` consists of error conditions that occur because of floating-point overflow.

---

**floating-point-underflow**

*Condition Type*

Class Precedence List:

floating-point-underflow, arithmetic-error, error, serious-condition, condition, t

Description:

The type `floating-point-underflow` consists of error conditions that occur because of floating-point underflow.
Programming Language—Common Lisp

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.

Figure 13–1 lists some defined names relating to character attributes and character predicates.

<table>
<thead>
<tr>
<th>Defined Name</th>
<th>Defined Name</th>
<th>Defined Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha-char-p</td>
<td>char-not-equal</td>
<td>char&gt;</td>
</tr>
<tr>
<td>alphanumericp</td>
<td>char-not-greaterp</td>
<td>char&gt;:=</td>
</tr>
<tr>
<td>both-case-p</td>
<td>char-not-lessp</td>
<td>digit-char-p</td>
</tr>
<tr>
<td>char-code-limit</td>
<td>char/=</td>
<td>graphic-char-p</td>
</tr>
<tr>
<td>char-equal</td>
<td>char&lt;</td>
<td>lower-case-p</td>
</tr>
<tr>
<td>char-greaterp</td>
<td>char&lt;=</td>
<td>standard-char-p</td>
</tr>
<tr>
<td>char-lessp</td>
<td>char=</td>
<td>upper-case-p</td>
</tr>
</tbody>
</table>

Figure 13–1. Character defined names – 1

Figure 13–2 lists some character construction and conversion defined names.

<table>
<thead>
<tr>
<th>Defined Name</th>
<th>Defined Name</th>
<th>Defined Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>char-code</td>
<td>char-name</td>
<td>code-char</td>
</tr>
<tr>
<td>char-downcase</td>
<td>char-upcase</td>
<td>digit-char</td>
</tr>
<tr>
<td>char-int</td>
<td>character</td>
<td>name-char</td>
</tr>
</tbody>
</table>

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\textsubscript{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 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\textsubscript{1} characters are a subset of the graphic characters. Of the standard characters, only these are the alphabetic\textsubscript{1} characters:

\begin{verbatim}
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
\end{verbatim}

Any implementation-defined character that has case must be alphabetic\textsubscript{1}. For each implementation-defined graphic character that has no case, it is implementation-defined whether that character is alphabetic\textsubscript{1}.

13.1.4.3 Characters With Case

The characters with case are a subset of the alphabetic\textsubscript{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 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 $0, 1, 2, \ldots, 35$. 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 alphabetic1 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*, alphabetic1 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 \#\texttt{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, \#\texttt{Linefeed} and \#\texttt{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 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.
character

System Class

Class Precedence List:
character, t

Description:
A character is an object that represents a unitary token in an aggregate quantity of text; see Section 13.1 (Character Concepts).

The types base-char and extended-char form an exhaustive partition of the type character.

See Also:
Section 13.1 (Character Concepts), Section 2.4.8.1 (Sharpsign Backslash), Section 22.1.3.2 (Printing Characters)

base-char

Type

Supertypes:
base-char, character, t

Description:
The type base-char is defined as the upgraded array element type of standard-char. An implementation can support additional subtypes of type character (besides the ones listed in this standard) that might or might not be supertypes of type base-char. In addition, an implementation can define base-char to be the same type as character.

Base characters are distinguished in the following respects:

1. The type standard-char is a subrepertoire of the type base-char.
2. The selection of base characters that are not standard characters is implementation defined.
3. Only objects of the type base-char can be elements of a base string.
4. No upper bound is specified for the number of characters in the base-char repertoire; the size of that repertoire is implementation-defined. The lower bound is 96, the number of standard characters.

Whether a character is a base character depends on the way that an implementation represents strings, and not any other properties of the implementation or the host operating system. For example, one implementation might encode all strings as characters having 16-bit encodings, and another might have two kinds of strings: those with characters having 8-bit encodings and those with characters having 16-bit encodings. In the first implementation, the type base-char is
equivalent to the type character: there is only one kind of string. In the second implementation, the base characters might be those characters that could be stored in a string of characters having 8-bit encodings. In such an implementation, the type base-char is a proper subtype of the type character.

The type standard-char is a subtype of type base-char.

---

**standard-char**

**Type**

**Supertypes:**
standard-char, base-char, character, t

**Description:**
A fixed set of 96 characters required to be present in all conforming implementations. Standard characters are defined in Section 2.1.3 (Standard Characters).

Any character that is not simple is not a standard character.

**See Also:**
Section 2.1.3 (Standard Characters)

---

**extended-char**

**Type**

**Supertypes:**
extended-char, character, t

**Description:**
The type extended-char is equivalent to the type (and character (not base-char)).

**Notes:**
The type extended-char might have no elements in implementations in which all characters are of type base-char.
char=, char/=, char<, char>, char<=, char>=, ...

char=, char/=, char<, char>, char<=, char>=,
char-equal, char-not-equal, char-lessp, char-greaterp, char-not-greaterp, char-not-lessp

Function

Syntax:

char = &rest characters+ → generalized-boolean
char /= &rest characters+ → generalized-boolean
char < &rest characters+ → generalized-boolean
char <= &rest characters+ → generalized-boolean
char > &rest characters+ → generalized-boolean
char >= &rest characters+ → generalized-boolean
char-equal &rest characters+ → generalized-boolean
char-not-equal &rest characters+ → generalized-boolean
char-lessp &rest characters+ → generalized-boolean
char-greaterp &rest characters+ → generalized-boolean
char-not-greaterp &rest characters+ → generalized-boolean
char-not-lessp &rest characters+ → generalized-boolean

Arguments and Values:

character—a character.

generalized-boolean—a generalized boolean.

Description:

These predicates compare characters.

char= returns true if all characters are the same; otherwise, it returns false. If two characters differ in any implementation-defined attributes, then they are not char=.

char/= returns true if all characters are different; otherwise, it returns false.

char< returns true if the characters are monotonically increasing; otherwise, it returns false. If two characters have identical implementation-defined attributes, then their ordering by char< is consistent with the numerical ordering by the predicate < on their codes.

char> returns true if the characters are monotonically decreasing; otherwise, it returns false. If two characters have identical implementation-defined attributes, then their ordering by char> is consistent with the numerical ordering by the predicate > on their codes.

char<= returns true if the characters are monotonically nondecreasing; otherwise, it returns false. If two characters have identical implementation-defined attributes, then their ordering by char<= is consistent with the numerical ordering by the predicate <= on their codes.

char>= returns true if the characters are monotonically nonincreasing; otherwise, it returns false.
If two characters have identical implementation-defined attributes, then their ordering by \texttt{char\>=} is consistent with the numerical ordering by the predicate \texttt{>=} on their codes.

\texttt{char-equal, char-not-equal, char-lessp, char-greaterp, char-not-greaterp, and char-not-lessp} are similar to \texttt{char=, char/=, char<, char>, char<=, char>=}, respectively, except that they ignore differences in case and might have an implementation-defined behavior for non-simple characters. For example, an implementation might define that \texttt{char-equal, etc.} ignore certain implementation-defined attributes. The effect, if any, of each implementation-defined attribute upon these functions must be specified as part of the definition of that attribute.

\textbf{Examples:}

\begin{verbatim}
(char= #\d #\d) → true
(char= #\A #\a) → false
(char= #\d #\x) → false
(char= #\d #\D) → false
(char/= #\d #\d) → false
(char/= #\d #\x) → true
(char/= #\d #\D) → true
(char= #\d #\d #\d #\d) → true
(char/= #\d #\d #\d #\d) → false
(char= #\d #\d #\x #\d) → false
(char/= #\d #\y #\x #\c) → false
(char/= #\d #\y #\x #\c) → true
(char= #\d #\c #\d) → false
(char/= #\d #\c #\d) → false
(char= #\d #\y #\x #\c) → false
(char/= #\d #\y #\x #\c) → true
(char< #\d #\x) → true
(char< #\d #\d) → false
(char< #\a #\e #\y #\z) → true
(char< #\a #\e #\y #\z) → false
(char< #\a #\e #\e #\y) → true
(char< #\e #\d) → true
(char= #\e #\d) → true
(char= #\d #\c #\d) → true
(char= #\d #\c #\b #\a) → true
(char= #\d #\c #\b #\a) → false
(char= #\d #\c #\b #\a) → true
(char= #\e #\d #\b #\c #\a) → false
(char= #\e #\d #\b #\c #\a) → true
(char= #\z #\A) → implementation-dependent
(char= #\Z #\a) → implementation-dependent
(char-equal #\A #\a) → true
\end{verbatim}
(stable-sort (list #\b #\A #\B #\a #\c #\C) #'char-lessp)
→ (#\A #\a #\b #\B #\c #\C)
(stable-sort (list #\b #\A #\B #\a #\c #\C) #'char<)
→ (#\a #\A #\b #\#B #\c #\C) ; Implementation A
→ (#\a #\b #\#c #\#A #\B #\C) ; Implementation B
→ (#\a #\A #\b #\B #\c #\C) ; Implementation C
→ (#\a #\B #\a #\#b #\C #\c) ; Implementation D
→ (#\a #\B #\a #\#b #\C #\C) ; Implementation E

Exceptional Situations:
Should signal an error of type program-error if at least one character is not supplied.

See Also:
Section 2.1 (Character Syntax), Section 13.1.10 (Documentation of Implementation-Defined Scripts)

Notes:
If characters differ in their code attribute or any implementation-defined attribute, they are considered to be different by char=.

There is no requirement that (eq c1 c2) be true merely because (char= c1 c2) is true. While eq can distinguish two characters that char= does not, it is distinguishing them not as characters, but in some sense on the basis of a lower level implementation characteristic. If (eq c1 c2) is true, then (char= c1 c2) is also true. eql and equal compare characters in the same way that char= does.

The manner in which case is used by char-equal, char-not-equal, char-lessp, char-greaterp, char-not-greaterp, and char-not-lessp implies an ordering for standard characters such that A=a, B=b, and so on, up to Z=z, and furthermore either 9<A or Z<0.

character

Function

Syntax:
character character  → denoted-character

Arguments and Values:
character—a character designator.

denoted-character—a character.

Description:
Returns the character denoted by the character designator.
Examples:

(character \a) → \a
(character "a") → \a
(character 'a) → \A
(character '\a) → \a
(character 65.) is an error.
(character 'apple) is an error.

Exceptional Situations:
Should signal an error of type type-error if object is not a character designator.

See Also:
coerce

Notes:

(character object) ≡ (coerce object 'character)

characterp

Function

Syntax:
characterp object → generalized-boolean

Arguments and Values:
object—an object.
generalized-boolean—a generalized boolean.

Description:
Returns true if object is of type character; otherwise, returns false.

Examples:

(character #\a) → true
(character 'a) → false
(character "a") → false
(character 65.) → false
(character #\\Newline) → true
;; This next example presupposes an implementation
;; in which #\Rubout is an implementation-defined character.
(character #\Rubout) → true
See Also:
character (type and function), typep

Notes:
(characterp object) ≡ (typep object 'character)

---

**alpha-char-p**

*Function*

**Syntax:**

\[
\text{alpha-char-p } \text{character} \rightarrow \text{generalized-boolean}
\]

**Arguments and Values:**

- **character**—a character.
- **generalized-boolean**—a generalized boolean.

**Description:**

Returns *true* if **character** is an alphabetic character; otherwise, returns *false*.

**Examples:**

\[
\begin{align*}
(\text{alpha-char-p } \text{#\a}) & \rightarrow \text{true} \\
(\text{alpha-char-p } \text{#\5}) & \rightarrow \text{false} \\
(\text{alpha-char-p } \text{#\Newline}) & \rightarrow \text{false} \\
;; \text{This next example presupposes an implementation} \\
;; \text{in which } \text{#\a} \text{ is a defined character.} \\
(\text{alpha-char-p } \text{#\a}) & \rightarrow \text{implementation-dependent}
\end{align*}
\]

**Affected By:**

None. (In particular, the results of this predicate are independent of any special syntax which might have been enabled in the current readtable.)

**Exceptional Situations:**

Should signal an error of *type type-error* if **character** is not a character.

**See Also:**

alphanumericp, Section 13.1.10 (Documentation of Implementation-Defined Scripts)
alphanumericp

Function

Syntax:
  alphanumericp character → generalized-boolean

Arguments and Values:
  character—a character.
  generalized-boolean—a generalized boolean.

Description:
  Returns true if character is an alphabetic character or a numeric character; otherwise, returns false.

Examples:

  (alphanumericp #\Z) → true
  (alphanumericp #\9) → true
  (alphanumericp #\Newline) → false
  (alphanumericp #\#) → false

Affected By:
  None. (In particular, the results of this predicate are independent of any special syntax which might have been enabled in the current readtable.)

Exceptional Situations:
  Should signal an error of type type-error if character is not a character.

See Also:
  alpha-char-p, graphic-char-p, digit-char-p

Notes:
  Alphanumeric characters are graphic as defined by graphic-char-p. The alphanumeric characters are a subset of the graphic characters. The standard characters A through Z, a through z, and 0 through 9 are alphanumeric characters.

  (alphanumericp x) ≡ (or (alpha-char-p x) (not (null (digit-char-p x))))
**digit-char**

**Function**

Syntax:

\[
\text{digit-char \ weight \ &optional \ radix \ → \ char}
\]

Arguments and Values:

- \(\text{weight}\)—a non-negative \text{integer}.
- \(\text{radix}\)—a \text{radix}. The default is 10.
- \(\text{char}\)—a \text{character} or \text{false}.

Description:

If \(\text{weight}\) is less than \(\text{radix}\), \text{digit-char} returns a \text{character} which has that \text{weight} when considered as a digit in the specified \text{radix}. If the resulting \text{character} is to be an \text{alphabetic} character, it will be an uppercase \text{character}.

If \(\text{weight}\) is greater than or equal to \(\text{radix}\), \text{digit-char} returns \text{false}.

Examples:

- \((\text{digit-char} 0) \rightarrow \#\0\)
- \((\text{digit-char} 10 11) \rightarrow \#\A\)
- \((\text{digit-char} 10 10) \rightarrow \text{false}\)
- \((\text{digit-char} 7) \rightarrow \#\7\)
- \((\text{digit-char} 12) \rightarrow \text{false}\)
- \((\text{digit-char} 12 16) \rightarrow \#\C\); not \#\c
- \((\text{digit-char} 6 2) \rightarrow \text{false}\)
- \((\text{digit-char} 1 2) \rightarrow \#\1\)

See Also:

digit-char-p, graphic-char-p, Section 2.1 (Character Syntax)

Notes:

digit-char-p, graphic-char-p, Section 2.1 (Character Syntax)

**digit-char-p**

**Function**

Syntax:

\[
\text{digit-char-p \ char \ &optional \ radix \ → \ weight}
\]

Arguments and Values:

- \(\text{char}\)—a \text{character}.

13–16 Programming Language—Common Lisp
radix—a radix. The default is 10.

weight—either a non-negative integer less than radix, or false.

**Description:**
Tests whether char is a digit in the specified radix (i.e., with a weight less than radix). If it is a digit in that radix, its weight is returned as an integer; otherwise nil is returned.

**Examples:**

```
(digit-char-p #\5)  →  5
(digit-char-p #\5 2)  →  false
(digit-char-p #\A)  →  false
(digit-char-p #\a)  →  false
(digit-char-p #\A 11)  →  10
(digit-char-p #\a 11)  →  10
(mapcar #'(lambda (radix)
    (map 'list #'(lambda (x) (digit-char-p x radix))
      "059AaFGZ"))
  '(2 8 10 16 36))
→  ((0 NIL NIL NIL NIL NIL NIL NIL)
    (0 5 NIL NIL NIL NIL NIL)
    (0 5 9 NIL NIL NIL NIL)
    (0 5 9 10 10 15 NIL NIL)
    (0 5 9 10 10 15 16 35))
```

**Affected By:**
None. (In particular, the results of this predicate are independent of any special syntax which might have been enabled in the current readtable.)

**See Also:**
alphanumericp

**Notes:**
Digits are graphic characters.

graphic-char-p

**Function**

**Syntax:**
```
graphic-char-p char  →  generalized-boolean
```

**Arguments and Values:**
char—a character.
generalized-boolean—a generalized boolean.

Description:
Returns true if character is a graphic character; otherwise, returns false.

Examples:

(graphic-char-p #\G) → true
(graphic-char-p #\#) → true
(graphic-char-p #\Space) → true
(graphic-char-p #\Newline) → false

Exceptional Situations:
Should signal an error of type type-error if character is not a character.

See Also:
read, Section 2.1 (Character Syntax), Section 13.1.10 (Documentation of Implementation-Defined Scripts)

standard-char-p Function

Syntax:
standard-char-p character → generalized-boolean

Arguments and Values:
character—a character.

generalized-boolean—a generalized boolean.

Description:
Returns true if character is of type standard-char; otherwise, returns false.

Examples:

(standard-char-p #\Space) → true
(standard-char-p #\~) → true
;; This next example presupposes an implementation
;; in which #\Bell is a defined character.
(standard-char-p #\Bell) → false

Exceptional Situations:
Should signal an error of type type-error if character is not a character.
char-upcase, char-downcase

Function

Syntax:

char-upcase character → corresponding-character
char-downcase character → corresponding-character

Arguments and Values:

character, corresponding-character—a character.

Description:

If character is a lowercase character, char-upcase returns the corresponding uppercase character. Otherwise, char-upcase just returns the given character.

If character is an uppercase character, char-downcase returns the corresponding lowercase character. Otherwise, char-downcase just returns the given character.

The result only ever differs from character in its code attribute; all implementation-defined attributes are preserved.

Examples:

(char-upcase #\a) → #\A
(char-upcase #\A) → #\A
(char-downcase #\a) → #\a
(char-downcase #\A) → #\a
(char-upcase #\9) → #\9
(char-downcase #\9) → #\9
(char-upcase #\0) → #\0
(char-downcase #\0) → #\0

;; Note that this next example might run for a very long time in
;; some implementations if CHAR-CODE-LIMIT happens to be very large
;; for that implementation.
(dotimes (code char-code-limit)
  (let ((char (code-char code)))
    (when char
      (unless (cond ((upper-case-p char) (char= (char-upcase (char-downcase char)) char))
                    ((lower-case-p char) (char= (char-downcase (char-upcase char)) char))
                    (t (and (char= (char-upcase (char-downcase char)) char)
                            (char= (char-downcase (char-upcase char)) char))))
        (return char))))
  → NIL

Exceptional Situations:

Should signal an error of type type-error if character is not a character.
upper-case-p, lower-case-p, both-case-p

**Function**

**Syntax:**
- upper-case-p character  $\rightarrow$ generalized-boolean
- lower-case-p character  $\rightarrow$ generalized-boolean
- both-case-p character  $\rightarrow$ generalized-boolean

**Arguments and Values:**
- character—a character.
- generalized-boolean—a generalized boolean.

**Description:**
These functions test the case of a given character.

- **upper-case-p** returns true if character is an uppercase character; otherwise, returns false.
- **lower-case-p** returns true if character is a lowercase character; otherwise, returns false.
- **both-case-p** returns true if character is a character with case; otherwise, returns false.

**Examples:**
- (upper-case-p #\A)  $\rightarrow$ true
- (upper-case-p #\a)  $\rightarrow$ false
- (both-case-p #\a)  $\rightarrow$ true
- (both-case-p #\5)  $\rightarrow$ false
- (lower-case-p #\5)  $\rightarrow$ false
- (upper-case-p #\5)  $\rightarrow$ false

;; This next example presupposes an implementation
;; in which #\Bell is an implementation-defined character.
- (lower-case-p #\Bell)  $\rightarrow$ false

---

**Notes:**
If the corresponding-char is different than character, then both the character and the corresponding-char have case.

Since char-equal ignores the case of the characters it compares, the corresponding-character is always the same as character under char-equal.

**See Also:**
- upper-case-p, alpha-char-p, Section 13.1.4.3 (Characters With Case), Section 13.1.10 (Documentation of Implementation-Defined Scripts)
Exceptional Situations:
Should signal an error of type type-error if character is not a character.

See Also:
char-upcase, char-downcase, Section 13.1.4.3 (Characters With Case), Section 13.1.10 (Documentation of Implementation-Defined Scripts)

char-code

Function

Syntax:

char-code character → code

Arguments and Values:

character—a character.

code—a character code.

Description:

char-code returns the code attribute of character.

Examples:

;; An implementation using ASCII character encoding
;; might return these values:
(char-code #\$) → 36
(char-code #\a) → 97

Exceptional Situations:

Should signal an error of type type-error if character is not a character.

See Also:

char-code-limit
char-int

Function

Syntax:

char-int character → integer

Arguments and Values:

character—a character.

integer—a non-negative integer.

Description:

Returns a non-negative integer encoding the character object. The manner in which the integer is computed is implementation-dependent. In contrast to sxhash, the result is not guaranteed to be independent of the particular Lisp image.

If character has no implementation-defined attributes, the results of char-int and char-code are the same.

\[(\text{char=} \, c1 \, \text{c2}) \equiv (= (\text{char-int} \, c1) (\text{char-int} \, c2))\]

for characters c1 and c2.

Examples:

(char-int #\A) → 65 ; implementation A
(char-int #\A) → 577 ; implementation B
(char-int #\A) → 262145 ; implementation C

See Also:

char-code

code-char

Function

Syntax:

code-char code → char-p

Arguments and Values:

code—a character code.

char-p—a character or nil.

13–22 Programming Language—Common Lisp
Description:
Returns a character with the code attribute given by code. If no such character exists and one cannot be created, nil is returned.

Examples:
(code-char 65.) -> #\A ; in an implementation using ASCII codes
(code-char (char-code #\Space)) -> #\Space ; in any implementation

Affected By:
The implementation's character encoding.

See Also:
char-code

Notes:

char-code-limit

Constant Variable

Constant Value:
A non-negative integer, the exact magnitude of which is implementation-dependent, but which is not less than 96 (the number of standard characters).

Description:
The upper exclusive bound on the value returned by the function char-code.

See Also:
char-code

Notes:
The value of char-code-limit might be larger than the actual number of characters supported by the implementation.
char-name

char-name

Function

Syntax:
char-name character → name

Arguments and Values:
character—a character.
name—a string or nil.

Description:
Returns a string that is the name of the character, or nil if the character has no name.

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.

The standard characters ⟨Newline⟩ and ⟨Space⟩ have the respective names "Newline" and "Space". The semi-standard characters ⟨Tab⟩, ⟨Page⟩, ⟨Rubout⟩, ⟨Linefeed⟩, ⟨Return⟩, and ⟨Backspace⟩ (if they are supported by the implementation) have the respective names "Tab", "Page", "Rubout", "Linefeed", "Return", and "Backspace" (in the indicated case, even though name lookup by "#\" and by the function name-char is not case sensitive).

Examples:

(char-name #\ ) → "Space"
(char-name #\Space) → "Space"
(char-name #\Page) → "Page"

(char-name #\a)
⇒ NIL
⇒ "LOWERCASE-a"
⇒ "Small-A"
⇒ "LA01"

(char-name #\A)
⇒ NIL
⇒ "UPPERCASE-A"
⇒ "Capital-A"
⇒ "LA02"

;; Even though its CHAR-NAME can vary, #\A prints as #\A
(prin1-to-string (read-from-string (format nil "#\A" (or (char-name #\A) "A"))))
⇒ "#\A"

13–24 Programming Language—Common Lisp
Exceptional Situations:
Should signal an error of type type-error if character is not a character.

See Also:
name-char, Section 22.1.3.2 (Printing Characters)

Notes:
Non-graphic characters having names are written by the Lisp printer as “#\” followed by the their name; see Section 22.1.3.2 (Printing Characters).

name-char

Function

Syntax:
name-char name → char-p

Arguments and Values:
name—a string designator.
char-p—a character or nil.

Description:
Returns the character object whose name is name (as determined by string-equal—i.e., lookup is not case sensitive). If such a character does not exist, nil is returned.

Examples:
(name-char 'space) → #\Space
(name-char "space") → #\Space
(name-char "Space") → #\Space
(let ((x (char-name #\a)))
  (or (not x) (eql (name-char x) #\a))) → true

Exceptional Situations:
Should signal an error of type type-error if name is not a string designator.

See Also:
char-name
Programming Language—Common Lisp

14. Conses
14.1 Cons Concepts

A cons is a compound data object having two components called the car and the cdr.

<table>
<thead>
<tr>
<th></th>
<th>car</th>
<th>cons</th>
<th>rplaca</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cdr</td>
<td></td>
<td>rplacd</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>caaar</th>
<th>caddar</th>
<th>cdar</th>
<th>nsubst</th>
</tr>
</thead>
<tbody>
<tr>
<td>caadr</td>
<td>cadddr</td>
<td>cddar</td>
<td>nsubst-if</td>
</tr>
<tr>
<td>caaar</td>
<td>caddr</td>
<td>cddadr</td>
<td>nsubst-if-not</td>
</tr>
<tr>
<td>caadar</td>
<td>cadr</td>
<td>cddar</td>
<td>nthcdr</td>
</tr>
<tr>
<td>caaddr</td>
<td>cdaaar</td>
<td>cdddar</td>
<td>sublis</td>
</tr>
<tr>
<td>caadr</td>
<td>cdaadr</td>
<td>cdddr</td>
<td>subst</td>
</tr>
<tr>
<td>caar</td>
<td>cdaar</td>
<td>cdddr</td>
<td>subst-if</td>
</tr>
<tr>
<td>cadaar</td>
<td>cdadar</td>
<td>cddr</td>
<td>subst-if-not</td>
</tr>
<tr>
<td>cadadr</td>
<td>cdaddr</td>
<td>copy-tree</td>
<td>tree-equal</td>
</tr>
<tr>
<td>cadar</td>
<td>cdadr</td>
<td>nsublis</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>append</th>
<th>last</th>
<th>nbutlast</th>
<th>rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>butlast</td>
<td>ldif</td>
<td>nconc</td>
<td>revappend</td>
</tr>
<tr>
<td>copy-alist</td>
<td>list</td>
<td>ninth</td>
<td>second</td>
</tr>
<tr>
<td>copy-list</td>
<td>list*</td>
<td>nreconc</td>
<td>seventh</td>
</tr>
<tr>
<td>eighth</td>
<td>list-length</td>
<td>nth</td>
<td>sixth</td>
</tr>
<tr>
<td>endp</td>
<td>make-list</td>
<td>nthcdr</td>
<td>tailp</td>
</tr>
<tr>
<td>fifth</td>
<td>member</td>
<td>pop</td>
<td>tenth</td>
</tr>
<tr>
<td>first</td>
<td>member-if</td>
<td>push</td>
<td>third</td>
</tr>
<tr>
<td>fourth</td>
<td>member-if-not</td>
<td>pushnew</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>acons</th>
<th>assoc-if</th>
<th>pairlis</th>
<th>rassoc-if</th>
</tr>
</thead>
<tbody>
<tr>
<td>assoc</td>
<td>assoc-if-not</td>
<td>rassoc</td>
<td>rassoc-if-not</td>
</tr>
</tbody>
</table>

Figure 14–4. Some defined names related to association 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.

<table>
<thead>
<tr>
<th>adjoin</th>
<th>nset-difference</th>
<th>set-difference</th>
<th>union</th>
</tr>
</thead>
<tbody>
<tr>
<td>intersection</td>
<td>nset-exclusive-or</td>
<td>set-exclusive-or</td>
<td></td>
</tr>
<tr>
<td>nintersection</td>
<td>nunion</td>
<td></td>
<td>subsetp</td>
</tr>
</tbody>
</table>

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.
**list**

*System Class*

Class Precedence List:
list, sequence, t

Description:
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 chain of *conses* terminated by the *empty list*, (), which is itself a *proper list*. A *dotted list* is a *list* which has a terminating *atom* that is not the *empty 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*.

*Dotted lists and circular lists* are also *lists*, but usually the unqualified term “list” within this specification means *proper list*. Nevertheless, the *type* *list* unambiguously includes *dotted lists* and *circular lists*.

For each *element* of a *list* there is a *cons*. The *empty list* has no *elements* and is not a *cons*.

The *types* *cons* and *null* form an *exhaustive partition* of the *type* *list*.

See Also:
Section 2.4.1 (Left-Parenthesis), Section 22.1.3.5 (Printing Lists and Conses)

---

**null**

*System Class*

Class Precedence List:
null, symbol, list, sequence, t

Description:
The only *object* of *type* *null* is *nil*, which represents the *empty list* and can also be notated ()

See Also:
Section 2.3.4 (Symbols as Tokens), Section 2.4.1 (Left-Parenthesis), Section 22.1.3.3 (Printing Symbols)

---

14–4  Programming Language—Common Lisp
**cons**

*System Class*

**Class Precedence List:**
cons, list, sequence, t

**Description:**
A cons is a compound object having two components, called the car and cdr. These form a dotted pair. Each component can be any object.

**Compound Type Specifier Kind:**
Specializing.

**Compound Type Specifier Syntax:**
(cons [car-typespec [cdr-typespec]])

**Compound Type Specifier Arguments:**
- *car-typespec*—a type specifier, or the symbol *. The default is the symbol *.
- *cdr-typespec*—a type specifier, or the symbol *. The default is the symbol *.

**Compound Type Specifier Description:**
This denotes the set of conses whose car is constrained to be of type car-typespec and whose cdr is constrained to be of type cdr-typespec. (If either car-typespec or cdr-typespec is *, it is as if the type t had been denoted.)

**See Also:**
Section 2.4.1 (Left-Parenthesis), Section 22.1.3.5 (Printing Lists and Conses)

**atom**

*Type*

**Supertypes:**
atom, t

**Description:**
It is equivalent to (not cons).
cons

Function

Syntax:

\[ \text{cons } \text{object-1} \text{ object-2 } \rightarrow \text{cons} \]

Arguments and Values:

\text{object-1}—an \text{object}.

\text{object-2}—an \text{object}.

\text{cons}—a \text{cons}.

Description:

Creates a fresh \text{cons}, the \text{car} of which is \text{object-1} and the \text{cdr} of which is \text{object-2}.

Examples:

\begin{align*}
(\text{cons 1 2}) & \rightarrow (1 . 2) \\
(\text{cons 1 nil}) & \rightarrow (1) \\
(\text{cons nil 2}) & \rightarrow (\text{NIL} . 2) \\
(\text{cons nil nil}) & \rightarrow (\text{NIL}) \\
(\text{cons 1 (cons 2 (cons 3 (cons 4 nil))))) & \rightarrow (1 \ 2 \ 3 \ 4) \\
(\text{cons 'a 'b}) & \rightarrow (\text{A} . \text{B}) \\
(\text{cons 'a (cons 'b (cons 'c '()))}) & \rightarrow (\text{A} \ \text{B} \ \text{C}) \\
(\text{cons 'a '(b c d)}) & \rightarrow (\text{A} \ \text{B} \ \text{C} \ \text{D})
\end{align*}

See Also:

\text{list}

Notes:

If \text{object-2} is a \text{list}, \text{cons} can be thought of as producing a new \text{list} which is like it but has \text{object-1} prepended.

consp

Function

Syntax:

\[ \text{consp } \text{object} \rightarrow \text{generalized-boolean} \]

Arguments and Values:

\text{object}—an \text{object}.

\text{generalized-boolean}—a \text{generalized boolean}.
Description:

Returns true if object is of type cons; otherwise, returns false.

Examples:

(consp nil) → false
(consp (cons 1 2)) → true

The empty list is not a cons, so

(consp '()) ≡ (consp 'nil) → false

See Also:

listp

Notes:

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

atom Function

Syntax:

atom object → generalized-boolean

Arguments and Values:

object—an object.

generalized-boolean—a generalized boolean.

Description:

Returns true if object is of type atom; otherwise, returns false.

Examples:

(atom 'sss) → true
(atom (cons 1 2)) → false
(atom nil) → true
(atom '()) → true
(atom 3) → true
rplaca, rplacd

Function

Syntax:
\[
\begin{align*}
\text{rplaca} & \quad \text{cons object} \rightarrow \text{cons} \\
\text{rplacd} & \quad \text{cons object} \rightarrow \text{cons}
\end{align*}
\]

Pronunciation:
\[
\begin{align*}
\text{rplaca}: & \quad [\text{r} \text{ë} \text{plak} \epsilon] \text{ or } [\text{r} \epsilon \text{ plak} \epsilon] \\
\text{rplacd}: & \quad [\text{r} \text{ë} \text{ plakd} \epsilon] \text{ or } [\text{r} \epsilon \text{ plakd} \epsilon] \text{ or } [\text{r} \text{ë} \text{ plakd} \epsilon] \text{ or } [\text{r} \epsilon \text{ plakd} \epsilon]
\end{align*}
\]

Arguments and Values:
- \textit{cons}—a \textit{cons}.
- \textit{object}—an \textit{object}.

Description:
\[
\begin{align*}
\text{rplaca} & \quad \text{replaces the car of the cons with object.} \\
\text{rplacd} & \quad \text{replaces the cdr of the cons with object.}
\end{align*}
\]

Examples:
\[
\begin{align*}
\text{(defparameter *some-list* (list* 'one 'two 'three 'four))} & \rightarrow \text{*some-list*} \\
*\text{some-list*} & \rightarrow \text{(ONE TWO THREE . FOUR)} \\
(\text{rplaca *some-list* 'uno}) & \rightarrow \text{(UNO TWO THREE . FOUR)} \\
*\text{some-list*} & \rightarrow \text{(UNO TWO THREE . FOUR)} \\
(\text{rplacd (last *some-list*) (list 'IV))} & \rightarrow \text{(THREE IV)} \\
*\text{some-list*} & \rightarrow \text{(UNO TWO THREE IV)}
\end{align*}
\]

Side Effects:
\[
\begin{align*}
The \textit{cons} \text{ is modified.} \\
\text{Should signal an error of type type-error if cons is not a cons.}
\end{align*}
\]
car, cdr, caar, cadr, cdar, cddr, caaar, caadr, cadar, ...  
car, cdr, caar, cadr, cdar, cddr, caaar, caadr, cadar, caddr, cdaar, cdadr, cddar, cdddr, caaaar, caaadr, caadar, caaddr, cadaar, cadadr, caddar, cadddr, cdaaar, cdaadr, cdadar, cdaddr, cddaar, cddadr, cdddar, cddddd

Accessor

Syntax:

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>car x</td>
<td>object</td>
<td>(setf (car x) new-object)</td>
</tr>
<tr>
<td>cdr x</td>
<td>object</td>
<td>(setf (cdr x) new-object)</td>
</tr>
<tr>
<td>caar x</td>
<td>object</td>
<td>(setf (caar x) new-object)</td>
</tr>
<tr>
<td>cadr x</td>
<td>object</td>
<td>(setf (cadr x) new-object)</td>
</tr>
<tr>
<td>cdar x</td>
<td>object</td>
<td>(setf (cdar x) new-object)</td>
</tr>
<tr>
<td>cddr x</td>
<td>object</td>
<td>(setf (cddr x) new-object)</td>
</tr>
<tr>
<td>caaar x</td>
<td>object</td>
<td>(setf (caaar x) new-object)</td>
</tr>
<tr>
<td>caadr x</td>
<td>object</td>
<td>(setf (caadr x) new-object)</td>
</tr>
<tr>
<td>cadar x</td>
<td>object</td>
<td>(setf (cadar x) new-object)</td>
</tr>
<tr>
<td>caddr x</td>
<td>object</td>
<td>(setf (caddr x) new-object)</td>
</tr>
<tr>
<td>cdaar x</td>
<td>object</td>
<td>(setf (cdaar x) new-object)</td>
</tr>
<tr>
<td>cdaadr x</td>
<td>object</td>
<td>(setf (cdaadr x) new-object)</td>
</tr>
<tr>
<td>cddar x</td>
<td>object</td>
<td>(setf (cddar x) new-object)</td>
</tr>
<tr>
<td>cddadr x</td>
<td>object</td>
<td>(setf (cddadr x) new-object)</td>
</tr>
<tr>
<td>cdaaar x</td>
<td>object</td>
<td>(setf (cdaaar x) new-object)</td>
</tr>
<tr>
<td>cdaadr x</td>
<td>object</td>
<td>(setf (cdaadr x) new-object)</td>
</tr>
<tr>
<td>cddadr x</td>
<td>object</td>
<td>(setf (cddadr x) new-object)</td>
</tr>
<tr>
<td>cdddar x</td>
<td>object</td>
<td>(setf (cdddar x) new-object)</td>
</tr>
<tr>
<td>cddddd x</td>
<td>object</td>
<td>(setf (cddddd x) new-object)</td>
</tr>
</tbody>
</table>

Pronunciation:

cadr: [¹ka,dr]
car, cdr, caar, cadr, cdar, cddr, caaar, caadr, cadar, ...  

\[
\text{caddr: } [\text{'kader}] \text{ or } [\text{'kadur}]
\]
\[
\text{cdr: } [\text{'kuder}]
\]
\[
\text{cddr: } [\text{'kudur}] \text{ or } [\text{'kudurer}]
\]

**Arguments and Values:**

\(x\)—a list.

\(\text{object}\)—an object.

\(\text{new-object}\)—an object.

**Description:**

If \(x\) is a \textit{cons}, \textit{car} returns the \textit{car} of that \textit{cons}. If \(x\) is \textit{nil}, \textit{car} returns \textit{nil}.

If \(x\) is a \textit{cons}, \textit{cdr} returns the \textit{cdr} of that \textit{cons}. If \(x\) is \textit{nil}, \textit{cdr} returns \textit{nil}.

Functions are provided which perform compositions of up to four \textit{car} and \textit{cdr} operations. Their \textit{names} consist of a \textit{C}, followed by two, three, or four occurrences of \textit{A} or \textit{D}, and finally an \textit{R}. The series of \textit{A}'s and \textit{D}'s in each \textit{function's name} is chosen to identify the series of \textit{car} and \textit{cdr} operations that is performed by the function. The order in which the \textit{A}'s and \textit{D}'s appear is the inverse of the order in which the corresponding operations are performed. Figure 14–6 defines the relationships precisely.
**car, cdr, caar, cadr, cdar, cddr, caaar, caadr, cadar, ...**

<table>
<thead>
<tr>
<th>This place ...</th>
<th>Is equivalent to this place ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>(caar x)</td>
<td>(car (car x))</td>
</tr>
<tr>
<td>(cadr x)</td>
<td>(car (cdr x))</td>
</tr>
<tr>
<td>(cdar x)</td>
<td>(cdr (car x))</td>
</tr>
<tr>
<td>(cddr x)</td>
<td>(cdr (cdr x))</td>
</tr>
<tr>
<td>(caaar x)</td>
<td>(car (car (car x)))</td>
</tr>
<tr>
<td>(caadr x)</td>
<td>(car (car (cdr x)))</td>
</tr>
<tr>
<td>(cadar x)</td>
<td>(car (cdr (car x)))</td>
</tr>
<tr>
<td>(caddr x)</td>
<td>(car (cdr (cdr x)))</td>
</tr>
<tr>
<td>(cdaar x)</td>
<td>(cdr (car (car x)))</td>
</tr>
<tr>
<td>(cdadr x)</td>
<td>(cdr (car (cdr x)))</td>
</tr>
<tr>
<td>(cdar x)</td>
<td>(cdr (cdr (car x)))</td>
</tr>
<tr>
<td>(cddr x)</td>
<td>(cdr (cdr (cdr x)))</td>
</tr>
</tbody>
</table>

Figure 14–6. CAR and CDR variants

**setf** can also be used with any of these functions to change an existing component of \( x \), but **setf** will not make new components. So, for example, the **car** of a **cons** can be assigned with **setf** of **car**, but the **car** of **nil** cannot be assigned with **setf** of **car**. Similarly, the **car** of the **car** of a **cons** whose **car** is a **cons** can be assigned with **setf** of **caar**, but neither **nil** nor a **cons** whose **car** is **nil** can be assigned with **setf** of **caar**.

The argument \( x \) is permitted to be a **dotted list** or a **circular list**.

**Examples:**

\[
\begin{align*}
\text{(car nil)} & \rightarrow \text{NIL} \\
\text{(cdr '(1 . 2))} & \rightarrow 2 \\
\text{(cdr '(1 2))} & \rightarrow (2)
\end{align*}
\]
Exceptional Situations:

The functions car and cdr should signal type-error if they receive an argument which is not a list. The other functions (caar, cadr, . . . cddddr) should behave for the purpose of error checking as if defined by appropriate calls to car and cdr.

Notes:

The car of a cons can also be altered by using rplaca, and the cdr of a cons can be altered by using rplacd.

\[
\begin{align*}
\text{car} x & \equiv \text{first} x \\
\text{cadr} x & \equiv (\text{second} x) \equiv (\text{car} (\text{cdr} x)) \\
\text{caddr} x & \equiv (\text{third} x) \equiv (\text{car} (\text{cdr} (\text{cdr} x))) \\
\text{cadddr} x & \equiv (\text{fourth} x) \equiv (\text{car} (\text{cdr} (\text{cdr} (\text{cdr} x))))
\end{align*}
\]

copy-tree

Function

Syntax:

\[
\text{copy-tree } \text{tree} \rightarrow \text{new-tree}
\]

Arguments and Values:

- \text{tree}—a \text{tree}.
- \text{new-tree}—a \text{tree}.

Description:

Creates a copy of a tree of conses.

If \text{tree} is not a cons, it is returned; otherwise, the result is a new cons of the results of calling \text{copy-tree} on the car and cdr of \text{tree}. In other words, all conses in the tree represented by \text{tree} are copied recursively, stopping only when non-conses are encountered.

\text{copy-tree} does not preserve circularities and the sharing of substructure.

Examples:

\[
\text{(setq object (list (cons 1 "one"))}
\]
(cons 2 (list 'a 'b 'c)))
→ ((1 . "one") (2 A B C))
(setq object-too object) → ((1 . "one") (2 A B C))
(setq copy-as-list (copy-list object))
(setq copy-as-alist (copy-alist object))
(setq copy-as-tree (copy-tree object))
(eq object object-too) → true
(eq copy-as-tree object) → false
(eql copy-as-tree object) → false
(equal copy-as-tree object) → true
(setf (first (cdr (second object))) "a"
      (car (second object)) "two"
      (car object) '(one . 1)) → (ONE . 1)
object → ((ONE . 1) "two" "a" B C)
object-too → ((ONE . 1) "two" "a" B C)
copy-as-list → ((1 . "one") ("two" "a" B C))
copy-as-alist → ((1 . "one") (2 "a" B C))
copy-as-tree → ((1 . "one") (2 A B C))

See Also:
  tree-equal

sublis, nsublis  
Function

Syntax:
  sublis alist tree &key key test test-not → new-tree
  nsublis alist tree &key key test test-not → new-tree

Arguments and Values:
  alist—an association list.
  tree—a tree.
  test—a designator for a function of two arguments that returns a generalized boolean.
  test-not—a designator for a function of two arguments that returns a generalized boolean.
  key—a designator for a function of one argument, or nil.
  new-tree—a tree.
sublis, nsublis

Description:

sublis makes substitutions for objects in tree (a structure of conses). nsublis is like sublis but destructively modifies the relevant parts of the tree.

sublis looks at all subtrees and leaves of tree; if a subtree or leaf appears as a key in alist (that is, the key and the subtree or leaf satisfy the test), it is replaced by the object with which that key is associated. This operation is non-destructive. In effect, sublis can perform several subst operations simultaneously.

If sublis succeeds, a new copy of tree is returned in which each occurrence of such a subtree or leaf is replaced by the object with which it is associated. If no changes are made, the original tree is returned. The original tree is left unchanged, but the result tree may share cells with it.

nsublis is permitted to modify tree but otherwise returns the same values as sublis.

Examples:

(sublis '((x . 100) (z . zprime))
  '(plus x (minus g z x p) 4 . x))
→ (PLUS 100 (MINUS G ZPRIME 100 P) 4 . 100)
(sublis '(((+ x y) . (- x y)) ((- x y) . (+ x y)))
  '(* (/ (+ x y) (+ x p)) (- x y)) :test #'equal)
→ (* (/ (- X Y) (+ X P)) (+ X Y))
(setq tree1 '((1 . 2) ((1 2 3) (((1 2 3 4)))))
→ (1 2) ((1 2 "three")) (((1 2 "three" 4)))))
(sublis '(((3 . "three")) tree1)
→ (1 2) ((1 2 "three")) (((1 2 "three" 4)))))
(sublis '((t . "string")
  (sublis '(((1 . "") (4 . 44)) tree1)
  :key #'stringp))
→ ("string" ("string" 2) ("string" 2 3) ("string" 2 3 4)))
setq tree2 '("one" ("one" "two") ("one" "Two" "three"))
→ ("one" ("one" "two") ("one" "Two" "three"))
(sublis '(("two" . 2)) tree2)
→ ("one" ("one" "two") ("one" "Two" "three"))
(setq tree2 '("one" ("one" "two") ("one" "Two" "three"))
→ ("one" ("one" "two") ("one" "Two" "three"))
(sublis '(("one" . 3)) tree2 :test 'equal)
→ ("one" ("one" "two") ("one" "Two" "three")))

(nsublis '((t . 'temp))
  tree1
  :key #'(lambda (x) (or (atom x) (< (list-length x) 3))))
→ ((QUOTE TEMP) (QUOTE TEMP) QUOTE TEMP)

14–14 Programming Language—Common Lisp
Side Effects:

nsublis modifies tree.

See Also:

subst, Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects)

Notes:

The :test-not parameter is deprecated.

Because the side-effecting variants (e.g., nsublis) potentially change the path that is being traversed, their effects in the presence of shared or circular structure may vary in surprising ways when compared to their non-side-effecting alternatives. To see this, consider the following side-effect behavior, which might be exhibited by some implementations:

```lisp
(defun test-it (fn)
  (let* ((shared-piece (list 'a 'b))
         (data (list shared-piece shared-piece)))
    (funcall fn '((a . b) (b . a)) data)))
(test-it #'sublis) → ((B A) (B A))
(test-it #'nsublis) → ((A B) (A B))
```

### subst, subst-if, subst-if-not, nsubst, nsubst-if, nsubst-if-not

#### Function

**Syntax:**

```lisp
subst new old tree &key key test test-not → new-tree
subst-if new predicate tree &key key → new-tree
subst-if-not new predicate tree &key key → new-tree
nsubst new old tree &key key test test-not → new-tree
nsubst-if new predicate tree &key key → new-tree
nsubst-if-not new predicate tree &key key → new-tree
```

**Arguments and Values:**

- `new`—an object.
- `old`—an object.
subst, subst-if, subst-if-not, nsubst, nsubst-if, ...

predicate—a symbol that names a function, or a function of one argument that returns a generalized boolean value.

tree—a tree.

test—a designator for a function of two arguments that returns a generalized boolean.

test-not—a designator for a function of two arguments that returns a generalized boolean.

key—a designator for a function of one argument, or nil.

new-tree—a tree.

Description:

subst, subst-if, and subst-if-not perform substitution operations on tree. Each function searches tree for occurrences of a particular old item of an element or subexpression that satisfies the test.

nsubst, nsubst-if, and nsubst-if-not are like subst, subst-if, and subst-if-not respectively, except that the original tree is modified.

subst makes a copy of tree, substituting new for every subtree or leaf of tree (whether the subtree or leaf is a car or a cdr of its parent) such that old and the subtree or leaf satisfy the test.

nsubst is a destructive version of subst. The list structure of tree is altered by destructively replacing with new each leaf of the tree such that old and the leaf satisfy the test.

For subst, subst-if, and subst-if-not, if the functions succeed, a new copy of the tree is returned in which each occurrence of such an element is replaced by the new element or subexpression. If no changes are made, the original tree may be returned. The original tree is left unchanged, but the result tree may share storage with it.

For nsubst, nsubst-if, and nsubst-if-not the original tree is modified and returned as the function result, but the result may not be eq to tree.

Examples:

(setq tree1 '((1 (1 2) (1 2 3) (1 2 3 4))) -> (1 (1 2) (1 2 3) (1 2 3 4))
(subst "two" 2 tree1) -> (1 (1 "two") (1 "two" 3) (1 "two" 3 4))
(subst "five" 5 tree1) -> (1 (1 2) (1 2 3) (1 2 3 4))
(eq tree1 (subst "five" 5 tree1)) -> implementation-dependent
(subst 'tempest 'hurricane
  '(shakespea_re wrote (the hurricane)))
  -> (SHAKESPEARE WROTE (THE TEMPEST))
(subst 'foo 'nil '(shakespea_re wrote (twelfth night)))
  -> (SHAKESPEARE WROTE (TWELFTH NIGHT . FOO) . FOO)
(subst '(a . cons) '(old . pair)
  '((old . spice) ((old . shoes) old . pair) (old . pair))
:test #'equal
→ (($OLD . SPICE) (($OLD . SHOES) A . CONS) (A . CONS))

(subst-if 5 #'listp tree1) → 5
(subst-if-not '(x) #'consp tree1)
→ (1 X)

tree1 → (1 (1 2) (1 2 3) (1 2 3 4))
(nesubst 'x 3 tree1 :key #'(lambda (y) (and (listp y) (third y))))
→ (1 (1 2) X X)
tree1 → (1 (1 2) X X)

Side Effects:
sesubst, subst-if, and nesubst-if-not might alter the tree structure of tree.

See Also:
subst, nsubstitute, Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects)

Notes:
The :test-not parameter is deprecated.
The functions subst-if-not and nesubst-if-not are deprecated.

One possible definition of subst:

(defun subst (old new tree &rest x &key test test-not key)
  (cond ((satisfies-the-test old tree :test test
                   :test-not test-not :key key)
         new)
        ((atom tree) tree)
        (t (let ((a (apply #'subst old new (car tree) x))
             (d (apply #'subst old new (cdr tree) x)))
            (if (and (eql a (car tree))
                     (eql d (cdr tree)))
                tree
                (cons a d))))))
tree-equal

Syntax:

```
(tree-equal tree-1 tree-2 &key test test-not) → generalized-boolean
```

Arguments and Values:

- `tree-1`—a `tree`.
- `tree-2`—a `tree`.
- `test`—a `designator` for a function of two arguments that returns a `generalized boolean`.
- `test-not`—a `designator` for a function of two arguments that returns a `generalized boolean`.
- `generalized-boolean`—a `generalized boolean`.

Description:

`tree-equal` tests whether two trees are of the same shape and have the same leaves. `tree-equal` returns `true` if `tree-1` and `tree-2` are both `atoms` and satisfy the `test`, or if they are both `conses` and the `car` of `tree-1` is `tree-equal` to the `car` of `tree-2` and the `cdr` of `tree-1` is `tree-equal` to the `cdr` of `tree-2`. Otherwise, `tree-equal` returns `false`.

`tree-equal` recursively compares `conses` but not any other `objects` that have components.

The first argument to the `:test` or `:test-not` function is `tree-1` or a `car` or `cdr` of `tree-1`; the second argument is `tree-2` or a `car` or `cdr` of `tree-2`.

Examples:

```
(setq tree1 '(1 (1 2))
    tree2 '(1 (1 2)))
(tree-equal tree1 tree2) → true
(eql tree1 tree2) → false
(setq tree1 '((a (b 'c))
    tree2 '((a (b 'c)))
    (tree-equal tree1 tree2 :test 'eq))
→ ((QUOTE A) ((QUOTE B) (QUOTE C)))
(tree-equal tree1 tree2 :test 'eq) → true
```

Exceptional Situations:

The consequences are undefined if both `tree-1` and `tree-2` are circular.

See Also:

- `equal`, Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` parameter is deprecated.
Function

copy-list

Syntax:

\[
\text{copy-list } list \rightarrow \text{copy}
\]

Arguments and Values:

- \(list\) — a proper list or a dotted list.
- \(copy\) — a list.

Description:

Returns a copy of \(list\). If \(list\) is a dotted list, the resulting \(list\) will also be a dotted list.

Only the list structure of \(list\) is copied; the elements of the resulting list are the same as the corresponding elements of the given \(list\).

Examples:

\[
\begin{align*}
\text{(setq lst (list 1 (list 2 3)))} & \rightarrow (1 (2 3)) \\
\text{(setq slst lst)} & \rightarrow (1 (2 3)) \\
\text{(setq clst (copy-list lst))} & \rightarrow (1 (2 3)) \\
\text{(eq slst lst)} & \rightarrow \text{true} \\
\text{(eq clst lst)} & \rightarrow \text{false} \\
\text{(equal clst lst)} & \rightarrow \text{true} \\
\text{(rplaca lst "one")} & \rightarrow (\"one\" (2 3)) \\
\text{slst} & \rightarrow (\"one\" (2 3)) \\
\text{clst} & \rightarrow (1 (2 3)) \\
\text{(setf (caadr lst) \"two\")} & \rightarrow \"two\" \\
\text{lst} & \rightarrow (\"one\" (\"two\" 3)) \\
\text{slst} & \rightarrow (\"one\" (\"two\" 3)) \\
\text{clst} & \rightarrow (1 (\"two\" 3))
\end{align*}
\]

Exceptional Situations:

The consequences are undefined if \(list\) is a circular list.

See Also:

- copy-alist, copy-seq, copy-tree

Notes:

The copy created is equal to \(list\), but not eq.
list, list*

Syntax:

\[
\text{list} \; \&\text{rest} \; \text{objects} \rightarrow \text{list} \\
\text{list*} \; \&\text{rest} \; \text{objects}^+ \rightarrow \text{result}
\]

Arguments and Values:

\- \text{object} --- an \text{object}.

\- \text{list} --- a \text{list}.

\- \text{result} --- an \text{object}.

Description:

\text{list} returns a \text{list} containing the supplied \text{objects}.

\text{list*} is like \text{list} except that the last \text{argument} to \text{list} becomes the \text{car} of the last \text{cons} constructed, while the last \text{argument} to \text{list*} becomes the \text{cdr} of the last \text{cons} constructed. Hence, any given call to \text{list*} always produces one fewer \text{conses} than a call to \text{list} with the same number of arguments.

If the last \text{argument} to \text{list*} is a \text{list}, the effect is to construct a new \text{list} which is similar, but which has additional elements added to the front corresponding to the preceding \text{arguments} of \text{list*}.

If \text{list*} receives only one \text{object}, that \text{object} is returned, regardless of whether or not it is a \text{list}.

Examples:

\[
\begin{align*}
(\text{list} \; 1) & \rightarrow (1) \\
(\text{list*} \; 1) & \rightarrow 1 \\
(\text{setq} \; a \; 1) & \rightarrow 1 \\
(\text{list} \; a \; 2) & \rightarrow (1 \; 2) \\
'(a \; 2) & \rightarrow (A \; 2) \\
(\text{list} \; 'a \; 2) & \rightarrow (A \; 2) \\
(\text{list*} \; a \; 2) & \rightarrow (1 \; . \; 2) \\
(\text{list}) & \rightarrow \text{NIL} \; ; \text{i.e.,} \; () \\
(\text{setq} \; a \; '(1 \; 2)) & \rightarrow (1 \; 2) \\
(\text{eq} \; a \; (\text{list*} \; a)) & \rightarrow \text{true} \\
(\text{list} \; 3 \; 4 \; 'a \; (\text{car} \; '(b \; . \; c))) \rightarrow (3 \; 4 \; A \; B \; D) \\
(\text{list*} \; 'a \; 'b \; 'c \; 'd) & \equiv (\text{cons} \; 'a \; (\text{cons} \; 'b \; (\text{cons} \; 'c \; 'd))) \rightarrow (A \; B \; C \; . \; D) \\
(\text{list*} \; 'a \; 'b \; 'c \; (d \; e \; f)) & \rightarrow (A \; B \; C \; D \; E \; F)
\end{align*}
\]

See Also:

\- cons

14–20 Programming Language—Common Lisp
Notes:

\[(\text{list}^* \ x) \equiv x\]

---

**list-length**

*Function*

**Syntax:**

\[
\text{list-length } \text{list} \rightarrow \text{length}
\]

**Arguments and Values:**

- \(\text{list}\) — a *proper list* or a *circular list*.
- \(\text{length}\) — a non-negative integer, or \(\text{nil}\).

**Description:**

Returns the *length* of \(\text{list}\) if \(\text{list}\) is a *proper list*. Returns \(\text{nil}\) if \(\text{list}\) is a *circular list*.

**Examples:**

- \((\text{list-length } '(a \ b \ c \ d)) \rightarrow 4\)
- \((\text{list-length } '(a \ (b \ c) \ d)) \rightarrow 3\)
- \((\text{list-length } '()) \rightarrow 0\)
- \((\text{list-length } \text{nil}) \rightarrow 0\)
- \(\text{(defun circular-list } \text{ (&rest elements)}\)
  \(\text{(let } \text{((cycle } (\text{copy-list elements}))}\)
  \(\text{(nconc cycle cycle)))}\)
- \((\text{list-length } (\text{circular-list } 'a \ 'b)) \rightarrow \text{NIL}\)
- \((\text{list-length } (\text{circular-list } 'a)) \rightarrow \text{NIL}\)
- \((\text{list-length } (\text{circular-list})) \rightarrow 0\)

**Exceptional Situations:**

Should signal an error of *type* \(\text{type-error}\) if \(\text{list}\) is not a *proper list* or a *circular list*.

**See Also:**

- \(\text{length}\)

**Notes:**

\(\text{list-length}\) could be implemented as follows:

\(\text{(defun list-length } \text{ (x)}\)
  \(\text{(do } \text{((n 0 (+ n 2))) } ; \text{Counter.}\)
  \(\text{(fast x (cddr fast))) } ; \text{Fast pointer: leaps by 2.}\)
(slow x (cdr slow)) ; Slow pointer: leaps by 1.
(nil)
;; If fast pointer hits the end, return the count.
(when (endp fast) (return n))
(when (endp (cdr fast)) (return (+ n 1)))
;; If fast pointer eventually equals slow pointer,
;; then we must be stuck in a circular list.
;; (A deeper property is the converse: if we are
;; stuck in a circular list, then eventually the
;; fast pointer will equal the slow pointer.
;; That fact justifies this implementation.)
(when (and (eq fast slow) (> n 0)) (return nil)))

---

listp

Function

Syntax:

listp object → generalized-boolean

Arguments and Values:

object—an object.

generalized-boolean—a generalized boolean.

Description:

Returns true if object is of type list; otherwise, returns false.

Examples:

(listp nil) → true
(listp (cons 1 2)) → true
(listp (make-array 6)) → false
(listp t) → false

See Also:

consp

Notes:

If object is a cons, listp does not check whether object is a proper list; it returns true for any kind of list.

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

14–22 Programming Language—Common Lisp
make-list

Function

Syntax:

make-list size &key initial-element → list

Arguments and Values:

size—a non-negative integer.

initial-element—an object. The default is nil.

list—a list.

Description:

Returns a list of length given by size, each of the elements of which is initial-element.

Examples:

(make-list 5) → (NIL NIL NIL NIL NIL)
(make-list 3 :initial-element 'rah) → (RAH RAH RAH)
(make-list 2 :initial-element '(1 2 3)) → ((1 2 3) (1 2 3))
(make-list 0) → NIL ;i.e., ()
(make-list 0 :initial-element 'new-element) → NIL

Exceptional Situations:

Should signal an error of type type-error if size is not a non-negative integer.

See Also:

cons, list

push

Macro

Syntax:

push item place → new-place-value

Arguments and Values:

item—an object.

place—a place, the value of which may be any object.

new-place-value—a list (the new value of place).
Description:

*push* prepends *item* to the *list* that is stored in *place*, stores the resulting *list* in *place*, and returns the *list*.

For information about the evaluation of subforms of *place*, see Section 5.1.1.1 (Evaluation of Subforms to Places).

Examples:

```
(setq lst '(nil))  \rightarrow  (NIL)
(push 1 (car lst)) \rightarrow  (1)
lst        \rightarrow  ((1))
(push 1 (car lst)) \rightarrow  (1 1)
lst        \rightarrow  ((1 1))
(setq x '(a (b c) d)) \rightarrow  (A (B C) D)
(push 5 (cadr x)) \rightarrow  (5 B C)
x        \rightarrow  (A (5 B C) D)
```

Side Effects:
The contents of *place* are modified.

See Also:
pop, pushnew, Section 5.1 (Generalized Reference)

Notes:
The effect of *(push item place)* is equivalent to

```
(setf place (cons item place))
```

except that the subforms of *place* are evaluated only once, and *item* is evaluated before *place*.

---

### pop

*Macro*

**Syntax:**

```
pop place  \rightarrow  element
```

**Arguments and Values:**

*place*—a *place*, the value of which is a *list* (possibly, but necessarily, a *dotted list* or *circular list*).

*element*—an *object* (the car of the contents of *place*).

**Description:**

*pop* reads the *value* of *place*, remembers the *car* of the *list* which was retrieved, *writes* the *cdr* of the *list* back into the *place*, and finally *yields* the *car* of the originally retrieved *list*.

14–24  Programming Language—Common Lisp
For information about the evaluation of subforms of place, see Section 5.1.1.1 (Evaluation of Subforms to Places).

Examples:

(setq stack '(a b c)) → (A B C)  
(pop stack) → A  
stack → (B C)  
(setq llst '((1 2 3 4))) → ((1 2 3 4))  
(pop (car llst)) → 1  
l1st → ((2 3 4))

Side Effects:

The contents of place are modified.

See Also:

push, pushnew, Section 5.1 (Generalized Reference)

Notes:

The effect of (pop place) is roughly equivalent to

(prog1 (car place) (setf place (cdr place)))

except that the latter would evaluate any subforms of place three times, while pop evaluates them only once.

first, second, third, fourth, fifth, sixth, seventh, eighth, ninth, tenth  

Argument Syntax:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>first list</td>
<td>object</td>
</tr>
<tr>
<td>second list</td>
<td>object</td>
</tr>
<tr>
<td>third list</td>
<td>object</td>
</tr>
<tr>
<td>fourth list</td>
<td>object</td>
</tr>
<tr>
<td>fifth list</td>
<td>object</td>
</tr>
<tr>
<td>sixth list</td>
<td>object</td>
</tr>
<tr>
<td>seventh list</td>
<td>object</td>
</tr>
<tr>
<td>eighth list</td>
<td>object</td>
</tr>
<tr>
<td>ninth list</td>
<td>object</td>
</tr>
<tr>
<td>tenth list</td>
<td>object</td>
</tr>
</tbody>
</table>

Arguments and Values:

list—a list, which might be a dotted list or a circular list.
first, second, third, fourth, fifth, sixth, seventh, ...

object, new-object—an object.

Description:
The functions first, second, third, fourth, fifth, sixth, seventh, eighth, ninth, and tenth access the first, second, third, fourth, fifth, sixth, seventh, eighth, ninth, and tenth elements of list, respectively. Specifically,

(first list)  ≡  (car list)
(second list)  ≡  (car (cdr list))
(third list)  ≡  (car (cddr list))
(fourth list)  ≡  (car (cddddr list))
(fifth list)  ≡  (car (cddddr list))
(sixth list)  ≡  (car (cdr (cddddr list)))
(seventh list)  ≡  (car (cdddr (cddddr list))))
(eighth list)  ≡  (car (cddr (cdddr (cddddr list))))
(ninth list)  ≡  (car (cdddr (cdddr (cddddr list))))
(tenth list)  ≡  (car (cdr (cddddr (cddddr list))))

setf can also be used with any of these functions to change an existing component. The same equivalences apply. For example:

(setf (fifth list) new-object) ≡ (setf (car (cddddr list))) new-object

Examples:

(setq lst '(1 2 3 (4 5 6) ((V)) vi 7 8 9 10))
→ (1 2 3 (4 5 6) ((V)) VI 7 8 9 10)
(first lst) → 1
(tenth lst) → 10
(fifth lst) → ((V))
(second (fourth lst)) → 5
(sixth '(1 2 3)) → NIL
(setf (fourth lst) "four") → "four"
lst → (1 2 3 "four" ((V)) VI 7 8 9 10)

See Also:
car, nth

Notes:
first is functionally equivalent to car, second is functionally equivalent to cadr, third is functionally equivalent to caddr, and fourth is functionally equivalent to cadddr.

The ordinal numbering used here is one-origin, as opposed to the zero-origin numbering used by nth:

(fifth x) ≡ (nth 4 x)
nth

Syntax:

\[ \text{nth } n \text{ list } \rightarrow \text{ object} \]

\[ \text{(setf (nth } n \text{ list) new-object) } \]

Arguments and Values:

- \[ n \]— a non-negative integer.
- \[ \text{list} \]— a list, which might be a dotted list or a circular list.
- \[ \text{object} \]— an object.
- \[ \text{new-object} \]— an object.

Description:

\[ \text{nth} \] locates the \( n \)th element of \( \text{list} \), where the \text{car} of the \( \text{list} \) is the “zeroth” element. Specifically,

\[ (\text{nth } n \text{ list}) \equiv (\text{car (nthcdr } n \text{ list})) \]

\[ \text{nth} \] may be used to specify a place to \text{setf}. Specifically,

\[ (\text{setf (nth } n \text{ list) new-object}) \equiv (\text{setf (car (nthcdr } n \text{ list))) new-object}) \]

Examples:

\[ (\text{nth 0 '(foo bar baz)}) \rightarrow \text{FOO} \]
\[ (\text{nth 1 '(foo bar baz)}) \rightarrow \text{BAR} \]
\[ (\text{nth 3 '(foo bar baz)}) \rightarrow \text{NIL} \]
\[ (\text{setq 0-to-3 (list 0 1 2 3)}) \rightarrow (0 1 2 3) \]
\[ (\text{setf (nth 2 0-to-3) "two") \rightarrow "two"} \]
\[ 0-to-3 \rightarrow (0 1 "two" 3) \]

See Also:

\[ \text{elt, first, nthcdr} \]
endp

Function

Syntax:

endp list → generalized-boolean

Arguments and Values:

list—a list, which might be a dotted list or a circular list.

generalized-boolean—a generalized boolean.

Description:

Returns true if list is the empty list. Returns false if list is a cons.

Examples:

(endp nil) → true
(endp '(1 2)) → false
(endp (cddr '(1 2))) → true

Exceptional Situations:

Should signal an error of type type-error if list is not a list.

Notes:

The purpose of endp is to test for the end of proper list. Since endp does not descend into a cons, it is well-defined to pass it a dotted list. However, if shorter “lists” are iteratively produced by calling cdr on such a dotted list and those “lists” are tested with endp, a situation that has undefined consequences will eventually result when the non-nil atom (which is not in fact a list) finally becomes the argument to endp. Since this is the usual way in which endp is used, it is conservative programming style and consistent with the intent of endp to treat endp as simply a function on proper lists which happens not to enforce an argument type of proper list except when the argument is atomic.

null

Function

Syntax:

null object → boolean

Arguments and Values:

object—an object.

boolean—a boolean.
Description:

Returns t if object is the empty list; otherwise, returns nil.

Examples:

- (null '()) → T
- (null nil) → T
- (null t) → NIL
- (null 1) → NIL

See Also:

not

Notes:

null is intended to be used to test for the empty list whereas not is intended to be used to invert a boolean (or generalized boolean). Operationally, null and not compute the same result; which to use is a matter of style.

- (null object) ≡ (typep object 'null) ≡ (eq object '())

nconc Function

Syntax:

nconc &rest lists → concatenated-list

Arguments and Values:

- list—each but the last must be a list (which might be a dotted list but must not be a circular list); the last list may be any object.
- concatenated-list—a list.

Description:

Returns a list that is the concatenation of lists. If no lists are supplied, (nconc) returns nil. nconc is defined using the following recursive relationship:

- (nconc) → ()
- (nconc nil . lists) ≡ (nconc . lists)
- (nconc list) → list
- (nconc list-1 list-2) ≡ (progn (rplacd (last list-1) list-2) list-1)
- (nconc list-1 list-2 . lists) ≡ (nconc (nconc list-1 list-2) . lists)
Examples:

\[
\begin{align*}
&\text{(nconc)} \rightarrow \text{NIL} \\
&(\text{setq } x \ (\text{'a b c})) \rightarrow (\text{A B C}) \\
&(\text{setq } y \ (\text{'d e f})) \rightarrow (\text{D E F}) \\
&(\text{nconc } x \ y) \rightarrow (\text{A B C D E F}) \\
&x \rightarrow (\text{A B C D E F})
\end{align*}
\]

Note, in the example, that the value of \(x\) is now different, since its last \textit{cons} has been \texttt{rplacd}'d to the value of \(y\). If \(\text{(nconc } x \ y)\) were evaluated again, it would yield a piece of a \textit{circular list}, whose printed representation would be \((\text{A B C D E F D E F ...})\), repeating forever; if the \texttt{*print-circle*} switch were \texttt{non-nil}, it would be printed as \((\text{A B C . #1=(D E F . #1#)})\).

\[
\begin{align*}
&(\text{setq } \text{foo} \ (\text{list 'a 'b 'c 'd 'e}) \\
&\quad \text{bar} \ (\text{list 'f 'g 'h 'i 'j}) \\
&\quad \text{baz} \ (\text{list 'k 'l 'm}) \rightarrow (\text{K L M}) \\
&(\text{setq } \text{foo} \ (\text{nconc } \text{foo} \ \text{bar} \ \text{baz})) \rightarrow (\text{A B C D E F G H I J K L M}) \\
&\text{foo} \rightarrow (\text{A B C D E F G H I J K L M}) \\
&\text{bar} \rightarrow (\text{F G H I J K L M}) \\
&\text{baz} \rightarrow (\text{K L M})
\end{align*}
\]

\[
\begin{align*}
&(\text{setq } \text{foo} \ (\text{list 'a 'b 'c 'd 'e}) \\
&\quad \text{bar} \ (\text{list 'f 'g 'h 'i 'j}) \\
&\quad \text{baz} \ (\text{list 'k 'l 'm}) \rightarrow (\text{K L M}) \\
&(\text{setq } \text{foo} \ (\text{nconc } \text{nil } \text{foo} \ \text{bar} \ \text{nil} \ \text{baz})) \rightarrow (\text{A B C D E F G H I J K L M}) \\
&\text{foo} \rightarrow (\text{A B C D E F G H I J K L M}) \\
&\text{bar} \rightarrow (\text{F G H I J K L M}) \\
&\text{baz} \rightarrow (\text{K L M})
\end{align*}
\]

Side Effects:

The \textit{lists} are modified rather than copied.

See Also:

\begin{itemize}
\item append, concatenate
\end{itemize}

append \hspace{1cm} Function

Syntax:

\[
\text{append } \&\text{rest } \text{lists} \rightarrow \text{result}
\]

Arguments and Values:

\textit{list}—each must be a \textit{proper list} except the last, which may be any \textit{object}.

14–30 Programming Language—Common Lisp
result—an object. This will be a list unless the last list was not a list and all preceding lists were null.

**Description:**
append returns a new list that is the concatenation of the copies. lists are left unchanged; the list structure of each of lists except the last is copied. The last argument is not copied; it becomes the cdr of the final dotted pair of the concatenation of the preceding lists, or is returned directly if there are no preceding non-empty lists.

**Examples:**
- `(append '(a b c) '(d e f) '() 'g)) → (A B C D E F G)
- `(append '(a b c) 'd) → (A B C . D)
- `(setq lst '(a b c)) → (A B C)
- `(append lst 'd)) → (A B C D)
- `lst → (A B C)
- `(append) → NIL
- `(append 'a) → A

**See Also:**
nconc, concatenate

---

**revappend, nreconc Function**

**Syntax:**
- `revappend list tail → result-list`
- `nreconc list tail → result-list`

**Arguments and Values:**
- `list`—a proper list.
- `tail`—an object.
- `result-list`—an object.

**Description:**
revappend constructs a copy of list, but with the elements in reverse order. It then appends (as if by nconc) the tail to that reversed list and returns the result.

nreconc reverses the order of elements in list (as if by nreverse). It then appends (as if by nconc) the tail to that reversed list and returns the result.

The resulting list shares list structure with tail.
revappend, nreconc

Examples:

(let ((list-1 (list 1 2 3))
     (list-2 (list 'a 'b 'c)))
  (print (revappend list-1 list-2))
  (print (equal list-1 '(1 2 3)))
  (print (equal list-2 '(a b c))))
▷ (3 2 1 A B C)
▷ T
▷ T
→ T

(revappend '(1 2 3) '()) → (3 2 1)
(revappend '(1 2 3) '(a . b)) → (3 2 1 A . B)
(revappend '() '(a b c)) → (A B C)
(revappend '(1 2 3) 'a) → (3 2 1 . A)
(revappend '() 'a) → A ;degenerate case

(let ((list-1 '(1 2 3))
     (list-2 '(a b c)))
  (print (nreconc list-1 list-2))
  (print (equal list-1 '(1 2 3)))
  (print (equal list-2 '(a b c))))
▷ (3 2 1 A B C)
▷ NIL
▷ T
→ T

Side Effects:

revappend does not modify either of its arguments. nreconc is permitted to modify list but not tail.

Although it might be implemented differently, nreconc is constrained to have side-effect behavior equivalent to:

(nconc (nreverse list) tail)

See Also:

reverse, nreverse, nconc

Notes:

The following functional equivalences are true, although good implementations will typically use a faster algorithm for achieving the same effect:

14–32 Programming Language—Common Lisp
(revappend list tail) ≡ (nconc (reverse list) tail)
(nreconc list tail) ≡ (nconc (nreverse list) tail)

butlast, nbutlast

Function

Syntax:

butlast list &optional n → result-list
nbutlast list &optional n → result-list

Arguments and Values:

list—a list, which might be a dotted list but must not be a circular list.
n—a non-negative integer.
result-list—a list.

Description:

butlast returns a copy of list from which the last n conses have been omitted. If n is not supplied, its value is 1. If there are fewer than n conses in list, nil is returned and, in the case of nbutlast, list is not modified.

nbutlast is like butlast, but nbutlast may modify list. It changes the cdr of the cons n+1 from the end of the list to nil.

Examples:

(setq lst '(1 2 3 4 5 6 7 8 9)) → (1 2 3 4 5 6 7 8 9)
(butlast lst) → (1 2 3 4 5 6 7 8)
(butlast lst 5) → (1 2 3 4)
(butlast lst (+ 5 5)) → NIL
lst → (1 2 3 4 5 6 7 8 9)
(nbutlast lst 3) → (1 2 3 4 5 6)
lst → (1 2 3 4 5 6)
(nbutlast lst 99) → NIL
lst → (1 2 3 4 5 6)
(butlast '((a b c d)) → (A B C)
(butlast '((a b) (c d))) → ((A B))
(butlast '(a)) → NIL
(butlast nil) → NIL
(setq foo (list 'a 'b 'c 'd)) → (A B C D)
(nbutlast foo) → (A B C)
foo → (A B C)
Exceptional Situations:
Should signal an error of type type-error if list is not a proper list or a dotted list. Should signal an error of type type-error if n is not a non-negative integer.

Notes:
(butlast list n) ≡ (ldiff list (last list n))
(last '(a b c) 2) → (B C)
(last '(a b c) 3) → (A B C)
(last '(a b c) 4) → (A B C)

(last '(a . b) 0) → B
(last '(a . b) 1) → (A . B)
(last '(a . b) 2) → (A . B)

Exceptional Situations:
The consequences are undefined if list is a circular list. Should signal an error of type type-error if n is not a non-negative integer.

See Also:
butlast, nth

Notes:
The following code could be used to define last.

(defun last (list &optional (n 1))
  (check-type n (integer 0))
  (do ((l list (cdr l))
       (r list)
       (i 0 (+ i 1)))
      ((atom l) r)
    (if (>= i n) (pop r))))

ldiff, tailp

Function

Syntax:
  ldiff list object  → result-list
  tailp object list  → generalized-boolean

Arguments and Values:
  list—a list, which might be a dotted list.
  object—an object.
  result-list—a list.
  generalized-boolean—a generalized boolean.
ldiff, tailp

Description:
If object is the same as some tail of list, tailp returns true; otherwise, it returns false.

If object is the same as some tail of list, ldiff returns a fresh list of the elements of list that precede object in the list structure of list; otherwise, it returns a copy of list.

Examples:

(let ((lists `#((a b c) (a b c . d))))
  (dotimes (i (length lists)) ()
    (let ((list (aref lists i))
      (format t ""2&list='S "21T(tailp object list)"
        "44T(ldiff list object)"X" list)
      (let ((objects (vector list (cddr list) (copy-list (cddr list))
            '(f g h) '() 'd 'x)))
        (dotimes (j (length objects)) ()
          (let ((object (aref objects j)))
            (format t "&object='S "21T"S "44T"S"            
              object (tailp object list) (ldiff list object))))))))

> list=(A B C)  (tailp object list)  (ldiff list object)
> object=(A B C) T NIL
> object=(C) T (A B)
> object=(C) NIL (A B C)
> object=(F G H) NIL (A B C)
> object=NIL T (A B C)
> object=D NIL (A B C)
> object=X NIL (A B C)

> list=(A B C . D)  (tailp object list)  (ldiff list object)
> object=(A B C . D) T NIL
> object=(C . D) T (A B)
> object=(C . D) NIL (A B C . D)
> object=(F G H) NIL (A B C . D)
> object=NIL NIL (A B C . D)
> object=D T (A B C)
> object=X NIL (A B C . D)
→ NIL

Side Effects:
Neither ldiff nor tailp modifies either of its arguments.

Exceptional Situations:
Should be prepared to signal an error of type type-error if list is not a proper list or a dotted list.
See Also:

set-difference

Notes:

If the list is a circular list, tailp will reliably yield a value only if the given object is in fact a tail of list. Otherwise, the consequences are unspecified: a given implementation which detects the circularity must return false, but since an implementation is not obliged to detect such a situation, tailp might just loop indefinitely without returning in that case.

tailp could be defined as follows:

\[
\text{(defun tailp (object list)}
\text{  (do ((list list (cdr list)))
\text{    ((atom list) (eql list object))
\text{      (if (eql object list)
\text{        (return t))))))}
\]

and ldiff could be defined by:

\[
\text{(defun ldiff (list object)}
\text{  (do ((list list (cdr list)))
\text{    (r '() (cons (car list) r)))
\text{    (atom list)
\text{      (if (eql list object) (nreverse r) (nreconc r list)))
\text{    (when (eql object list)
\text{      (return (nreverse r))))))}
\]

nthcdr

Syntax:

\[
\text{nthcdr } n \text{ list } \rightarrow \text{ tail}
\]

Arguments and Values:

\[
n-\text{a non-negative integer.}
\]

\[
\text{list-\text{a list, which might be a dotted list or a circular list.}}
\]

\[
\text{tail-\text{an object.}}
\]

Description:

Returns the tail of list that would be obtained by calling cdr \( n \) times in succession.
Examples:

(nthcdr 0 '()) → NIL
(nthcdr 3 '()) → NIL
(nthcdr 0 '(a b c)) → (A B C)
(nthcdr 2 '(a b c)) → (C)
(nthcdr 4 '(a b c)) → ()
(nthcdr 1 '(0 . 1)) → 1

(locally (declare (optimize (safety 3)))
  (nthcdr 3 '(0 . 1)))
Error: Attempted to take CDR of 1.

Exceptional Situations:

Should signal an error of type type-error if \( n \) is not a non-negative integer.

For \( n \) being an integer greater than 1, the error checking done by \( \text{nthcdr} \ n \ \text{list} \) is the same as for \( \text{nthcdr} \ (-n\ 1) \ \text{cdr} \ \text{list} \)); see the function cdr.

See Also:
cdr, nth, rest

---

**rest**

Syntax:

rest list → tail

(setf (rest list) new-tail)

Arguments and Values:

- list—a list, which might be a dotted list or a circular list.
- tail—an object.

Description:

rest performs the same operation as cdr, but mnemonically complements first. Specifically,

\[
\text{rest list} \equiv \text{cdr list}
\]
\[
\text{(setf (rest list) new-tail)} \equiv \text{(setf (cdr list) new-tail)}
\]

Examples:

(rest '(1 2)) → (2)
(rest '(1 . 2)) → 2
(rest '(1)) → NIL
(setq *cons* '(1 . 2)) → (1 . 2)
(setf (rest *cons*) "two") → "two"
*cons* → (1 . "two")

See Also:
cdr, nthcdr

Notes:
rest is often preferred stylistically over cdr when the argument is to being subjectively viewed as a list rather than as a cons.

---

class="FUNCTION"

member, member-if, member-if-not

Syntax:

member item list &key key test test-not → tail
member-if predicate list &key key → tail
member-if-not predicate list &key key → tail

Arguments and Values:

item—an object.
list—a proper list.
predicate—a designator for a function of one argument that returns a generalized boolean.
test—a designator for a function of two arguments that returns a generalized boolean.
test-not—a designator for a function of two arguments that returns a generalized boolean.
key—a designator for a function of one argument, or nil.
tail—a list.

description:

member, member-if, and member-if-not each search list for item or for a top-level element that satisfies the test. The argument to the predicate function is an element of list.

If some element satisfies the test, the tail of list beginning with this element is returned; otherwise nil is returned.

list is searched on the top level only.
Examples:

- `(member 2 '(1 2 3)) → (2 3)
- `(member 2 '((1 . 2) (3 . 4)) :test-not #'= :key #'cdr) → ((3 . 4))
- `(member 'e '(a b c d)) → NIL
- `(member-if #'listp '(a b nil c d)) → (NIL C D)
- `(member-if #'numberp '(a #\Space 5/3 foo)) → (5/3 FOO)
- `(member-if-not #'zerop '(3 6 9 11 . 12) :key #'(lambda (x) (mod x 3))) → (11 . 12)

Exceptional Situations:

Should be prepared to signal an error of type type-error if list is not a proper list.

See Also:

find, position, Section 3.6 (Traversal Rules and Side Effects)

Notes:

The :test-not parameter is deprecated.

The function member-if-not is deprecated.

In the following

- `(member 'a '(g (a y) c a d e a f)) → (A D E A F)

the value returned by member is identical to the portion of the list beginning with a. Thus rplaca on the result of member can be used to alter the part of the list where a was found (assuming a check has been made that member did not return nil).

mapc, mapcar, mapcan, mapl, maplist, mapcon

Function

Syntax:

- `mapc function &rest lists+ → list-1`
- `mapcar function &rest lists+ → result-list`
- `mapcan function &rest lists+ → concatenated-results`
- `mapl function &rest lists+ → list-1`
- `maplist function &rest lists+ → result-list`
mapc, mapcar, mapcan, mpl, maplist, mapcon

mapcon function &rest lists+  →  concatenated-results

Arguments and Values:
  function—a designator for a function that must take as many arguments as there are lists.
  list—a proper list.
  list-1—the first list (which must be a proper list).
  result-list—a list.
  concatenated-results—a list.

Description:
The mapping operation involves applying function to successive sets of arguments in which one argument is obtained from each sequence. Except for mapc and mapl, the result contains the results returned by function. In the cases of mapc and mapl, the resulting sequence is list.

function is called first on all the elements with index 0, then on all those with index 1, and so on. result-type specifies the type of the resulting sequence. If function is a symbol, it is coerced to a function as if by symbol-function.

mapcar operates on successive elements of the lists. function is applied to the first element of each list, then to the second element of each list, and so on. The iteration terminates when the shortest list runs out, and excess elements in other lists are ignored. The value returned by mapcar is a list of the results of successive calls to function.

mapc is like mapcar except that the results of applying function are not accumulated. The list argument is returned.

maplist is like mapcar except that function is applied to successive sublists of the lists. function is first applied to the lists themselves, and then to the cdr of each list, and then to the cdr of the cdr of each list, and so on.

mapl is like maplist except that the results of applying function are not accumulated; list-1 is returned.

mapcan and mapcon are like mapcar and maplist respectively, except that the results of applying function are combined into a list by the use of nconc rather than list. That is,

(mapcon f x1 ... xn) ≡ (apply #'nconc (maplist f x1 ... xn))

and similarly for the relationship between mapcan and mapcar.

Examples:

(mapcar #'car '((1 a) (2 b) (3 c))) → (1 2 3)
(mapcar #'abs '((3 -4 2 -5 -6)) → (3 4 2 5 6)
(mapcar #'cons '(a b c) '((1 2 3)) → ((A . 1) (B . 2) (C . 3))

(maplist #'append '(1 2 3 4) '(1 2) → ((1 2 3 4 1 2 1 2 3) 2 3 4 2 2 3))
(maplist #'(lambda (x) (cons 'foo x)) '(a b c d)) → ((FOO A B C D) (FOO B C D) (FOO C D) (FOO D))
(maplist #'(lambda (x) (if (member (car x) (cdr x)) 0 1)) '(a b a c d b c)) → (0 0 1 0 1 1 1)
; An entry is 1 if the corresponding element of the input
; list was the last instance of that element in the input list.

(setq dummy nil) → NIL
(mapc #'(lambda (&rest x) (setq dummy (append dummy x)))
' '(1 2 3 4)
  '(a b c d e)
  '(x y z)) → (1 2 3 4)
dummy → (1 A X 2 B Y 3 C Z)

(setq dummy nil) → NIL
(mapl #'(lambda (x) (push x dummy)) '(1 2 3 4)) → (1 2 3 4)
dummy → ((4) (3 4) (2 3 4) (1 2 3 4))

(mapcan #'(lambda (x y) (if (null x) nil (list x y)))
  '(nil nil nil d e)
  '(1 2 3 4 5 6)) → (D 4 E 5)
(mapcan #'(lambda (x) (and (numberp x) (list x)))
  '(a 1 b c 3 4 d 5)) → (1 3 4 5)

In this case the function serves as a filter; this is a standard Lisp idiom using mapcan.

(mapcon #'list '(1 2 3 4)) → ((1 2 3 4) (2 3 4) (3 4) (4))

Exceptional Situations:
Should be prepared to signal an error of type type-error if any list is not a proper list.

See Also:
dolist, map, Section 3.6 (Traversal Rules and Side Effects)
acons

Function

Syntax:
acons key datum alist → new-alist

Arguments and Values:
key—an object.
datum—an object.
alist—an association list.
new-alist—an association list.

Description:
Creates a fresh cons, the cdr of which is alist and the car of which is another fresh cons, the car of which is key and the cdr of which is datum.

Examples:
(setq alist '()) → NIL
(acons 1 "one" alist) → ((1 . "one"))
alist → NIL
(setq alist (acons 1 "one" (acons 2 "two" alist))) → ((1 . "one") (2 . "two"))
(assoc 1 alist) → (1 . "one")
(setq alist (acons 1 "uno" alist)) → ((1 . "uno") (1 . "one") (2 . "two"))
(assoc 1 alist) → (1 . "uno")

See Also:
assoc, pairlis

Notes:
(acons key datum alist) ≡ (cons (cons key datum) alist)

assoc, assoc-if, assoc-if-not

Function

Syntax:
assoc item alist &key key test test-not → entry
assoc-if predicate alist &key key → entry
assoc, assoc-if, assoc-if-not

assoc-if-not predicate alist &key key → entry

Arguments and Values:

item—an object.

alist—an association list.

predicate—a designator for a function of one argument that returns a generalized boolean.

test—a designator for a function of two arguments that returns a generalized boolean.

test-not—a designator for a function of two arguments that returns a generalized boolean.

key—a designator for a function of one argument, or nil.

entry—a cons that is an element of alist, or nil.

Description:

assoc, assoc-if, and assoc-if-not return the first cons in alist whose car satisfies the test, or nil if no such cons is found.

For assoc, assoc-if, and assoc-if-not, if nil appears in alist in place of a pair, it is ignored.

Examples:

```
(setq values '((x . 100) (y . 200) (z . 50))) → ((X . 100) (Y . 200) (Z . 50))
(assoc 'y values) → (Y . 200)
(rplacd (assoc 'y values) 201) → (Y . 201)
(assoc 'y values) → (Y . 201)
(setq alist '(((1 . "one") (2 . "two") (3 . "three")) → ((1 . "one") (2 . "two") (3 . "three"))
(assoc 2 alist) → (2 . "two")
(assoc-if #'evenp alist) → (2 . "two")
(assoc-if-not #'(lambda (x) (< x 3)) alist) → (3 . "three")
(setq alist '(("one" . 1) ("two" . 2)) → (("one" . 1) ("two" . 2))
(assoc "one" alist) → NIL
(assoc "one" alist :test #'equalp) → ("one" . 1)
(assoc "two" alist :key #'(lambda (x) (char x 2))) → NIL
(assoc "\0" alist :key #'(lambda (x) (char x 2))) → ("two" . 2)
(assoc 'r '((a . b) (c . d) (r . x) (s . y) (r . z)) → (R . X)
(assoc 'goo '((foo . bar) (zoo . goo))) → NIL
(assoc '2 '((1 a b c) (2 b c d) (3 y z)) → (2 b c d)
(setq alist '(((one" . 1) ("two" . 2) ("three" . 3))
→ (("one" . 1) ("two" . 2) ("three" . 3))
(assoc-if-not #'alpha-char-p alist
 :key #'(lambda (x) (char x 0))) → ("two" . 2)
```

14–44 Programming Language—Common Lisp
Exceptional Situations:
Should be prepared to signal an error of type type-error if alist is not an association list.

See Also:
sassoc, find, member, position, Section 3.6 (Traversal Rules and Side Effects)

Notes:
The :test-not parameter is deprecated.
The function assoc-if-not is deprecated.
It is possible to rplacd the result of assoc, provided that it is not nil, in order to “update” alist.
The two expressions
(assoc item list :test fn)
and
(find item list :test fn :key #'car)
are equivalent in meaning with one exception: if nil appears in alist in place of a pair, and item is nil, find will compute the car of the nil in alist, find that it is equal to item, and return nil, whereas assoc will ignore the nil in alist and continue to search for an actual cons whose car is nil.

copy-alist

Function

Syntax:
copy-alist alist → new-alist

Arguments and Values:
alist—an association list.
new-alist—an association list.

Description:
copy-alist returns a copy of alist.
The list structure of alist is copied, and the elements of alist which are conses are also copied (as conses only). Any other objects which are referred to, whether directly or indirectly, by the alist continue to be shared.
Examples:

```
(defparameter *alist* (acons 1 "one" (acons 2 "two" '())))
*alist* → ((1 . "one") (2 . "two"))
(defparameter *list-copy* (copy-list *alist*))
*list-copy* → ((1 . "one") (2 . "two"))
(defparameter *alist-copy* (copy-alist *alist*))
*alist-copy* → ((1 . "one") (2 . "two"))
(setf (cdr (assoc 2 *alist-copy*)) "deux") → "deux"
*alist-copy* → ((1 . "one") (2 . "deux"))
*alist* → (((1 . "one") (2 . "two"))
(setf (cdr (assoc 1 *list-copy*)) "uno") → "uno"
*list-copy* → ((1 . "uno") (2 . "two"))
*alist* → (((1 . "uno") (2 . "two"))
```

See Also:

- `copy-list`

## pairlis

**Function**

**Syntax:**

```
pairlis keys data &optional alist → new-alist
```

**Arguments and Values:**

- `keys`—a proper list.
- `data`—a proper list.
- `alist`—an association list. The default is the empty list.
- `new-alist`—an association list.

**Description:**

Returns an association list that associates elements of `keys` to corresponding elements of `data`. The consequences are undefined if `keys` and `data` are not of the same length.

If `alist` is supplied, `pairlis` returns a modified `alist` with the new pairs prepended to it. The new pairs may appear in the resulting association list in either forward or backward order. The result of

```
(pairlis '(one two) '(1 2) '((three . 3) (four . 19)))
```

might be

```
((one . 1) (two . 2) (three . 3) (four . 19))
```
or

```lisp
((two . 2) (one . 1) (three . 3) (four . 19))
```

**Examples:**

```lisp
(setq keys '(1 2 3)
data ('"one" "two" "three")
alist '((4 . "four")) → ((4 . "four"))
(pairlis keys data) → ((3 . "three") (2 . "two") (1 . "one"))
(pairlis keys data alist)
→ ((3 . "three") (2 . "two") (1 . "one") (4 . "four"))
alist → ((4 . "four"))
```

**Exceptional Situations:**

Should be prepared to signal an error of type `type-error` if `keys` and `data` are not `proper lists`.

**See Also:**

`acons`

---

**rassoc, rassoc-if, rassoc-if-not**

**Function**

**Syntax:**

```lisp
rassoc item alist &key key test test-not → entry
rassoc-if predicate alist &key key → entry
rassoc-if-not predicate alist &key key → entry
```

**Arguments and Values:**

- `item`—an `object`.
- `alist`—an `association list`.
- `predicate`—a `designator` for a function of one argument that returns a `generalized boolean`.
- `test`—a `designator` for a function of two arguments that returns a `generalized boolean`.
- `test-not`—a `designator` for a function of two arguments that returns a `generalized boolean`.
- `key`—a `designator` for a function of one argument, or `nil`.
- `entry`—a `cons` that is an `element` of the `alist`, or `nil`.
Description:

`rassoc`, `rassoc-if`, and `rassoc-if-not` return the first `cons` whose `cdr` satisfies the test. If no such `cons` is found, `nil` is returned.

If `nil` appears in `alist` in place of a pair, it is ignored.

Examples:

```lisp
(setq alist '(((1 . "one") (2 . "two") (3 . 3)))
→ ((1 . "one") (2 . "two") (3 . 3))
(rassoc 3 alist) → (3 . 3)
(rassoc "two" alist) → NIL
(rassoc "two" alist :test 'equal) → (2 . "two")
(rassoc 1 alist :key #'(lambda (x) (if (numberp x) (/ x 3)))) → (3 . 3)
(rassoc 'a '((a . b) (b . c) (c . a) (z . a))) → (C . A)
(rassoc-if #'stringp alist) → (1 . "one")
(rassoc-if-not #'vectorp alist) → (3 . 3)
```

See Also:

`assoc`, Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` parameter is deprecated.

The function `rassoc-if-not` is deprecated.

It is possible to `rplaca` the result of `rassoc`, provided that it is not `nil`, in order to “update” `alist`.

The expressions

```lisp
(rassoc item list :test fn)
```

and

```lisp
(find item list :test fn :key #'cdr)
```

are equivalent in meaning, except when the `item` is `nil` and `nil` appears in place of a pair in the `alist`. See the function `assoc`.

---

**get-properties**

*Function*

Syntax:

`get-properties` *plist indicator-list* → *indicator, value, tail*

Arguments and Values:
plist—a property list.

indicator-list—a proper list (of indicators).

indicator—an object that is an element of indicator-list.

value—an object.

tail—a list.

Description:

get-properties is used to look up any of several property list entries all at once.

It searches the plist for the first entry whose indicator is identical to one of the objects in indicator-list. If such an entry is found, the indicator and value returned are the property indicator and its associated property value, and the tail returned is the tail of the plist that begins with the found entry (i.e., whose car is the indicator). If no such entry is found, the indicator, value, and tail are all nil.

Examples:

(setq x '()) → NIL
(setq *indicator-list* '(prop1 prop2)) → (PROP1 PROP2)
(getf x 'prop1) → NIL
(setf (getf x 'prop1) 'val1) → VAL1
(eq (getf x 'prop1) 'val1) → true
(get-properties x *indicator-list*) → PROP1, VAL1, (PROP1 VAL1)

x → (PROP1 VAL1)

See Also:

get, getf

getf

Accessor

Syntax:

getf plist indicator &optional default → value

(setf (getf place indicator &optional default) new-value)

Arguments and Values:

plist—a property list.

place—a place, the value of which is a property list.
**getf**

*indicator*—an object.

*default*—an object. The default is `nil`.

*value*—an object.

*new-value*—an object.

**Description:**

`getf` finds a *property* on the *plist* whose *property indicator* is identical to *indicator*, and returns its corresponding *property value*. If there are multiple *properties* with that *property indicator*, `getf` uses the first such *property*. If there is no *property* with that *property indicator*, *default* is returned.

`setf` of `getf` may be used to associate a new *object* with an existing indicator in the *property list* held by *place*, or to create a new association if none exists. If there are multiple *properties* with that *property indicator*, `setf` of `getf` associates the *new-value* with the first such *property*. When a `getf` form is used as a `setf` place, any *default* which is supplied is evaluated according to normal left-to-right evaluation rules, but its *value* is ignored.

`setf` of `getf` is permitted to either write the *value* of *place* itself, or modify any part, *car* or *cdr*, of the list structure held by *place*.

**Examples:**

```lisp
(setq x '())  →  NIL
(getf x 'prop1)  →  NIL
(getf x 'prop1 7)  →  7
(getf x 'prop1)  →  NIL
(setf (getf x 'prop1) 'val1)  →  VAL1
(eq (getf x 'prop1) 'val1)  →  true
(getf x 'prop1)  →  VAL1
(getf x 'prop1 7)  →  VAL1
x  →  (PROP1 VAL1)
```

`;; Examples of implementation variation permitted.
(setq foo (list 'a 'b 'c 'd 'e 'f))  →  (A B C D E F)
(setq bar (cddr foo))  →  (C D E F)
(remove foo 'c)  →  true
foo  →  (A B E F)
bar  →  (C D E F)
\[c\]  →  (C)
\[c\]  →  (NIL)
\[c\]  →  (C NIL)
\[c\]  →  (C D)
```

14–50  Programming Language—Common Lisp
See Also:
get, get-properties, setf, Section 5.1.2.2 (Function Call Forms as Places)

Notes:
There is no way (using getf) to distinguish an absent property from one whose value is default; but see get-properties.

Note that while supplying a default argument to getf in a setf situation is sometimes not very interesting, it is still important because some macros, such as push and incf, require a place argument which data is both read from and written to. In such a context, if a default argument is to be supplied for the read situation, it must be syntactically valid for the write situation as well. For example,

(let ((plist '()))
  (incf (getf plist 'count 0))
  plist) → (COUNT 1)

remf

Macro

Syntax:
remf place indicator  →  generalized-boolean

Arguments and Values:
place—a place.
indicator—an object.
generalized-boolean—a generalized boolean.

Description:
remf removes from the property list stored in place a property1 with a property indicator identical to indicator. If there are multiple properties1 with the identical key, remf only removes the first such property. remf returns false if no such property was found, or true if a property was found.

The property indicator and the corresponding property value are removed in an undefined order by destructively splicing the property list. remf is permitted to either setf place or to setf any part, car or cdr, of the list structure held by that place.

For information about the evaluation of subforms of place, see Section 5.1.1.1 (Evaluation of Subforms to Places).
Examples:

- `(setq x (cons () ()))` → `(NIL)
- `(setf (getf (car x) 'prop1) 'val1)` → VAL1
- `(remf (car x) 'prop1)` → `true`
- `(remf (car x) 'prop1)` → `false`

Side Effects:

The property list stored in `place` is modified.

See Also:

`remprop`, `getf`

---

**intersection, nintersection**

*Function*

**Syntax:**

```
intersection list-1 list-2 &key key test test-not  → result-list
nintersection list-1 list-2 &key key test test-not  → result-list
```

**Arguments and Values:**

- `list-1`—a proper list.
- `list-2`—a proper list.
- `test`—a designator for a function of two arguments that returns a generalized boolean.
- `test-not`—a designator for a function of two arguments that returns a generalized boolean.
- `key`—a designator for a function of one argument, or `nil`.
- `result-list`—a list.

**Description:**

`intersection` and `nintersection` return a list that contains every element that occurs in both `list-1` and `list-2`.

`nintersection` is the destructive version of `intersection`. It performs the same operation, but may destroy `list-1` using its cells to construct the result. `list-2` is not destroyed.

The intersection operation is described as follows. For all possible ordered pairs consisting of one element from `list-1` and one element from `list-2`, `:test` or `:test-not` are used to determine whether they satisfy the test. The first argument to the `:test` or `:test-not` function is an element of `list-1`; the second argument is an element of `list-2`. If `:test` or `:test-not` is not supplied, `eql` is used. It is an error if `:test` and `:test-not` are supplied in the same function call.
intersection, nintersection

If :key is supplied (and not nil), it is used to extract the part to be tested from the list element. The argument to the :key function is an element of either list-1 or list-2; the :key function typically returns part of the supplied element. If :key is not supplied or nil, the list-1 and list-2 elements are used.

For every pair that satisfies the test, exactly one of the two elements of the pair will be put in the result. No element from either list appears in the result that does not satisfy the test for an element from the other list. If one of the lists contains duplicate elements, there may be duplication in the result.

There is no guarantee that the order of elements in the result will reflect the ordering of the arguments in any particular way. The result list may share cells with, or be eq to, either list-1 or list-2 if appropriate.

Examples:

```
(setq list1 (list 1 1 2 3 4 a b c "A" "B" "C" "d")
    list2 (list 1 4 5 b c d "a" "B" "c" "D"))
→ (1 4 5 B C D "a" "B" "c" "D")
(intersection list1 list2) → (C B 4 1 1)
(intersection list1 list2 :test 'equal) → ("B" C B 4 1 1)
(intersection list1 list2 :test #'equalp) → ("d" "C" "B" "A" C B 4 1 1)
(nintersection list1 list2) → (1 1 4 B C)
list1 → implementation-dependent ; e.g., (1 1 4 B C)
list2 → implementation-dependent ; e.g., (1 4 5 B C D "a" "B" "c" "D")
(setq list1 (copy-list '((1 . 2) (2 . 3) (3 . 4) (4 . 5))))
→ ((1 . 2) (2 . 3) (3 . 4) (4 . 5))
(setq list2 (copy-list '((1 . 3) (2 . 4) (3 . 6) (4 . 8))))
→ ((1 . 3) (2 . 4) (3 . 6) (4 . 8))
(nintersection list1 list2 :key #'cdr) → ((2 . 3) (3 . 4))
list1 → implementation-dependent ; e.g., ((1 . 2) (2 . 3) (3 . 4))
list2 → implementation-dependent ; e.g., ((1 . 3) (2 . 4) (3 . 6) (4 . 8))
```

Side Effects:

nintersection can modify list-1, but not list-2.

Exceptional Situations:

Should be prepared to signal an error of type type-error if list-1 and list-2 are not proper lists.

See Also:

union, Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects)

Notes:

The :test-not parameter is deprecated.

Since the nintersection side effect is not required, it should not be used in for-effect-only posi-
adjoin  

Function

Syntax:

```
adjoin item list &key key test test-not → new-list
```

Arguments and Values:

- **item**—an object.
- **list**—a proper list.
- **test**—a designator for a function of two arguments that returns a generalized boolean.
- **test-not**—a designator for a function of two arguments that returns a generalized boolean.
- **key**—a designator for a function of one argument, or nil.
- **new-list**—a list.

Description:

Tests whether **item** is the same as an existing element of **list**. If the **item** is not an existing element, **adjoin** adds it to **list** (as if by cons) and returns the resulting **list**; otherwise, nothing is added and the original **list** is returned.

The **test**, **test-not**, and **key** affect how it is determined whether **item** is the same as an element of **list**. For details, see Section 17.2.1 (Satisfying a Two-Argument Test).

Examples:

```
(setq slist '()) → NIL
(adjoin 'a slist) → (A)
slist → NIL
(setq slist (adjoin '(test-item 1) slist)) → ((TEST-ITEM 1))
(adjoin '(test-item 1) slist) → ((TEST-ITEM 1) (TEST-ITEM 1))
(adjoin '(test-item 1) slist :test 'equal) → ((TEST-ITEM 1))
(adjoin '(new-test-item 1) slist :key #'cadr) → ((TEST-ITEM 1))
(adjoin '(new-test-item 1) slist) → ((NEW-TEST-ITEM 1) (TEST-ITEM 1))
```

Exceptional Situations:

Should be prepared to signal an error of type **type-error** if **list** is not a proper list.

See Also:

- **pushnew**, Section 3.6 (Traversal Rules and Side Effects)

14–54 Programming Language—Common Lisp
pushnew  

Syntax:

```
pushnew item place &key key test test-not  
→ new-place-value
```

Arguments and Values:

- `item`—an object.
- `place`—a place, the value of which is a proper list.
- `test`—a designator for a function of two arguments that returns a generalized boolean.
- `test-not`—a designator for a function of two arguments that returns a generalized boolean.
- `key`—a designator for a function of one argument, or `nil`.
- `new-place-value`—a list (the new value of `place`).

Description:

`pushnew` tests whether `item` is the same as any existing element of the `list` stored in `place`. If `item` is not, it is prepended to the `list`, and the new `list` is stored in `place`.

`pushnew` returns the new `list` that is stored in `place`.

Whether or not `item` is already a member of the `list` that is in `place` is determined by comparisons using `:test` or `:test-not`. The first argument to the `:test` or `:test-not` function is `item`; the second argument is an element of the `list` in `place` as returned by the `:key` function (if supplied).

If `:key` is supplied, it is used to extract the part to be tested from both `item` and the `list` element, as for `adjoin`.

The argument to the `:key` function is an element of the `list` stored in `place`. The `:key` function typically returns part part of the element of the `list`. If `:key` is not supplied or `nil`, the `list` element is used.

For information about the evaluation of subforms of `place`, see Section 5.1.1.1 (Evaluation of Subforms to Places).
It is *implementation-dependent* whether or not `pushnew` actually executes the storing form for its place in the situation where the item is already a member of the list held by `place`.

Examples:

```lisp
(setq x '(a (b c) d)) → (A (B C) D)
pushnew 5 (cadr x)) → (5 B C)
x → (A (5 B C) D)
pushnew 'b (cadr x)) → (5 B C)
x → (A (5 B C) D)
(setq lst '((1) (1 2) (1 2 3))) → ((1) (1 2) (1 2 3))
pushnew '(2) lst) → ((2) (1) (1 2) (1 2 3))
pushnew '(1) lst) → ((1) (2) (1) (1 2) (1 2 3))
pushnew '(1) lst :test 'equal) → ((1) (2) (1) (1 2) (1 2 3))
pushnew '(1) lst :key #'car) → ((1) (2) (1) (1 2) (1 2 3))
```

Side Effects:

The contents of `place` may be modified.

See Also:

push, adjoin, Section 5.1 (Generalized Reference)

Notes:

The effect of `(pushnew item place :test p)`

is roughly equivalent to `(setf place (adjoin item place :test p))`

except that the subforms of `place` are evaluated only once, and `item` is evaluated before `place`.

---

**set-difference, nset-difference**

*Function*

**Syntax:**

```
set-difference list-1 list-2 &key key test test-not → result-list
nset-difference list-1 list-2 &key key test test-not → result-list
```

**Arguments and Values:**

- `list-1`—a proper list.
- `list-2`—a proper list.
- `test`—a designator for a function of two arguments that returns a generalized boolean.
- `test-not`—a designator for a function of two arguments that returns a generalized boolean.
set-difference, nset-difference

key—a designator for a function of one argument, or nil.

result-list—a list.

Description:
set-difference returns a list of elements of list-1 that do not appear in list-2.
nset-difference is the destructive version of set-difference. It may destroy list-1.

For all possible ordered pairs consisting of one element from list-1 and one element from list-2, the :test or :test-not function is used to determine whether they satisfy the test. The first argument to the :test or :test-not function is the part of an element of list-1 that is returned by the :key function (if supplied); the second argument is the part of an element of list-2 that is returned by the :key function (if supplied).

If :key is supplied, its argument is a list-1 or list-2 element. The :key function typically returns part of the supplied element. If :key is not supplied, the list-1 or list-2 element is used.

An element of list-1 appears in the result if and only if it does not match any element of list-2.

There is no guarantee that the order of elements in the result will reflect the ordering of the arguments in any particular way. The result list may share cells with, or be eq to, either of list-1 or list-2, if appropriate.

Examples:

(setq lst1 (list "A" "b" "C" "d")
   lst2 (list "a" "B" "C" "d")) → ("a" "B" "C" "d")
(set-difference lst1 lst2) → ("d" "C" "b" "A")
(set-difference lst1 lst2 :test 'equal) → ("b" "A")
(set-difference lst1 lst2 :test #'equalp) → NIL
(nset-difference lst1 lst2 :test #'string= :key #'cdr)
→ ("c" "d")

(setq lst1 (list "A" "b" "C" "d")
   lst2 (list "a" "B" "C" "d")) → ("a" "B" "C" "d")
(set-difference lst1 lst2 :test :test-not #'equalp)
→ ("A" "b")
(nset-difference lst1 lst2 :test :test-not #'equalp)
→ ("a" "b")

(conses 14–57)
Side Effects:
  nset-difference may destroy list-1.

Exceptional Situations:
  Should be prepared to signal an error of type type-error if list-1 and list-2 are not proper lists.

See Also:
  Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects)

Notes:
  The :test-not parameter is deprecated.

set-exclusive-or, nset-exclusive-or  

Function

Syntax:
  set-exclusive-or  list-1 list-2 &key key test test-not  →  result-list  
  nset-exclusive-or  list-1 list-2 &key key test test-not  →  result-list

Arguments and Values:
  list-1—a proper list.
  list-2—a proper list.
  test—a designator for a function of two arguments that returns a generalized boolean.
  test-not—a designator for a function of two arguments that returns a generalized boolean.
  key—a designator for a function of one argument, or nil.
  result-list—a list.

Description:
  set-exclusive-or returns a list of elements that appear in exactly one of list-1 and list-2.
  nset-exclusive-or is the destructive version of set-exclusive-or.

For all possible ordered pairs consisting of one element from list-1 and one element from list-2, the :test or :test-not function is used to determine whether they satisfy the test.

If :key is supplied, it is used to extract the part to be tested from the list-1 or list-2 element. The first argument to the :test or :test-not function is the part of an element of list-1 extracted by the :key function (if supplied); the second argument is the part of an element of list-2 extracted by the :key function (if supplied). If :key is not supplied or nil, the list-1 or list-2 element is used.
The result contains precisely those elements of list-1 and list-2 that appear in no matching pair.

The result list of set-exclusive-or might share storage with one of list-1 or list-2.

Examples:

(setq lst1 (list 1 "a" "b")
        lst2 (list 1 "A" "b")) → (1 "A" "b")
(set-exclusive-or lst1 lst2) → ("b" "A" "b" "a")
(set-exclusive-or lst1 lst2 :test #'equal) → ("A" "a")
(set-exclusive-or lst1 lst2 :test 'equalp) → NIL
(set-exclusive-or lst1 lst2) → ("a" "b" "A" "b")
(set-exclusive-or lst1 lst2 :test #'equal) → ("A" "a")
(set-exclusive-or lst1 lst2 :test #'equalp) → NIL

Examples (with association lists):

(setq lst1 (list (("a" . "b") ("c" . "d") ("e" . "f")))
        lst2 (list (("c" . "a") ("e" . "b") ("d" . "a")))))

(set-exclusive-or lst1 lst2 :test #'string= :key #'cdr)

Side Effects:
nset-exclusive-or is permitted to modify any part, car or cdr, of the list structure of list-1 or list-2.

Exceptional Situations:
Should be prepared to signal an error of type type-error if list-1 and list-2 are not proper lists.

See Also:
Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects)

Notes:
The :test-not parameter is deprecated.

Since the nset-exclusive-or side effect is not required, it should not be used in for-effect-only positions in portable code.

subsetp

Function

Syntax:

subsetp list-1 list-2 &key key test test-not → generalized-boolean
Arguments and Values:

- `list-1`—a proper list.
- `list-2`—a proper list.
- `test`—a designator for a function of two arguments that returns a generalized boolean.
- `test-not`—a designator for a function of two arguments that returns a generalized boolean.
- `key`—a designator for a function of one argument, or `nil`.
- `generalized-boolean`—a generalized boolean.

Description:

`subsetp` returns `true` if every element of `list-1` matches some element of `list-2`, and `false` otherwise.

Whether a list element is the same as another list element is determined by the functions specified by the keyword arguments. The first argument to the `:test` or `:test-not` function is typically part of an element of `list-1` extracted by the `:key` function; the second argument is typically part of an element of `list-2` extracted by the `:key` function.

The argument to the `:key` function is an element of either `list-1` or `list-2`; the return value is part of the element of the supplied list element. If `:key` is not supplied or `nil`, the `list-1` or `list-2` element itself is supplied to the `:test` or `:test-not` function.

Examples:

```lisp
(setq cosmos '(1 "a" (1 2))) → (1 "a" (1 2))
(subsetp '((1) cosmos) → true
(subsetp '(((1 2)) cosmos) → false
(subsetp '((1 2)) cosmos :test 'equal) → true
(subsetp '(1 "A") cosmos :test #'equalp) → true
(subsetp '((1) (2)) '(((1) (2))) → false
(subsetp '((1) (2)) '(((1) (2)) :key #'car) → true
```

Exceptional Situations:

Should be prepared to signal an error of type `type-error` if `list-1` and `list-2` are not proper lists.

See Also:

Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` parameter is deprecated.

14–60 Programming Language—Common Lisp
union, nunion

Function

Syntax:

union list-1 list-2 &key key test test-not → result-list
nunion list-1 list-2 &key key test test-not → result-list

Arguments and Values:

- list-1—a proper list.
- list-2—a proper list.
- test—a designator for a function of two arguments that returns a generalized boolean.
- test-not—a designator for a function of two arguments that returns a generalized boolean.
- key—a designator for a function of one argument, or nil.
- result-list—a list.

Description:

union and nunion return a list that contains every element that occurs in either list-1 or list-2.

For all possible ordered pairs consisting of one element from list-1 and one element from list-2, :test or :test-not is used to determine whether they satisfy the test. The first argument to the :test or :test-not function is the part of the element of list-1 extracted by the :key function (if supplied); the second argument is the part of the element of list-2 extracted by the :key function (if supplied).

The argument to the :key function is an element of list-1 or list-2; the return value is part of the supplied element. If :key is not supplied or nil, the element of list-1 or list-2 itself is supplied to the :test or :test-not function.

For every matching pair, one of the two elements of the pair will be in the result. Any element from either list-1 or list-2 that matches no element of the other will appear in the result.

If there is a duplication between list-1 and list-2, only one of the duplicate instances will be in the result. If either list-1 or list-2 has duplicate entries within it, the redundant entries might or might not appear in the result.

The order of elements in the result do not have to reflect the ordering of list-1 or list-2 in any way. The result list may be eq to either list-1 or list-2 if appropriate.
union, nunion

Examples:

\[ \text{(union '(a b c) '(f a d))} \]
\[ \rightarrow (A B C F D) \]
\[ \rightarrow (B C F A D) \]
\[ \rightarrow (D F A B C) \]

\[ \text{(union '((x 5) (y 6)) '((z 2) (x 4)) :key #'car)} \]
\[ \rightarrow ((X 5) (Y 6) (Z 2)) \]
\[ \rightarrow ((X 4) (Y 6) (Z 2)) \]

\[ \text{(setq lst1 (list 1 2 '(1 2) "a" "b")} \]
\[ \text{lst2 (list 2 3 '(2 3) "B" "C")} \]
\[ \rightarrow (2 3 (2 3) "B" "C") \]
\[ \text{(nunion lst1 lst2)} \]
\[ \rightarrow (1 (1 2) "a" "b" 2 3 (2 3) "B" "C") \]
\[ \rightarrow or (1 2 (1 2) "a" "b" "C" "B" (2 3) 3) \]

Side Effects:

nunion is permitted to modify any part, car or cdr, of the list structure of list-1 or list-2.

Exceptional Situations:

Should be prepared to signal an error of type type-error if list-1 and list-2 are not proper lists.

See Also:

intersection, Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects)

Notes:

The :test-not parameter is deprecated.

Since the nunion side effect is not required, it should not be used in for-effect-only positions in portable code.
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 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.

Figure 15–1 lists some defined names that are applicable to array creation, access, and information operations.

<table>
<thead>
<tr>
<th>adjust-array</th>
<th>array-has-fill-pointer-p</th>
<th>make-array</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjustable-array-p</td>
<td>array-in-bounds-p</td>
<td>svref</td>
</tr>
<tr>
<td>aref</td>
<td>array-rank</td>
<td>upgraded-array-element-type</td>
</tr>
<tr>
<td>array-dimension</td>
<td>array-rank-limit</td>
<td>upgraded-complex-part-type</td>
</tr>
<tr>
<td>array-dimension-limit</td>
<td>array-row-major-index</td>
<td>vector</td>
</tr>
<tr>
<td>array-dimensions</td>
<td>array-total-size</td>
<td>vector-pop</td>
</tr>
<tr>
<td>array-displacement</td>
<td>array-total-size-limit</td>
<td>vector-push</td>
</tr>
<tr>
<td>array-element-type</td>
<td>fill-pointer</td>
<td>vector-push-extend</td>
</tr>
</tbody>
</table>

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 Tᵧ, then the upgraded array element type of Tₓ must be a subtype of the upgraded array element type of Tᵧ. 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. Figure 15–2 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 Chapter 16 (Strings).

<table>
<thead>
<tr>
<th>char</th>
<th>string-equal</th>
<th>string-upcase</th>
</tr>
</thead>
<tbody>
<tr>
<td>make-string</td>
<td>string-greaterp</td>
<td>string/=</td>
</tr>
<tr>
<td>nstring-capsitalize</td>
<td>string-lessp</td>
<td>string&lt;</td>
</tr>
<tr>
<td>nstring-downcase</td>
<td>string-not-equal</td>
<td>string=</td>
</tr>
<tr>
<td>nstring-upcase</td>
<td>string-not-greaterp</td>
<td>string&gt;</td>
</tr>
<tr>
<td>schar</td>
<td>string-not-lessp</td>
<td>string&gt;</td>
</tr>
<tr>
<td>string</td>
<td>string-right-trim</td>
<td>string-trim</td>
</tr>
</tbody>
</table>

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. Figure 15–3 lists some defined names for operations on bit arrays.
<table>
<thead>
<tr>
<th>bit</th>
<th>bit-ior</th>
<th>bit-orc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit-and</td>
<td>bit-nand</td>
<td>bit-xor</td>
</tr>
<tr>
<td>bit-andc1</td>
<td>bit-nor</td>
<td>sbit</td>
</tr>
<tr>
<td>bit-andc2</td>
<td>bit-not</td>
<td></td>
</tr>
<tr>
<td>bit-eqv</td>
<td>bit-orc1</td>
<td></td>
</tr>
</tbody>
</table>

Figure 15–3. Operators that Manipulate Bit Arrays
**array**

*System Class*

**Class Precedence List:**

array, t

**Description:**

An *array* contains objects arranged according to a Cartesian coordinate system. An array provides mappings from a set of fixnums \( \{i_0, i_1, \ldots, i_{r-1}\} \) to corresponding elements of the array, where \( 0 \leq i_j < d_j \), \( r \) is the rank of the array, and \( d_j \) is the size of dimension \( j \) of the array.

When an array is created, the program requesting its creation may declare that all elements are of a particular type, called the *expressed array element type*. The implementation is permitted to upgrade this type in order to produce the actual array element type, which is the element type for the array is actually specialized. See the function `upgraded-array-element-type`.

**Compound TypeSpecifier Kind:**

Specializing.

**Compound TypeSpecifier Syntax:**

\[
\text{array } ([\text{element-type } | *] [\text{dimension-spec}])
\]

\[
dimension-spec ::= \text{rank } | * | ({\text{dimension } | *})^*
\]

**Compound TypeSpecifier Arguments:**

*dimension*—a valid array dimension.

*element-type*—a type specifier.

*rank*—a non-negative fixnum.

**Compound TypeSpecifier Description:**

This denotes the set of arrays whose element type, rank, and dimensions match any given element-type, rank, and dimensions. Specifically:

If *element-type* is the symbol `*`, arrays are not excluded on the basis of their element type. Otherwise, only those arrays are included whose actual array element type is the result of upgrading element-type; see Section 15.1.2.1 (Array Upgrading).

If the *dimension-spec* is a rank, the set includes only those arrays having that rank. If the *dimension-spec* is a list of dimensions, the set includes only those arrays having a rank given by the length of the dimensions, and having the indicated dimensions; in this case, `*` matches any value for the corresponding dimension. If the *dimension-spec* is the symbol `*`, the set is not restricted on the basis of rank or dimension.
See Also:

*print-array*, aref, make-array, vector, Section 2.4.8.12 (Sharpsign A), Section 22.1.3.8 (Printing Other Arrays)

Notes:

Note that the type (array t) is a proper subtype of the type (array *). The reason is that the type (array t) is the set of arrays that can hold any object (the elements are of type t, which includes all objects). On the other hand, the type (array *) is the set of all arrays whatsoever, including for example arrays that can hold only characters. The type (array character) is not a subtype of the type (array t): the two sets are disjoint because the type (array character) is not the set of all arrays that can hold characters, but rather the set of arrays that are specialized to hold precisely characters and no other objects.

simple-array

Type

Supertypes:
simple-array, array, t

Description:

The type of an array that is not displaced to another array, has no fill pointer, and is not expressly adjustable is a subtype of type simple-array. The concept of a simple array exists to allow the implementation to use a specialized representation and to allow the user to declare that certain values will always be simple arrays.

The types simple-vector, simple-string, and simple-bit-vector are disjoint subtypes of type simple-array, for they respectively mean (simple-array t (*)), the union of all (simple-array c (*)) for any c being a subtype of type character, and (simple-array bit (*)).

Compound TypeSpecifier Kind:

Specializing.

Compound TypeSpecifier Syntax:

(simple-array [[element-type | *] [dimension-spec]])

dimension-spec ::= rank | * | ([dimension | *]*)

Compound TypeSpecifier Arguments:

dimension—a valid array dimension.

element-type—a type specifier.

rank—a non-negative fixnum.

15–6 Programming Language—Common Lisp
Compound Type Specifier Description:
This compound type specifier is treated exactly as the corresponding compound type specifier for type array would be treated, except that the set is further constrained to include only simple arrays.

Notes:
It is implementation-dependent whether displaced arrays, vectors with fill pointers, or arrays that are actually adjustable are simple arrays.

(simple-array *) refers to all simple arrays regardless of element type, (simple-array type-specifier) refers only to those simple arrays that can result from giving type-specifier as the :element-type argument to make-array.

vector

Class Precedence List:
vector, array, sequence, t

Description:
Any one-dimensional array is a vector.
The type vector is a subtype of type array; for all types x, (vector x) is the same as (array x (*)).
The type (vector t), the type string, and the type bit-vector are disjoint subtypes of type vector.

Compound Type Specifier Kind:
Specializing.

Compound Type Specifier Syntax:
(vector [{element-type | *} [{size | *]}])

Compound Type Specifier Arguments:
size—a non-negative fixnum.
element-type—a type specifier.

Compound Type Specifier Description:
This denotes the set of specialized vectors whose element type and dimension match the specified values. Specifically:
If `element-type` is the symbol `*`, vectors are not excluded on the basis of their `element type`. Otherwise, only those vectors are included whose `actual array element type` is the result of `upgrading element-type`; see Section 15.1.2.1 (Array Upgrading).

If a `size` is specified, the set includes only those vectors whose only `dimension` is `size`. If the symbol `*` is specified instead of a `size`, the set is not restricted on the basis of `dimension`.

See Also:
Section 15.1.2.2 (Required Kinds of Specialized Arrays), Section 2.4.8.3 (Sharpsign Left-Parenthesis), Section 22.1.3.7 (Printing Other Vectors), Section 2.4.8.12 (Sharpsign A)

Notes:
The `type (vector e s)` is equivalent to the `type (array e (s))`.

The `type (vector bit)` has the name `bit-vector`.

The union of all types `(vector C)`, where `C` is any `subtype` of `character`, has the name `string`.

`(vector *)` refers to all vectors regardless of element type, `(vector type-specifier)` refers only to those vectors that can result from giving `type-specifier` as the `:element-type` argument to `make-array`.

---

### simple-vector

**Type**

**Supertypes:**
simple-vector, vector, simple-array, array, sequence, t

**Description:**
The `type` of a vector that is not displaced to another array, has no fill pointer, is not expressly adjustable and is able to hold elements of any type is a subtype of type `simple-vector`.

The `type simple-vector` is a subtype of `type vector`, and is a subtype of `type (vector t)`.

**Compound Type Specifier Kind:**
Specializing.

**Compound Type Specifier Syntax:**
(simple-vector [size])

**Compound Type Specifier Arguments:**
`size`—a non-negative `fixnum`, or the symbol `*`. The default is the symbol `*`.  

15–8 Programming Language—Common Lisp
Compound Type Specifier Description:
This is the same as `(simple-array t (size))`.

bit-vector

System Class

Class Precedence List:
bit-vector, vector, array, sequence, t

Description:
A bit vector is a vector the element type of which is bit.
The type bit-vector is a subtype of type vector, for bit-vector means (vector bit).

Compound Type Specifier Kind:
Abbreviating.

Compound Type Specifier Syntax:
(bit-vector [size])

Compound Type Specifier Arguments:
size—a non-negative fixnum, or the symbol *.

Compound Type Specifier Description:
This denotes the same type as the type (array bit (size)); that is, the set of bit vectors of size size.

See Also:
Section 2.4.8.4 (Sharpsign Asterisk), Section 22.1.3.6 (Printing Bit Vectors), Section 15.1.2.2 (Required Kinds of Specialized Arrays)
simple-bit-vector

Supertypes:
   simple-bit-vector, bit-vector, vector, simple-array, array, sequence, t

Description:
The type of a bit vector that is not displaced to another array, has no fill pointer, and is not expressly adjustable is a subtype of type simple-bit-vector.

Compound TypeSpecifier Kind:
   Abbreviating.

Compound TypeSpecifier Syntax:
   (simple-bit-vector [size])

Compound TypeSpecifier Arguments:
   size—a non-negative fixnum, or the symbol *. The default is the symbol *.

Compound TypeSpecifier Description:
   This denotes the same type as the type (simple-array bit (size)); that is, the set of simple bit vectors of size size.

make-array

Syntax:
   make-array dimensions &key element-type
      initial-element
      initial-contents
      adjustable
      fill-pointer
      displaced-to
      displaced-index-offset
   → new-array

Arguments and Values:
   dimensions—a designator for a list of valid array dimensions.

   element-type—a type specifier. The default is t.

   initial-element—an object.
make-array

initial-contents—an object.

adjustable—a generalized boolean. The default is nil.

fill-pointer—a valid fill pointer for the array to be created, or t or nil. The default is nil.

displaced-to—an array or nil. The default is nil. This option must not be supplied if either initial-element or initial-contents is supplied.

displaced-index-offset—a valid array row-major index for displaced-to. The default is 0. This option must not be supplied unless a non-nil displaced-to is supplied.

new-array—an array.

Description:

Creates and returns an array constructed of the most specialized type that can accommodate elements of type given by element-type. If dimensions is nil then a zero-dimensional array is created.

Dimensions represents the dimensionality of the new array.

element-type indicates the type of the elements intended to be stored in the new-array. The new-array can actually store any objects of the type which results from upgrading element-type; see Section 15.1.2.1 (Array Upgrading).

If initial-element is supplied, it is used to initialize each element of new-array. If initial-element is supplied, it must be of the type given by element-type. initial-element cannot be supplied if either the :initial-contents option is supplied or displaced-to is non-nil. If initial-element is not supplied, the consequences of later reading an uninitialized element of new-array are undefined unless either initial-contents is supplied or displaced-to is non-nil.

initial-contents is used to initialize the contents of array. For example:

(make-array '(4 2 3) :initial-contents
 '(((a b c) (1 2 3))
 ((d e f) (3 1 2))
 ((g h i) (2 3 1))
 ((j k l) (0 0 0))))

initial-contents is composed of a nested structure of sequences. The numbers of levels in the structure must equal the rank of array. Each leaf of the nested structure must be of the type given by element-type. If array is zero-dimensional, then initial-contents specifies the single element. Otherwise, initial-contents must be a sequence whose length is equal to the first dimension; each element must be a nested structure for an array whose dimensions are the remaining dimensions, and so on. Initial-contents cannot be supplied if either initial-element is supplied or displaced-to is non-nil. If initial-contents is not supplied, the consequences of later reading an uninitialized element of new-array are undefined unless either initial-element is supplied or displaced-to is non-nil.

If adjustable is non-nil, the array is expressly adjustable (and so actually adjustable); otherwise,
make-array

the array is not expressly adjustable (and it is implementation-dependent whether the array is actually adjustable).

If fill-pointer is non-nil, the array must be one-dimensional; that is, the array must be a vector. If fill-pointer is t, the length of the vector is used to initialize the fill pointer. If fill-pointer is an integer, it becomes the initial fill pointer for the vector.

If displaced-to is non-nil, make-array will create a displaced array and displaced-to is the target of that displaced array. In that case, the consequences are undefined if the actual array element type of displaced-to is not type equivalent to the actual array element type of the array being created. If displaced-to is nil, the array is not a displaced array.

The displaced-index-offset is made to be the index offset of the array. When an array A is given as the :displaced-to argument to make-array when creating array B, then array B is said to be displaced to array A. The total number of elements in an array, called the total size of the array, is calculated as the product of all the dimensions. It is required that the total size of A be no smaller than the sum of the total size of B plus the offset n supplied by the displaced-index-offset. The effect of displacing is that array B does not have any elements of its own, but instead maps accesses to itself into accesses to array A. The mapping treats both arrays as if they were one-dimensional by taking the elements in row-major order, and then maps an access to element k of array B to an access to element k+n of array A.

If make-array is called with adjustable, fill-pointer, and displaced-to each nil, then the result is a simple array. If make-array is called with one or more of adjustable, fill-pointer, or displaced-to being true, whether the resulting array is a simple array is implementation-dependent.

When an array A is given as the :displaced-to argument to make-array when creating array B, then array B is said to be displaced to array A. The total number of elements in an array, called the total size of the array, is calculated as the product of all the dimensions. The consequences are unspecified if the total size of A is smaller than the sum of the total size of B plus the offset n supplied by the displaced-index-offset. The effect of displacing is that array B does not have any elements of its own, but instead maps accesses to itself into accesses to array A. The mapping treats both arrays as if they were one-dimensional by taking the elements in row-major order, and then maps an access to element k of array B to an access to element k+n of array A.

Examples:

(make-array 5) ;; Creates a one-dimensional array of five elements.
(make-array '(3 4) :element-type '(mod 16)) ;; Creates a two-dimensional array, 3 by 4, with four-bit elements.
(make-array 5 :element-type 'single-float) ;; Creates an array of single-floats.

(make-array nil :initial-element nil) → #OANIL
(make-array 4 :initial-element nil) → #(NIL NIL NIL NIL)
(make-array '(2 4) :element-type '(unsigned-byte 2)
make-array

:initial-contents '((0 1 2 3) (3 2 1 0))
→ #2A((0 1 2 3) (3 2 1 0))

(make-array 6
  :element-type 'character
  :initial-element #\a
  :fill-pointer 3) → "aaa"

The following is an example of making a displaced array.

(setq a (make-array '(4 3)))
→ #<ARRAY 4x3 simple 32546632>
(dotimes (i 4)
  (dotimes (j 3)
    (setf (aref a i j) (list i 'x j '=' (* i j))))
→ NIL
(setq b (make-array 8 :displaced-to a
  :displaced-index-offset 2))
→ #<ARRAY 8 indirect 32550757>
(dotimes (i 8)
  (print (list i (aref b i))))
→ (0 (0 X 2 = 0))
→ (1 (1 X 0 = 0))
→ (2 (1 X 1 = 1))
→ (3 (1 X 2 = 2))
→ (4 (2 X 0 = 0))
→ (5 (2 X 1 = 2))
→ (6 (2 X 2 = 4))
→ (7 (3 X 0 = 0))
→ NIL

The last example depends on the fact that arrays are, in effect, stored in row-major order.

(setq a1 (make-array 50))
→ #<ARRAY 50 simple 32562043>
(setq b1 (make-array 20 :displaced-to a1 :displaced-index-offset 10))
→ #<ARRAY 20 indirect 32563346>
(length b1) → 20

(setq a2 (make-array 50 :fill-pointer 10))
→ #<ARRAY 50 fill-pointer 10 46100216>
(setq b2 (make-array 20 :displaced-to a2 :displaced-index-offset 10))
→ #<ARRAY 20 indirect 46104010>
(length a2) → 10
(length b2) → 20
(setq a3 (make-array 50 :fill-pointer 10))
→ #<ARRAY 50 fill-pointer 10 46105663>
(setq b3 (make-array 20 :displaced-to a3 :displaced-index-offset 10
 :fill-pointer 5))
→ #<ARRAY 20 indirect, fill-pointer 5 46107432>
(length a3) → 10
(length b3) → 5

See Also:
adjustable-array-p, aref, arrayp, array-element-type, array-rank-limit, array-dimension-limit,
fill-pointer, upgraded-array-element-type

Notes:
There is no specified way to create an array for which adjustable-array-p definitely returns false.
There is no specified way to create an array that is not a simple array.

adjust-array

Function

Syntax:

adjust-array array new-dimensions &key element-type
initial-element
initial-contents
fill-pointer
displaced-to
displaced-index-offset

→ adjusted-array

Arguments and Values:
array—an array.
new-dimensions—a valid array dimension or a list of valid array dimensions.

element-type—a type specifier.
initial-element—an object. Initial-element must not be supplied if either initial-contents or
displaced-to is supplied.

initial-contents—an object. If array has rank greater than zero, then initial-contents is composed
of nested sequences, the depth of which must equal the rank of array. Otherwise, array is zero-
dimensional and initial-contents supplies the single element. initial-contents must not be supplied if
either initial-element or displaced-to is given.

fill-pointer—a valid fill pointer for the array to be created, or t, or nil. The default is nil.
adjust-array

-displaced-to—an array or nil. initial-elements and initial-contents must not be supplied if displaced-to is supplied.

displaced-index-offset—an object of type (fixnum 0 n) where n is (array-total-size displaced-to). displaced-index-offset may be supplied only if displaced-to is supplied.

adjusted-array—an array.

Description:
adjust-array changes the dimensions or elements of array. The result is an array of the same type and rank as array, that is either the modified array, or a newly created array to which array can be displaced, and that has the given new-dimensions.

New-dimensions specify the size of each dimension of array.

Element-type specifies the type of the elements of the resulting array. If element-type is supplied, the consequences are unspecified if the upgraded array element type of element-type is not the same as the actual array element type of array.

If initial-contents is supplied, it is treated as for make-array. In this case none of the original contents of array appears in the resulting array.

If fill-pointer is an integer, it becomes the fill pointer for the resulting array. If fill-pointer is the symbol t, it indicates that the size of the resulting array should be used as the fill pointer. If fill-pointer is nil, it indicates that the fill pointer should be left as it is.

If displaced-to non-nil, a displaced array is created. The resulting array shares its contents with the array given by displaced-to. The resulting array cannot contain more elements than the array it is displaced to. If displaced-to is not supplied or nil, the resulting array is not a displaced array. If array A is created displaced to array B and subsequently array B is given to adjust-array, array A will still be displaced to array B. Although array might be a displaced array, the resulting array is not a displaced array unless displaced-to is supplied and not nil. The interaction between adjust-array and displaced arrays is as follows given three arrays, A, B, and C:

A is not displaced before or after the call

(adjust-array A ...)

The dimensions of A are altered, and the contents rearranged as appropriate. Additional elements of A are taken from initial-element. The use of initial-contents causes all old contents to be discarded.

A is not displaced before, but is displaced to C after the call

(adjust-array A ... :displaced-to C)

None of the original contents of A appears in A afterwards; A now contains the contents of C, without any rearrangement of C.
adjust-array

\[A\] is displaced to \(B\) before the call, and is displaced to \(C\) after the call:

\[
\text{(adjust-array } A \ldots \text{ :displaced-to } B)\]
\[
\text{(adjust-array } A \ldots \text{ :displaced-to } C)\]

\(B\) and \(C\) might be the same. The contents of \(B\) do not appear in \(A\) afterward unless such contents also happen to be in \(C\). If \text{displaced-index-offset} is not supplied in the \text{adjust-array} call, it defaults to zero; the old offset into \(B\) is not retained.

\(A\) is displaced to \(B\) before the call, but not displaced afterward.

\[
\text{(adjust-array } A \ldots \text{ :displaced-to } B)\]
\[
\text{(adjust-array } A \ldots \text{ :displaced-to nil)}\]

\(A\) gets a new “data region,” and contents of \(B\) are copied into it as appropriate to maintain the existing old contents; additional elements of \(A\) are taken from \text{initial-element} if supplied. However, the use of \text{initial-contents} causes all old contents to be discarded.

If \text{displaced-index-offset} is supplied, it specifies the offset of the resulting \text{array} from the beginning of the \text{array} that it is displaced to. If \text{displaced-index-offset} is not supplied, the offset is 0. The size of the resulting \text{array} plus the offset value cannot exceed the size of the \text{array} that it is displaced to.

If only \text{new-dimensions} and an \text{initial-element} argument are supplied, those elements of \text{array} that are still in bounds appear in the resulting \text{array}. The elements of the resulting \text{array} that are not in the bounds of \text{array} are initialized to \text{initial-element}; if \text{initial-element} is not provided, the consequences of later reading any such new element of \text{new-array} before it has been initialized are undefined.

If \text{initial-contents} or \text{displaced-to} is supplied, then none of the original contents of \text{array} appears in the new \text{array}.

The consequences are unspecified if \text{array} is adjusted to a size smaller than its \text{fill pointer} without supplying the \text{fill-pointer} argument so that its \text{fill-pointer} is properly adjusted in the process.

If \(A\) is displaced to \(B\), the consequences are unspecified if \(B\) is adjusted in such a way that it no longer has enough elements to satisfy \(A\).

If \text{adjust-array} is applied to an \text{array} that is \text{actually adjustable}, the \text{array} returned is \text{identical} to \text{array}. If the \text{array} returned by \text{adjust-array} is \text{distinct} from \text{array}, then the argument \text{array} is unchanged.

Note that if an \text{array} \(A\) is displaced to another \text{array} \(B\), and \(B\) is displaced to another \text{array} \(C\), and \(B\) is altered by \text{adjust-array}, \(A\) must now refer to the adjust contents of \(B\). This means that an implementation cannot collapse the chain to make \(A\) refer to \(C\) directly and forget that the chain of reference passes through \(B\). However, caching techniques are permitted as long as they preserve the semantics specified here.
Examples:

```lisp
(adjustable-array-p
 (setq ada (adjust-array
       (make-array '(2 3)
         :adjustable t
         :initial-contents '((a b c) (1 2 3)))
       '(4 6))) ) → T
(array-dimensions ada) → (4 6)
(aref ada 1 1) → 2
(setq beta (make-array '(2 3) :adjustable t))
→ #2A((NIL NIL NIL) (NIL NIL NIL))
(adjust-array beta '(4 6) :displaced-to ada)
→ #2A((A B C NIL NIL)
       (1 2 3 NIL NIL NIL)
       (NIL NIL NIL NIL NIL NIL))
(array-dimensions beta) → (4 6)
(aref beta 1 1) → 2

Suppose that the 4-by-4 array in m looks like this:

#2A(( alpha beta gamma delta )
    ( epsilon zeta eta theta )
    ( iota kappa lambda mu )
    ( nu xi omicron pi ))

Then the result of

(adjust-array m '(3 5) :initial-element 'baz)

is a 3-by-5 array with contents

#2A(( alpha beta gamma delta baz )
    ( epsilon zeta eta theta baz )
    ( iota kappa lambda mu baz ))
```

Exceptional Situations:

An error of type error is signaled if fill-pointer is supplied and non-nil but array has no fill pointer.

See Also:

adjustable-array-p, make-array, array-dimension-limit, array-total-size-limit, array
adjustable-array-p

Function

Syntax:
adjustable-array-p array → generalized-boolean

Arguments and Values:
array—an array.
generalized-boolean—a generalized boolean.

Description:
Returns true if and only if adjust-array could return a value which is identical to array when given that array as its first argument.

Examples:
(adjustable-array-p (make-array 5
        :element-type 'character
        :adjustable t
        :fill-pointer 3)) → true
(adjustable-array-p (make-array 4)) → implementation-dependent

Exceptional Situations:
Should signal an error of type type-error if its argument is not an array.

See Also:
adjust-array, make-array

aref

Accessor

Syntax:
aref array &rest subscripts → element
(setf (aref array &rest subscripts) new-element)

Arguments and Values:
array—an array.

subscripts—a list of valid array indices for the array.

element, new-element—an object.
Description:

Accesses the array element specified by the subscripts. If no subscripts are supplied and array is zero rank, aref accesses the sole element of array.

aref ignores fill pointers. It is permissible to use aref to access any array element, whether active or not.

Examples:

If the variable foo names a 3-by-5 array, then the first index could be 0, 1, or 2, and then second index could be 0, 1, 2, 3, or 4. The array elements can be referred to by using the function aref; for example, (aref foo 2 1) refers to element (2, 1) of the array.

(aref (setq alpha (make-array 4)) 3) → implementation-dependent
(setf (aref alpha 3) 'sirens) → SIRENS
(aref alpha 3) → SIRENS
(aref (setq beta (make-array '(2 4)
:element-type '(unsigned-byte 2)
:initial-contents '((0 1 2 3) (3 2 1 0)))) 1 2) → 1
(setq gamma '(0 2))
(apply #'aref beta gamma) → 2
(setf (apply #'aref beta gamma) 3) → 3
(apply #'aref beta gamma) → 3
(aref beta 0 2) → 3

See Also:
bit, char, elt, row-major-aref, svref, Section 3.2.1 (Compiler Terminology)

array-dimension

Function

Syntax:

array-dimension array axis-number → dimension

Arguments and Values:

array—an array.

axis-number—an integer greater than or equal to zero and less than the rank of the array.

dimension—a non-negative integer.

Description:

array-dimension returns the axis-number dimension of array. (Any fill pointer is ignored.)
Examples:

(array-dimension (make-array 4) 0) → 4
(array-dimension (make-array '(2 3)) 1) → 3

Affected By:
None.

See Also:
array-dimensions, length

Notes:

(array-dimension array n) ≡ (nth n (array-dimensions array))
array-element-type

Syntax:  
\[
\text{array-element-type} \ \text{array} \rightarrow \text{typespec}
\]

Arguments and Values:  
- \text{array}—an \text{array}.
- \text{typespec}—a \text{type specifier}.

Description:  
Returns a \text{type specifier} which represents the \text{actual array element type} of the array, which is the set of \text{objects} that such an \text{array} can hold. (Because of \text{array upgrading}, this \text{type specifier} can in some cases denote a \text{supertype} of the \text{expressed array element type} of the \text{array}.)

Examples:  
\[
\begin{align*}
\text{(array-element-type (make-array 4))} & \rightarrow \text{T} \\
\text{(array-element-type (make-array 12 :element-type '(unsigned-byte 8)))} & \rightarrow \text{implementation-dependent} \\
\text{(array-element-type (make-array 12 :element-type '(unsigned-byte 5)))} & \rightarrow \text{implementation-dependent} \\
\text{(array-element-type (make-array 5 :element-type '(mod 5)))} & \text{could be (mod 5), (mod 8), fixnum, t, or any other type of which (mod 5) is a subtype.}
\end{align*}
\]

Affected By:  
The \text{implementation}.

Exceptional Situations:  
Should signal an error of \text{type type-error} if its argument is not an \text{array}.

See Also:  
array, make-array, subtypep, upgraded-array-element-type

array-has-fill-pointer-p

Syntax:  
\[
\text{array-has-fill-pointer-p} \ \text{array} \rightarrow \text{generalized-boolean}
\]
Arguments and Values:

array—an array.

generalized-boolean—a generalized boolean.

Description:

Returns true if array has a fill pointer; otherwise returns false.

Examples:

(array-has-fill-pointer-p (make-array 4)) \rightarrow \text{implementation-dependent}
(array-has-fill-pointer-p (make-array '(2 3))) \rightarrow \text{false}
(array-has-fill-pointer-p
(make-array 8
 :fill-pointer 2
 :initial-element 'filler)) \rightarrow \text{true}

Exceptional Situations:

Should signal an error of type type-error if its argument is not an array.

See Also:
make-array, fill-pointer

Notes:

Since arrays of rank other than one cannot have a fill pointer, array-has-fill-pointer-p always returns nil when its argument is such an array.

array-displacement

Syntax:

array-displacement array \rightarrow \text{displaced-to, displaced-index-offset}

Arguments and Values:

array—an array.

displaced-to—an array or nil.

displaced-index-offset—a non-negative fixnum.

Description:

If the array is a displaced array, returns the values of the :displaced-to and :displaced-index-offset options for the array (see the functions make-array and adjust-array).

If the array is not a displaced array, nil and 0 are returned.
If `array-displacement` is called on an `array` for which a non-nil object was provided as the `:displaced-to` argument to `make-array` or `adjust-array`, it must return that object as its first value. It is implementation-dependent whether `array-displacement` returns a non-nil primary value for any other `array`.

Examples:

```lisp
(setq a1 (make-array 5)) → #<ARRAY 5 simple 46115576>
(setq a2 (make-array 4 :displaced-to a1
           :displaced-index-offset 1))
→ #<ARRAY 4 indirect 46117134>
(array-displacement a2)
→ #<ARRAY 5 simple 46115576>, 1
(setq a3 (make-array 2 :displaced-to a2
           :displaced-index-offset 2))
→ #<ARRAY 2 indirect 46122527>
(array-displacement a3)
→ #<ARRAY 4 indirect 46117134>, 2
```

Exceptional Situations:
Should signal an error of type `type-error` if `array` is not an `array`.

See Also:
`make-array`

---

**array-in-bounds-p**

*Function*

Syntax:

```
array-in-bounds-p array &rest subscripts → generalized-boolean
```

Arguments and Values:

- `array`—an `array`.
- `subscripts`—a list of integers of length equal to the rank of the `array`.
- `generalized-boolean`—a generalized boolean.

Description:
Returns `true` if the `subscripts` are all in bounds for `array`; otherwise returns `false`. (If `array` is a `vector` with a fill pointer, that fill pointer is ignored.)

Examples:

```
(setq a (make-array '(7 11) :element-type 'string-char))
```
(array-in-bounds-p a 0 0) → true
(array-in-bounds-p a 6 10) → true
(array-in-bounds-p a 0 -1) → false
(array-in-bounds-p a 0 11) → false
(array-in-bounds-p a 7 0) → false

See Also:
array-dimensions

Notes:

(array-in-bounds-p array subscripts)
≡ (and (not (some #'minusp (list subscripts)))
      (every #'< (list subscripts) (array-dimensions array)))

array-rank

Syntax:
array-rank array  → rank

Arguments and Values:
array—an array.
rank—a non-negative integer.

Description:
Returns the number of dimensions of array.

Examples:
(array-rank (make-array '())) → 0
(array-rank (make-array 4)) → 1
(array-rank (make-array '(4))) → 1
(array-rank (make-array '(2 3))) → 2

Exceptional Situations:
Should signal an error of type type-error if its argument is not an array.

See Also:
array-rank-limit, make-array
array-row-major-index  

Function

Syntax:

array-row-major-index array &rest subscripts → index

Arguments and Values:

array—an array.

subscripts—a list of valid array indices for the array.

index—a valid array row-major index for the array.

Description:

Computes the position according to the row-major ordering of array for the element that is specified by subscripts, and returns the offset of the element in the computed position from the beginning of array.

For a one-dimensional array, the result of array-row-major-index equals subscript.

array-row-major-index ignores fill pointers.

Examples:

(setq a (make-array '(4 7) :element-type '(unsigned-byte 8)))
(array-row-major-index a 1 2) → 9
(array-row-major-index
  (make-array '(2 3 4)
    :element-type '(unsigned-byte 8)
    :displaced-to a
    :displaced-index-offset 4)
  0 2 1) → 9

Notes:

A possible definition of array-row-major-index, with no error-checking, is

(defun array-row-major-index (a &rest subscripts)
  (apply #'+ (maplist #'(lambda (x y)
                       (* (car x) (apply #'* (cdr y))))
                  subscripts
                  (array-dimensions a))))
array-total-size  

Function

Syntax:

array-total-size array → size

Arguments and Values:

array—an array.

size—a non-negative integer.

Description:

Returns the array total size of the array.

Examples:

(array-total-size (make-array 4)) → 4
(array-total-size (make-array 4 :fill-pointer 2)) → 4
(array-total-size (make-array 0)) → 0
(array-total-size (make-array '(4 2))) → 8
(array-total-size (make-array '(4 0))) → 0
(array-total-size (make-array '())) → 1

Exceptional Situations:

Should signal an error of type type-error if its argument is not an array.

See Also:

make-array, array-dimensions

Notes:

If the array is a vector with a fill pointer, the fill pointer is ignored when calculating the array total size.

Since the product of no arguments is one, the array total size of a zero-dimensional array is one.

(array-total-size x)
≡ (apply #'* (array-dimensions x))
≡ (reduce #'* (array-dimensions x))
arrayp

Function

Syntax:

arrayp object  → generalized-boolean

Arguments and Values:

object—an object.

generalized-boolean—a generalized boolean.

Description:

Returns true if object is of type array; otherwise, returns false.

Examples:

(arrayp (make-array '(2 3 4) :adjustable t)) → true
(arrayp (make-array 6)) → true
(arrayp #*1011) → true
(arrayp "hi") → true
(arrayp 'hi) → false
(arrayp 12) → false

See Also:

typep

Notes:

(arrayp object) ≡ (typep object 'array)

fill-pointer

Accessor

Syntax:

fill-pointer vector  → fill-pointer

(setf (fill-pointer vector) new-fill-pointer)

Arguments and Values:

vector—a vector with a fill pointer.

fill-pointer, new-fill-pointer—a valid fill pointer for the vector.
Description:

Accesses the fill pointer of vector.

Examples:

```lisp
(setq a (make-array 8 :fill-pointer 4)) → #(NIL NIL NIL NIL)
(fill-pointer a) → 4
(dotimes (i (length a)) (setf (aref a i) (* i i))) → NIL
a → #(0 1 4 9)
(setf (fill-pointer a) 3) → 3
(fill-pointer a) → 3
a → #(0 1 4)
(setf (fill-pointer a) 8) → 8
a → #(0 1 4 9 NIL NIL NIL NIL)
```

Exceptional Situations:

Should signal an error of type type-error if vector is not a vector with a fill pointer.

See Also:

make-array, length

Notes:

There is no operator that will remove a vector’s fill pointer.

---

**row-major-aref**

*Accessor*

Syntax:

```
row-major-aref array index  → element
(setf (row-major-aref array index) new-element)
```

Arguments and Values:

- **array**—an array.
- **index**—a valid array row-major index for the array.
- **element, new-element**—an object.

Description:

Considers array as a vector by viewing its elements in row-major order, and returns the element of that vector which is referred to by the given index.

row-major-aref is valid for use with setf.
See Also:

aref, array-row-major-index

Notes:

(row-major-aref array index) ≡
(aref (make-array (array-total-size array)
 :displaced-to array
 :element-type (array-element-type array))
 index)

(aref array i1 i2 ...) ≡
(row-major-aref array (array-row-major-index array i1 i2))

upgraded-array-element-type

Function

Syntax:

upgraded-array-element-type typespec &optional environment → upgraded-typespec

Arguments and Values:

typespec—a type specifier.

evironment—an environment object. The default is nil, denoting the null lexical environment

and the current global environment.

upgraded-typespec—a type specifier.

Description:

Returns the element type of the most specialized array representation capable of holding items of
the type denoted by typespec.

The typespec is a subtype of (and possibly type equivalent to) the upgraded-typespec.

If typespec is bit, the result is type equivalent to bit. If typespec is base-char, the result is type
equivalent to base-char. If typespec is character, the result is type equivalent to character.

The purpose of upgraded-array-element-type is to reveal how an implementation does its
upgrading.

The environment is used to expand any derived type specifiers that are mentioned in the typespec.

See Also:

array-element-type, make-array

Arrays 15–29
Notes:

Except for storage allocation consequences and dealing correctly with the optional `environment` argument, `upgraded-array-element-type` could be defined as:

```lisp
(defun upgraded-array-element-type (type &optional environment)
  (array-element-type (make-array 0 :element-type type)))
```

---

**array-dimension-limit**

*Constant Variable*

**Constant Value:**

A positive `fixnum`, the exact magnitude of which is `implementation-dependent`, but which is not less than 1024.

**Description:**

The upper exclusive bound on each individual `dimension` of an `array`.

**See Also:**

`make-array`

---

**array-rank-limit**

*Constant Variable*

**Constant Value:**

A positive `fixnum`, the exact magnitude of which is `implementation-dependent`, but which is not less than 8.

**Description:**

The upper exclusive bound on the `rank` of an `array`.

**See Also:**

`make-array`

---

15–30 Programming Language—Common Lisp
array-total-size-limit  

**Constant Variable**

**Constant Value:**
A positive *fixnum*, the exact magnitude of which is *implementation-dependent*, but which is not less than 1024.

**Description:**
The upper exclusive bound on the *array total size* of an *array*.

The actual limit on the *array total size* imposed by the *implementation* might vary according the *element type* of the *array*; in this case, the value of `array-total-size-limit` will be the smallest of these possible limits.

**See Also:**
- `make-array`, `array-element-type`

---

**simple-vector-p**  

**Function**

**Syntax:**
```
simple-vector-p object  → generalized-boolean
```

**Arguments and Values:**
- `object`—an *object*.
- `generalized-boolean`—a *generalized boolean*.

**Description:**
Returns *true* if `object` is of *type* `simple-vector`; otherwise, returns *false*.

**Examples:**
```
(simple-vector-p (make-array 6)) → true
(simple-vector-p "aaaaaa") → false
(simple-vector-p (make-array 6 :fill-pointer t)) → false
```

**See Also:**
- `simple-vector`

**Notes:**
```
(simple-vector-p object) ⇒ (typep object 'simple-vector)
```
svref

Syntax:

svref simple-vector index → element

(setf (svref simple-vector index) new-element)

Arguments and Values:

simple-vector—a simple vector.

index—a valid array index for the simple-vector.

element, new-element—an object (whose type is a subtype of the array element type of the simple-vector).

Description:

Accesses the element of simple-vector specified by index.

Examples:

(simple-vector-p (setq v (vector 1 2 'sirens))) → true
(svref v 0) → 1
(svref v 2) → SIRENS
(setf (svref v 1) 'newcomer) → NEWCOMER
v → #(1 NEWCOMER SIRENS)

See Also:

aref, sbit, schar, vector, Section 3.2.1 (Compiler Terminology)

Notes:

svref is identical to aref except that it requires its first argument to be a simple vector.

(svref v i) ≡ (aref (the simple-vector v) i)
vector Function

Syntax:

vector &rest objects → vector

Arguments and Values:

object—an object.

vector—a vector of type (vector t *).

Description:

Creates a fresh simple general vector whose size corresponds to the number of objects. The vector is initialized to contain the objects.

Examples:

(arrayp (setq v (vector 1 2 'sirens))) → true
(vectorp v) → true
(simple-vector-p v) → true
(length v) → 3

See Also:

make-array

Notes:

vector is analogous to list.

(vector a₁ a₂ ... aₙ)

≡ (make-array (list n) :element-type t

:initial-contents

(list a₁ a₂ ... aₙ))

vector-pop Function

Syntax:

vector-pop vector → element

Arguments and Values:

vector—a vector with a fill pointer.

element—an object.
Description:
Decreases the fill pointer of vector by one, and retrieves the element of vector that is designated by the new fill pointer.

Examples:
(vector-push (setq fable (list 'fable))
  (setq fa (make-array 8
       :fill-pointer 2
       :initial-element 'sisyphus))) → 2
(fill-pointer fa) → 3
(eq (vector-pop fa) fable) → true
(vector-pop fa) → SISYPHUS
(fill-pointer fa) → 1

Side Effects:
The fill pointer is decreased by one.

Affected By:
The value of the fill pointer.

Exceptional Situations:
An error of type type-error is signaled if vector does not have a fill pointer.
If the fill pointer is zero, vector-pop signals an error of type error.

See Also:
vector-push, vector-push-extend, fill-pointer

---

**vector-push, vector-push-extend**

**Function**

**Syntax:**
vector-push new-element vector → new-index-p
vector-push-extend new-element vector &optional extension → new-index

**Arguments and Values:**

new-element—an object.

vector—a vector with a fill pointer.

extension—a positive integer. The default is implementation-dependent.

new-index-p—a valid array index for vector, or nil.

15–34 Programming Language—Common Lisp
vector-push, vector-push-extend

new-index—a valid array index for vector.

Description:

vector-push and vector-push-extend store new-element in vector. vector-push attempts to store new-element in the element of vector designated by the fill pointer, and to increase the fill pointer by one. If the (>= (fill-pointer vector) (array-dimension vector 0)), neither vector nor its fill pointer are affected. Otherwise, the store and increment take place and vector-push returns the former value of the fill pointer which is one less than the one it leaves in vector.

vector-push-extend is just like vector-push except that if the fill pointer gets too large, vector is extended using adjust-array so that it can contain more elements. Extension is the minimum number of elements to be added to vector if it must be extended.


Examples:

(vector-push (setq fable (list 'fable))
 (setq fa (make-array 8
 :fill-pointer 2
 :initial-element 'first-one)))  →  2

(fill-pointer fa)  →  3
(eq (aref fa 2) fable)  →  true

(vector-push-extend \\X
 (setq aa
 (make-array 5
 :element-type 'character
 :adjustable t
 :fill-pointer 3)))  →  3

(fill-pointer aa)  →  4
(vector-push-extend \\Y aa 4)  →  4
(array-total-size aa)  →  at least 5
(vector-push-extend \\Z aa 4)  →  5
(array-total-size aa)  →  9 ;(or more)

Affected By:

The value of the fill pointer.

How vector was created.

Exceptional Situations:

An error of type error is signaled by vector-push-extend if it tries to extend vector and vector is not actually adjustable.

An error of type error is signaled if vector does not have a fill pointer.
See Also:
    adjustable-array-p, fill-pointer, vector-pop

vectorp

Function

Syntax:
    vectorp object  →  generalized-boolean

Arguments and Values:
    object—an object.

    generalized-boolean—a generalized boolean.

Description:
    Returns true if object is of type vector; otherwise, returns false.

Examples:
    (vectorp "aaaaaa") → true
    (vectorp (make-array 6 :fill-pointer t)) → true
    (vectorp (make-array '(2 3 4))) → false
    (vectorp #*11) → true
    (vectorp #b11) → false

Notes:
    (vectorp object) ≡ (typep object 'vector)

bit, sbit

Accessor

Syntax:
    bit  bit-array &rest subscripts  →  bit
    sbit  bit-array &rest subscripts  →  bit

    (setf (bit bit-array &rest subscripts) new-bit)
    (setf (sbit bit-array &rest subscripts) new-bit)

Arguments and Values:
    bit-array—for bit, a bit array; for sbit, a simple bit array.

15–36  Programming Language—Common Lisp
subscripts—a list of valid array indices for the *bit-array*.

*bit*—a *bit*.

**Description:**

*bit* and *sbit* access the *bit-array* element specified by *subscripts*.

These functions ignore the fill pointer when accessing elements.

**Examples:**

```lisp
(bit (setq ba (make-array 8
             :element-type 'bit
             :initial-element 1))
   3) → 1
(setf (bit ba 3) 0) → 0
(bit ba 3) → 0
(sbit ba 5) → 1
(setf (sbit ba 5) 1) → 1
(sbit ba 5) → 1
```

**See Also:**

*aref*, Section 3.2.1 (Compiler Terminology)

**Notes:**

*bit* and *sbit* are like *aref* except that they require *arrays* to be a *bit array* and a *simple bit array*, respectively.

*bit* and *sbit*, unlike *char* and *schar*, allow the first argument to be an *array* of any *rank*.

---

*bit-and, bit-andc1, bit-andc2, bit-eqv, bit-ior, bit-nand, bit-nor, bit-not, bit-orc1, bit-orc2, bit-xor*

**Function**

**Syntax:**

- `bit-and bit-array1 bit-array2 koptional opt-arg` → `resulting-bit-array`
- `bit-andc1 bit-array1 bit-array2 koptional opt-arg` → `resulting-bit-array`
- `bit-andc2 bit-array1 bit-array2 koptional opt-arg` → `resulting-bit-array`
- `bit-eqv bit-array1 bit-array2 koptional opt-arg` → `resulting-bit-array`
- `bit-ior bit-array1 bit-array2 koptional opt-arg` → `resulting-bit-array`
- `bit-nand bit-array1 bit-array2 koptional opt-arg` → `resulting-bit-array`
- `bit-nor bit-array1 bit-array2 koptional opt-arg` → `resulting-bit-array`
- `bit-orc1 bit-array1 bit-array2 koptional opt-arg` → `resulting-bit-array`
- `bit-orc2 bit-array1 bit-array2 koptional opt-arg` → `resulting-bit-array`
- `bit-xor bit-array1 bit-array2 koptional opt-arg` → `resulting-bit-array`
bit-and, bit-andc1, bit-andc2, bit-eqv, bit-ior, ...

bit-orc2 bit-array1 bit-array2 &optional opt-arg  →  resulting-bit-array
bit-xor bit-array1 bit-array2 &optional opt-arg  →  resulting-bit-array
bit-not bit-array &optional opt-arg  →  resulting-bit-array

Arguments and Values:

bit-array, bit-array1, bit-array2—a bit array.

Opt-arg—a bit array, or t, or nil. The default is nil.

Bit-array, bit-array1, bit-array2, and opt-arg (if an array) must all be of the same rank and dimensions.

resulting-bit-array—a bit array.

Description:

These functions perform bit-wise logical operations on bit-array1 and bit-array2 and return an array of matching rank and dimensions, such that any given bit of the result is produced by operating on corresponding bits from each of the arguments.

In the case of bit-not, an array of rank and dimensions matching bit-array is returned that contains a copy of bit-array with all the bits inverted.

If opt-arg is of type (array bit) the contents of the result are destructively placed into opt-arg. If opt-arg is the symbol t, bit-array or bit-array1 is replaced with the result; if opt-arg is nil or omitted, a new array is created to contain the result.

Figure 15–4 indicates the logical operation performed by each of the functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit-and</td>
<td>and</td>
</tr>
<tr>
<td>bit-eqv</td>
<td>equivalence (exclusive nor)</td>
</tr>
<tr>
<td>bit-not</td>
<td>complement</td>
</tr>
<tr>
<td>bit-ior</td>
<td>inclusive or</td>
</tr>
<tr>
<td>bit-xor</td>
<td>exclusive or</td>
</tr>
<tr>
<td>bit-nand</td>
<td>complement of bit-array1 and bit-array2</td>
</tr>
<tr>
<td>bit-nor</td>
<td>complement of bit-array1 or bit-array2</td>
</tr>
<tr>
<td>bit-andc1</td>
<td>and complement of bit-array1 with bit-array2</td>
</tr>
<tr>
<td>bit-andc2</td>
<td>and bit-array1 with complement of bit-array2</td>
</tr>
<tr>
<td>bit-orc1</td>
<td>or complement of bit-array1 with bit-array2</td>
</tr>
<tr>
<td>bit-orc2</td>
<td>or bit-array1 with complement of bit-array2</td>
</tr>
</tbody>
</table>

Figure 15–4. Bit-wise Logical Operations on Bit Arrays
Examples:

\[
\begin{align*}
\text{(bit-and (setq ba #*11101010) #*01101011)} & \rightarrow \ #*01101010 \\
\text{(bit-and #*1100 #*1010)} & \rightarrow \ #*1000 \\
\text{(bit-andc1 #*1100 #*1010)} & \rightarrow \ #*0010 \\
\text{(setq rba (bit-andc2 ba #*00110011 t))} & \rightarrow \ #*11001000 \\
\text{(eq rba ba)} & \rightarrow \ true \\
\text{(bit-not (setq ba #*11101010))} & \rightarrow \ #*00010101 \\
\text{(setq rba (bit-not ba)} & \rightarrow \ #*0010 \\
\text{(setq tba (make-array 8 :element-type 'bit)))} & \rightarrow \ #*00010101 \\
\text{(equal rba tba)} & \rightarrow \ true \\
\text{(bit-xor #*1100 #*1010)} & \rightarrow \ #*0110
\end{align*}
\]

See Also:

lognot, logand

---

\textbf{bit-vector-p} \hfill \textit{Function}

Syntax:

\[
\text{bit-vector-p object } \rightarrow \text{generalized-boolean}
\]

Arguments and Values:

\text{object}—an \textit{object}.

\text{generalized-boolean}—a \textit{generalized boolean}.

Description:

Returns \textit{true} if \text{object} is of \textit{type} \text{bit-vector}; otherwise, returns \textit{false}.

Examples:

\[
\begin{align*}
\text{(bit-vector-p (make-array 6 :element-type 'bit :fill-pointer t))} & \rightarrow \ true \\
\text{(bit-vector-p #*)} & \rightarrow \ true \\
\text{(bit-vector-p (make-array 6))} & \rightarrow \ false
\end{align*}
\]

See Also:

typep

---

Arrays 15–39
Notes:

(bit-vector-p object) ≡ (typep object 'bit-vector)

### simple-bit-vector-p

**Function**

**Syntax:**

simple-bit-vector-p object → generalized-boolean

**Arguments and Values:**

- `object`—an object.
- `generalized-boolean`—a generalized boolean.

**Description:**

Returns true if `object` is of type `simple-bit-vector`; otherwise, returns false.

**Examples:**

- `(simple-bit-vector-p (make-array 6)) → false`
- `(simple-bit-vector-p #*) → true`

**See Also:**

simple-vector-p

**Notes:**

(simple-bit-vector-p object) ≡ (typep object 'simple-bit-vector)
Programming Language—Common Lisp

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.
**string**

*System Class*

**Class Precedence List:**
string, vector, array, sequence, t

**Description:**
A string is a specialized vector whose elements are of type character or a subtype of type character. When used as a type specifier for object creation, string means (vector character).

**Compound Type Specifier Kind:**
Abbreviating.

**Compound Type Specifier Syntax:**
(string [size])

**Compound Type Specifier Arguments:**
size—a non-negative fixnum, or the symbol *.

**Compound Type Specifier Description:**
This denotes the union of all types (array c (size)) for all subtypes c of character; that is, the set of strings of size size.

**See Also:**
Section 16.1 (String Concepts), Section 2.4.5 (Double-Quote), Section 22.1.3.4 (Printing Strings)

---

**base-string**

*Type*

**Supertypes:**
base-string, string, vector, array, sequence, t

**Description:**
The type base-string is equivalent to (vector base-char). The base string representation is the most efficient string representation that can hold an arbitrary sequence of standard characters.

**Compound Type Specifier Kind:**
Abbreviating.

**Compound Type Specifier Syntax:**
(base-string [size])

16–2  Programming Language—Common Lisp
Compound Type Specifier Arguments:

- size—a non-negative fixnum, or the symbol *.

Compound Type Specifier Description:

This is equivalent to the type (vector base-char size); that is, the set of base strings of size size.

**simple-string**

**Type**

Supertypes:

- simple-string, string, vector, simple-array, array, sequence, t

Description:

A *simple string* is a specialized one-dimensional *simple array* whose elements are of type *character* or a subtype of type *character*. When used as a *type specifier* for object creation, *simple-string* means (simple-array character (size)).

Compound Type Specifier Kind:

Abbreviating.

Compound Type Specifier Syntax:

(simple-string [size])

Compound Type Specifier Arguments:

- size—a non-negative fixnum, or the symbol *.

Compound Type Specifier Description:

This denotes the union of all types (simple-array c (size)) for all subtypes c of character; that is, the set of *simple strings* of size size.
simple-base-string

Supertypes:
  simple-base-string, base-string, simple-string, string, vector, simple-array, array, sequence, t

Description:
The type simple-base-string is equivalent to (simple-array base-char (*)).

Compound TypeSpecifier Kind:
  Abbreviating.

Compound TypeSpecifier Syntax:
  (simple-base-string [size])

Compound TypeSpecifier Arguments:
  size—a non-negative fixnum, or the symbol *.

Compound TypeSpecifier Description:
  This is equivalent to the type (simple-array base-char (size)); that is, the set of simple base strings of size size.

simple-string-p

Syntax:
  simple-string-p object → generalized-boolean

Arguments and Values:
  object—an object.
  generalized-boolean—a generalized boolean.

Description:
  Returns true if object is of type simple-string; otherwise, returns false.

Examples:
  (simple-string-p "aaaaaa") → true
  (simple-string-p (make-array 6
    :element-type 'character
    :fill-pointer t)) → false
Notes:

\[(\text{simple-string-p } \text{object}) \equiv (\text{typep } \text{object } \text{simple-string})\]

---

**char, schar**

*Accessor*

**Syntax:**

- `char string index → character`
- `schar string index → character`

\[(\text{setf } (\text{char } \text{string index}) \text{new-character})\]
\[(\text{setf } (\text{schar } \text{string index}) \text{new-character})\]

**Arguments and Values:**

- `string`—for `char`, a string; for `schar`, a simple string.
- `index`—a valid array index for the `string`.
- `character, new-character`—a character.

**Description:**

`char` and `schar` access the element of `string` specified by `index`.

`char` ignores fill pointers when accessing elements.

**Examples:**

```lisp
(setq my-simple-string (make-string 6 :initial-element #\A)) → "AAAAAA"
(schar my-simple-string 4) → #\A
(setf (schar my-simple-string 4) #\B) → #\B
my-simple-string → "AAAABA"
(setq my-filled-string
  (make-array 6 :element-type 'character
    :fill-pointer 5
    :initial-contents my-simple-string))
→ "AAAAB"
(char my-filled-string 4) → #\B
(char my-filled-string 5) → #\A
(setf (char my-filled-string 3) #\C) → #\C
(setf (char my-filled-string 5) #\D) → #\D
(setf (fill-pointer my-filled-string) 6) → 6
my-filled-string → "AAACBD"
```
See Also:
aref, elt, Section 3.2.1 (Compiler Terminology)

Notes:

(char s j) ≡ (aref (the string s) j)

---

string

Function

Syntax:

\[ \text{string } x \rightarrow \text{string} \]

Arguments and Values:

- \( x \) — a string, a symbol, or a character.
- \( \text{string} \) — a string.

Description:

Returns a string described by \( x \); specifically:

- If \( x \) is a string, it is returned.
- If \( x \) is a symbol, its name is returned.
- If \( x \) is a character, then a string containing that one character is returned.
- \( \text{string} \) might perform additional, implementation-defined conversions.

Examples:

\[
\begin{align*}
\text{(string "already a string")} & \rightarrow \text{"already a string"} \\
\text{(string 'elm)} & \rightarrow \text{"ELM"} \\
\text{(string \('#c\') \rightarrow \text{"c"}}
\end{align*}
\]

Exceptional Situations:

In the case where a conversion is defined neither by this specification nor by the implementation, an error of type type-error is signaled.

See Also:
coerce, string (type).

Notes:

coerce can be used to convert a sequence of characters to a string.
string-upcase, string-downcase, string-capitalize, nstring-upcase, nstring-downcase, nstring-capitalize

Function

Syntax:

- string-upcase string &key start end → cased-string
- string-downcase string &key start end → cased-string
- string-capitalize string &key start end → cased-string
- nstring-upcase string &key start end → string
- nstring-downcase string &key start end → string
- nstring-capitalize string &key start end → string

Arguments and Values:

- **string**—a string designator. For nstring-upcase, nstring-downcase, and nstring-capitalize, the string designator must be a string.
- **start, end**—bounding index designators of string. The defaults for start and end are 0 and nil, respectively.
- **cased-string**—a string.

Description:

string-upcase, string-downcase, string-capitalize, nstring-upcase, nstring-downcase, nstring-capitalize change the case of the subsequence of string bounded by start and end as follows:

string-upcase

string-upcase returns a string just like string with all lowercase characters replaced by the corresponding uppercase characters. More precisely, each character of the result string is produced by applying the function char-upcase to the corresponding character of string.

string-downcase

string-downcase is like string-upcase except that all uppercase characters are replaced by the corresponding lowercase characters (using char-downcase).
string-upcase, string-downcase, string-capitalize, ...

string-capitalize

string-capitalize produces a copy of string such that, for every word in the copy, the first character of the “word,” if it has case, is uppercase and any other characters with case in the word are lowercase. For the purposes of string-capitalize, a “word” is defined to be a consecutive subsequence consisting of alphanumeric characters, delimited at each end either by a non-alphanumeric character or by an end of the string.

nstring-upcase, nstring-downcase, nstring-capitalize

nstring-upcase, nstring-downcase, and nstring-capitalize are identical to string-upcase, string-downcase, and string-capitalize respectively except that they modify string.

For string-upcase, string-downcase, and string-capitalize, string is not modified. However, if no characters in string require conversion, the result may be either string or a copy of it, at the implementation’s discretion.

Examples:

```
(string-upcase "abcde") → "ABCDE"
(string-upcase "Dr. Livingston, I presume?")
  → "DR. LIVINGSTON, I PRESUME?"
(string-upcase "Dr. Livingston, I presume?" :start 6 :end 10)
  → "Dr. LiVINGston, I presume?"
(string-downcase "Dr. Livingston, I presume?")
  → "dr. livingston, i presume?"

(string-capitalize "elm 13c arthur;fig don’t") → "Elm 13c Arthur;Fig Don’T"
(string-capitalize " hello ") → " Hello "
(string-capitalize "occlUDeD cASEmenTs FOreSTall iNADVertent DEFenestraTION")
  → "Occluded Casements Forestall Inadvertent Defenestration"
(string-capitalize 'kludgy-hash-search) → "Kludgy-Hash-Search"
(string-capitalize "DON’T!") → "Don’T!" ;not "Don’t!"
(string-capitalize "pipe 13a, foo16c") → "Pipe 13a, Foo16c"

(setq str (copy-seq "0123abcd890a"))
  → "0123abcd890a"
(nstring-downcase str :start 5 :end 7)
  → "0123abcD890a"
(str "0123abcd890a"

Side Effects:

nstring-upcase, nstring-downcase, and nstring-capitalize modify string as appropriate rather than constructing a new string.

See Also:

char-upcase, char-downcase

16–8 Programming Language—Common Lisp
Notes:
The result is always of the same length as \texttt{string}.

\textbf{string-trim, string-left-trim, string-right-trim} \textit{Function}

\textbf{Syntax:}

\begin{verbatim}
string-trim character-bag string → trimmed-string
string-left-trim character-bag string → trimmed-string
string-right-trim character-bag string → trimmed-string
\end{verbatim}

\textbf{Arguments and Values:}

- \texttt{character-bag}—a sequence containing characters.
- \texttt{string}—a string designator.
- \texttt{trimmed-string}—a string.

\textbf{Description:}

\texttt{string-trim} returns a substring of \texttt{string}, with all characters in \texttt{character-bag} stripped off the beginning and end. \texttt{string-left-trim} is similar but strips characters off only the beginning; \texttt{string-right-trim} strips off only the end.

If no characters need to be trimmed from the \texttt{string}, then either \texttt{string} itself or a copy of it may be returned, at the discretion of the implementation.

All of these \textit{functions} observe the fill pointer.

\textbf{Examples:}

\begin{verbatim}
(string-trim "abc" "abcaakaaakabcaaa") → "kaak"
(string-trim '(#\Space #\Tab #\Newline) " garbanzo beans"
" ) → "garbanzo beans"
(string-trim " (*)" " ( *three (silly) words* ) ")
→ "three (silly) words"

(string-left-trim "abc" "labcababc") → "labcababc"
(string-left-trim " (*)" " ( *three (silly) words* ) ")
→ "three (silly) words*"

(string-right-trim " (*)" " ( *three (silly) words* ) ")
→ " ( *three (silly) words"
\end{verbatim}
string=, string/=, string<, string>, string<=, string>=, string-equal, string-not-equal, string-lessp, string-greaterp, string-not-greaterp, string-not-lessp

Function

Syntax:

\[
\text{string} = \text{string1 string2} \text{ &key start1 end1 start2 end2} \rightarrow \text{generalized-boolean}
\]

\[
\text{string}/= \text{string1 string2} \text{ &key start1 end1 start2 end2} \rightarrow \text{mismatch-index}
\]

\[
\text{string} < \text{string1 string2} \text{ &key start1 end1 start2 end2} \rightarrow \text{mismatch-index}
\]

\[
\text{string} > \text{string1 string2} \text{ &key start1 end1 start2 end2} \rightarrow \text{mismatch-index}
\]

\[
\text{string} <= \text{string1 string2} \text{ &key start1 end1 start2 end2} \rightarrow \text{mismatch-index}
\]

\[
\text{string} >= \text{string1 string2} \text{ &key start1 end1 start2 end2} \rightarrow \text{mismatch-index}
\]

\[
\text{string-equal} \text{ string1 string2} \text{ &key start1 end1 start2 end2} \rightarrow \text{generalized-boolean}
\]

\[
\text{string-not-equal} \text{ string1 string2} \text{ &key start1 end1 start2 end2} \rightarrow \text{mismatch-index}
\]

\[
\text{string-lessp} \text{ string1 string2} \text{ &key start1 end1 start2 end2} \rightarrow \text{mismatch-index}
\]

\[
\text{string-greaterp} \text{ string1 string2} \text{ &key start1 end1 start2 end2} \rightarrow \text{mismatch-index}
\]

\[
\text{string-not-greaterp} \text{ string1 string2} \text{ &key start1 end1 start2 end2} \rightarrow \text{mismatch-index}
\]

\[
\text{string-not-lessp} \text{ string1 string2} \text{ &key start1 end1 start2 end2} \rightarrow \text{mismatch-index}
\]

Arguments and Values:

- \text{string1}—\text{a string designator}.
- \text{string2}—\text{a string designator}.
- \text{start1, end1}—\text{bounding index designators of string1}. The defaults for \text{start} and \text{end} are 0 and \text{nil}, respectively.
- \text{start2, end2}—\text{bounding index designators of string2}. The defaults for \text{start} and \text{end} are 0 and \text{nil}, respectively.
- \text{generalized-boolean}—\text{a generalized boolean}.
- \text{mismatch-index}—\text{a bounding index of string1}, or \text{nil}.

Description:

These functions perform lexicographic comparisons on \text{string1} and \text{string2}. \text{string=} and \text{string-equal} are called equality functions; the others are called inequality functions. The compar-
### string=, string/=, string<, string>, string<=, ...

ison operations these functions perform are restricted to the subsequence of \texttt{string1} bounded by \texttt{start1} and \texttt{end1} and to the subsequence of \texttt{string2} bounded by \texttt{start2} and \texttt{end2}.

A string \texttt{a} is equal to a string \texttt{b} if it contains the same number of characters, and the corresponding characters are the same under \texttt{char=} or \texttt{char-equal}, as appropriate.

A string \texttt{a} is less than a string \texttt{b} if in the first position in which they differ the character of \texttt{a} is less than the corresponding character of \texttt{b} according to \texttt{char<} or \texttt{char-lessp} as appropriate, or if string \texttt{a} is a proper prefix of string \texttt{b} (of shorter length and matching in all the characters of \texttt{a}).

The equality functions return a \texttt{generalized boolean} that is \texttt{true} if the strings are equal, or \texttt{false} otherwise.

The inequality functions return a \texttt{mismatch-index} that is \texttt{true} if the strings are not equal, or \texttt{false} otherwise. When the \texttt{mismatch-index} is \texttt{true}, it is an \texttt{integer} representing the first character position at which the two substrings differ, as an offset from the beginning of \texttt{string1}.

The comparison has one of the following results:

\begin{itemize}
\item \texttt{string=} is \texttt{true} if the supplied substrings are of the same length and contain the same characters in corresponding positions; otherwise it is \texttt{false}.
\item \texttt{string/=} is \texttt{true} if the supplied substrings are different; otherwise it is \texttt{false}.
\item \texttt{string-equal} is just like \texttt{string=} except that differences in case are ignored; two characters are considered to be the same if \texttt{char-equal} is \texttt{true} of them.
\item \texttt{string<} is \texttt{true} if substring1 is less than substring2; otherwise it is \texttt{false}.
\item \texttt{string>} is \texttt{true} if substring1 is greater than substring2; otherwise it is \texttt{false}.
\item \texttt{string-lessp}, \texttt{string-greaterp} \texttt{string-lessp} and \texttt{string-greaterp} are exactly like \texttt{string<} and \texttt{string>}, respectively, except that distinctions between uppercase and lowercase letters are ignored. It is as if \texttt{char-lessp} were used instead of \texttt{char<} for comparing characters.
\item \texttt{string<=} is \texttt{true} if substring1 is less than or equal to substring2; otherwise it is \texttt{false}.
\end{itemize}
**string>=**

string>= is true if substring1 is greater than or equal to substring2; otherwise it is false.

**string-not-greaterp, string-not-lessp**

string-not-greaterp and string-not-lessp are exactly like string< and string>=, respectively, except that distinctions between uppercase and lowercase letters are ignored. It is as if char-lessp were used instead of char< for comparing characters.

**Examples:**

- `(string= "foo" "foo")` → true
- `(string= "foo" "Foo")` → false
- `(string= "foo" "bar")` → false
- `(string= "together" "frog" :start1 1 :end1 3 :start2 2)` → true
- `(string=equal "foo" "Foo")` → true
- `(string= "abcd" "01234abcd9012" :start2 5 :end2 9)` → true
- `(string< "aaaa" "aaab")` → 3
- `(string>= "aaaaa" "aaaa")` → 4
- `(string-not-greaterp "Abcde" "abcdE")` → 5
- `(string-lessp "012AAA789" "01aab6" :start1 3 :end1 7 :start2 2 :end2 6)` → 6
- `(string-not-equal "AAAA" "aaaA")` → false

**See Also:**

char=

equal calls string= if applied to two strings.

---

**stringp**

*Function*

**Syntax:**

```
stringp object  → generalized-boolean
```

**Arguments and Values:**

- **object**—an object.

  generalized-boolean—a generalized boolean.

**Description:**

Returns true if object is of type string; otherwise, returns false.

---

16–12 Programming Language—Common Lisp
Examples:

```lisp
(stringp "aaaaaa") → true
(stringp #\a) → false
```

See Also:

typep, string (type)

Notes:

```
(stringp object) ≡ (typep object 'string)
```

### make-string

**Function**

**Syntax:**

```
(make-string size &key initial-element element-type) → string
```

**Arguments and Values:**

- `size`—a valid array dimension.
- `initial-element`—a character. The default is implementation-dependent.
- `element-type`—a type specifier. The default is character.
- `string`—a simple string.

**Description:**

`make-string` returns a simple string of length `size` whose elements have been initialized to `initial-element`.

The `element-type` names the type of the elements of the string; a string is constructed of the most specialized type that can accommodate elements of the given type.

**Examples:**

```
(make-string 10 :initial-element #\5) → "5555555555"
(length (make-string 10)) → 10
```

**Affected By:**

The implementation.
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**.

<table>
<thead>
<tr>
<th>concatenate</th>
<th>length</th>
<th>remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy-seq</td>
<td>map</td>
<td>remove-duplicates</td>
</tr>
<tr>
<td>count</td>
<td>map-into</td>
<td>remove-if</td>
</tr>
<tr>
<td>count-if</td>
<td>merge</td>
<td>remove-if-not</td>
</tr>
<tr>
<td>count-if-not</td>
<td>mismatch</td>
<td>replace</td>
</tr>
<tr>
<td>delete</td>
<td>notany</td>
<td>reverse</td>
</tr>
<tr>
<td>delete-duplicates</td>
<td>notevery</td>
<td>search</td>
</tr>
<tr>
<td>delete-if</td>
<td>nreverse</td>
<td>some</td>
</tr>
<tr>
<td>delete-if-not</td>
<td>ns SUBSTITUTE</td>
<td>sort</td>
</tr>
<tr>
<td>elt</td>
<td>ns SUBSTITUTE-IF</td>
<td>stable-sort</td>
</tr>
<tr>
<td>every</td>
<td>ns SUBSTITUTE-IF-NOT</td>
<td>subseq</td>
</tr>
<tr>
<td>fill</td>
<td>position</td>
<td>substitute</td>
</tr>
<tr>
<td>find</td>
<td>position-if</td>
<td>substitute-if</td>
</tr>
<tr>
<td>find-if</td>
<td>position-if-not</td>
<td>substitute-if-not</td>
</tr>
<tr>
<td>find-if-not</td>
<td>reduce</td>
<td></td>
</tr>
</tbody>
</table>

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 E_i of a sequence S by an operator F listed in Figure 17–2, 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Designator</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjoin</td>
<td>nset-exclusive-or</td>
<td>search</td>
</tr>
<tr>
<td>assoc</td>
<td>nsublis</td>
<td>set-difference</td>
</tr>
<tr>
<td>count</td>
<td>nsubst</td>
<td>set-exclusive-or</td>
</tr>
<tr>
<td>delete</td>
<td>nsubst</td>
<td>subst</td>
</tr>
<tr>
<td>find</td>
<td>nunion</td>
<td>subsetp</td>
</tr>
<tr>
<td>intersection</td>
<td>position</td>
<td>subst</td>
</tr>
<tr>
<td>member</td>
<td>pushnew</td>
<td>substitute</td>
</tr>
<tr>
<td>mismatch</td>
<td>rassoc</td>
<td>tree-equal</td>
</tr>
<tr>
<td>nintersection</td>
<td>remove</td>
<td>union</td>
</tr>
<tr>
<td>nset-difference</td>
<td>remove-duplicates</td>
<td></td>
</tr>
</tbody>
</table>

Figure 17–2. Operators that have Two-Argument Tests to be Satisfied

The object O might not be compared directly to E_i. If a :key argument is provided, it is a designator for a function of one argument to be called with each E_i as an argument, and yielding an object Z_i to be used for comparison. (If there is no :key argument, Z_i is E_i.)

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 Z_i. An E_i is said (or, sometimes, an O and an E_i 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 Z_i. An E_i is said (or, sometimes, an O and an E_i 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 #'eqv 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 Figure 17–3, 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.

<table>
<thead>
<tr>
<th>assoc-if</th>
<th>member-if</th>
<th>rassoc-if</th>
</tr>
</thead>
<tbody>
<tr>
<td>assoc-if-not</td>
<td>member-if-not</td>
<td>rassoc-if-not</td>
</tr>
<tr>
<td>count-if</td>
<td>nsubst-if</td>
<td>remove-if</td>
</tr>
<tr>
<td>count-if-not</td>
<td>nsubst-if-not</td>
<td>remove-if-not</td>
</tr>
<tr>
<td>delete-if</td>
<td>nsubstitute-if</td>
<td>subst-if</td>
</tr>
<tr>
<td>delete-if-not</td>
<td>nsubstitute-if-not</td>
<td>subst-if-not</td>
</tr>
<tr>
<td>find-if</td>
<td>position-if</td>
<td>substitute-if</td>
</tr>
<tr>
<td>find-if-not</td>
<td>position-if-not</td>
<td>substitute-if-not</td>
</tr>
</tbody>
</table>

Figure 17–3. Operators that have One-Argument Tests to be Satisfied
The element \( E_i \) might not be considered directly. If a \( \text{:key} \) argument is provided, it is a designator for a function of one argument to be called with each \( E_i \) as an argument, and yielding an object \( Z_i \) to be used for comparison. (If there is no \( \text{:key} \) argument, \( Z_i \) is \( E_i \).)

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, \( Z_i \). An \( E_i \) 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, \( Z_i \). An \( E_i \) 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

\[
\begin{align*}
\text{(count-if #'zerop '(1 #C(0.0 0.0) 0 0.0d0 0.0s0 3))} & \rightarrow 4 \\
\text{(remove-if-not #'symbolp '(0 1 2 3 4 5 6 7 8 9 A B C D E F))} & \rightarrow (A B C D E F) \\
\text{(remove-if (complement #'symbolp) '(0 1 2 3 4 5 6 7 8 9 A B C D E F))} & \rightarrow (A B C D E F) \\
\text{(count-if #'zerop '("foo" "bar" "baz" "quux") :key #'length)} & \rightarrow 3
\end{align*}
\]
**sequence**

**System Class**

**Class Precedence List:**

sequence, t

**Description:**

*Sequences* are ordered collections of *objects*, called the *elements* of the *sequence*.

The *types* vector and the *type* list are *disjoint subtypes* of *type* sequence, but are not necessarily an *exhaustive partition* of sequence.

When viewing a vector as a sequence, only the *active elements* of that vector are considered elements of the sequence; that is, *sequence* operations respect the *fill pointer* when given sequences represented as vectors.

**copy-seq**

**Function**

**Syntax:**

\[
\text{copy-seq sequence} \rightarrow \text{copied-sequence}
\]

**Arguments and Values:**

- **sequence**—a *proper sequence*.
- **copied-sequence**—a *proper sequence*.

**Description:**

Creates a copy of *sequence*. The *elements* of the new *sequence* are the *same* as the corresponding *elements* of the given *sequence*.

If *sequence* is a vector, the result is a *fresh simple array* of rank one that has the same *actual array element type as sequence*. If *sequence* is a list, the result is a *fresh list*.

**Examples:**

\[
\begin{align*}
\text{setq str "a string"} & \rightarrow \text{"a string"} \\
\text{equalp str (copy-seq str)} & \rightarrow \text{true} \\
\text{eql str (copy-seq str)} & \rightarrow \text{false}
\end{align*}
\]

**Exceptional Situations:**

Should be prepared to signal an error of *type* type-error if *sequence* is not a *proper sequence*.

**See Also:**

copy-list
Notes:

From a functional standpoint, \((\text{copy-seq } x) \equiv (\text{subseq } x 0)\)

However, the programmer intent is typically very different in these two cases.

\[\text{elt} \quad \text{Accessor}\]

Syntax:

\[
\text{elt} \ \text{sequence} \ \text{index} \rightarrow \text{object}\\
\]

\[
(\text{setf} \ (\text{elt} \ \text{sequence} \ \text{index}) \ \text{new-object})
\]

Arguments and Values:

- sequence—a \text{proper sequence}.
- index—a \text{valid sequence index} for sequence.
- object—an object.
- new-object—an object.

Description:

Accesses the element of \text{sequence} specified by \text{index}.

Examples:

\[
(\text{setq str (copy-seq "0123456789"))} \rightarrow "0123456789"\\
(\text{elt str 6}) \rightarrow \#\text{\textbackslash 6}\\
(\text{setf (elt str 0) \#\textbackslash \text{\#})} \rightarrow \#\text{\textbackslash \text{\#}}\\
\text{str} \rightarrow "\#123456789"
\]

Exceptional Situations:

Should be prepared to signal an error of type \text{type-error} if sequence is not a \text{proper sequence}.

Should signal an error of type \text{type-error} if index is not a \text{valid sequence index} for sequence.

See Also:

\text{aref, nth, Section 3.2.1 (Compiler Terminology)}

Notes:

\text{aref} may be used to access vector elements that are beyond the vector’s fill pointer.

17–6 Programming Language—Common Lisp
fill

Syntax:

\[ \text{fill} \ sequence \ item \ \&\&key \ start \ end \ \rightarrow \ sequence \]

Arguments and Values:

- \( \text{sequence} \) — a proper sequence.
- \( \text{item} \) — a sequence.
- \( \text{start}, \text{end} \) — bounding index designators of \( \text{sequence} \). The defaults for \( \text{start} \) and \( \text{end} \) are 0 and \text{nil}, respectively.

Description:

Replaces the elements of \( \text{sequence} \) bounded by \( \text{start} \) and \( \text{end} \) with \( \text{item} \).

Examples:

\[
\begin{align*}
(\text{fill} \ (\text{list} \ 0 \ 1 \ 2 \ 3 \ 4 \ 5) \ '(444)) & \rightarrow ((444) \ (444) \ (444) \ (444) \ (444)) \\
(\text{fill} \ (\text{copy-seq} \ "01234") \ \&\text{e} :\text{start} \ 3) & \rightarrow "012ee" \\
(\text{setq} \ x \ (\text{vector} \ 'a \ 'b \ 'c \ 'd \ 'e)) & \rightarrow #(A \ B \ C \ D \ E) \\
(\text{fill} \ x \ 'z \ :\text{start} \ 1 \ :\text{end} \ 3) & \rightarrow #(A \ Z \ Z \ D \ E) \\
x & \rightarrow #(A \ Z \ Z \ D \ E) \\
(\text{fill} \ x \ 'p) & \rightarrow #(P \ P \ P \ P \ P) \\
x & \rightarrow #(P \ P \ P \ P \ P)
\end{align*}
\]

Side Effects:

\( \text{Sequence} \) is destructively modified.

Exceptional Situations:

Should be prepared to signal an error of \text{type} \text{type-error} if \( \text{sequence} \) is not a proper sequence.

Should signal an error of \text{type} \text{type-error} if \( \text{start} \) is not a non-negative integer. Should signal an error of \text{type} \text{type-error} if \( \text{end} \) is not a non-negative integer or \text{nil}.

See Also:

replace, nsubstitute

Notes:

\[(\text{fill} \ \text{sequence} \ \text{item}) \equiv (\text{nsubstitute-if} \ \text{item} \ (\text{constantly} \ t) \ \text{sequence})\]
make-sequence

Syntax:

```
(make-sequence result-type size &key initial-element) → sequence
```

Arguments and Values:
- `result-type`: a sequence type specifier.
- `size`: a non-negative integer.
- `initial-element`: an object. The default is implementation-dependent.
- `sequence`: a proper sequence.

Description:

Returns a sequence of the type `result-type` and of length `size`, each of the elements of which has been initialized to `initial-element`.

If the `result-type` is a subtype of list, the result will be a list.

If the `result-type` is a subtype of vector, then if the implementation can determine the element type specified for the `result-type`, the element type of the resulting array is the result of upgrading that element type; or, if the implementation can determine that the element type is unspecified (or *), the element type of the resulting array is t; otherwise, an error is signaled.

Examples:

```
(make-sequence 'list 0) → ()
(make-sequence 'string 26 :initial-element #\.)
→ "....................."
(make-sequence '(vector double-float) 2
:initial-element 1d0)
→ #(1.0d0 1.0d0)
```

(make-sequence '(vector * 2) 3) should signal an error
(make-sequence '(vector * 4) 3) should signal an error

Affected By:

The implementation.

Exceptional Situations:

The consequences are unspecified if `initial-element` is not an object which can be stored in the resulting sequence.
An error of type \texttt{type-error} must be signaled if the \texttt{result-type} is neither a \texttt{recognizable subtype} of \texttt{list}, nor a \texttt{recognizable subtype} of \texttt{vector}.

An error of type \texttt{type-error} should be signaled if \texttt{result-type} specifies the number of elements and \texttt{size} is different from that number.

\textbf{See Also:}

\texttt{make-array}, \texttt{make-list}

\textbf{Notes:}

\begin{verbatim}
(make-sequence 'string 5) ≡ (make-string 5)
\end{verbatim}

\section*{subseq \textit{Accessor}}

\textbf{Syntax:}

\begin{verbatim}
subseq sequence start &optional end → subsequence
(setf (subseq sequence start &optional end) new-subsequence)
\end{verbatim}

\textbf{Arguments and Values:}

- \texttt{sequence} — a \texttt{proper sequence}.
- \texttt{start, end} — bounding index designators of \texttt{sequence}. The default for \texttt{end} is \texttt{nil}.
- \texttt{subsequence} — a \texttt{proper sequence}.
- \texttt{new-subsequence} — a \texttt{proper sequence}.

\textbf{Description:}

\texttt{subseq} creates a \texttt{sequence} that is a copy of the subsequence of \texttt{sequence} bounded by \texttt{start} and \texttt{end}.

\texttt{Start} specifies an offset into the original \texttt{sequence} and marks the beginning position of the subsequence. \texttt{end} marks the position following the last element of the subsequence.

\texttt{subseq} always allocates a new \texttt{sequence} for a result; it never shares storage with an old \texttt{sequence}. The result subsequence is always of the same \texttt{type} as \texttt{sequence}.

If \texttt{sequence} is a \texttt{vector}, the result is a fresh \texttt{simple array} of \texttt{rank} one that has the same \texttt{actual array element type} as \texttt{sequence}. If \texttt{sequence} is a \texttt{list}, the result is a fresh \texttt{list}.
**setf** may be used with **subseq** to destructively replace **elements** of a subsequence with **elements** taken from a **sequence** of new values. If the subsequence and the new sequence are not of equal length, the shorter length determines the number of elements that are replaced. The remaining **elements** at the end of the longer sequence are not modified in the operation.

**Examples:**

```
(setq str "012345") → "012345"
(subseq str 2) → "2345"
(subseq str 3 5) → "34"
(setf (subseq str 4) "abc") → "abc"
str → "0123ab"
(setf (subseq str 0 2) "A") → "A"
str → "A123ab"
```

**Exceptional Situations:**

Should be prepared to signal an error of **type** **type-error** if **sequence** is not a **proper sequence**.

Should be prepared to signal an error of **type** **type-error** if **new-subsequence** is not a **proper** **sequence**.

**See Also:**

replace

---

**map**

**Function**

**Syntax:**

```
map result-type function &rest sequences → result
```

**Arguments and Values:**

- **result-type** — a **sequence** type specifier, or nil.
- **function** — a **function designator**. **function** must take as many arguments as there are **sequences**.
- **sequence** — a **proper sequence**.
- **result** — if **result-type** is a **type specifier** other than nil, then a **sequence** of the **type** it denotes; otherwise (if the **result-type** is nil), nil.

**Description:**

Applies **function** to successive sets of arguments in which one argument is obtained from each **sequence**. The **function** is called first on all the elements with index 0, then on all those with index 1, and so on. The **result-type** specifies the **type** of the resulting **sequence**.

---

17–10  Programming Language—Common Lisp
map returns nil if result-type is nil. Otherwise, map returns a sequence such that element \( j \) is the result of applying function to element \( j \) of each of the sequences. The result sequence is as long as the shortest of the sequences. The consequences are undefined if the result of applying function to the successive elements of the sequences cannot be contained in a sequence of the type given by result-type.

If the result-type is a subtype of list, the result will be a list.

If the result-type is a subtype of vector, then if the implementation can determine the element type specified for the result-type, the element type of the resulting array is the result of upgrading that element type; or, if the implementation can determine that the element type is unspecified (or *), the element type of the resulting array is t; otherwise, an error is signaled.

Examples:

\[
\begin{align*}
\text{map 'string #'(lambda (x y) (char "01234567890ABCDEF" (mod (+ x y) 16)))} & \rightarrow "AAAA" \\
\text{(setq seq '("lower" "UPPER" "" "123"))} & \rightarrow ("lower" "UPPER" "" "123") \\
\text{(map nil #'nstring-upcase seq)} & \rightarrow NIL \\
\text{(map 'list #'- '(1 2 3 4))} & \rightarrow (-1 -2 -3 -4) \\
\text{(map 'string #'(lambda (x) (if (oddp x) #\1 #\0)) '(1 2 3 4))} & \rightarrow "1010" \\
\text{(map '(<vector * 4) #'cons "abc" "de") should signal an error}
\end{align*}
\]

Exceptional Situations:

An error of type type-error must be signaled if the result-type is not a recognizable subtype of list, not a recognizable subtype of vector, and not nil.

Should be prepared to signal an error of type type-error if any sequence is not a proper sequence.

An error of type type-error should be signaled if result-type specifies the number of elements and the minimum length of the sequences is different from that number.

See Also:

Section 3.6 (Traversal Rules and Side Effects)
Function

map-into

Syntax:

map-into result-sequence function &rest sequences → result-sequence

Arguments and Values:

result-sequence—a proper sequence.

tooltip

function—a designator for a function of as many arguments as there are sequences.

sequence—a proper sequence.

Description:

Destructively modifies result-sequence to contain the results of applying function to each element in the argument sequences in turn.

result-sequence and each element of sequences can each be either a list or a vector. If result-sequence and each element of sequences are not all the same length, the iteration terminates when the shortest sequence (of any of the sequences or the result-sequence) is exhausted. If result-sequence is a vector with a fill pointer, the fill pointer is ignored when deciding how many iterations to perform, and afterwards the fill pointer is set to the number of times function was applied. If result-sequence is longer than the shortest element of sequences, extra elements at the end of result-sequence are left unchanged. If result-sequence is nil, map-into immediately returns nil, since nil is a sequence of length zero.

If function has side effects, it can count on being called first on all of the elements with index 0, then on all of those numbered 1, and so on.

Examples:

(setq a (list 1 2 3 4) b (list 10 10 10 10)) → (10 10 10 10)
(map-into a #'+' a b) → (11 12 13 14)
a → (11 12 13 14)
b → (10 10 10 10)
(setq k '#(one two three)) → (ONE TWO THREE)
(map-into a #'cons k a) → ((ONE . 11) (TWO . 12) (THREE . 13) 14)
(map-into a #'gensym) → (#G9090 #:G9091 #:G9092 #:G9093)
a → (#G9090 #:G9091 #:G9092 #:G9093)

Exceptional Situations:

Should be prepared to signal an error of type type-error if result-sequence is not a proper sequence. Should be prepared to signal an error of type type-error if sequence is not a proper sequence.

Notes:

map-into differs from map in that it modifies an existing sequence rather than creating a new

17–12 Programming Language—Common Lisp
one. In addition, **map-into** can be called with only two arguments, while **map** requires at least three arguments.

**map-into** could be defined by:

```lisp
(defun map-into (result-sequence function &rest sequences)
  (loop for index below (apply #'min
                              (length result-sequence)
                              (mapcar #'length sequences))
    do (setf (elt result-sequence index)
             (apply function
                    (mapcar #'(lambda (seq) (elt seq index))
                             sequences))))
  result-sequence)
```

---

### reduce

**Function**

**Syntax:**

```
reduce function sequence &key key from-end start end initial-value → result
```

**Arguments and Values:**

- **function**—a designator for a function that might be called with either zero or two arguments.
- **sequence**—a proper sequence.
- **key**—a designator for a function of one argument, or nil.
- **from-end**—a generalized boolean. The default is false.
- **start, end**—bounding index designators of sequence. The defaults for start and end are 0 and nil, respectively.
- **initial-value**—an object.
- **result**—an object.

**Description:**

**reduce** uses a binary operation, **function**, to combine the elements of **sequence** bounded by **start** and **end**.

The **function** must accept as arguments two elements of **sequence** or the results from combining those elements. The **function** must also be able to accept no arguments.

If **key** is supplied, it is used to extract the values to reduce. The **key** function is applied exactly once to each element of **sequence** in the order implied by the reduction order but not to...
the value of `initial-value`, if supplied. The `key` function typically returns part of the `element` of `sequence`. If `key` is not supplied or is `nil`, the `sequence` element itself is used.

The reduction is left-associative, unless `from-end` is `true` in which case it is right-associative.

If `initial-value` is supplied, it is logically placed before the subsequence (or after it if `from-end` is `true`) and included in the reduction operation.

In the normal case, the result of `reduce` is the combined result of `function`'s being applied to successive pairs of `elements` of `sequence`. If the subsequence contains exactly one `element` and no `initial-value` is given, then that `element` is returned and `function` is not called. If the subsequence is empty and an `initial-value` is given, then the `initial-value` is returned and `function` is not called. If the subsequence is empty and no `initial-value` is given, then the `function` is called with zero arguments, and `reduce` returns whatever `function` does. This is the only case where the `function` is called with other than two arguments.

Examples:

```
(reduce #** '(1 2 3 4 5))  →  120
(reduce #'append '(((1) (2))) :initial-value '(i n i t))  →  (I N I T 1 2)
(reduce #'append '(((1) (2))) :from-end t
   :initial-value '(i n i t))  →  (1 2 I N I T)
(reduce #'- '(1 2 3 4))  ≡  (- (- (- 1 2) 3) 4)  →  -8
(reduce #'- '(1 2 3 4) :from-end t) ; Alternating sum. ≡  (- 1 (- 2 (- 3 4)))  →  -2
(reduce #'+ '())  →  0
(reduce #'+ '(0))  →  3
(reduce #'+ '(foo))  →  FOO
(reduce #'list '(1 2 3 4))  →  (((1 2) 3) 4)
(reduce #'list '(1 2 3 4) :from-end t)  →  (1 (2 (3 4)))
(reduce #'list '(1 2 3 4) :initial-value 'foo)  →  (((foo 1) 2) 3) 4)
(reduce #'list '(1 2 3 4)
   :from-end t :initial-value 'foo)  →  (1 (2 (3 (4 foo)))))
```

Exceptional Situations:

Should be prepared to signal an error of `type type-error` if `sequence` is not a `proper sequence`.

See Also:

Section 3.6 (Traversal Rules and Side Effects)
count, count-if, count-if-not

Function

Syntax:

\[
\begin{align*}
\text{count} & \quad \text{item sequence} \& \text{key} \text{ from-end start end key test test-not} \quad \rightarrow n \\
\text{count-if} & \quad \text{predicate sequence} \& \text{key} \text{ from-end start end key} \quad \rightarrow n \\
\text{count-if-not} & \quad \text{predicate sequence} \& \text{key} \text{ from-end start end key} \quad \rightarrow n
\end{align*}
\]

Arguments and Values:

- item—an object.
- sequence—a proper sequence.
- predicate—a designator for a function of one argument that returns a generalized boolean.
- from-end—a generalized boolean. The default is false.
- test—a designator for a function of two arguments that returns a generalized boolean.
- test-not—a designator for a function of two arguments that returns a generalized boolean.
- start, end—bounding index designators of sequence. The defaults for start and end are 0 and nil, respectively.
- key—a designator for a function of one argument, or nil.
- n—a non-negative integer less than or equal to the length of sequence.

Description:

The count, count-if, and count-if-not count and return the number of elements in the sequence bounded by start and end that satisfy the test.

The from-end has no direct effect on the result. However, if from-end is true, the elements of sequence will be supplied as arguments to the test, test-not, and key in reverse order, which may change the side-effects, if any, of those functions.

Examples:

\[
\begin{align*}
\text{(count #\textbackslash{a} "how many A's are there in here?")} & \rightarrow 2 \\
\text{(count-if-not #'oddp '(1) (2) (3) (4)) :key #'car} & \rightarrow 2 \\
\text{(count-if #'upper-case-p "The Crying of Lot 49" :start 4)} & \rightarrow 2
\end{align*}
\]

Exceptional Situations:

Should be prepared to signal an error of type type-error if sequence is not a proper sequence.

See Also:
Section 17.2 (Rules about Test Functions), Section 3.6 (Traversal Rules and Side Effects)

Notes:
The :test-not argument is deprecated.
The function count-if-not is deprecated.

### length

**Function**

**Syntax:**

\[
\text{length sequence } \rightarrow n
\]

**Arguments and Values:**

- `sequence` — a proper sequence.
- `n` — a non-negative integer.

**Description:**

Returns the number of elements in `sequence`.

If `sequence` is a vector with a fill pointer, the active length as specified by the fill pointer is returned.

**Examples:**

\[
\begin{align*}
\text{(length "abc") } & \rightarrow 3 \\
\text{(setq str (make-array '(3) :element-type 'character}
& \text{ :initial-contents "abc"
& \text{ :fill-pointer t})) } & \rightarrow \text{"abc"}
\end{align*}
\]

\[
\begin{align*}
\text{(length str) } & \rightarrow 3 \\
\text{(setf (fill-pointer str) 2) } & \rightarrow 2 \\
\text{(length str) } & \rightarrow 2
\end{align*}
\]

**Exceptional Situations:**

Should be prepared to signal an error of type type-error if `sequence` is not a proper sequence.

**See Also:**

list-length, sequence

---

17–16  Programming Language—Common Lisp
reverse, nreverse

Function

Syntax:

\[\text{reverse} \quad \text{sequence} \rightarrow \text{reversed-sequence}\]
\[\text{nreverse} \quad \text{sequence} \rightarrow \text{reversed-sequence}\]

Arguments and Values:

- \text{sequence} — a proper sequence.
- \text{reversed-sequence} — a sequence.

Description:

\text{reverse} and \text{nreverse} return a new sequence of the same kind as \text{sequence}, containing the same elements, but in reverse order.

\text{reverse} and \text{nreverse} differ in that \text{reverse} always creates and returns a new sequence, whereas \text{nreverse} might modify and return the given \text{sequence}. \text{reverse} never modifies the given \text{sequence}.

For \text{reverse}, if \text{sequence} is a vector, the result is a fresh simple array of rank one that has the same actual array element type as \text{sequence}. If \text{sequence} is a list, the result is a fresh list.

For \text{nreverse}, if \text{sequence} is a vector, the result is a vector that has the same actual array element type as \text{sequence}. If \text{sequence} is a list, the result is a list.

For \text{nreverse}, \text{sequence} might be destroyed and re-used to produce the result. The result might or might not be identical to \text{sequence}. Specifically, when \text{sequence} is a list, \text{nreverse} is permitted to setf any part, \text{car} or \text{cdr}, of any \text{cons} that is part of the list structure of \text{sequence}. When \text{sequence} is a vector, \text{nreverse} is permitted to re-order the elements of \text{sequence} in order to produce the resulting vector.

Examples:

\begin{verbatim}
(setq str "abc") → "abc"
(reverse str) → "cba"
str → "abc"
(setq str (copy-seq str)) → "abc"
(reverse str) → "cba"
str → implementation-dependent
(setq 1 (list 1 2 3)) → (1 2 3)
(nreverse 1) → (3 2 1)
1 → implementation-dependent
\end{verbatim}

Side Effects:

\text{nreverse} might either create a new \text{sequence}, modify the argument \text{sequence}, or both. (\text{reverse} does not modify \text{sequence}.)
Exceptional Situations:
Should be prepared to signal an error of type type-error if sequence is not a proper sequence.

sort, stable-sort
Function

Syntax:
sort sequence predicate &key key  → sorted-sequence
stable-sort sequence predicate &key key  → sorted-sequence

Arguments and Values:
sequence—a proper sequence.
predicate—a designator for a function of two arguments that returns a generalized boolean.
key—a designator for a function of one argument, or nil.
sorted-sequence—a sequence.

Description:
sort and stable-sort destructively sort sequences according to the order determined by the predicate function.

If sequence is a vector, the result is a vector that has the same actual array element type as sequence. If sequence is a list, the result is a list.

sort determines the relationship between two elements by giving keys extracted from the elements to the predicate. The first argument to the predicate function is the part of one element of sequence extracted by the key function (if supplied); the second argument is the part of another element of sequence extracted by the key function (if supplied). Predicate should return true if and only if the first argument is strictly less than the second (in some appropriate sense). If the first argument is greater than or equal to the second (in the appropriate sense), then the predicate should return false.

The argument to the key function is the sequence element. The return value of the key function becomes an argument to predicate. If key is not supplied or nil, the sequence element itself is used. There is no guarantee on the number of times the key will be called.

If the key and predicate always return, then the sorting operation will always terminate, producing a sequence containing the same elements as sequence (that is, the result is a permutation of sequence). This is guaranteed even if the predicate does not really consistently represent a total order (in which case the elements will be scrambled in some unpredictable way, but no element will be lost). If the key consistently returns meaningful keys, and the predicate does reflect some total ordering criterion on those keys, then the elements of the sorted-sequence will be properly sorted according to that ordering.
The sorting operation performed by `sort` is not guaranteed stable. Elements considered equal by the `predicate` might or might not stay in their original order. The `predicate` is assumed to consider two elements `x` and `y` to be equal if `(funcall predicate x y)` and `(funcall predicate y x)` are both `false`. `stable-sort` guarantees stability.

The sorting operation can be destructive in all cases. In the case of a `vector` argument, this is accomplished by permuting the elements in place. In the case of a `list`, the `list` is destructively reordered in the same manner as for `nreverse`.

Examples:

```lisp
(setq tester (copy-seq "lkjashd")) → "lkjashd"
(sort tester #\'char-lessp) → "adhkjsl"
(setq tester (list '(1 2 3) '(4 5 6) '(7 8 9))) → ((1 2 3) (4 5 6) (7 8 9))
(sort tester #\'> :key #\'car) → ((7 8 9) (4 5 6) (1 2 3))
(setq tester (list 1 2 3 4 5 6 7 8 9 0)) → (1 2 3 4 5 6 7 8 9 0)
(stable-sort tester #\'(lambda (x y) (and (oddp x) (evenp y))))
→ (1 3 5 7 9 2 4 6 8 0)
```

```
;; Sequences 17-19
```
find, find-if, find-if-not

Function

Syntax:

find item sequence &key from-end test-not start end key → element
find-if predicate sequence &key from-end start end key → element
find-if-not predicate sequence &key from-end start end key → element

Arguments and Values:

item—an object.
sequence—a proper sequence.
predicate—a designator for a function of one argument that returns a generalized boolean.
from-end—a generalized boolean. The default is false.
test—a designator for a function of two arguments that returns a generalized boolean.
test-not—a designator for a function of two arguments that returns a generalized boolean.
start, end—bounding index designators of sequence. The defaults for start and end are 0 and nil, respectively.
key—a designator for a function of one argument, or nil.
element—an element of the sequence, or nil.

Exceptional Situations:

Should be prepared to signal an error of type type-error if sequence is not a proper sequence.

See Also:
merge, Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects), Section 3.7 (Destructive Operations)

Notes:
If sequence is a vector, the result might or might not be simple, and might or might not be identical to sequence.
Description:

find, find-if, and find-if-not each search for an element of the sequence bounded by start and end that satisfies the predicate predicate or that satisfies the test test or test-not, as appropriate.

If from-end is true, then the result is the rightmost element that satisfies the test.

If the sequence contains an element that satisfies the test, then the leftmost or rightmost sequence element, depending on from-end, is returned; otherwise nil is returned.

Examples:

(find #\d "here are some letters that can be looked at" :test #'char>)
→ #\Space
(find-if #'oddp '(1 2 3 4 5) :end 3 :from-end t) → 3
(find-if-not #'complexp #(3.5 2 #C(1.0 0.0) #C(0.0 1.0))
  :start 2) → NIL

Exceptional Situations:

Should be prepared to signal an error of type type-error if sequence is not a proper sequence.

See Also:

position, Section 17.2 (Rules about Test Functions), Section 3.6 (Traversal Rules and Side Effects)

Notes:

The :test-not argument is deprecated.

The function find-if-not is deprecated.

---

**position, position-if, position-if-not**

*Function*

**Syntax:**

position item sequence &key from-end test test-not start end key → position

position-if predicate sequence &key from-end start end key → position

position-if-not predicate sequence &key from-end start end key → position

**Arguments and Values:**

item—an object.

sequence—a proper sequence.

predicate—a designator for a function of one argument that returns a generalized boolean.
from-end—a generalized boolean. The default is false.

test—a designator for a function of two arguments that returns a generalized boolean.

test-not—a designator for a function of two arguments that returns a generalized boolean.

start, end—bounding index designators of sequence. The defaults for start and end are 0 and nil, respectively.

key—a designator for a function of one argument, or nil.

position—a bounding index of sequence, or nil.

Description:

position, position-if, and position-if-not each search sequence for an element that satisfies the test.

The position returned is the index within sequence of the leftmost (if from-end is true) or of the rightmost (if from-end is false) element that satisfies the test; otherwise nil is returned. The index returned is relative to the left-hand end of the entire sequence, regardless of the value of start, end, or from-end.

Examples:

(position #\a "baobab" :from-end t) → 4
(position-if #'oddp '((1) (2) (3) (4)) :start 1 :key #'car) → 2
(position 595 '()) → NIL
(position-if-not #'integerp '(1 2 3 4 5.0)) → 4

Exceptional Situations:

Should be prepared to signal an error of type type-error if sequence is not a proper sequence.

See Also:

find, Section 3.6 (Traversal Rules and Side Effects)

Notes:

The :test-not argument is deprecated.

The function position-if-not is deprecated.
search

Function

Syntax:

\[
\text{search } \text{sequence-1} \text{ sequence-2 } \&\text{key from-end test test-not} \\
\quad \text{key start1 start2} \\
\quad \text{end1 end2}
\]

→ position

Arguments and Values:

- Sequence-1—a sequence.
- Sequence-2—a sequence.
- from-end—a generalized boolean. The default is false.
- test—a designator for a function of two arguments that returns a generalized boolean.
- test-not—a designator for a function of two arguments that returns a generalized boolean.
- key—a designator for a function of one argument, or nil.
- start1, end1—bounding index designators of sequence-1. The defaults for start1 and end1 are 0 and nil, respectively.
- start2, end2—bounding index designators of sequence-2. The defaults for start2 and end2 are 0 and nil, respectively.
- position—a bounding index of sequence-2, or nil.

Description:

Searches sequence-2 for a subsequence that matches sequence-1.

The implementation may choose to search sequence-2 in any order; there is no guarantee on the number of times the test is made. For example, when start-end is true, the sequence might actually be searched from left to right instead of from right to left (but in either case would return the rightmost matching subsequence). If the search succeeds, search returns the offset into sequence-2 of the first element of the leftmost or rightmost matching subsequence, depending on from-end; otherwise search returns nil.

If from-end is true, the index of the leftmost element of the rightmost matching subsequence is returned.

Examples:

\[
\begin{align*}
\text{(search "dog" "it's a dog's life")} & \rightarrow 7 \\
\text{(search '(0 1) '(2 4 6 1 3 5) :key #'oddp)} & \rightarrow 2
\end{align*}
\]
See Also:
Section 3.6 (Traversal Rules and Side Effects)

Notes:
The :test-not argument is deprecated.

---

**mismatch**

*Function*

**Syntax:**

```
mismatch sequence-1 sequence-2 &key from-end test test-not key start1 start2 end1 end2  → position
```

**Arguments and Values:**

*Sequence-1—a sequence.*

*Sequence-2—a sequence.*

*from-end—a generalized boolean.* The default is `false`.

*test—a designator for a function of two arguments that returns a generalized boolean.*

*test-not—a designator for a function of two arguments that returns a generalized boolean.*

*start1, end1—bounding index designators of sequence-1.* The defaults for `start1` and `end1` are 0 and nil, respectively.

*start2, end2—bounding index designators of sequence-2.* The defaults for `start2` and `end2` are 0 and nil, respectively.

*key—a designator for a function of one argument, or nil.*

*position—a bounding index of sequence-1, or nil.*

**Description:**

The specified subsequences of `sequence-1` and `sequence-2` are compared element-wise.

The `key` argument is used for both the `sequence-1` and the `sequence-2`.

If `sequence-1` and `sequence-2` are of equal length and match in every element, the result is `false`.

Otherwise, the result is a non-negative integer, the index within `sequence-1` of the leftmost or rightmost position, depending on `from-end`, at which the two subsequences fail to match. If one subsequence is shorter than and a matching prefix of the other, the result is the index relative to `sequence-1` beyond the last position tested.
If `from-end` is `true`, then one plus the index of the rightmost position in which the sequences differ is returned. In effect, the subsequences are aligned at their right-hand ends; then, the last elements are compared, the penultimate elements, and so on. The index returned is an index relative to `sequence-1`.

Examples:

- `(mismatch "abcd" "ABCDE" :test #'char-equal) → 4`
- `(mismatch '(3 2 1 2 3) '(1 2 3) :from-end t) → 3`
- `(mismatch '(1 2 3) '(2 3 4) :test-not #'eq :key #'oddp) → NIL`
- `(mismatch '(1 2 3 4 5 6) '(3 4 5 6 7) :start1 2 :end2 4) → NIL`

See Also:

- Section 3.6 (Traversal Rules and Side Effects)

Notes:

The :test-not argument is deprecated.

---

**replace**

*Function*

**Syntax:**

```
replace sequence-1 sequence-2 &key start1 end1 start2 end2 → sequence-1
```

**Arguments and Values:**

- `sequence-1`—a sequence.
- `sequence-2`—a sequence.
- `start1, end1`—bounding index designators of `sequence-1`. The defaults for `start1` and `end1` are 0 and nil, respectively.
- `start2, end2`—bounding index designators of `sequence-2`. The defaults for `start2` and `end2` are 0 and nil, respectively.

**Description:**

Destructively modifies `sequence-1` by replacing the elements of `subsequence-1` bounded by `start1` and `end1` with the elements of `subsequence-2` bounded by `start2` and `end2`.

`Sequence-1` is destructively modified by copying successive elements into it from `sequence-2`. Elements of the subsequence of `sequence-2` bounded by `start2` and `end2` are copied into the subsequence of `sequence-1` bounded by `start1` and `end1`. If these subsequences are not of the same length, then the shorter length determines how many elements are copied; the extra elements near the end of the longer subsequence are not involved in the operation. The number of elements copied can be expressed as:
If `sequence-1` and `sequence-2` are the same object and the region being modified overlaps the region being copied from, then it is as if the entire source region were copied to another place and only then copied back into the target region. However, if `sequence-1` and `sequence-2` are not the same, but the region being modified overlaps the region being copied from (perhaps because of shared list structure or displaced arrays), then after the `replace` operation the subsequence of `sequence-1` being modified will have unpredictable contents. It is an error if the elements of `sequence-2` are not of a type that can be stored into `sequence-1`.

**Examples:**

```
(replace "abcdefghij" "0123456789" :start1 4 :end1 7 :start2 4)
→ "abcd456hij"
```

```
(setq lst "012345678")
→ "012345678"
```

```
(replace lst lst :start1 2 :start2 0)
→ "010123456"
```

```
(lst → "010123456"
```

**Side Effects:**

The `sequence-1` is modified.

**See Also:**

`fill`

---

**substitute, substitute-if, substitute-if-not, nsubstitute, nsubstitute-if, nsubstitute-if-not**

**Function**

**Syntax:**

```
substitute newitem olditem sequence &key from-end test test-not start end count key
→ result-sequence
```

```
substitute-if newitem predicate sequence &key from-end start end count key
→ result-sequence
```

```
substitute-if-not newitem predicate sequence &key from-end start end count key
→ result-sequence
```

```
nsubstitute newitem olditem sequence &key from-end test-not start end count key
→ sequence
```

---

17–26  Programming Language—Common Lisp
substitute, substitute-if, substitute-if-not, …

\[
\text{nsubstitute-if } \text{newitem predicate sequence } \&\text{key from-end start end count key} \\
\quad \to \quad \text{sequence}
\]

\[
\text{nsubstitute-if-not } \text{newitem predicate sequence } \&\text{key from-end start end count key} \\
\quad \to \quad \text{sequence}
\]

Arguments and Values:
- **newitem**—an object.
- **olditem**—an object.
- **sequence**—a proper sequence.
- **predicate**—a designator for a function of one argument that returns a generalized boolean.
- **from-end**—a generalized boolean. The default is false.
- **test**—a designator for a function of two arguments that returns a generalized boolean.
- **test-not**—a designator for a function of two arguments that returns a generalized boolean.
- **start, end**—bounding index designators of sequence. The defaults for start and end are 0 and nil, respectively.
- **count**—an integer or nil. The default is nil.
- **key**—a designator for a function of one argument, or nil.
- **result-sequence**—a sequence.

Description:
- **substitute, substitute-if, and substitute-if-not** return a copy of sequence in which each element that satisfies the test has been replaced with newitem.

\[
\text{nsubstitute, nsubstitute-if, and nsubstitute-if-not} \text{ are like substitute, substitute-if, and substitute-if-not respectively, but they may modify sequence.}
\]

If sequence is a vector, the result is a vector that has the same actual array element type as sequence. If sequence is a list, the result is a list.

**Count**, if supplied, limits the number of elements altered; if more than count elements satisfy the test, then of these elements only the leftmost or rightmost, depending on from-end, are replaced, as many as specified by count. If count is supplied and negative, the behavior is as if zero had been supplied instead. If count is nil, all matching items are affected.

Supplying a from-end of true matters only when the count is provided (and non-nil); in that case, only the rightmost count elements satisfying the test are removed (instead of the leftmost).

**predicate, test, and test-not** might be called more than once for each sequence element, and their side effects can happen in any order.
The result of all these functions is a sequence of the same type as the input sequence that has the same elements except that those in the subsequence bounded by start and end and satisfying the test have been replaced by newitem.

substitute, substitute-if, and substitute-if-not return a sequence which can share with the input sequence or may be identical to the input sequence if no elements need to be changed.

nsubstitute and nsubstitute-if are required to setf any car (if sequence is a list) or aref (if sequence is a vector) of the sequence that is required to be replaced with newitem. If sequence is a list, none of the cdrs of the top-level list can be modified.

Examples:

(substitute #\. \SPACE "0 2 4 6") → "0.2.4.6"
(substitute 9 4 '(1 2 4 1 3 4 5)) → (1 2 9 1 3 9 5)
(substitute 9 4 '(1 2 4 1 3 4 5) :count 1) → (1 2 9 1 3 4 5)
(substitute 9 4 '(1 2 4 1 3 4 5) :count 1 :from-end t) → (1 2 4 1 3 9 5)
(substitute 9 3 '(1 2 4 1 3 4 5) :test #'>) → (9 9 4 9 3 4 5)

(substitute-if 0 #'evenp '((1) (2) (3) (4)) :start 2 :key #'car) → ((1) (2) (3) 0)
(substitute-if 9 #'oddp '(1 2 4 1 3 4 5)) → (9 2 4 9 9 4 9)
(substitute-if 9 #'evenp '(1 2 4 1 3 4 5) :count 1 :from-end t) → (1 2 4 1 3 9 5)

(setq some-things (list 'a 'car 'b 'cdr 'c)) → (A CAR B CDR C)
(nsubstitute-if "function was here" #'fboundp some-things :count 1 :from-end t) → (A CAR B "function was here" C)
some-things → (A CAR B "function was here" C)
(setq alpha-tester (copy-seq "ab ")) → "ab "
(nsubstitute-if-not #\z #'alpha-char-p alpha-tester) → "abz"
alpha-tester → "abz"

Side Effects:

nsubstitute, nsubstitute-if, and nsubstitute-if-not modify sequence.

Exceptional Situations:

Should be prepared to signal an error of type type-error if sequence is not a proper sequence.

See Also:

subst, nsust, Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects)

Notes:

If sequence is a vector, the result might or might not be simple, and might or might not be
identical to sequence.

The :test-not argument is deprecated.

The functions substitute-if-not and nsubstitute-if-not are deprecated.

nsubstitute and nsubstitute-if can be used in for-effect-only positions in code.

Because the side-effecting variants (e.g., nsubstitute) potentially change the path that is being traversed, their effects in the presence of shared or circular structure may vary in surprising ways when compared to their non-side-effecting alternatives. To see this, consider the following side-effect behavior, which might be exhibited by some implementations:

```lisp
(defun test-it (fn)
  (let ((x (cons 'b nil)))
    (rplacd x x)
    (funcall fn 'a 'b x :count 1)))
(test-it #'substitute) → (A . #1=(B . #1#))
(test-it #'nsubstitute) → (A . #1#)
```

### concatenate

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
</table>

**Syntax:**

```lisp
concatenate result-type krest sequences → result-sequence
```

**Arguments and Values:**

- **result-type**—a sequence type specifier.
- **sequences**—a sequence.
- **result-sequence**—a proper sequence of type result-type.

**Description:**

concatenate returns a sequence that contains all the individual elements of all the sequences in the order that they are supplied. The sequence is of type result-type, which must be a subtype of type sequence.

All of the sequences are copied from; the result does not share any structure with any of the sequences. Therefore, if only one sequence is provided and it is of type result-type, concatenate is required to copy sequence rather than simply returning it.

It is an error if any element of the sequences cannot be an element of the sequence result.

If the result-type is a subtype of list, the result will be a list.
If the result-type is a subtype of vector, then if the implementation can determine the element type specified for the result-type, the element type of the resulting array is the result of upgrading that element type; or, if the implementation can determine that the element type is unspecified (or *), the element type of the resulting array is t; otherwise, an error is signaled.

Examples:

(concatenate 'string "all" "together" "now") → "all together now"
(concatenate 'list "ABC" '(d e f) #(1 2 3) #*1011)
→ (#\A #\B #\C D E #\F #\1 #\2 #\3 #\1 #\0 #\1 #\1)
(concatenate 'list) → NIL

(concatenate '(vector * 2) "a" "bc") should signal an error

Exceptional Situations:

An error is signaled if the result-type is neither a recognizable subtype of list, nor a recognizable subtype of vector.

An error of type type-error should be signaled if result-type specifies the number of elements and the sum of sequences is different from that number.

See Also:
append

merge Function

Syntax:

merge result-type sequence-1 sequence-2 predicate &key key → result-sequence

Arguments and Values:

result-type—a sequence type specifier.
sequence-1—a sequence.
sequence-2—a sequence.
predicate—a designator for a function of two arguments that returns a generalized boolean.
key—a designator for a function of one argument, or nil.
result-sequence—a proper sequence of type result-type.
merge

Description:
Destructively merges sequence-1 with sequence-2 according to an order determined by the predicate. merge determines the relationship between two elements by giving keys extracted from the sequence elements to the predicate.

The first argument to the predicate function is an element of sequence-1 as returned by the key (if supplied); the second argument is an element of sequence-2 as returned by the key (if supplied). Predicate should return true if and only if its first argument is strictly less than the second (in some appropriate sense). If the first argument is greater than or equal to the second (in the appropriate sense), then predicate should return false. merge considers two elements x and y to be equal if (funcall predicate x y) and (funcall predicate y x) both yield false.

The argument to the key is the sequence element. Typically, the return value of the key becomes the argument to predicate. If key is not supplied or nil, the sequence element itself is used. The key may be executed more than once for each sequence element, and its side effects may occur in any order.

If key and predicate return, then the merging operation will terminate. The result of merging two sequences x and y is a new sequence of type result-type z, such that the length of z is the sum of the lengths of x and y, and z contains all the elements of x and y. If x1 and x2 are two elements of x, and x1 precedes x2 in x, then x1 precedes x2 in z, and similarly for elements of y. In short, z is an interleaving of x and y.

If x and y were correctly sorted according to the predicate, then z will also be correctly sorted. If x or y is not so sorted, then z will not be sorted, but will nevertheless be an interleaving of x and y.

The merging operation is guaranteed stable; if two or more elements are considered equal by the predicate, then the elements from sequence-1 will precede those from sequence-2 in the result.

sequence-1 and/or sequence-2 may be destroyed.

If the result-type is a subtype of list, the result will be a list.

If the result-type is a subtype of vector, then if the implementation can determine the element type specified for the result-type, the element type of the resulting array is the result of upgrading that element type; or, if the implementation can determine that the element type is unspecified (or *), the element type of the resulting array is t; otherwise, an error is signaled.

Examples:

(setq test1 (list 1 3 4 6 7))
(setq test2 (list 2 5 8))
(merge 'list test1 test2 #'<) → (1 2 3 4 5 6 7 8)
(setq test1 (copy-seq "BOY"))
(setq test2 (copy-seq :nosy"))
(merge 'string test1 test2 #'char-lessp) → "BnOosYy"
(setq test1 (vector ((red . 1) (blue . 4))))
(setq test2 (vector ((yellow . 2) (green . 7))))
(merge 'vector test1 test2 #'< :key #'cdr)
→ #((RED . 1) (YELLOW . 2) (BLUE . 4) (GREEN . 7))

(merge '(vector * 4) '(1 5) '(2 4 6) #'<) should signal an error

Exceptional Situations:
An error must be signaled if the result-type is neither a recognizable subtype of list, nor a recognizable subtype of vector.

An error of type type-error should be signaled if result-type specifies the number of elements and the sum of the lengths of sequence-1 and sequence-2 is different from that number.

See Also:
sort, stable-sort, Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects)

remove, remove-if, remove-if-not, delete, delete-if, delete-if-not

Function

Syntax:
remove item sequence &key from-end test test-not start end count key → result-sequence
remove-if test sequence &key from-end start end count key → result-sequence
remove-if-not test sequence &key from-end start end count key → result-sequence
delete item sequence &key from-end test test-not start end count key → result-sequence
delete-if test sequence &key from-end test-not start end count key → result-sequence
delete-if-not test sequence &key from-end start end count key → result-sequence

Arguments and Values:
item—an object.
sequence—a proper sequence.
test—a designator for a function of one argument that returns a generalized boolean.
from-end—a generalized boolean. The default is false.
**remove, remove-if, remove-if-not, delete, delete-if, ...**

**test**—a designator for a function of two arguments that returns a generalized boolean.

**test-not**—a designator for a function of two arguments that returns a generalized boolean.

**start, end**—bounding index designators of sequence. The defaults for start and end are 0 and nil, respectively.

**count**—an integer or nil. The default is nil.

**key**—a designator for a function of one argument, or nil.

**result-sequence**—a sequence.

**Description:**

**remove, remove-if, and remove-if-not** return a sequence from which the elements that satisfy the test have been removed.

**delete, delete-if, and delete-if-not** are like remove, remove-if, and remove-if-not respectively, but they may modify sequence.

If **sequence** is a vector, the result is a vector that has the same actual array element type as sequence. If **sequence** is a list, the result is a list.

Supplying a from-end of true matters only when the count is provided; in that case only the rightmost count elements satisfying the test are deleted.

Count, if supplied, limits the number of elements removed or deleted; if more than count elements satisfy the test, then of these elements only the leftmost or rightmost, depending on from-end, are deleted or removed, as many as specified by count. If count is supplied and negative, the behavior is as if zero had been supplied instead. If count is nil, all matching items are affected.

For all these functions, elements not removed or deleted occur in the same order in the result as they did in sequence.

**remove, remove-if, remove-if-not** return a sequence of the same type as sequence that has the same elements except that those in the subsequence bounded by start and end and satisfying the test have been removed. This is a non-destructive operation. If any elements need to be removed, the result will be a copy. The result of remove may share with sequence; the result may be identical to the input sequence if no elements need to be removed.

**delete, delete-if, and delete-if-not** return a sequence of the same type as sequence that has the same elements except that those in the subsequence bounded by start and end and satisfying the test have been deleted. Sequence may be destroyed and used to construct the result; however, the result might or might not be identical to sequence.

**delete**, when sequence is a list, is permitted to setf any part, car or cdr, of the top-level list structure in that sequence. When sequence is a vector, delete is permitted to change the dimensions of the vector and to slide its elements into new positions without permuting them to produce the resulting vector.

Sequences 17–33
remove, remove-if, remove-if-not, delete, delete-if, ...

**delete-if** is constrained to behave exactly as follows:

\[
\text{(delete nil sequence :test #'(lambda (ignore item) (funcall test item)) ...)}
\]

Examples:

- \((\text{remove 4 '(1 3 4 5 9)}) \rightarrow (1 3 5 9)\)
- \((\text{remove 4 '(1 2 4 1 3 4 5) :count 1}) \rightarrow (1 2 1 3 4 5)\)
- \((\text{remove 4 '(1 2 4 1 3 4 5) :count 1 :from-end t}) \rightarrow (1 2 4 1 3 5)\)
- \((\text{remove 3 '(1 2 4 1 3 4 5) :test #'>}) \rightarrow (4 3 4 5)\)
- \((\text{setq lst '(list of four elements)}) \rightarrow (\text{LIST OF FOUR ELEMENTS})\)
- \((\text{setq lst2 (copy-seq lst)}) \rightarrow (\text{LIST OF FOUR ELEMENTS})\)
- \((\text{setq lst3 (delete 'four lst)}) \rightarrow (\text{LIST OF ELEMENTS})\)
- \((\text{equal lst lst2}) \rightarrow \text{false}\)
- \((\text{remove-if #'oddp '(1 2 4 1 3 4 5)}) \rightarrow (2 4 4)\)
- \((\text{remove-if #'evenp '(1 2 4 1 3 4 5) :count 1 :from-end t}) \rightarrow (1 2 4 1 3 5)\)
- \((\text{remove-if-not #'evenp '(1 2 3 4 5 6 7 8 9) :count 2 :from-end t}) \rightarrow (1 2 3 4 5 6 8)\)
- \((\text{setq tester (list 1 2 4 1 3 4 5)}) \rightarrow (1 2 4 1 3 4 5)\)
- \((\text{delete 4 tester}) \rightarrow (1 2 1 3 5)\)
- \((\text{delete 4 tester :count 1}) \rightarrow (1 2 1 3 4 5)\)
- \((\text{delete 4 tester :count 1 :from-end t}) \rightarrow (1 2 4 1 3 5)\)
- \((\text{delete 3 tester :test #'>}) \rightarrow (4 3 4 5)\)
- \((\text{eq (cdr foo) (car bar)) \rightarrow T or ...}\)
- \((\text{setq tester (list 1 2 4 1 3 4 5 6)) \rightarrow (1 2 4 1 3 4 5 6)\)
- \((\text{delete-if #'evenp tester}) \rightarrow (1 3 5)\)

\[\text{tester} \rightarrow \text{implementation-dependent}\]

---

17–34 Programming Language—Common Lisp
Side Effects:
   For delete, delete-if, and delete-if-not, sequence may be destroyed and used to construct the result.

Exceptional Situations:
   Should be prepared to signal an error of type type-error if sequence is not a proper sequence.

See Also:
   Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects)

Notes:
   If sequence is a vector, the result might or might not be simple, and might or might not be identical to sequence.

   The :test-not argument is deprecated.

   The functions delete-if-not and remove-if-not are deprecated.

---

remove-duplicates, delete-duplicates  Function

Syntax:

\[
\begin{align*}
\text{remove-duplicates} & \quad \text{sequence} \&\text{key} \quad \text{from-end} \quad \text{test} \quad \text{test-not} \\
& \quad \text{start} \quad \text{end} \quad \text{key} \\
& \quad \rightarrow \quad \text{result-sequence} \\
\text{delete-duplicates} & \quad \text{sequence} \&\text{key} \quad \text{from-end} \quad \text{test} \quad \text{test-not} \\
& \quad \text{start} \quad \text{end} \quad \text{key} \\
& \quad \rightarrow \quad \text{result-sequence}
\end{align*}
\]

Arguments and Values:

sequence—a proper sequence.

from-end—a generalized boolean. The default is false.

test—a designator for a function of two arguments that returns a generalized boolean.

test-not—a designator for a function of two arguments that returns a generalized boolean.

start, end—bounding index designators of sequence. The defaults for start and end are 0 and nil, respectively.

key—a designator for a function of one argument, or nil.
remove-duplicates, delete-duplicates

result-sequence—a sequence.

Description:
remove-duplicates returns a modified copy of sequence from which any element that matches another element occurring in sequence has been removed.

If sequence is a vector, the result is a vector that has the same actual array element type as sequence. If sequence is a list, the result is a list.

delete-duplicates is like remove-duplicates, but delete-duplicates may modify sequence.

The elements of sequence are compared pairwise, and if any two match, then the one occurring earlier in sequence is discarded, unless from-end is true, in which case the one later in sequence is discarded.

remove-duplicates and delete-duplicates return a sequence of the same type as sequence with enough elements removed so that no two of the remaining elements match. The order of the elements remaining in the result is the same as the order in which they appear in sequence.

remove-duplicates returns a sequence that may share with sequence or may be identical to sequence if no elements need to be removed.

delete-duplicates, when sequence is a list, is permitted to setf any part, car or cdr, of the top-level list structure in that sequence. When sequence is a vector, delete-duplicates is permitted to change the dimensions of the vector and to slide its elements into new positions without permuting them to produce the resulting vector.

Examples:

```
(remove-duplicates "aBcDAbCd" :test #'char-equal :from-end t) → "aBcD"
(remove-duplicates '(a b c b d d e)) → (A C B D E)
(remove-duplicates '(a b c b d d e) :from-end t) → (A B C D E)
(remove-duplicates '(((foo #\a) (bar #\%) (baz #\A)) :test #'char-equal :key #'cadr) → ((BAR #\%) (BAZ #\A))
(remove-duplicates '(((foo #\a) (bar #\%) (baz #\A)) :test #'char-equal :key #'cadr :from-end t) → ((FOO #\a) (BAR #\%))
(setq tester (list 0 1 2 3 4 5 6))
(delete-duplicates tester :key #'oddp :start 1 :end 6) → (0 4 5 6)
```

Side Effects:
delete-duplicates might destructively modify sequence.

Exceptional Situations:
Should signal an error of type type-error if sequence is not a proper sequence.

See Also:
Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects)
remove-duplicates, delete-duplicates

Notes:

If sequence is a vector, the result might or might not be simple, and might or might not be identical to sequence.

The :test-not argument is deprecated.

These functions are useful for converting sequence into a canonical form suitable for representing a set.
Programming Language—Common Lisp

18. Hash Tables
18.1 Hash Table Concepts

18.1.1 Hash-Table Operations

Figure 18–1 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:

```lisp
(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 $O_1$ 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 $O_1$, or any object $O_2$ equivalent to $O_1$ under the equivalence test (either before or after the modification), is used as a key in further operations on $H$. The consequences of using $O_1$ as a key are unspecified even if $O_1$ 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.
hash-table

Class Precedence List:
  hash-table, t

Description:
  Hash tables provide a way of mapping any object (a key) to an associated object (a value).

See Also:
  Section 18.1 (Hash Table Concepts), Section 22.1.3.13 (Printing Other Objects)

Notes:
  The intent is that this mapping be implemented by a hashing mechanism, such as that described in Section 6.4 “Hashing” of The Art of Computer Programming, Volume 3 (pp506-549). In spite of this intent, no conforming implementation is required to use any particular technique to implement the mapping.

make-hash-table

Syntax:
  make-hash-table &key test size rehash-size rehash-threshold  →  hash-table

Arguments and Values:
  test—a designator for one of the functions eq, eql, equal, or equalp. The default is eql.
  size—a non-negative integer. The default is implementation-dependent.
  rehash-size—a real of type (or (integer 1 *) (float (1.0) *)). The default is implementation-dependent.
  rehash-threshold—a real of type (real 0 1). The default is implementation-dependent.
  hash-table—a hash table.

Description:
  Creates and returns a new hash table.

  test determines how keys are compared. An object is said to be present in the hash-table if that object is the same under the test as the key for some entry in the hash-table.

  size is a hint to the implementation about how much initial space to allocate in the hash-table. This information, taken together with the rehash-threshold, controls the approximate number of entries which it should be possible to insert before the table has to grow. The actual size might
be rounded up from size to the next ‘good’ size; for example, some implementations might round to the next prime number.

rehash-size specifies a minimum amount to increase the size of the hash-table when it becomes full enough to require rehashing; see rehash-threshold below. If rehash-size is an integer, the expected growth rate for the table is additive and the integer is the number of entries to add; if it is a float, the expected growth rate for the table is multiplicative and the float is the ratio of the new size to the old size. As with size, the actual size of the increase might be rounded up.

rehash-threshold specifies how full the hash-table can get before it must grow. It specifies the maximum desired hash-table occupancy level.

The values of rehash-size and rehash-threshold do not constrain the implementation to use any particular method for computing when and by how much the size of hash-table should be enlarged. Such decisions are implementation-dependent, and these values only hints from the programmer to the implementation, and the implementation is permitted to ignore them.

Examples:

```lisp
(setq table (make-hash-table)) → #<HASH-TABLE EQL 0/120 46142754>
(setf (gethash "one" table) 1) → 1
(gethash "one" table) → NIL, false
(setq table (make-hash-table :test 'equal)) → #<HASH-TABLE EQUAL 0/139 46145547>
(setf (gethash "one" table) 1) → 1
(gethash "one" table) → 1, T
(make-hash-table :rehash-size 1.5 :rehash-threshold 0.7) → #<HASH-TABLE EQL 0/120 46156620>
```

See Also:

gethash, hash-table

---

**hash-table-p**

*Function*

**Syntax:**

hash-table-p object → generalized-boolean

**Arguments and Values:**

object—an object.

generalized-boolean—a generalized boolean.

**Description:**

Returns true if object is of type hash-table; otherwise, returns false.
Examples:

```lisp
(setq table (make-hash-table)) → #<HASH-TABLE EQL 0/120 32511220>
(hash-table-p table) → true
(hash-table-p 37) → false
(hash-table-p '((a . 1) (b . 2))) → false
```

Notes:

```
(hash-table-p object) ≡ (typep object 'hash-table)
```

---

### hash-table-count

**Function**

**Syntax:**

```
hash-table-count hash-table → count
```

**Arguments and Values:**

- `hash-table`—a hash table.
- `count`—a non-negative integer.

**Description:**

Returns the number of entries in the `hash-table`. If `hash-table` has just been created or newly cleared (see `clrhash`) the entry count is 0.

**Examples:**

```
(setq table (make-hash-table)) → #<HASH-TABLE EQL 0/120 32115135>
(hash-table-count table) → 0
(setf (gethash 57 table) "fifty-seven") → "fifty-seven"
(hash-table-count table) → 1
(dotimes (i 100) (setf (gethash i table) i)) → NIL
(hash-table-count table) → 100
```

**Affected By:**

- `clrhash`, `remhash`, `setf` of `gethash`

**See Also:**

- `hash-table-size`

---

18–6 Programming Language—Common Lisp
Notes:
The following relationships are functionally correct, although in practice using `hash-table-count` is probably much faster:

\[
\text{(hash-table-count table)} \equiv \\
\text{(loop for value being the hash-values of table count t)} \equiv \\
\text{(let ((total 0))} \\
\text{\quad (maphash #'(lambda (key value) \\
\text{\quad \quad (declare (ignore key value)) \\
\text{\quad \quad (incf total)) \\
\text{\quad table)} \\
\text{\quad total))}
\]

**hash-table-rehash-size**

*Function*

**Syntax:**

`hash-table-rehash-size hash-table → rehash-size`

**Arguments and Values:**

- `hash-table`—a hash table.
- `rehash-size`—a real of type `(or (integer 1 *) (float (1.0) *))`.

**Description:**

Returns the current rehash size of `hash-table`, suitable for use in a call to `make-hash-table` in order to produce a hash table with state corresponding to the current state of the `hash-table`.

**Examples:**

```
(setq table (make-hash-table :size 100 :rehash-size 1.4))
→ #<HASH-TABLE EQL 0/100 2556371>
(hash-table-rehash-size table) → 1.4
```

**Exceptional Situations:**

Should signal an error of `type type-error` if `hash-table` is not a `hash table`.

**See Also:**

`make-hash-table`, `hash-table-rehash-threshold`

**Notes:**

If the hash table was created with an integer rehash size, the result is an integer, indicating that the rate of growth of the `hash-table` when rehashed is intended to be additive; otherwise, the
result is a float, indicating that the rate of growth of the hash-table when rehashed is intended to
be multiplicative. However, this value is only advice to the implementation; the actual amount by
which the hash-table will grow upon rehash is implementation-dependent.

### hash-table-rehash-threshold

**Function**

**Syntax:**

\[
\text{hash-table-rehash-threshold} \quad \text{hash-table} \rightarrow \text{rehash-threshold}
\]

**Arguments and Values:**

- **hash-table** — a hash table.
- **rehash-threshold** — a real of type (real 0 1).

**Description:**

Returns the current rehash threshold of hash-table, which is suitable for use in a call to
make-hash-table in order to produce a hash table with state corresponding to the current state of
the hash-table.

**Examples:**

\[
\begin{align*}
\text{(setq table (make-hash-table :size 100 :rehash-threshold 0.5))} \\
\rightarrow \#<\text{HASH-TABLE EQL 0/100 2562446}> \\
\text{(hash-table-rehash-threshold table)} \rightarrow 0.5
\end{align*}
\]

**Exceptional Situations:**

Should signal an error of type type-error if hash-table is not a hash table.

**See Also:**

make-hash-table, hash-table-rehash-size
hash-table-size  

Function

Syntax:

\[
\text{hash-table-size hash-table} \rightarrow \text{size}
\]

Arguments and Values:

- \text{hash-table} — a hash table.
- \text{size} — a non-negative integer.

Description:

Returns the current size of \text{hash-table}, which is suitable for use in a call to \text{make-hash-table} in order to produce a hash table with state corresponding to the current state of the \text{hash-table}.

Exceptional Situations:

Should signal an error of type \text{type-error} if \text{hash-table} is not a hash table.

See Also:

- hash-table-count, make-hash-table

hash-table-test  

Function

Syntax:

\[
\text{hash-table-test hash-table} \rightarrow \text{test}
\]

Arguments and Values:

- \text{hash-table} — a hash table.
- \text{test} — a function designator. For the four standardized hash table test functions (see \text{make-hash-table}), the test value returned is always a symbol. If an implementation permits additional tests, it is implementation-dependent whether such tests are returned as function objects or function names.

Description:

Returns the test used for comparing keys in \text{hash-table}.

Exceptional Situations:

Should signal an error of type \text{type-error} if \text{hash-table} is not a hash table.

See Also:

- make-hash-table
gethash  

Syntax:

\[
\text{gethash} \hspace{1mm} \text{key hash-table} \hspace{1mm} \text{&optional default} \hspace{1mm} \rightarrow \hspace{1mm} \text{value, present-p}
\]

(setf (gethash key hash-table &optional default) new-value)

Arguments and Values:

- **key**—an object.
- **hash-table**—a hash table.
- **default**—an object. The default is nil.
- **value**—an object.
- **present-p**—a generalized boolean.

Description:

Value is the object in hash-table whose key is the same as key under the hash-table’s equivalence test. If there is no such entry, value is the default.

Present-p is true if an entry is found; otherwise, it is false.

setf may be used with gethash to modify the value associated with a given key, or to add a new entry. When a gethash form is used as a setf place, any default which is supplied is evaluated according to normal left-to-right evaluation rules, but its value is ignored.

Examples:

(setq table (make-hash-table)) → #<HASH-TABLE EQL 0/120 32206334>

(gethash 1 table) → NIL, false
(gethash 1 table 2) → 2, false
(setf (gethash 1 table) "one") → "one"
(setf (gethash 2 table "two") "two") → "two"
(gethash 1 table) → "one", true
(gethash 2 table) → "two", true
(gethash nil table) → NIL, false
(setf (gethash nil table) nil) → NIL
(gethash nil table) → NIL, true
(defvar *counters* (make-hash-table)) → *COUNTERS*
(gethash 'foo *counters*) → NIL, false
(gethash 'foo *counters* 0) → 0, false
(defmacro how-many (obj) `(values (gethash ,obj *counters* 0))) → HOW-MANY
(defun count-it (obj) (incf (how-many obj))) → COUNT-IT
(dolist (x '(bar foo foo bar bar baz)) (count-it x))
(how-many 'foo) → 2
(how-many 'bar) → 3
(how-many 'quux) → 0

See Also:
remhash

Notes:
The secondary value, present-p, can be used to distinguish the absence of an entry from the presence of an entry that has a value of default.

remhash  Function

Syntax:
remhash key hash-table  → generalized-boolean

Arguments and Values:
key—an object.
hash-table—a hash table.
generalized-boolean—a generalized boolean.

Description:
Removes the entry for key in hash-table, if any. Returns true if there was such an entry, or false otherwise.

Examples:
(setq table (make-hash-table)) → #<HASH-TABLE EQL 0/120 32115666>
(setf (gethash 100 table) "C") → "C"
(gethash 100 table) → "C", true
(remhash 100 table) → true
(gethash 100 table) → NIL, false
(remhash 100 table) → false

Side Effects:
The hash-table is modified.
maphash

Syntax:

maphash function hash-table → nil

Arguments and Values:

function—a designator for a function of two arguments, the key and the value.
hash-table—a hash table.

Description:

Iterates over all entries in the hash-table. For each entry, the function is called with two arguments—the key and the value of that entry.

The consequences are unspecified if any attempt is made to add or remove an entry from the hash-table while a maphash is in progress, with two exceptions: the function can use setf of gethash to change the value part of the entry currently being processed, or it can use remhash to remove that entry.

Examples:

(setq table (make-hash-table)) → #<HASH-TABLE EQL 0/120 32304110>
(dotimes (i 10) (setf (gethash i table) i)) → NIL
(let ((sum-of-squares 0))
  (maphash #'(lambda (key val)
               (let ((square (* val val)))
                 (incf sum-of-squares square)
                 (setf (gethash key table) square)))
    table)
  sum-of-squares) → 285
(hash-table-count table) → 10
(maphash #'(lambda (key val)
               (when (oddp val) (remhash key table)))
    table) → NIL
(hash-table-count table) → 5
(maphash #'(lambda (k v) (print (list k v))) table)
  (0 0)
  (8 64)
  (2 4)
  (6 36)
  (4 16)
  → NIL

Side Effects:

None, other than any which might be done by the function.

18–12 Programming Language—Common Lisp
See Also:
  loop, with-hash-table-iterator, Section 3.6 (Traversal Rules and Side Effects)

with-hash-table-iterator

Macro

Syntax:

with-hash-table-iterator (name hash-table) {declaration}*[form]* → {result}*

Arguments and Values:

name—a name suitable for the first argument to macrolet.
hash-table—a form, evaluated once, that should produce a hash table.
declaration—a declare expression; not evaluated.
forms—an implicit progn.
results—the values returned by forms.

Description:

Within the lexical scope of the body, name is defined via macrolet such that successive invocations of (name) return the items, one by one, from the hash table that is obtained by evaluating hash-table only once.

An invocation (name) returns three values as follows:

1. A generalized boolean that is true if an entry is returned.
2. The key from the hash-table entry.
3. The value from the hash-table entry.

After all entries have been returned by successive invocations of (name), then only one value is returned, namely nil.

It is unspecified what happens if any of the implicit interior state of an iteration is returned outside the dynamic extent of the with-hash-table-iterator form such as by returning some closure over the invocation form.

Any number of invocations of with-hash-table-iterator can be nested, and the body of the innermost one can invoke all of the locally established macros, provided all of those macros have distinct names.

Examples:

The following function should return t on any hash table, and signal an error if the usage of with-hash-table-iterator does not agree with the corresponding usage of maphash.
(defun test-hash-table-iterator (hash-table)
  (let ((all-entries '())
         (generated-entries '())
         (unique (list nil)))
    (maphash #'(lambda (key value) (push (list key value) all-entries))
             hash-table)
    (with-hash-table-iterator (generator-fn hash-table)
      (loop
        (multiple-value-bind (more? key value) (generator-fn)
          (unless more? (return))
          (unless (eql value (gethash key hash-table unique))
            (error "Key ~S not found for value ~S" key value))
          (push (list key value) generated-entries))))
    (unless (= (length all-entries)
               (length generated-entries)
               (length (union all-entries generated-entries
                              :key #'car :test (hash-table-test hash-table))))
      (error "Generated entries and Maphash entries don't correspond"))
    t))

The following could be an acceptable definition of maphash, implemented by
with-hash-table-iterator.

(defun maphash (function hash-table)
  (with-hash-table-iterator (next-entry hash-table)
    (loop (multiple-value-bind (more key value) (next-entry)
           (unless more (return nil))
           (funcall function key value))))

Exceptional Situations:
The consequences are undefined if the local function named name established by
with-hash-table-iterator is called after it has returned false as its primary value.

See Also:
Section 3.6 (Traversal Rules and Side Effects)

clrhash

Function

Syntax:
clrhash hash-table  →  hash-table

Arguments and Values:
hash-table—a hash table.
Description:
Removes all entries from hash-table, and then returns that empty hash table.

Examples:

```lisp
(setq table (make-hash-table)) \rightarrow \#<HASH-TABLE EQL 0/120 32004073>
(dotimes (i 100) (setf (gethash i table) (format nil "~R" i))) \rightarrow NIL
(hash-table-count table) \rightarrow 100
(gethash 57 table) \rightarrow "fifty-seven", true
(clrhash table) \rightarrow \#<HASH-TABLE EQL 0/120 32004073>
(hash-table-count table) \rightarrow 0
(gethash 57 table) \rightarrow NIL, false
```

Side Effects:
The hash-table is modified.

sxhash  Function

Syntax:
```
sxhash object \rightarrow hash-code
```

Arguments and Values:
- object—an object.
- hash-code—a non-negative fixnum.

Description:
sxhash returns a hash code for object.

The manner in which the hash code is computed is implementation-dependent, but subject to certain constraints:

1. (equal x y) implies (= (sxhash x) (sxhash y)).

2. For any two objects, x and y, both of which are bit vectors, characters, conses, numbers, pathnames, strings, or symbols, and which are similar, (sxhash x) and (sxhash y) yield the same mathematical value even if x and y exist in different Lisp images of the same implementation. See Section 3.2.4 (Literal Objects in Compiled Files).

3. The hash-code for an object is always the same within a single session provided that the object is not visibly modified with regard to the equivalence test equal. See Section 18.1.2 (Modifying Hash Table Keys).

Hash Tables 18–15
sxhash

4. The hash-code is intended for hashing. This places no verifiable constraint on a conforming implementation, but the intent is that an implementation should make a good-faith effort to produce hash-codes that are well distributed within the range of non-negative fixnums.

5. Computation of the hash-code must terminate, even if the object contains circularities.

Examples:

(= (sxhash (list 'list "ab")) (sxhash (list 'list "ab"))) → true
(= (sxhash "a") (sxhash (make-string 1 :initial-element #\a))) → true
(let ((r (make-random-state)))
  (= (sxhash r) (sxhash (make-random-state r))))
→ implementation-dependent

Affected By:
The implementation.

Notes:

Many common hashing needs are satisfied by make-hash-table and the related functions on hash tables. sxhash is intended for use where the pre-defined abstractions are insufficient. Its main intent is to allow the user a convenient means of implementing more complicated hashing paradigms than are provided through hash tables.

The hash codes returned by sxhash are not necessarily related to any hashing strategy used by any other function in Common Lisp.

For objects of types that equal compares with eq, item 3 requires that the hash-code be based on some immutable quality of the identity of the object. Another legitimate implementation technique would be to have sxhash assign (and cache) a random hash code for these objects, since there is no requirement that similar but non-eq objects have the same hash code.

Although similarity is defined for symbols in terms of both the symbol’s name and the packages in which the symbol is accessible, item 3 disallows using package information to compute the hash code, since changes to the package status of a symbol are not visible to equal.
Programming Language—Common Lisp

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*.

Figure 19–1 lists some defined names that are applicable to *pathnames*.

<table>
<thead>
<tr>
<th><em>default-pathname-defaults</em></th>
<th>namestring</th>
<th>pathname-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>directory-namestring</td>
<td>open</td>
<td>pathname-type</td>
</tr>
<tr>
<td>enough-namestring</td>
<td>parse-namestring</td>
<td>pathname-version</td>
</tr>
<tr>
<td>file-namestring</td>
<td>pathname</td>
<td>pathnamep</td>
</tr>
<tr>
<td>file-string-length</td>
<td>pathname-device</td>
<td>translate-pathname</td>
</tr>
<tr>
<td>host-namestring</td>
<td>pathname-directory</td>
<td>truename</td>
</tr>
<tr>
<td>make-pathname</td>
<td>pathname-host</td>
<td>user-homedir-pathname</td>
</tr>
<tr>
<td>merge-pathnames</td>
<td>pathname-match-p</td>
<td>wild-pathname-p</td>
</tr>
</tbody>
</table>

Figure 19–1. Pathname Operations

### 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. Figure 19–2 lists the functions relating to pathnames that permit a :case argument:

<table>
<thead>
<tr>
<th>make-pathname</th>
<th>pathname-directory</th>
<th>pathname-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pathname-device</td>
<td>pathname-host</td>
<td>pathname-type</td>
</tr>
</tbody>
</table>

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.

19–4  Programming Language—Common Lisp
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, the value of any pathname component, conforming programs should be prepared for the value to be :unspecific.

When writing, 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.

19.2.2.3.1 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).

19.2.3 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 as the value of a pathname component is substantially larger than the space of possible objects that a conforming program is permitted to write 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 the pathname component, even though portable programs are restricted from writing 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. Figure 19–3 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.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>:wild</td>
<td>Wildcard match of one level of directory structure</td>
</tr>
<tr>
<td>:wild-inferiors</td>
<td>Wildcard match of any number of directory levels</td>
</tr>
<tr>
<td>:up</td>
<td>Go upward in directory structure (semantic)</td>
</tr>
<tr>
<td>:back</td>
<td>Go upward in directory structure (syntactic)</td>
</tr>
</tbody>
</table>

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")
19.2.4.3.1 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.

19.2.4.4 Restrictions on Examining a Pathname Name Component

The name might be a string, :wild, :unspecific, or nil.

19.2.4.5 Restrictions on Examining a Pathname Type Component

The type might be a string, :wild, :unspecific, or nil.

19.2.4.6 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.

19.2.4.7 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 and which permit four-letter type components, they serve to illustrate the basic concept of pathname merging.

```lisp
(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.)

\[ \text{logical-pathname} := [\text{host host-marker}] \\
\quad [\text{relative-directory-marker}] \{ [\text{directory directory-marker}]^* \\
\quad [\text{name}] [\text{type-marker}] [\text{type}] [\text{version-marker}] \text{version} ] \]

\[
\text{host} := \text{word} \\
\text{directory} := \text{word} | \text{wildcard-word} | \text{wild-inferiors-word} \\
\text{name} := \text{word} | \text{wildcard-word} \\
\text{type} := \text{word} | \text{wildcard-word} \\
\text{version} := \text{pos-int} | \text{newest-word} | \text{wildcard-version} \\
\text{host-marker} — a colon. \\
\text{relative-directory-marker} — a semicolon. \\
\text{directory-marker} — a semicolon. \\
\text{type-marker} — a dot. \\
\text{version-marker} — a dot. \\
\text{wild-inferiors-word} — The two character sequence “**” (two asterisks). \\
\text{newest-word} — The six character sequence “newest” or the six character sequence “NEWEST”. \\
\text{wildcard-version} — an asterisk. \\
\text{wildcard-word} — one or more asterisks, uppercase letters, digits, and hyphens, including at least one asterisk, with no two asterisks adjacent. \\
\text{word} — one or more uppercase letters, digits, and hyphens. \\
\text{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*.
pathname 

System Class

Class Precedence List:
pathname, t

Description:
A pathname is a structured object which represents a filename.
There are two kinds of pathnames—physical pathnames and logical pathnames.

logical-pathname 

System Class

Class Precedence List:
logical-pathname, pathname, t

Description:
A pathname that uses a namestring syntax that is implementation-independent, and that has component values that are implementation-independent. Logical pathnames do not refer directly to filenames.

See Also:
Section 20.1 (File System Concepts), Section 2.4.8.14 (Sharpsign P), Section 22.1.3.11 (Printing Pathnames)

pathname 

Function

Syntax:
pathname pathspec → pathname

Arguments and Values:
pathspec—a pathname designator.
pathname—a pathname.

Description:
Returns the pathname denoted by pathspec.
pathname

If the pathspec designator is a stream, the stream can be either open or closed; in both cases, the pathname returned corresponds to the filename used to open the file. pathname returns the same pathname for a file stream after it is closed as it did when it was open.

If the pathspec designator is a file stream created by opening a logical pathname, a logical pathname is returned.

Examples:

;; There is a great degree of variability permitted here. The next
;; several examples are intended to illustrate just a few of the many
;; possibilities. Whether the name is canonicalized to a particular
;; case (either upper or lower) depends on both the file system and the
;; implementation since two different implementations using the same
;; file system might differ on many issues. How information is stored
;; internally (and possibly presented in #S notation) might vary,
;; possibly requiring ‘accessors’ such as PATHNAME-NAME to perform case
;; conversion upon access. The format of a namestring is dependent both
;; on the file system and the implementation since, for example, one
;; implementation might include the host name in a namestring, and
;; another might not. #S notation would generally only be used in a
;; situation where no appropriate namestring could be constructed for use
;; with #P.

(setq p1 (pathname "test")
→ #P"CHOCOLATE:TEST" ; with case canonicalization (e.g., VMS)
or
→ #P"VANILLA:test" ; without case canonicalization (e.g., Unix)
or
→ #P"test"
or
→ #S(PATHNAME :HOST "STRAWBERRY" :NAME "TEST")
or
→ #S(PATHNAME :HOST "BELGIAN-CHOCOLATE" :NAME "test")

(setq p2 (pathname "test")
→ #P"CHOCOLATE:TEST"
or
→ #P"VANILLA:test"
or
→ #P"test"
or
→ #S(PATHNAME :HOST "STRAWBERRY" :NAME "TEST")
or
→ #S(PATHNAME :HOST "BELGIAN-CHOCOLATE" :NAME "test")

(pathnamep p1) → true
(eq p1 (pathname p1)) → true
(eq p1 p2)
→ true
or
false
(with-open-file (stream "test" :direction :output)
  (pathname stream))
→ #P"ORANGE-CHOCOLATE:Gus>test.lisp.newest"

See Also:
pathname, logical-pathname, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as
**make-pathname**  

### Syntax:

```lisp
(make-pathname &key host device directory name type version defaults case) → pathname
```

### Arguments and Values:

- **host**—a valid physical pathname host. Complicated defaulting behavior; see below.
- **device**—a valid pathname device. Complicated defaulting behavior; see below.
- **directory**—a valid pathname directory. Complicated defaulting behavior; see below.
- **name**—a valid pathname name. Complicated defaulting behavior; see below.
- **type**—a valid pathname type. Complicated defaulting behavior; see below.
- **version**—a valid pathname version. Complicated defaulting behavior; see below.
- **defaults**—a pathname designator. The default is a pathname whose host component is the same as the host component of the value of *default-pathname-defaults*, and whose other components are all nil.
- **case**—one of :common or :local. The default is :local.
- **pathname**—a pathname.

### Description:

Constructs and returns a pathname from the supplied keyword arguments.

After the components supplied explicitly by `host`, `device`, `directory`, `name`, `type`, and `version` are filled in, the merging rules used by `merge-pathnames` are used to fill in any unsupplied components from the defaults supplied by `defaults`.

Whenever a pathname is constructed the components may be canonicalized if appropriate. For the explanation of the arguments that can be supplied for each component, see Section 19.2.1 (Pathname Components).

If `case` is supplied, it is treated as described in Section 19.2.2.1.2 (Case in Pathname Components).

The resulting pathname is a logical pathname if and only its host component is a logical host or a string that names a defined logical host.
make-pathname

If the directory is a string, it should be the name of a top level directory, and should not contain any punctuation characters; that is, specifying a string, str, is equivalent to specifying the list (:absolute str). Specifying the symbol :wild is equivalent to specifying the list (:absolute :wild-inferiors), or (:absolute :wild) in a file system that does not support :wild-inferiors.

Examples:

;; Implementation A -- an implementation with access to a single
;; Unix file system. This implementation happens to never display
;; the 'host' information in a namestring, since there is only one host.
(make-pathname :directory '(:absolute "public" "games")
:name "chess" :type "db")
→ #P"/public/games/chess.db"

;; Implementation B -- an implementation with access to one or more
;; VMS file systems. This implementation displays 'host' information
;; in the namestring only when the host is not the local host.
;; It uses a double colon to separate a host name from the host's local
;; file name.
(make-pathname :directory '(:absolute "PUBLIC" "GAMES")
:name "CHESS" :type "DB")
→ #P"SYS$DISK:[PUBLIC.GAMES]CHESS.DB"
(make-pathname :host "BOBBY"
:directory '(:absolute "PUBLIC" "GAMES")
:name "CHESS" :type "DB")
→ #P"BOBBY::SYS$DISK:[PUBLIC.GAMES]CHESS.DB"

;; Implementation C -- an implementation with simultaneous access to
;; multiple file systems from the same Lisp image. In this
;; implementation, there is a convention that any text preceding the
;; first colon in a pathname namestring is a host name.
(dolist (case '(:common :local))
 (dolist (host '("MY-LISP" "MY-VAX" "MY-UNIX"))
   (print (make-pathname :host host :case case
    :directory '(:absolute "PUBLIC" "GAMES")
    :name "CHESS" :type "DB"))))
▷ #P"MY-LISP::public>games>chess.db"
▷ #P"MY-VAX:SYS$DISK:[PUBLIC.GAMES]CHESS.DB"
▷ #P"MY-UNIX:/public/games/chess.db"
▷ #P"MY-LISP::public>games>chess.db"
▷ #P"MY-VAX:SYS$DISK:[PUBLIC.GAMES]CHESS.DB"

19–18  Programming Language—Common Lisp
pathnamep

Function

Syntax:

pathnamep object $\rightarrow$ generalized-boolean

Arguments and Values:

object—an object.
generalized-boolean—a generalized boolean.

Description:

Returns true if object is of type pathname; otherwise, returns false.

Examples:

(setq q "test") $\rightarrow$ "test"
(pathnamep q) $\rightarrow$ false
(setq q (pathname "test"))
$\rightarrow$ #S(PATHNAME :HOST NIL :DEVICE NIL :DIRECTORY NIL :NAME "test" :TYPE NIL :VERSION NIL)
(pathnamep q) $\rightarrow$ true
(setq q (logical-pathname "SYS:SITE;FOO.SYSTEM"))
$\rightarrow$ #P"SYS:SITE;FOO.SYSTEM"
(pathnamep q) $\rightarrow$ true

Notes:

(pathname object) $\equiv$ (typep object 'pathname)
pathname-host, pathname-device, pathname-directory, pathname-name, pathname-type, pathname-version

Function

Syntax:

- pathname-host pathname &key case → host
- pathname-device pathname &key case → device
- pathname-directory pathname &key case → directory
- pathname-name pathname &key case → name
- pathname-type pathname &key case → type
- pathname-version pathname → version

Arguments and Values:

- pathname—a pathname designator.
- case—one of :local or :common. The default is :local.
- host—a valid pathname host.
- device—a valid pathname device.
- directory—a valid pathname directory.
- name—a valid pathname name.
- type—a valid pathname type.
- version—a valid pathname version.

Description:

These functions return the components of pathname.

If the pathname designator is a pathname, it represents the name used to open the file. This may be, but is not required to be, the actual name of the file.

If case is supplied, it is treated as described in Section 19.2.1.2 (Case in Pathname Components).

Examples:

19–20 Programming Language—Common Lisp
pathname-host, pathname-device, ...

(setq q (make-pathname :host "KATHY"
:directory "CHAPMAN"
:name "LOGIN" :type "COM"))
→ #P"KATHY::[CHAPMAN]LOGIN.COM"
(pathname-host q) → "KATHY"
(pathname-name q) → "LOGIN"
(pathname-type q) → "COM"

;; Because namestrings are used, the results shown in the remaining
;; examples are not necessarily the only possible results. Mappings
;; from namestring representation to pathname representation are
;; dependent both on the file system involved and on the implementation
;; (since there may be several implementations which can manipulate the
;; the same file system, and those implementations are not constrained
;; to agree on all details). Consult the documentation for each
;; implementation for specific information on how namestrings are treated
;; that implementation.

;; VMS
(pathname-directory (parse-namestring "[FOO.*.BAR]BAZ.LSP"))
→ (:ABSOLUTE "FOO" "BAR")
(pathname-directory (parse-namestring "[FOO.*.BAR]BAZ.LSP") :case :common)
→ (:ABSOLUTE "FOO" "BAR")

;; Unix
(pathname-directory "foo.l") → NIL
(pathname-device "foo.l") → UNSPECIFIC
(pathname-name "foo.l") → "foo"
(pathname-name "foo.l" :case :local) → "foo"
(pathname-name "foo.l" :case :common) → "FOO"
(pathname-type "foo.l") → "l"
(pathname-type "foo.l" :case :local) → "l"
(pathname-type "foo.l" :case :common) → "L"
(pathname-type "foo") → UNSPECIFIC
(pathname-type "foo" :case :common) → "FOO"
(pathname-type "foo") → "FOO"
(pathname-name "foo" :case :common) → "FOO"
(pathname-directory (parse-namestring "/foo/bar/baz.lisp") :case :local)
→ (:ABSOLUTE "foo" "bar")
(pathname-directory (parse-namestring "/foo/bar/baz.lisp") :case :local)
→ (:ABSOLUTE "foo" "BAR")
(pathname-directory (parse-namestring "/.baz.lisp")
→ (:RELATIVE :UP)
(PATHNAME-DIRECTORY (PARSE-NAMESTRING "/foo/Bar/./Mum/baz")
→ (:ABSOLUTE "foo" "BAR" :UP "Mum")
Affected By:
The implementation and the host file system.

Exceptional Situations:
Should signal an error of type type-error if its first argument is not a pathname.

See Also:
pathname, logical-pathname, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

load-logical-pathname-translations Function

Syntax:
load-logical-pathname-translations host → just-loaded

Arguments and Values:
host—a string.
just-loaded—a generalized boolean.

Description:
Searches for and loads the definition of a logical host named host, if it is not already defined. The specific nature of the search is implementation-defined.
If the `host` is already defined, no attempt to find or load a definition is attempted, and `false` is returned. If the `host` is not already defined, but a definition is successfully found and loaded, `true` is returned. Otherwise, an error is signaled.

Examples:

```lisp
(translate-logical-pathname "hacks:weather;barometer.lisp.newest")
▷ Error: The logical host HACKS is not defined.
(load-logical-pathname-translations "HACKS")
▷ ;; Loading SYS:SITE;HACKS.TRANSLATIONS
▷ ;; Loading done.
   → true
(translate-logical-pathname "hacks:weather;barometer.lisp.newest")
   → #P"HELIUM: [SHARED.HACKS.WEATHER]BAROMETER.LSP;0"
(load-logical-pathname-translations "HACKS")
   → false
```

Exceptional Situations:
If no definition is found, an error of type `error` is signaled.

See Also:
logical-pathname

Notes:
`Logical pathname` definitions will be created not just by `implementors` but also by `programmers`. As such, it is important that the search strategy be documented. For example, an `implementation` might define that the definition of a `host` is to be found in a file called “`host.translations`” in some specifically named directory.

---

**logical-pathname-translations**  
*Accessor*

Syntax:

```lisp
logical-pathname-translations host → translations

(setf (logical-pathname-translations host) new-translations)
```

Arguments and Values:

- `host`—a `logical host designator`.
- `translations, new-translations`—a `list`.
**logical-pathname-translations**

**Description:**
Returns the host’s list of translations. Each translation is a list of at least two elements: from-wildcard and to-wildcard. Any additional elements are implementation-defined. From-wildcard is a logical pathname whose host is host. To-wildcard is a pathname.

(setf (logical-pathname-translations host) translations) sets a logical pathname host’s list of translations. If host is a string that has not been previously used as a logical pathname host, a new logical pathname host is defined; otherwise an existing host’s translations are replaced. logical pathname host names are compared with string-equal.

When setting the translations list, each from-wildcard can be a logical pathname whose host is host or a logical pathname namestring parseable by (parse-namestring string host), where host represents the appropriate object as defined by parse-namestring. Each to-wildcard can be anything coercible to a pathname by (pathname to-wildcard). If to-wildcard coerces to a logical pathname, translate-logical-pathname will perform repeated translation steps when it uses it.

host is either the host component of a logical pathname or a string that has been defined as a logical pathname host name by setf of logical-pathname-translations.

**Examples:**

;;; A very simple example of setting up a logical pathname host. No
;;; translations are necessary to get around file system restrictions, so
;;; all that is necessary is to specify the root of the physical directory
;;; tree that contains the logical file system.
;;; The namestring syntax on the right-hand side is implementation-dependent.
(setf (logical-pathname-translations "foo")
'(('"**;*.**" "MY-LISP:>library>foo>**>"))))

;;; Sample use of that logical pathname. The return value
;;; is implementation-dependent.
(translate-logical-pathname "foo:bar;baz;mum.quux.3")
→ #P"MY-LISP:>library>foo>bar>baz>mum.quux.3"

;;; A more complex example, dividing the files among two file servers
;;; and several different directories. This Unix doesn’t support
;;; :WILD-INFERIORS in the directory, so each directory level must
;;; be translated individually. No file name or type translations
;;; are required except for .MAIL to .MBX.
;;; The namestring syntax on the right-hand side is implementation-dependent.
(setf (logical-pathname-translations "prog")
'(('"RELEASED;*.**" "MY-UNIX:/sys/bin/my-prog/"
 ("RELEASED;*.**" "MY-UNIX:/sys/bin/my-prog/*/"
 ("EXPERIMENTAL;*.**" "MY-UNIX:/usr/Joe/development/prog/"))

19–24  Programming Language—Common Lisp
logical-pathname-translations

("EXPERIMENTAL;DOCUMENTATION;**;*.MAIL" "MY-VAX:SYS$DISK:[JOE.MAIL.PROG...]*.MBX"))

;;;Sample use of that logical pathname. The return value
;;;is implementation-dependent.
(translate-logical-pathname "prog:mail;save;ideas.mail.3")
→ #P"MY-VAX:SYS$DISK:[JOE.MAIL.PROG.SAVE]IDEAS.MBX.3"

;;;Example translations for a program that uses three files main.lisp,
;;;auxiliary.lisp, and documentation.lisp. These translations might be
;;;supplied by a software supplier as examples.

;;;For Unix with long file names
(setf (logical-pathname-translations "prog")
  '(("CODE;*.*.*" "/lib/prog/")
    ("CODE;DOCUMENTATION.*.*.*" "/lib/prog/docum.*")
    ("CODE;*.*.*" "/lib/prog/"))

;;;Sample use of that logical pathname. The return value
;;;is implementation-dependent.
(translate-logical-pathname "prog:code;documentation.lisp")
→ #P"/lib/prog/documentation.lisp"

;;;For Unix with 14-character file names, using .lisp as the type
(setf (logical-pathname-translations "prog")
  '(("CODE;DOCUMENTATION.*.*.*" "/lib/prog/docum.*")
    ("CODE;*.*.*" "/lib/prog/")
    ("**;*.LISP.*" ,(logical-pathname "PROG:**;*.L.*")
      ,(compile-file-pathname (logical-pathname "PROG:**;*.LISP.*"))

Файлennames 19–25
logical-pathname-translations

;;;Sample use of that logical pathname. The return value
;;;is implementation-dependent.
(translate-logical-pathname "prog:code;documentation.lisp")
→ #P"/lib/prog/documentatio.l"

;;;For a Cray with 6 character names and no directories, types, or versions.
(setf (logical-pathname-translations "prog")
  (let ((l '(("MAIN" "PGMN")
           ("AUXILIARY" "PGAUX")
           ("DOCUMENTATION" "PGDOC")))
       (logpath (logical-pathname "prog:code;"))
       (phypath (pathname "XXX")))
    (append
     ;; Translations for source files
     (mapcar #'(lambda (x)
                (let ((log (first x))
                      (phy (second x)))
                  (list (make-pathname :name log
                                         :type "LISP"
                                         :version :wild
                                         :defaults logpath)
                       (make-pathname :name phy
                                      :defaults phypath))))
              l))
     ;; Translations for compiled files
     (mapcar #'(lambda (x)
               (let* ((log (first x))
                      (phy (second x)))
                  (com (compile-file-pathname
                         (make-pathname :name log
                                         :type "LISP"
                                         :version :wild
                                         :defaults logpath)))
                  (setq phy (concatenate 'string phy "B"))
                  (list com
                         (make-pathname :name phy
                                        :defaults phypath))))
              l)))))

19–26  Programming Language—Common Lisp
Sample use of that logical pathname. The return value
is implementation-dependent.

(translate-logical-pathname "prog:code;documentation.lisp")
→ #P"PGDOC"

Exceptional Situations:
If host is incorrectly supplied, an error of type type-error is signaled.

See Also:
logical-pathname, Section 19.1.2 (Pathnames as Filenames)

Notes:
Implementations can define additional functions that operate on logical pathway names, for example to specify additional translation rules or options.

**logical-pathname**

*Function*

Syntax:

```
logical-pathname pathspec  → logical-pathname
```

Arguments and Values:

- **pathspec**—a logical pathname, a logical path name name string, or a stream.
- **logical-pathname**—a logical pathname.

Description:

**logical-pathname** converts **pathspec** to a logical pathname and returns the new logical pathname. If **pathspec** is a logical path name name string, it should contain a host component and its following colon. If **pathspec** is a stream, it should be one for which **pathname** returns a logical pathname.

If **pathspec** is a stream, the stream can be either open or closed. **logical-pathname** returns the same logical pathname after a file is closed as it did when the file was open. It is an error if **pathspec** is a stream that is created with make-two-way-stream, make-echo-stream, make-broadcast-stream, make-concatenated-stream, make-string-input-stream, or make-string-output-stream.

Exceptional Situations:
Signals an error of type type-error if **pathspec** isn’t supplied correctly.

See Also:
logical-pathname, translate-logical-pathname, Section 19.3 (Logical Pathnames)
*default-pathname-defaults*  

**Variable**

**Value Type:**

*a pathname object.*

**Initial Value:**

An *implementation-dependent pathname*, typically in the working directory that was current when Common Lisp was started up.

**Description:**

*a pathname*, used as the default whenever a *function* needs a default *pathname* and one is not supplied.

**Examples:**

```lisp
;; This example illustrates a possible usage for a hypothetical Lisp running on a
;; DEC TUPS-20 file system. Since pathname conventions vary between Lisp
;; implementations and host file system types, it is not possible to provide a
;; general-purpose, conforming example.
*default-pathname-defaults*  →  #P"PS:<FRED>"
(merge-pathnames (make-pathname :name "CALENDAR"))
  →  #P"PS:<FRED>CALENDAR"
(let ((*default-pathname-defaults* (pathname "<MARY>")))
  (merge-pathnames (make-pathname :name "CALENDAR")))
  →  #P"<MARY>CALENDAR"
```

**Affected By:**

The *implementation*.

**namestring, file-namestring, directory-namestring, host-namestring, enough-namestring**  

**Function**

**Syntax:**

- `namestring pathname  →  namestring`
- `file-namestring pathname  →  namestring`
- `directory-namestring pathname  →  namestring`
- `host-namestring pathname  →  namestring`
- `enough-namestring pathname &optional defaults  →  namestring`
Arguments and Values:

pathname—a pathname designator.

defaults—a pathname designator. The default is the value of *default-pathname-defaults*.

namestring—a string or nil.

Description:

These functions convert pathname into a namestring. The name represented by pathname is returned as a namestring in an implementation-dependent canonical form.

namestring returns the full form of pathname.

file-namestring returns just the name, type, and version components of pathname.

directory-namestring returns the directory name portion.

host-namestring returns the host name.

enough-namestring returns an abbreviated namestring that is just sufficient to identify the file named by pathname when considered relative to the defaults. It is required that

\[(\text{merge-pathnames } (\text{enough-namestring } \text{pathname } \text{defaults}) \text{ defaults}) \equiv (\text{merge-pathnames } (\text{parse-namestring } \text{pathname } \text{nil } \text{defaults}) \text{ defaults})\]

in all cases, and the result of enough-namestring is the shortest reasonable string that will satisfy this criterion.

It is not necessarily possible to construct a valid namestring by concatenating some of the three shorter namestrings in some order.

Examples:

```
(namestring "getty")
→ "getty"
(setq q (make-pathname :host "kathy"
 :directory
 (pathname-directory *default-pathname-defaults*)
 :name "getty")
→ #S(PATHNAME :HOST "kathy" :DEVICE NIL :DIRECTORY directory-name
 :NAME "getty" :TYPE NIL :VERSION NIL)
(file-namestring q) → "getty"
(directory-namestring q) → directory-name
(host-namestring q) → "kathy"
```

;;;Using Unix syntax and the wildcard conventions used by the
;;;particular version of Unix on which this example was created:
(namestring
  (translate-pathname "/usr/dmr/hacks/frob.l"
    "*/usr/d*/hacks/backup/*.*")
  "*/usr/d*/backup/hacks/backup-frob.l"
  (namestring
    (translate-pathname "/usr/dmr/hacks/frob.l"
      "*/usr/d*/hacks/fr*.l"
      "*/usr/d*/backup/hacks/backup-*.l")
    "*/usr/dmr/backup/hacks/backup-ob.l"
  ))

;;;This is similar to the above example but uses two different hosts, ;;;U: which is a Unix and V: which is a VMS. Note the translation ;;;of file type and alphabetic case conventions.
  (namestring
    (translate-pathname "U:/usr/dmr/hacks/frob.l"
      "U:/usr/d*/hacks/backup/*.*")
    "V:SYS$DISK:[DMR.BACKUP.HACKS]BACKUP-*.LSP"
    (namestring
      (translate-pathname "U:/usr/dmr/hacks/frob.l"
        "U:/usr/d*/hacks/fr*.l"
        "V:SYS$DISK:[DMR.BACKUP.HACKS]BACKUP-*.LSP")
      "V:SYS$DISK:[DMR.BACKUP.HACKS]BACKUP-OB.LSP"
    ))

See Also:
  truename, merge-pathnames, pathname, logical-pathname, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

---

### parse-namestring Function

**Syntax:**

```lisp
parse-namestring thing &optional host default-pathname &key start end junk-allowed
  → pathname, position
```

**Arguments and Values:**

- `thing`—a string, a pathname, or a stream associated with a file.
- `host`—a valid pathname host, a logical host, or `nil`.
- `default-pathname`—a pathname designator. The default is the value of `*default-pathname-defaults*`.

---

19–30 Programming Language—Common Lisp
parse-namestring

**start**, **end**—bounding index designators of **thing**. The defaults for **start** and **end** are 0 and **nil**, respectively.

**junk-allowed**—a generalized boolean. The default is **false**.

**pathname**—a pathname, or **nil**.

**position**—a bounding index designator for **thing**.

**Description:**

Converts **thing** into a **pathname**.

The **host** supplies a host name with respect to which the parsing occurs.

If **thing** is a stream associated with a file, processing proceeds as if the **pathname** used to open that file had been supplied instead.

If **thing** is a pathname, the **host** and the host component of **thing** are compared. If they match, two values are immediately returned: **thing** and **start**; otherwise (if they do not match), an error is signaled.

Otherwise (if **thing** is a string), **parse-namestring** parses the name of a file within the substring of **thing** bounded by **start** and **end**.

If **thing** is a string then the substring of **thing** bounded by **start** and **end** is parsed into a **pathname** as follows:

- If **host** is a logical host then **thing** is parsed as a logical pathname namestring on the **host**.
- If **host** is **nil** and **thing** is a syntactically valid logical pathname namestring containing an explicit host, then it is parsed as a logical pathname namestring.
- If **host** is **nil**, **default-pathname** is a logical pathname, and **thing** is a syntactically valid logical pathname namestring without an explicit host, then it is parsed as a logical pathname namestring on the host that is the host component of **default-pathname**.
- Otherwise, the parsing of **thing** is implementation-defined.

In the first of these cases, the host portion of the logical pathname namestring and its following colon are optional.

If the host portion of the namestring and **host** are both present and do not match, an error is signaled.

If **junk-allowed** is **true**, then the primary value is the **pathname** parsed or, if no syntactically correct **pathname** was seen, **nil**. If **junk-allowed** is **false**, then the entire substring is scanned, and the primary value is the **pathname** parsed.

In either case, the secondary value is the index into **thing** of the delimiter that terminated the
parse, or the index beyond the substring if the parse terminated at the end of the substring (as will always be the case if junk-allowed is false).

Parsing a null string always succeeds, producing a pathname with all components (except the host) equal to nil.

If thing contains an explicit host name and no explicit device name, then it is implementation-defined whether parse-namestring will supply the standard default device for that host as the device component of the resulting pathname.

Examples:

```
(setq q (parse-namestring "test"))
→ #S(PATHNAME :HOST NIL :DEVICE NIL :DIRECTORY NIL :NAME "test"
      :TYPE NIL :VERSION NIL)
(pathname q) → true
(parse-namestring "test")
→ #S(PATHNAME :HOST NIL :DEVICE NIL :DIRECTORY NIL :NAME "test"
      :TYPE NIL :VERSION NIL), 4
(setq s (open xxx)) → #<Input File Stream...>
(parse-namestring s)
→ #S(PATHNAME :HOST NIL :DEVICE NIL :DIRECTORY NIL :NAME xxx
      :TYPE NIL :VERSION NIL), 0
(parse-namestring "test" nil nil :start 2 :end 4 )
→ #S(PATHNAME ...), 15
(parse-namestring "foo.lisp")
→ #P"foo.lisp"
```

Exceptional Situations:

If junk-allowed is false, an error of type parse-error is signaled if thing does not consist entirely of the representation of a pathname, possibly surrounded on either side by whitespace characters if that is appropriate to the cultural conventions of the implementation.

If host is supplied and not nil, and thing contains a manifest host name, an error of type error is signaled if the hosts do not match.

If thing is a logical pathname namestring and if the host portion of the namestring and host are both present and do not match, an error of type error is signaled.

See Also:

pathname, logical-pathname, Section 20.1 (File System Concepts), Section 19.2.2.2.3 (:UNSPECIFIC as a Component Value), Section 19.1.2 (Pathnames as Filenames)
wild-pathname-p

Function

Syntax:

wild-pathname-p pathname &optional field-key → generalized-boolean

Arguments and Values:

pathname—a pathname designator.

Field-key—one of :host, :device, :directory, :name, :type, :version, or nil.

generalized-boolean—a generalized boolean.

Description:

wild-pathname-p tests pathname for the presence of wildcard components.

If pathname is a pathname (as returned by pathname) it represents the name used to open the file. This may be, but is not required to be, the actual name of the file.

If field-key is not supplied or nil, wild-pathname-p returns true if pathname has any wildcard components, nil if pathname has none. If field-key is non-nil, wild-pathname-p returns true if the indicated component of pathname is a wildcard, nil if the component is not a wildcard.

Examples:

;;;The following examples are not portable. They are written to run
;;;with particular file systems and particular wildcard conventions.
;;;Other implementations will behave differently. These examples are
;;;intended to be illustrative, not to be prescriptive.

(wild-pathname-p (make-pathname :name :wild)) → true
(wild-pathname-p (make-pathname :name :wild) :name) → true
(wild-pathname-p (make-pathname :name :wild) :type) → false
(wild-pathname-p (pathname "s:>foo>***")) → true ;Lisp
(wild-pathname-p (pathname :name "F*O")) → true ;Most places

Exceptional Situations:

If pathname is not a pathname, a string, or a stream associated with a file an error of type type-error is signaled.

See Also:

pathname, logical-pathname, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

Notes:

Not all implementations support wildcards in all fields. See Section 19.2.2.2.2 (:WILD as a
Component Value) and Section 19.2.2.3 (Restrictions on Wildcard Pathnames).

### pathname-match-p

**Function**

**Syntax:**

```
pathname-match-p pathname wildcard → generalized-boolean
```

**Arguments and Values:**

- `pathname`—a pathname designator.
- `wildcard`—a designator for a wild pathname.
- `generalized-boolean`—a generalized boolean.

**Description:**

`pathname-match-p` returns true if `pathname` matches `wildcard`, otherwise `nil`. The matching rules are `implementation-defined` but should be consistent with `directory`. Missing components of `wildcard` default to `:wild`.

It is valid for `pathname` to be a wild `pathname`; a wildcard field in `pathname` only matches a wildcard field in `wildcard` (i.e., `pathname-match-p` is not commutative). It is valid for `wildcard` to be a non-wild `pathname`.

**Exceptional Situations:**

If `pathname` or `wildcard` is not a `pathname`, `string`, or `stream associated with a file` an error of `type `type-error` is signaled.

**See Also:**

`directory`, `pathname`, `logical-pathname`, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)
translate-logical-pathname

Syntax:

\[
\text{translate-logical-pathname} \quad \text{pathname} \quad \& \text{key} \quad \rightarrow \quad \text{physical-pathname}
\]

Arguments and Values:

- **pathname**: a pathname designator, or a logical pathname namestring.
- **physical-pathname**: a physical pathname.

Description:

Translates *pathname* to a physical pathname, which it returns.

If *pathname* is a stream, the stream can be either open or closed. *translate-logical-pathname* returns the same physical pathname after a file is closed as it did when the file was open. It is an error if *pathname* is a stream that is created with *make-two-way-stream*, *make-echo-stream*, *make-broadcast-stream*, *make-concatenated-stream*, *make-string-input-stream*, or *make-string-output-stream*.

If *pathname* is a logical pathname namestring, the host portion of the logical pathname namestring and its following colon are required.

*Pathname* is first coerced to a pathname. If the coerced *pathname* is a physical pathname, it is returned. If the coerced *pathname* is a logical pathname, the first matching translation (according to *pathname-match-p*) of the logical pathname host is applied, as if by calling *translate-pathname*. If the result is a logical pathname, this process is repeated. When the result is finally a physical pathname, it is returned. If no translation matches, an error is signaled.

*translate-logical-pathname* might perform additional translations, typically to provide translation of file types to local naming conventions, to accomodate physical file systems with limited length names, or to deal with special character requirements such as translating hyphens to underscores or uppercase letters to lowercase. Any such additional translations are implementation-defined. Some implementations do no additional translations.

There are no specified keyword arguments for *translate-logical-pathname*, but implementations are permitted to extend it by adding keyword arguments.

Examples:

See *logical-pathname-translations*.

Exceptional Situations:

If *pathname* is incorrectly supplied, an error of type *type-error* is signaled.

If no translation matches, an error of type *file-error* is signaled.
See Also:
logical-pathname, logical-pathname-translations, logical-pathname, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

**translate-pathname**  
*Function*

**Syntax:**
```
translate-pathname source from-wildcard to-wildcard &key
→ translated-pathname
```

**Arguments and Values:**
- `source`—a pathname designator.
- `from-wildcard`—a pathname designator.
- `to-wildcard`—a pathname designator.
- `translated-pathname`—a pathname.

**Description:**
`translate-pathname` translates `source` (that matches `from-wildcard`) into a corresponding pathname that matches `to-wildcard`, and returns the corresponding pathname.

The resulting pathname is `to-wildcard` with each wildcard or missing field replaced by a portion of `source`. A “wildcard field” is a pathname component with a value of `:wild`, a `:wild` element of a list-valued directory component, or an implementation-defined portion of a component, such as the `"*"` in the complex wildcard string `"foo*bar"` that some implementations support. An implementation that adds other wildcard features, such as regular expressions, must define how `translate-pathname` extends to those features. A “missing field” is a pathname component with a value of `nil`.

The portion of `source` that is copied into the resulting pathname is implementation-defined. Typically it is determined by the user interface conventions of the file systems involved. Usually it is the portion of `source` that matches a wildcard field of `from-wildcard` that is in the same position as the wildcard or missing field of `to-wildcard`. If there is no wildcard field in `from-wildcard` at that position, then usually it is the entire corresponding pathname component of `source`, or in the case of a list-valued directory component, the entire corresponding list element.

During the copying of a portion of `source` into the resulting pathname, additional implementation-defined translations of case or file naming conventions might occur, especially when `from-wildcard` and `to-wildcard` are for different hosts.

It is valid for `source` to be a wild pathname; in general this will produce a wild result. It is valid for `from-wildcard` and/or `to-wildcard` to be non-wild pathnames.
translate-pathname

There are no specified keyword arguments for translate-pathname, but implementations are permitted to extend it by adding keyword arguments.

translate-pathname maps customary case in source into customary case in the output pathname.

Examples:

;; The results of the following five forms are all implementation-dependent.
;; The second item in particular is shown with multiple results just to
;; emphasize one of many particular variations which commonly occurs.
(pathname-name (translate-pathname "foobar" "foo*" "*baz")) → "barbaz"
(pathname-name (translate-pathname "foobar" "foo*" "*")) → "foobar"
or "bar"

(pathname-name (translate-pathname "foobar" "*" "foo*")) → "foofoobar"
(pathname-name (translate-pathname "bar" "*" "foo*")) → "barfoo"
(pathname-name (translate-pathname "foobar" "foo*" "baz*")) → "bazbar"

(defun translate-logical-pathname-1 (pathname rules)
  (let ((rule (assoc pathname rules :test #'pathname-match-p)))
    (unless rule (error "No translation rule for ~A" pathname))
    (translate-pathname pathname (first rule) (second rule))))

defun translate-logical-pathname-1 "FOO:CODE;BASIC.LISP"
(("FOO:DOCUMENTATION;" "MY-UNIX:/doc/foo/")
 ("FOO:CODE;" "MY-UNIX:/lib/foo/")
 ("FOO:PATCHES;*:" "MY-UNIX:/lib/foo/patch/*/"))
→ #P"MY-UNIX:/lib/foo/basic.l"

;; This example assumes one particular set of wildcard conventions
;; Not all file systems will run this example exactly as written
(defun rename-files (from to)
  (dolist (file (directory from))
    (rename-file file (translate-pathname file from to))))

(rename-files "/*/usr/me/*-lisp" "/*/dev/her/*-l")
; Renames /usr/me/init.lisp to /dev/her/init.l
(rename-files "/*/usr/me/*-lisp" "/*/sys/pcl/*")
; In some file systems the result might be /sys/pcl/5-may/low.lisp

(rename-files "/*/usr/me/*-lisp" "/*/sys/library/*")
; In some file systems the result might be /sys/library/5-may/low.lisp

(rename-files "/*/usr/me/foo.*" "/*/usr/me2/")
; Renames /usr/me/foo.bar to /usr/me2/foo.bar
(rename-files "/*/usr/joe/*-recipes.text" "/*/usr/jim/cookbook/joe's-*-rec.text")
; Renames /usr/joe/lamb-recipes.text to /usr/jim/cookbook/joe's-lamb-rec.text
Exceptional Situations:

If any of source, from-wildcard, or to-wildcard is not a pathname, a string, or a stream associated with a file an error of type type-error is signaled.

(pathname-match-p source from-wildcard) must be true or an error of type error is signaled.

See Also:

namestring, pathname-host, pathname, logical-pathname, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

Notes:

The exact behavior of translate-pathname cannot be dictated by the Common Lisp language and must be allowed to vary, depending on the user interface conventions of the file systems involved.

The following is an implementation guideline. One file system performs this operation by examining each piece of the three pathnames in turn, where a piece is a pathname component or a list element of a structured component such as a hierarchical directory. Hierarchical directory elements in from-wildcard and to-wildcard are matched by whether they are wildcards, not by depth in the directory hierarchy. If the piece in to-wildcard is present and not wild, it is copied into the result. If the piece in to-wildcard is :wild or nil, the piece in source is copied into the result. Otherwise, the piece in to-wildcard might be a complex wildcard such as "foo*bar" and the piece in from-wildcard should be wild; the portion of the piece in source that matches the wildcard portion of the piece in from-wildcard replaces the wildcard portion of the piece in to-wildcard and the value produced is used in the result.

merge-pathnames

Function

Syntax:

merge-pathnames pathname &optional default-pathname default-version
→ merged-pathname

Arguments and Values:

pathname—a pathname designator.

default-pathname—a pathname designator. The default is the value of *default-pathname-defaults*.

default-version—a valid pathname version. The default is :newest.

merged-pathname—a pathname.
merge-pathnames

Description:

Constructs a pathname from pathname by filling in any unsupplied components with the corresponding values from default-pathname and default-version.

Defaulting of pathname components is done by filling in components taken from another pathname. This is especially useful for cases such as a program that has an input file and an output file. Unspecified components of the output pathname will come from the input pathname, except that the type should not default to the type of the input pathname but rather to the appropriate default type for output from the program; for example, see the function compile-file-pathname.

If no version is supplied, default-version is used. If default-version is nil, the version component will remain unchanged.

If pathname explicitly specifies a host and not a device, and if the host component of default-pathname matches the host component of pathname, then the device is taken from the default file device for that host. If pathname does not specify a host, device, directory, name, or type, each such component is copied from default-pathname. If pathname does not specify a name, then the version, if not provided, will come from default-pathname, just like the other components. If pathname does specify a name, then the version is not affected by default-pathname. If this process leaves the version missing, the default-version is used. If the host’s file name syntax provides a way to input a version without a name or type, the user can let the name and type default but supply a version different from the one in default-pathname.

If pathname is a stream, pathname effectively becomes (pathname pathname). merge-pathnames can be used on either an open or a closed stream.

If pathname is a pathname it represents the name used to open the file. This may be, but is not required to be, the actual name of the file.

merge-pathnames recognizes a logical pathname namestring when default-pathname is a logical pathname, or when the namestring begins with the name of a defined logical host followed by a colon. In the first of these two cases, the host portion of the logical pathname namestring and its following colon are optional.

merge-pathnames returns a logical pathname if and only if its first argument is a logical pathname, or its first argument is a logical pathname namestring with an explicit host, or its first argument does not specify a host and the default-pathname is a logical pathname.

Pathname merging treats a relative directory specially. If (pathname-directory pathname) is a list whose car is :relative, and (pathname-directory default-pathname) is a list, then the merged directory is the value of

```
(append (pathname-directory default-pathname)
        (cdr ;remove :relative from the front
             (pathname-directory pathname)))
```

except that if the resulting list contains a string or :wild immediately followed by
merge-pathnames

:back, both of them are removed. This removal of redundant :back keywords is repeated as many times as possible. If (pathname-directory defaultpathname) is not a list or (pathname-directory pathname) is not a list whose car is :relative, the merged directory is (or (pathname-directory pathname) (pathname-directory defaultpathname))

merge-pathnames maps customary case in pathname into customary case in the output pathname.

Examples:

(merge-pathnames "CMUC::FORMAT" "CMUC::PS:<LISPIO>.FASL") → #P"CMUC::PS:<LISPIO>FORMAT.FASL.0"

See Also:

*default-pathname-defaults*, pathname, logical-pathname, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

Notes:

The net effect is that if just a name is supplied, the host, device, directory, and type will come from defaultpathname, but the version will come from default-version. If nothing or just a directory is supplied, the name, type, and version will come from defaultpathname together.
Programming Language—Common Lisp

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 Chapter 21 (Streams).

Figure 20–1 lists some operators that are applicable to *files* and directories.

<table>
<thead>
<tr>
<th>compile-file</th>
<th>file-length</th>
<th>open</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete-file</td>
<td>file-position</td>
<td>probe-file</td>
</tr>
<tr>
<td>directory</td>
<td>file-write-date</td>
<td>rename-file</td>
</tr>
<tr>
<td>file-author</td>
<td>load</td>
<td>with-open-file</td>
</tr>
</tbody>
</table>

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 noted 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 Figure 20–2 treat *open* and *closed streams* differently.

<table>
<thead>
<tr>
<th>delete-file</th>
<th>file-author</th>
<th>probe-file</th>
</tr>
</thead>
<tbody>
<tr>
<td>directory</td>
<td>file-write-date</td>
<td>truename</td>
</tr>
</tbody>
</table>

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 Figure 20–3. 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.

<table>
<thead>
<tr>
<th>directory</th>
<th>probe-file</th>
<th>truename</th>
</tr>
</thead>
</table>

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.

20–2 Programming Language—Common Lisp
**directory**

**Syntax:**

\[
\text{directory } \text{pathspec} \text{ &key } \rightarrow \text{pathnames}
\]

**Arguments and Values:**

- \textit{pathspec}—a \textit{pathname designator}, which may contain \textit{wild} components.
- \textit{pathnames}—a list of physical \textit{pathnames}.

**Description:**

Determines which, if any, \textit{files} that are present in the file system have names matching \textit{pathspec}, and returns a \textit{fresh list of pathnames} corresponding to the \textit{truenames} of those \textit{files}.

An \textit{implementation} may be extended to accept \textit{implementation-defined} keyword arguments to \textit{directory}.

**Affected By:**

The host computer’s file system.

**Exceptional Situations:**

If the attempt to obtain a directory listing is not successful, an error of \textit{type file-error} is signaled.

**See Also:**

\textit{pathname}, \textit{logical-pathname}, \textit{ensure-directories-exist}, Section 20.1 (File System Concepts), Section 21.1.1.1.2 (Open and Closed Streams), Section 19.1.2 (Pathnames as Filenames)

**Notes:**

If the \textit{pathspec} is not \textit{wild}, the resulting list will contain either zero or one elements.

Common Lisp specifies “\&key” in the argument list to \textit{directory} even though no \textit{standardized} keyword arguments to \textit{directory} are defined. “:\textit{allow-other-keys t}” may be used in \textit{conforming programs} in order to quietly ignore any additional keywords which are passed by the program but not supported by the \textit{implementation}.

---

**probe-file**

**Syntax:**

\[
\text{probe-file } \text{pathspec} \rightarrow \text{truename}
\]

**Arguments and Values:**

---

Files 20–3
**Pathspec**—a _pathname designator_.

**Truename**—a _physical pathname_ or _nil_.

**Description:**

- **probe-file** tests whether a file exists.
- **probe-file** returns _false_ if there is no file named _pathspec_, and otherwise returns the _truename_ of _pathspec_.

If the _pathspec_ designator is an open _stream_, then **probe-file** produces the _truename_ of its associated _file_. If _pathspec_ is a _stream_, whether open or closed, it is coerced to a _pathname_ as if by the function **pathname**.

**Affected By:**

The host computer’s file system.

**Exceptional Situations:**

- An error of _type_ [file-error] is signaled if _pathspec_ is _wild_.
- An error of _type_ [file-error] is signaled if the _file system_ cannot perform the requested operation.

**See Also:**

- _truename_, _open_, _ensure-directories-exist_, _pathname_, _logical-pathname_, Section 20.1 (File System Concepts), Section 21.1.1.1.2 (Open and Closed Streams), Section 19.1.2 (Pathnames as Filenames)

---

**ensure-directories-exist**

**Function**

**Syntax:**

```
ensure-directories-exist pathspec &key verbose → pathspec, created
```

**Arguments and Values:**

- _pathspec_—a _pathname designator_.
- _verbose_—a _generalized boolean_.
- _created_—a _generalized boolean_.

**Description:**

Tests whether the directories containing the specified _file_ actually exist, and attempts to create them if they do not.
If the containing directories do not exist and if \texttt{verbose} is \texttt{true}, then the \textit{implementation} is permitted (but not required) to perform output to \textit{standard output} saying what directories were created. If the containing directories exist, or if \texttt{verbose} is \texttt{false}, this function performs no output.

The \textit{primary value} is the given \textit{pathspec} so that this operation can be straightforwardly composed with other file manipulation expressions. The \textit{secondary value}, \textit{created}, is \texttt{true} if any directories were created.

\textbf{Affected By:}

The host computer’s file system.

\textbf{Exceptional Situations:}

An error of \textit{type file-error} is signaled if the host, device, or directory part of \textit{pathspec} is \textit{wild}.

If the directory creation attempt is not successful, an error of \textit{type file-error} is signaled; if this occurs, it might be the case that none, some, or all of the requested creations have actually occurred within the \textit{file system}.

\textbf{See Also:}

\texttt{probe-file}, \texttt{open}, \texttt{Section 19.1.2 (Pathnames as Filenames)}

---

\textbf{truename \hspace{1cm} Function}

\textbf{Syntax:}

\texttt{truename \ filespec $\rightarrow$ \ truename}

\textbf{Arguments and Values:}

\textit{filespec}—a \textit{pathname designator}.

\textit{truename}—a physical \textit{pathname}.

\textbf{Description:}

\texttt{truename} tries to find the \textit{file} indicated by \textit{filespec} and returns its \textit{truename}. If the \textit{filespec designator} is an open \textit{stream}, its associated \textit{file} is used. If \textit{filespec} is a \textit{stream}, \texttt{truename} can be used whether the \textit{stream} is open or closed. It is permissible for \texttt{truename} to return more specific information after the \textit{stream} is closed than when the \textit{stream} was open. If \textit{filespec} is a \textit{pathname} it represents the name used to open the file. This may be, but is not required to be, the actual name of the file.

\textbf{Examples:}

\begin{verbatim}
;; An example involving version numbers. Note that the precise nature of
;; the truename is implementation-dependent while the file is still open.
(with-open-file (stream ">vistor>test.text.newest")
\end{verbatim}
Exceptional Situations:

An error of type file-error is signaled if an appropriate file cannot be located within the file system for the given filespec, or if the file system cannot perform the requested operation.

An error of type file-error is signaled if pathname is wild.

See Also:

pathname, logical-pathname, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

Notes:

truename may be used to account for any filename translations performed by the file system.

file-author Function

Syntax:

file-author pathspec  →  author

Arguments and Values:

pathspec—a pathname designator.
**Author**—a string or nil.

**Description:**

Returns a string naming the author of the file specified by **pathspec**, or nil if the author’s name cannot be determined.

**Examples:**

```
(with-open-file (stream ">relativity>general.text")
    (file-author s))
→ "albert"
```

**Affected By:**

The host computer’s file system.

Other users of the file named by **pathspec**.

**Exceptional Situations:**

An error of type **file-error** is signaled if **pathspec** is wild.

An error of type **file-error** is signaled if the file system cannot perform the requested operation.

**See Also:**

pathname, logical-pathname, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

---

**file-write-date**

**Syntax:**

```
file-write-date pathspec → date
```

**Arguments and Values:**

- **pathspec**—a pathname designator.
- **date**—a universal time or nil.

**Description:**

Returns a universal time representing the time at which the file specified by **pathspec** was last written (or created), or returns nil if such a time cannot be determined.

**Examples:**

```
(with-open-file (a "noel.text"
    :direction :output :if-exists :error)
```

Files 20–7
(format s ""~&Dear Santa,""2%I was good this year. "
Please leave lots of toys."2%Love, Sue"
"2%attachments: milk, cookies"")
(truename s))
→ #P"CUPID:/susan/noel.text"
(with-open-file (s "noel.text")
(file-write-date s))
→ 2902600800

Affected By:
The host computer’s file system.

Exceptional Situations:
An error of type file-error is signaled if pathspec is wild.
An error of type file-error is signaled if the file system cannot perform the requested operation.

See Also:
Section 25.1.4.2 (Universal Time), Section 19.1.2 (Pathnames as Filenames)

rename-file
Function

Syntax:
rename-file filespec new-name → defaulted-new-name, old-truename, new-truename

Arguments and Values:
filespec—a pathname designator.
new-name—a pathname designator other than a stream.
defaulted-new-name—a pathname
old-truename—a physical pathname.
new-truename—a physical pathname.

Description:
rename-file modifies the file system in such a way that the file indicated by filespec is renamed to defaulted-new-name.

It is an error to specify a filename containing a wild component, for filespec to contain a nil component where the file system does not permit a nil component, or for the result of defaulting missing components of new-name from filespec to contain a nil component where the file system does not permit a nil component.
If new-name is a logical pathname, rename-file returns a logical pathname as its primary value. rename-file returns three values if successful. The primary value, defaulted-new-name, is the resulting name which is composed of new-name with any missing components filled in by performing a merge-pathnames operation using filespec as the defaults. The secondary value, old-truename, is the truename of the file before it was renamed. The tertiary value, new-truename, is the truename of the file after it was renamed.

If the filespec designator is an open stream, then the stream itself and the file associated with it are affected (if the file system permits).

**Examples:**

```lisp
;; An example involving logical pathnames.
(with-open-file (stream "sys:chemistry;lead.text"
  :direction :output :if-exists :error)
  (princ "eureka" stream)
  (values (pathname stream) (truename stream)))
→ #P"SYS:CHEMISTRY;LEAD.TEXT.NEWEST", #P"Q:>sys>chem>lead.text.1"
(rename-file "sys:chemistry;lead.text" "gold.text")
→ #P"SYS:CHEMISTRY;GOLD.TEXT.NEWEST",
  #P"Q:>sys>chem>lead.text.1",
  #P"Q:>sys>chem>gold.text.1"
```

**Exceptional Situations:**

If the renaming operation is not successful, an error of type file-error is signaled.

An error of type file-error might be signaled if filespec is wild.

**See Also:**

truename, pathname, logical-pathname, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

---

**delete-file**  
*Function*

**Syntax:**

```
delete-file filespec → t
```

**Arguments and Values:**

filespec—a pathname designator.

**Description:**

Deletes the file specified by filespec.
If the filespec designator is an open stream, then filespec and the file associated with it are affected (if the file system permits), in which case filespec might be closed immediately, and the deletion might be immediate or delayed until filespec is explicitly closed, depending on the requirements of the file system.

It is implementation-dependent whether an attempt to delete a nonexistent file is considered to be successful.

delete-file returns true if it succeeds, or signals an error of type file-error if it does not.

The consequences are undefined if filespec has a wild component, or if filespec has a nil component and the file system does not permit a nil component.

Examples:

(with-open-file (s "delete-me.text" :direction :output :if-exists :error))
→ NIL
(setq p (probe-file "delete-me.text")) → #'R:\fred\delete-me.text.1
(delete-file p) → T
(probe-file "delete-me.text") → false
(with-open-file (s "delete-me.text" :direction :output :if-exists :error)
(delete-file s))
→ T
(probe-file "delete-me.text") → false

Exceptional Situations:

If the deletion operation is not successful, an error of type file-error is signaled.

An error of type file-error might be signaled if filespec is wild.

See Also:

pathname, logical-pathname, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

file-error

Condition Type

Class Precedence List:

file-error, error, serious-condition, condition, t

Description:

The type file-error consists of error conditions that occur during an attempt to open or close a file, or during some low-level transactions with a file system. The “offending pathname” is initialized by the :pathname initialization argument to make-condition, and is accessed by the function file-error-pathname.
file-error-pathname

Function

Syntax:

file-error-pathname condition → pathspec

Arguments and Values:

condition—a condition of type file-error.

pathspec—a pathname designator.

Description:

Returns the “offending pathname” of a condition of type file-error.

Exceptional Situations:

See Also:

file-error, Chapter 9 (Conditions)
Programming Language—Common Lisp

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; Figure 21–1 provides a list of standardized operations that are potentially useful with any kind of stream.

<table>
<thead>
<tr>
<th>close</th>
<th>stream-element-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>input-stream-p</td>
<td>streamp</td>
</tr>
<tr>
<td>interactive-stream-p</td>
<td>with-open-stream</td>
</tr>
<tr>
<td>output-stream-p</td>
<td></td>
</tr>
</tbody>
</table>

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.

Figure 21–2 shows operators relating to input streams.

<table>
<thead>
<tr>
<th>clear-input</th>
<th>read-byte</th>
<th>read-from-string</th>
</tr>
</thead>
<tbody>
<tr>
<td>listen</td>
<td>read-char</td>
<td>read-line</td>
</tr>
<tr>
<td>peek-char</td>
<td>read-char-no-hang</td>
<td>read-preserving-whitespace</td>
</tr>
<tr>
<td>read</td>
<td>read-delimited-list</td>
<td>unread-char</td>
</tr>
</tbody>
</table>

Figure 21–2. Operators relating to Input Streams.
Figure 21–3 shows operators relating to output streams.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear-output</td>
<td>prin1</td>
</tr>
<tr>
<td>finish-output</td>
<td>prin1-to-string</td>
</tr>
<tr>
<td>force-output</td>
<td>princ</td>
</tr>
<tr>
<td>format</td>
<td>princ-to-string</td>
</tr>
<tr>
<td>fresh-line</td>
<td>print</td>
</tr>
<tr>
<td>pprint</td>
<td>terpri</td>
</tr>
</tbody>
</table>

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 can be used with bidirectional streams. In addition, Figure 21–4 shows a list of operators that relate specifically to bidirectional streams.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y-or-n-p</td>
<td>yes-or-no-p</td>
</tr>
</tbody>
</table>

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”.

- \texttt{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.

\section*{21.1.1.2 Abstract Classifications of Streams}

\subsection*{21.1.1.2.1 File Streams}

Some streams, called file streams, provide access to files. An object of class \texttt{file-stream} is used to represent a file stream.

The basic operation for opening a file is \texttt{open}, which typically returns a file stream (see its dictionary entry for details). The basic operation for closing a stream is \texttt{close}. The macro \texttt{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.

\subsection*{21.1.1.3 Other Subclasses of Stream}

The class \texttt{stream} has a number of subclasses defined by this specification. Figure 21–5 shows some information about these subclasses.
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 Figure 21–6. The consequences are undefined if at any time the value of any of these variables is not an open stream.

<table>
<thead>
<tr>
<th>Glossary Term</th>
<th>Variable Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug I/O</td>
<td><em>debug-io</em></td>
</tr>
<tr>
<td>error output</td>
<td><em>error-output</em></td>
</tr>
<tr>
<td>query I/O</td>
<td><em>query-io</em></td>
</tr>
<tr>
<td>standard input</td>
<td><em>standard-input</em></td>
</tr>
<tr>
<td>standard output</td>
<td><em>standard-output</em></td>
</tr>
<tr>
<td>terminal I/O</td>
<td><em>terminal-io</em></td>
</tr>
<tr>
<td>trace output</td>
<td><em>trace-output</em></td>
</tr>
</tbody>
</table>

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 \texttt{*terminal-io*}.

### 21.1.3 Stream Arguments to Standardized Functions

The operators in Figure 21–7 accept stream arguments that might be either open or closed streams.

<table>
<thead>
<tr>
<th>broadcast-stream-streams</th>
<th>file-author</th>
<th>pathnamelp</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
<td>file-namestring</td>
<td>probe-file</td>
</tr>
<tr>
<td>compile-file</td>
<td>file-write-date</td>
<td>rename-file</td>
</tr>
<tr>
<td>compile-file-pathname</td>
<td>host-namestring</td>
<td>streamp</td>
</tr>
<tr>
<td>concatenated-stream-streams</td>
<td>load</td>
<td>synonym-stream-symbol</td>
</tr>
<tr>
<td>delete-file</td>
<td>logical-pathname</td>
<td>translate-logical-pathname</td>
</tr>
<tr>
<td>directory</td>
<td>merge-pathnames</td>
<td>translate-pathname</td>
</tr>
<tr>
<td>directory-namestring</td>
<td>namestring</td>
<td>truename</td>
</tr>
<tr>
<td>dribble</td>
<td>open</td>
<td>two-way-stream-input-stream</td>
</tr>
<tr>
<td>echo-stream-input-stream</td>
<td>open-stream-p</td>
<td>two-way-stream-output-stream</td>
</tr>
<tr>
<td>echo-stream-ouput-stream</td>
<td>parse-namestring</td>
<td>wild-pathname-p</td>
</tr>
<tr>
<td>ed</td>
<td>pathname</td>
<td>with-open-file</td>
</tr>
<tr>
<td>enough-namestring</td>
<td>pathnamematch-p</td>
<td></td>
</tr>
</tbody>
</table>

Figure 21–7. Operators that accept either Open or Closed Streams

The operators in Figure 21–8 accept stream arguments that must be open streams.

<table>
<thead>
<tr>
<th>clear-input</th>
<th>output-stream-p</th>
<th>read-char-no-hang</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear-output</td>
<td>peek-char</td>
<td>read-delimited-list</td>
</tr>
<tr>
<td>file-length</td>
<td>pprint</td>
<td>read-line</td>
</tr>
<tr>
<td>file-position</td>
<td>pprint-fill</td>
<td>read-preserving-whitespace</td>
</tr>
<tr>
<td>file-string-length</td>
<td>pprint-indent</td>
<td>stream-element-type</td>
</tr>
<tr>
<td>finish-output</td>
<td>pprint-linear</td>
<td>stream-external-format</td>
</tr>
<tr>
<td>force-output</td>
<td>pprint-logical-block</td>
<td>terpri</td>
</tr>
<tr>
<td>format</td>
<td>pprint-newline</td>
<td>unread-char</td>
</tr>
<tr>
<td>fresh-line</td>
<td>pprint-tab</td>
<td>with-open-stream</td>
</tr>
<tr>
<td>get-output-stream-string</td>
<td>pprint-tabular</td>
<td>write</td>
</tr>
<tr>
<td>input-stream-p</td>
<td>prin1</td>
<td>write-byte</td>
</tr>
<tr>
<td>interactive-stream-p</td>
<td>princ</td>
<td>write-char</td>
</tr>
<tr>
<td>listen</td>
<td>print</td>
<td>write-line</td>
</tr>
<tr>
<td>make-broadcast-stream</td>
<td>print-object</td>
<td>write-string</td>
</tr>
<tr>
<td>make-concatenated-stream</td>
<td>print-unreadable-object</td>
<td>y-or-n-p</td>
</tr>
<tr>
<td>make-echo-stream</td>
<td>read</td>
<td>yes-or-no-p</td>
</tr>
<tr>
<td>make-synonym-stream</td>
<td>read-byte</td>
<td></td>
</tr>
<tr>
<td>make-two-way-stream</td>
<td>read-char</td>
<td></td>
</tr>
</tbody>
</table>

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.
stream

Class Precedence List:
stream, t

Description:
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.

For more complete information, see Section 21.1 (Stream Concepts).

See Also:
Section 21.1 (Stream Concepts), Section 22.1.3.13 (Printing Other Objects), Chapter 22 (Printer), Chapter 23 (Reader)

broadcast-stream

Class Precedence List:
broadcast-stream, stream, t

Description:
A broadcast stream is an output stream which has associated with it a set of zero or more output streams such that any output sent to the broadcast stream gets passed on as output to each of the associated output streams. (If a broadcast stream has no component streams, then all output to the broadcast stream is discarded.)

The set of operations that may be performed on a broadcast stream is the intersection of those for its associated output streams.

Some output operations (e.g., fresh-line) return values based on the state of the stream at the time of the operation. Since these values might differ for each of the component streams, it is necessary to describe their return value specifically:

- stream-element-type returns the value from the last component stream, or t if there are no component streams.
- fresh-line returns the value from the last component stream, or nil if there are no component streams.
• The functions `file-length`, `file-position`, `file-string-length`, and `stream-external-format` return the value from the last component stream; if there are no component streams, `file-length` and `file-position` return 0, `file-string-length` returns 1, and `stream-external-format` returns `:default`.

• The functions `streamp` and `output-stream-p` always return `true` for `broadcast streams`.

• The functions `open-stream-p` tests whether the `broadcast stream` is `open`, not whether its component streams are `open`.

• The functions `input-stream-p` and `interactive-stream-p` return an `implementation-defined`, `generalized boolean` value.

• For the input operations `clear-input`, `listen`, `peek-char`, `read-byte`, `read-char-no-hang`, `read-char`, `read-line`, and `unread-char`, the consequences are undefined if the indicated operation is performed. However, an `implementation` is permitted to define such a behavior as an `implementation-dependent` extension.

For any output operations not having their return values explicitly specified above or elsewhere in this document, it is defined that the `values` returned by such an operation are the `values` resulting from performing the operation on the last of its `component streams`; the `values` resulting from performing the operation on all preceding `streams` are discarded. If there are no `component streams`, the value is `implementation-dependent`.

See Also:

`broadcast-stream-streams`, `make-broadcast-stream`

**concatenated-stream**

*System Class*

**Class Precedence List:**

`concatenated-stream`, `stream`, `t`

**Description:**

A `concatenated stream` is an `input stream` which is a `composite stream` of zero or more other `input streams`, such that the sequence of data which can be read from the `concatenated stream` is the same as the concatenation of the sequences of data which could be read from each of the constituent `streams`.

Input from a `concatenated stream` is taken from the first of the associated `input streams` until it reaches `end of file`; then that `stream` is discarded, and subsequent input is taken from the next `input stream`, and so on. An `end of file` on the associated `input streams` is always managed invisibly by the `concatenated stream`—the only time a client of a `concatenated stream` sees an `end`
An attempt to obtain data from the concatenated stream but it has no remaining input streams from which to obtain such data.

See Also:
concatenated-stream-streams, make-concatenated-stream

echo-stream

Class Precedence List:
echo-stream, stream, t

Description:
An echo stream is a bidirectional stream that gets its input from an associated input stream and sends its output to an associated output stream.

All input taken from the input stream is echoed to the output stream. Whether the input is echoed immediately after it is encountered, or after it has been read from the input stream is implementation-dependent.

See Also:
echo-stream-input-stream, echo-stream-output-stream, make-echo-stream

file-stream

Class Precedence List:
file-stream, stream, t

Description:
An object of type file-stream is a stream the direct source or sink of which is a file. Such a stream is created explicitly by open and with-open-file, and implicitly by functions such as load that process files.

See Also:
load, open, with-open-file
**string-stream**  
*System Class*

Class Precedence List:
- string-stream
- stream
- t

Description:
A *string stream* is a *stream* which reads input from or writes output to an associated *string*.

The stream element type of a *string stream* is always a subtype of type *character*.

See Also:
- make-string-input-stream, make-string-output-stream, with-input-from-string, with-output-to-string

**synonym-stream**  
*System Class*

Class Precedence List:
- synonym-stream
- stream
- t

Description:
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.

Any operations on a *synonym stream* will be performed on the *stream* that is then the value of the dynamic variable named by the synonym stream symbol. If the value of the variable should change, or if the variable should be bound, then the *stream* will operate on the new value of the variable.

See Also:
- make-synonym-stream, synonym-stream-symbol
two-way-stream

System Class

Class Precedence List:
  two-way-stream, stream, t

Description:
  A bidirectional composite stream that receives its input from an associated input stream and sends its output to an associated output stream.

See Also:
  make-two-way-stream, two-way-stream-input-stream, two-way-stream-output-stream

input-stream-p, output-stream-p

Function

Syntax:
  input-stream-p stream  \rightarrow generalized-boolean
  output-stream-p stream  \rightarrow generalized-boolean

Arguments and Values:
  stream—a stream.
  generalized-boolean—a generalized boolean.

Description:
  input-stream-p returns true if stream is an input stream; otherwise, returns false.
  output-stream-p returns true if stream is an output stream; otherwise, returns false.

Examples:
  (input-stream-p *standard-input*)  \rightarrow true
  (input-stream-p *terminal-io*)  \rightarrow true
  (input-stream-p (make-string-output-stream))  \rightarrow false
  (output-stream-p *standard-output*)  \rightarrow true
  (output-stream-p *terminal-io*)  \rightarrow true
  (output-stream-p (make-string-input-stream "jr"))  \rightarrow false

Exceptional Situations:
  Should signal an error of type type-error if stream is not a stream.
interactive-stream-p

Function

Syntax:

interactive-stream-p stream → generalized-boolean

Arguments and Values:

stream—a stream.

generalized-boolean—a generalized boolean.

Description:

Returns true if stream is an interactive stream; otherwise, returns false.

Examples:

(when (> measured limit)
  (let ((error (round (* (- measured limit) 100)
    limit)))
    (unless (if (interactive-stream-p *query-io*)
      (yes-or-no-p "The frammis is out of tolerance by ~D%. Is it safe to proceed? " error)
    (< error 15))) ;15% is acceptable
      (error "The frammis is out of tolerance by " error)))))

Exceptional Situations:

Should signal an error of type type-error if stream is not a stream.

See Also:

Section 21.1 (Stream Concepts)

open-stream-p

Function

Syntax:

open-stream-p stream → generalized-boolean

Arguments and Values:

stream—a stream.

generalized-boolean—a generalized boolean.

Description:

Returns true if stream is an open stream; otherwise, returns false.
Streams are open until they have been explicitly closed with `close`, or until they are implicitly closed due to exit from a `with-output-to-string`, `with-open-file`, `with-input-from-string`, or `with-open-stream` form.

**Examples:**

\[
\text{(open-stream-p *standard-input*)} \rightarrow \text{true}
\]

**Affected By:**

`close`.

**Exceptional Situations:**

Should signal an error of type `type-error` if `stream` is not a `stream`.

---

### stream-element-type

*Function*

**Syntax:**

\[\text{stream-element-type } \text{stream} \rightarrow \text{typespec}\]

**Arguments and Values:**

- **stream**—a `stream`.
- **typespec**—a `type specifier`.

**Description:**

`stream-element-type` returns a `type specifier` that indicates the `types` of `objects` that may be read from or written to `stream`.

*Streams* created by `open` have an `element type` restricted to `integer` or a `subtype` of `type` `character`.

**Examples:**

```lisp
;; Note that the stream must accommodate at least the specified type,
;; but might accommodate other types. Further note that even if it does
;; accommodate exactly the specified type, the type might be specified in
;; any of several ways.
(with-open-file (s "test" :element-type '(integer 0 1)
  :if-exists :error
  :direction :output)
```

Streams 21–13
(stream-element-type s))
→ INTEGER
or (UNSIGNED-BYTE 16)
or (UNSIGNED-BYTE 8)
or BIT
or (UNSIGNED-BYTE 1)
or (INTEGER 0 1)
or (INTEGER 0 2))

Exceptional Situations:
Should signal an error of type type-error if stream is not a stream.

---

**streamp**  
*Function*

Syntax:
streamp object → generalized-boolean

Arguments and Values:
object—an object.

generalized-boolean—a generalized boolean.

Description:
Returns true if object is of type stream; otherwise, returns false.

streamp is unaffected by whether object, if it is a stream, is open or closed.

Examples:

(streamp *terminal-io*) → true
(streamp 1) → false

Notes:

(streamp object) ≡ (typep object 'stream)
read-byte

Function

Syntax:

```
read-byte stream &optional eof-error-p eof-value  → byte
```

Arguments and Values:

- `stream`—a binary input stream.
- `eof-error-p`—a generalized boolean. The default is `true`.
- `eof-value`—an object. The default is `nil`.
- `byte`—an integer, or the `eof-value`.

Description:

`read-byte` reads and returns one byte from `stream`. If an end of file occurs and `eof-error-p` is `false`, the `eof-value` is returned.

Examples:

```
(with-open-file (s "temp-bytes"
            :direction :output
            :element-type 'unsigned-byte)
  (write-byte 101 s)) → 101
(with-open-file (s "temp-bytes" :element-type 'unsigned-byte)
  (format t "~S ~S" (read-byte s) (read-byte s nil 'eof)))
▷ 101 EOF
       → NIL
```

Side Effects:

Modifies `stream`.

Exceptional Situations:

- Should signal an error of type `type-error` if `stream` is not a `stream`.
- Should signal an error of type `error` if `stream` is not a binary input stream.
- If there are no bytes remaining in the `stream` and `eof-error-p` is `true`, an error of type `end-of-file` is signaled.

See Also:

- `read-char`, `read-sequence`, `write-byte`
write-byte

Function

Syntax:

```
write-byte byte stream  → byte
```

Arguments and Values:

- `byte`—an integer of the stream element type of `stream`.
- `stream`—a binary output stream.

Description:

`write-byte` writes one byte, `byte`, to `stream`.

Examples:

```
(with-open-file (s "temp-bytes"
   :direction :output
   :element-type 'unsigned-byte)
   (write-byte 101 s))  →  101
```

Side Effects:

`stream` is modified.

Affected By:

The element type of the `stream`.

Exceptional Situations:

Should signal an error of type `type-error` if `stream` is not a `stream`. Should signal an error of type `error` if `stream` is not a binary output stream.

Might signal an error of type `type-error` if `byte` is not an integer of the stream element type of `stream`.

See Also:

`read-byte`, `write-char`, `write-sequence`
peek-char

Syntax:

```lisp
peek-char &optional peek-type input-stream eof-error-p → char
  eof-value recursive-p
```

Arguments and Values:

- **peek-type**: a character or `t` or `nil`.
- **input-stream**: input stream designator. The default is `standard input`.
- **eof-error-p**: a generalized boolean. The default is `true`.
- **eof-value**: an object. The default is `nil`.
- **recursive-p**: a generalized boolean. The default is `false`.
- **char**: a character or the `eof-value`.

Description:

**peek-char** obtains the next character in **input-stream** without actually reading it, thus leaving the character to be read at a later time. It can also be used to skip over and discard intervening characters in the **input-stream** until a particular character is found.

If **peek-type** is not supplied or `nil`, **peek-char** returns the next character to be read from **input-stream**, without actually removing it from **input-stream**. The next time input is done from **input-stream**, the character will still be there. If **peek-type** is `t`, then **peek-char** skips over whitespace characters, but not comments, and then performs the peeking operation on the next character. The last character examined, the one that starts an object, is not removed from **input-stream**.

If **peek-type** is a character, then **peek-char** skips over input characters until a character that is `char=` to that character is found; that character is left in **input-stream**.

If an end of file occurs and **eof-error-p** is `false`, **eof-value** is returned.

If **recursive-p** is `true`, this call is expected to be embedded in a higher-level call to **read** or a similar function used by the Lisp reader.

When **input-stream** is an echo stream, characters that are only peeked at are not echoed. In the case that **peek-type** is not `nil`, the characters that are passed by **peek-char** are treated as if by **read-char**, and so are echoed unless they have been marked otherwise by **unread-char**.

Examples:

```lisp
(with-input-from-string (input-stream " 1 2 3 4 5")
  (format t "" ~S "S "S"
  (peek-char t input-stream))
```

Streams 21–17
(read-char Function

Syntax:
read-char &optional input-stream eof-error-p eof-value recursive-p → char

Arguments and Values:
input-stream—an input stream designator. The default is standard input.
eof-error-p—a generalized boolean. The default is true.
eof-value—an object. The default is nil.
recursive-p—a generalized boolean. The default is false.
char—a character or the eof-value.

Description:
read-char returns the next character from input-stream.

When input-stream is an echo stream, the character is echoed on input-stream the first time the character is seen. Characters that are not echoed by read-char are those that were put there by unread-char and hence are assumed to have been echoed already by a previous call to read-char.

If recursive-p is true, this call is expected to be embedded in a higher-level call to read or a similar function used by the Lisp reader.

If an end of file occurs and eof-error-p is false, eof-value is returned.
Examples:

\[
\text{(with-input-from-string (is "0123")}
\text{(do ((c (read-char is) (read-char is nil 'the-end)))}
\text{((not (characterp c)))}
\text{(format t "\"S " c)))}
\]

\[
> \text{\#\0 \#\1 \#\2 \#\3}
\]

→ NIL

Affected By:

*standard-input*, *terminal-io*.

Exceptional Situations:

If an \text{end of file} occurs before a character can be read, and \text{eof-error-p} is \text{true}, an error of \text{type end-of-file} is signaled.

See Also:

read-byte, read-sequence, write-char, read

Notes:

The corresponding output function is \text{write-char}.

---

**read-char-no-hang**

\text{Function}

Syntax:

\[
\text{read-char-no-hang \&optional input-stream eof-error-p} \rightarrow \text{char}
\text{eof-value recursive-p}
\]

Arguments and Values:

- \text{input-stream} – an \text{input stream designator}. The default is \text{standard input}.
- \text{eof-error-p}—a \text{generalized boolean}. The default is \text{true}.
- \text{eof-value}—an \text{object}. The default is \text{nil}.
- \text{recursive-p}—a \text{generalized boolean}. The default is \text{false}.
- \text{char}—a \text{character} or \text{nil} or the \text{eof-value}.

Description:

\text{read-char-no-hang} returns a character from \text{input-stream} if such a character is available. If no character is available, \text{read-char-no-hang} returns \text{nil}.
If `recursive-p` is `true`, this call is expected to be embedded in a higher-level call to `read` or a similar function used by the Lisp reader.

If an end of file occurs and `eof-error-p` is `false`, `eof-value` is returned.

Examples:

```lisp
;; This code assumes an implementation in which a newline is not
;; required to terminate input from the console.
(defun test-it ()
  (unread-char (read-char))
  (list (read-char-no-hang)
        (read-char-no-hang)
        (read-char-no-hang))
→ TEST-IT
;; Implementation A, where a Newline is not required to terminate
;; interactive input on the console.
(test-it)
▷ a
→ (#\a NIL NIL)
;; Implementation B, where a Newline is required to terminate
;; interactive input on the console, and where that Newline remains
;; on the input stream.
(test-it)
▷ a
→ (#\a #\ newline NIL)
```

Affected By:

* `*standard-input*`, *`terminal-io`*.

Exceptional Situations:

If an end of file occurs when `eof-error-p` is `true`, an error of type `end-of-file` is signaled.

See Also:

`listen`

Notes:

`read-char-no-hang` is exactly like `read-char`, except that if it would be necessary to wait in order to get a character (as from a keyboard), `nil` is immediately returned without waiting.
terpri, fresh-line

Function

Syntax:

terpri &optional output-stream → nil
fresh-line &optional output-stream → generalized-boolean

Arguments and Values:

output-stream – an output stream designator. The default is standard output.

generalized-boolean—a generalized boolean.

Description:

terpri outputs a newline to output-stream.

fresh-line is similar to terpri but outputs a newline only if the output-stream is not already at the start of a line. If for some reason this cannot be determined, then a newline is output anyway. fresh-line returns true if it outputs a newline; otherwise it returns false.

Examples:

(with-output-to-string (s)
  (write-string "some text" s)
  (terpri s)
  (terpri s)
  (write-string "more text" s))
→ "some text
more text"

(with-output-to-string (s)
  (write-string "some text" s)
  (fresh-line s)
  (fresh-line s)
  (write-string "more text" s))
→ "some text
more text"

Side Effects:

The output-stream is modified.

Affected By:

*standard-output*, *terminal-io*.

Exceptional Situations:

None.

Streams  21–21
Notes:

terpri is identical in effect to

```
(write-char #\Newline output-stream)
```

---

**unread-char**

*Function*

**Syntax:**

```
unread-char character &optional input-stream → nil
```

**Arguments and Values:**

- `character`—a character; must be the last character that was read from `input-stream`.
- `input-stream`—an input stream designator. The default is standard input.

**Description:**

`unread-char` places `character` back onto the front of `input-stream` so that it will again be the next character in `input-stream`.

When `input-stream` is an *echo stream*, no attempt is made to undo any echoing of the character that might already have been done on `input-stream`. However, characters placed on `input-stream` by `unread-char` are marked in such a way as to inhibit later re-echo by `read-char`.

It is an error to invoke `unread-char` twice consecutively on the same `stream` without an intervening call to `read-char` (or some other input operation which implicitly reads characters) on that `stream`.

Invoking `peek-char` or `read-char` commits all previous characters. The consequences of invoking `unread-char` on any character preceding that which is returned by `peek-char` (including those passed over by `peek-char` that has a *non-nil* `peek-type`) are unspecified. In particular, the consequences of invoking `unread-char` after `peek-char` are unspecified.

**Examples:**

```
(with-input-from-string (is "0123")
  (dotimes (i 6)
    (let ((c (read-char is)))
      (if (evenp i) (format t "\"S\"\" i c) (unread-char c is))))))
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#\0</td>
</tr>
<tr>
<td>2</td>
<td>#\1</td>
</tr>
<tr>
<td>4</td>
<td>#\2</td>
</tr>
</tbody>
</table>
```

```
⇒ NIL
```

---

21–22 Programming Language—Common Lisp
Affected By:
*standard-input*, *terminal-io*.

See Also:
peek-char, read-char, Section 21.1 (Stream Concepts)

Notes:
unread-char is intended to be an efficient mechanism for allowing the Lisp reader and other parsers to perform one-character lookahead in input-stream.

write-char

Syntax:
write-char character &optional output-stream → character

Arguments and Values:
character—a character.
output-stream—an output stream designator. The default is standard output.

Description:
write-char outputs character to output-stream.

Examples:

```lisp
(write-char #\a)
▷ a
→ #\a
(with-output-to-string (s)
  (write-char #\a s)
  (write-char #\Space s)
  (write-char #\b s))
→ "a b"
```

Side Effects:
The output-stream is modified.

Affected By:
*standard-output*, *terminal-io*.

See Also:
read-char, write-byte, write-sequence
Function

**read-line**

Syntax:

```
read-line &optional input-stream eof-error-p eof-value recursive-p
→ line, missing-newline-p
```

Arguments and Values:

- `input-stream`—an input stream designator. The default is standard input.
- `eof-error-p`—a generalized boolean. The default is `true`.
- `eof-value`—an object. The default is `nil`.
- `recursive-p`—a generalized boolean. The default is `false`.
- `line`—a string or the `eof-value`.
- `missing-newline-p`—a generalized boolean.

Description:

Reads from `input-stream` a line of text that is terminated by a newline or end of file.

If `recursive-p` is `true`, this call is expected to be embedded in a higher-level call to `read` or a similar function used by the Lisp reader.

The primary value, `line`, is the line that is read, represented as a string (without the trailing newline, if any). If `eof-error-p` is `false` and the end of file for `input-stream` is reached before any characters are read, `eof-value` is returned as the `line`.

The secondary value, `missing-newline-p`, is a generalized boolean that is `false` if the `line` was terminated by a newline, or `true` if the `line` was terminated by the end of file for `input-stream` (or if the `line` is the `eof-value`).

Examples:

```
(setq a "line 1 line2")
→ "line 1 line2"
(read-line (setq input-stream (make-string-input-stream a)))
→ "line 1", false
(read-line input-stream)
→ "line2", true
(read-line input-stream nil nil)
→ NIL, true
```

21–24 Programming Language—Common Lisp
Affected By:

*standard-input*, *terminal-io*.

Exceptional Situations:

If an end of file occurs before any characters are read in the line, an error is signaled if eof-error-p is true.

See Also:

read

Notes:

The corresponding output function is write-line.

---

**write-string, write-line**

**Function**

Syntax:

write-string string &optional output-stream &key start end  → string
write-line string &optional output-stream &key start end  → string

Arguments and Values:

string— a string.

output-stream — an output stream designator. The default is standard output.

start, end—bounding index designators of string. The defaults for start and end are 0 and nil, respectively.

Description:

write-string writes the characters of the subsequence of string bounded by start and end to output-stream. write-line does the same thing, but then outputs a newline afterwards.

Examples:

 Zum (prog1 (write-string "books" nil :end 4) (write-string "worms"))
> bookworms
-> "books"
(progn (write-char \\*)
 (write-line "test12" *standard-output* :end 5)
 (write-line "*test2")
 (write-line "test 2")
 nil)
> *test1
受影响于：
*standard-output*, *terminal-io*.

**See Also:**
read-line, write-char

**Notes:**
write-line 和 write-string 返回 string，不是 start 和 end 之间的子串。

\[
\text{(write-string string)}
\equiv
(\text{dotimes (i (length string)}
\quad (\text{write-char (char string i)}))
\]

\[
\text{(write-line string)}
\equiv
(\text{progl (write-string string) (terpri)})
\]

---

**读取序列**  
*Function*

**Syntax:**

\[
\text{read-sequence sequence stream &key start end} \rightarrow \text{position}
\]

sequence—一个 sequence。

stream—一个 input stream。

start, end—bounding index designators of sequence。默认为 start 和 end 分别为 0 和 nil。

position—an integer greater than or equal to zero, and less than or equal to the length of the sequence。

**Description:**

Destructively modifies sequence by replacing the elements of sequence bounded by start and end with elements read from stream。

Sequence is destructively modified by copying successive elements into it from stream。If the end of file for stream is reached before copying all elements of the subsequence, then the extra elements near the end of sequence are not updated。

21–26  语言与实现—Common Lisp
Position is the index of the first element of sequence that was not updated, which might be less than end because the end of file was reached.

Examples:

(defvar *data* (make-array 15 :initial-element nil))
(values (read-sequence *data* (make-string-input-stream "test string")) *data*)
→ 11, #(#\t #\e #\s #\t #\Space #\s #\t #\r #\i #\n #\g NIL NIL NIL NIL)

Side Effects:

Modifies stream and sequence.

Exceptional Situations:

Should be prepared to signal an error of type type-error if sequence is not a proper sequence. Should signal an error of type type-error if start is not a non-negative integer. Should signal an error of type type-error if end is not a non-negative integer or nil. Might signal an error of type type-error if an element read from the stream is not a member of the element type of the sequence.

See Also:

Section 3.2.1 (Compiler Terminology), write-sequence, read-line

Notes:

read-sequence is identical in effect to iterating over the indicated subsequence and reading one element at a time from stream and storing it into sequence, but may be more efficient than the equivalent loop. An efficient implementation is more likely to exist for the case where the sequence is a vector with the same element type as the stream.

write-sequence

Function

Syntax:

write-sequence sequence stream &key start end → sequence
sequence—a sequence.
stream—an output stream.
start, end—bounding index designators of sequence. The defaults for start and end are 0 and nil, respectively.

Description:

write-sequence writes the elements of the subsequence of sequence bounded by start and end to stream.
Examples:

```
(write-sequence "bookworms" *standard-output* :end 4)
▷ book
→ "bookworms"
```

Side Effects:
Modifies stream.

Exceptional Situations:
Should be prepared to signal an error of type type-error if sequence is not a proper sequence. Should signal an error of type type-error if start is not a non-negative integer. Should signal an error of type type-error if end is not a non-negative integer or nil. Might signal an error of type type-error if an element of the bounded sequence is not a member of the stream element type of the stream.

See Also:
Section 3.2.1 (Compiler Terminology), read-sequence, write-string, write-line

Notes:
write-sequence is identical in effect to iterating over the indicated subsequence and writing one element at a time to stream, but may be more efficient than the equivalent loop. An efficient implementation is more likely to exist for the case where the sequence is a vector with the same element type as the stream.

file-length  

Syntax:
```
file-length stream → length
```

Arguments and Values:
```
stream—a stream associated with a file.
length—a non-negative integer or nil.
```

Description:
file-length returns the length of stream, or nil if the length cannot be determined. For a binary file, the length is measured in units of the element type of the stream.

Examples:

```
(with-open-file (s "decimal-digits.text"
```
Exceptional Situations:

Should signal an error of type type-error if stream is not a stream associated with a file.

See Also:

open

file-position

Function

Syntax:

file-position stream → position
file-position stream position-spec → success-p

Arguments and Values:

stream—a stream.
position-spec—a file position designator.
position—a file position or nil.
success-p—a generalized boolean.

Description:

Returns or changes the current position within a stream.

When position-spec is not supplied, file-position returns the current file position in the stream, or nil if this cannot be determined.

When position-spec is supplied, the file position in stream is set to that file position (if possible). file-position returns true if the repositioning is performed successfully, or false if it is not.

An integer returned by file-position of one argument should be acceptable as position-spec for use with the same file.

For a character file, performing a single read-char or write-char operation may cause the file position to be increased by more than 1 because of character-set translations (such as translating
between the Common Lisp \#\Newline character and an external ASCII carriage-return/line-feed sequence) and other aspects of the implementation. For a binary file, every read-byte or write-byte operation increases the file position by 1.

Examples:

```lisp
(defun tester ()
  (let ((noticed '()) file-written)
    (flet ((notice (x) (push x noticed) x))
      (with-open-file (s "test.bin"
                      :element-type '(unsigned-byte 8)
                      :direction :output
                      :if-exists :error)
        (notice (file-position s)) ;1
        (write-byte 5 s)
        (write-byte 6 s)
        (let ((p (file-position s)))
          (notice p) ;2
          (notice (when p (file-position s (1- p)))) ;3
          (write-byte 7 s)
          (notice (file-position s)) ;4
          (setq file-written (truename s)))
      (with-open-file (s file-written
                      :element-type '(unsigned-byte 8)
                      :direction :input)
        (notice (file-position s)) ;5
        (let ((length (file-length s)))
          (notice length) ;6
          (when length
            (dotimes (i length)
              (notice (read-byte s))))))))) ;7,...
    (nreverse noticed)))))
→ tester
(tester)
⇒ (0 2 T 2 0 2 5 7)
⇒ (0 2 NIL 3 0 3 5 6 7)
⇒ (NIL NIL NIL NIL NIL NIL)
```

**Side Effects:**

When the position-spec argument is supplied, the file position in the stream might be moved.

**Affected By:**

The value returned by file-position increases monotonically as input or output operations are performed.
Exceptional Situations:

If position-spec is supplied, but is too large or otherwise inappropriate, an error is signaled.

See Also:

file-length, file-string-length, open

Notes:

Implementations that have character files represented as a sequence of records of bounded size might choose to encode the file position as, for example, ⟨⟨record-number⟩⟩*⟨⟨max-record-size⟩⟩+⟨⟨character-within-record⟩⟩. This is a valid encoding because it increases monotonically as each character is read or written, though not necessarily by 1 at each step. An integer might then be considered “inappropriate” as position-spec to file-position if, when decoded into record number and character number, it turned out that the supplied record was too short for the specified character number.

---

file-string-length

Function

Syntax:

file-string-length stream object  → length

Arguments and Values:

stream—an output character file stream.

object—a string or a character.

length—a non-negative integer, or nil.

Description:

file-string-length returns the difference between what (file-position stream) would be after writing object and its current value, or nil if this cannot be determined.

The returned value corresponds to the current state of stream at the time of the call and might not be the same if it is called again when the state of the stream has changed.
open

Syntax:

open filespec &key direction element-type
  if-exists if-does-not-exist external-format

→ stream

Arguments and Values:

filespec—a pathname designator.

direction—one of :input, :output, :io, or :probe. The default is :input.

element-type—a type specifier for recognizable subtype of character; or a type specifier for a finite
  recognizable subtype of integer; or one of the symbols signed-byte, unsigned-byte, or :default.
  The default is character.

  :supersede, or nil. The default is :new-version if the version component of filespec is :newest,
  or :error otherwise.

if-does-not-exist—one of :error, :create, or nil. The default is :error if direction is :input or
  if-exists is :overwrite or :append; :create if direction is :output or :io, and if-exists is neither
  :overwrite nor :append; or nil when direction is :probe.

external-format—an external file format designator. The default is :default.

stream—a file stream or nil.

Description:

open creates, opens, and returns a file stream that is connected to the file specified by filespec.
Filespec is the name of the file to be opened. If the filespec designator is a stream, that stream
is not closed first or otherwise affected.

The keyword arguments to open specify the characteristics of the file stream that is returned, and
how to handle errors.

If direction is :input or :probe, or if if-exists is not :new-version and the version component of the
filespec is :newest, then the file opened is that file already existing in the file system that has a
version greater than that of any other file in the file system whose other pathname components
are the same as those of filespec.

An implementation is required to recognize all of the open keyword options and to do something
reasonable in the context of the host operating system. For example, if a file system does not
support distinct file versions and does not distinguish the notions of deletion and expunging,
:new-version might be treated the same as :rename or :supersede, and :rename-and-delete might
be treated the same as :supersede.
open

:direction

These are the possible values for direction, and how they affect the nature of the stream that is created:

:input
Causes the creation of an input file stream.

:output
Causes the creation of an output file stream.

:io
Causes the creation of a bidirectional file stream.

:probe
Causes the creation of a “no-directional” file stream; in effect, the file stream is created and then closed prior to being returned by open.

:element-type

The element-type specifies the unit of transaction for the file stream. If it is :default, the unit is determined by file system, possibly based on the file.

:if-exists

if-exists specifies the action to be taken if direction is :output or :io and a file of the name filespec already exists. If direction is :input, not supplied, or :probe, if-exists is ignored. These are the results of open as modified by if-exists:

:error
An error of type file-error is signaled.

:new-version
A new file is created with a larger version number.

:rename
The existing file is renamed to some other name and then a new file is created.

:rename-and-delete
The existing file is renamed to some other name, then it is deleted but not expunged, and then a new file is created.

Streams 21–33
open

:overwrite

Output operations on the stream destructively modify the existing file. If direction is :io the file is opened in a bidirectional mode that allows both reading and writing. The file pointer is initially positioned at the beginning of the file; however, the file is not truncated back to length zero when it is opened.

:append

Output operations on the stream destructively modify the existing file. The file pointer is initially positioned at the end of the file.

If direction is :io, the file is opened in a bidirectional mode that allows both reading and writing.

:supersede

The existing file is superseded; that is, a new file with the same name as the old one is created. If possible, the implementation should not destroy the old file until the new stream is closed.

nil

No file or stream is created; instead, nil is returned to indicate failure.

:if-does-not-exist

if-does-not-exist specifies the action to be taken if a file of name filespec does not already exist. These are the results of open as modified by if-does-not-exist:

:error

An error of type file-error is signaled.

:create

An empty file is created. Processing continues as if the file had already existed but no processing as directed by if-exists is performed.

nil

No file or stream is created; instead, nil is returned to indicate failure.

:external-format

This option selects an external file format for the file: The only standardized value for this option is :default, although implementations are permitted to define additional external file formats and implementation-dependent values returned by stream-external-format can also be used by conforming programs.
The **external-format** is meaningful for any kind of file stream whose **element type** is a subtype of character. This option is ignored for streams for which it is not meaningful; however, implementations may define other **element types** for which it is meaningful. The consequences are unspecified if a character is written that cannot be represented by the given external file format.

When a file is opened, a **file stream** is constructed to serve as the file system’s ambassador to the Lisp environment; operations on the file stream are reflected by operations on the file in the file system.

A file can be deleted, renamed, or destructively modified by **open**.

For information about opening relative pathnames, see Section 19.2.3 (Merging Pathnames).

**Examples:**

```
(open filespec :direction :probe)  →  #<Closed Probe File Stream...>
(setq q (merge-pathnames (user-homedir-pathname) "test"))
→  #<PATHNAME :HOST NIL :DEVICE device-name :DIRECTORY directory-name
   :NAME "test" :TYPE NIL :VERSION :NEWEST>

(open filespec :if-does-not-exist :create)  →  #<Input File Stream...>
(setq s (open filespec :direction :probe))  →  #<Closed Probe File Stream...>
(truename s)  →  #<PATHNAME :HOST NIL :DEVICE device-name :DIRECTORY
directory-name :NAME filespec :TYPE extension :VERSION 1>

(open s :direction :output :if-exists nil)  →  NIL
```

**Affected By:**

The nature and state of the host computer’s file system.

**Exceptional Situations:**

If **if-exists** is **:error**, (subject to the constraints on the meaning of **if-exists** listed above), an error of type **file-error** is signaled.

If **if-does-not-exist** is **:error** (subject to the constraints on the meaning of **if-does-not-exist** listed above), an error of type **file-error** is signaled.

If it is impossible for an implementation to handle some option in a manner close to what is specified here, an error of type **error** might be signaled.

An error of type **file-error** is signaled if (**wild-pathname-p filespec**) returns true.

An error of type **error** is signaled if the **external-format** is not understood by the **implementation**.

The various **file systems** in existence today have widely differing capabilities, and some aspects of the **file system** are beyond the scope of this specification to define. A given **implementation** might not be able to support all of these options in exactly the manner stated. An **implementation** is required to recognize all of these option keywords and to try to do something “reasonable” in the context of the host file system. Where necessary to accommodate the file system, an implementation...
tion deviate slightly from the semantics specified here without being disqualified for consideration as a conforming implementation. If it is utterly impossible for an implementation to handle some option in a manner similar to what is specified here, it may simply signal an error.

With regard to the :element-type option, if a type is requested that is not supported by the file system, a substitution of types such as that which goes on in upgrading is permissible. As a minimum requirement, it should be the case that opening an output stream to a file in a given element type and later opening an input stream to the same file in the same element type should work compatibly.

See Also:
with-open-file, close, pathname, logical-pathname, Section 19.2.3 (Merging Pathnames), Section 19.1.2 (Pathnames as Filenames)

Notes:
open does not automatically close the file when an abnormal exit occurs.

When element-type is a subtype of character, read-char and/or write-char can be used on the resulting file stream.

When element-type is a subtype of integer, read-byte and/or write-byte can be used on the resulting file stream.

When element-type is :default, the type can be determined by using stream-element-type.

---

stream-external-format

Function

Syntax:
stream-external-format stream → format

Arguments and Values:
stream—a file stream.
format—an external file format.

Description:
Returns an external file format designator for the stream.

Examples:
(with-open-file (stream "test" :direction :output)
(stream-external-format stream))
→ :DEFAULT
or :ISO8859/1-1987
or (:ASCII :SAIL)
or ACME::PROPRIETARY-FILE-FORMAT-17
or #<FILE-FORMAT :ISO646-1983 2343673>

See Also:
the :external-format argument to the function open and the with-open-file macro.

Notes:
The format returned is not necessarily meaningful to other implementations.

with-open-file

with-open-file (stream filespec {options}*) {declaration}* {form}* → results

Arguments and Values:
stream – a variable.
filespec—a pathname designator.
options – forms; evaluated.
declaration—a declare expression; not evaluated.
forms—an implicit progn.
results—the values returned by the forms.

Description:
with-open-file uses open to create a file stream to file named by filespec. Filespec is the name of the file to be opened. Options are used as keyword arguments to open.

The stream object to which the stream variable is bound has dynamic extent; its extent ends when the form is exited.

with-open-file evaluates the forms as an implicit prog with stream bound to the value returned by open.

When control leaves the body, either normally or abnormally (such as by use of throw), the file is automatically closed. If a new output file is being written, and control leaves abnormally, the file is aborted and the file system is left, so far as possible, as if the file had never been opened.
with-open-file

It is possible by the use of :if-exists nil or :if-does-not-exist nil for stream to be bound to nil. Users of :if-does-not-exist nil should check for a valid stream.

The consequences are undefined if an attempt is made to assign the stream variable. The compiler may choose to issue a warning if such an attempt is detected.

Examples:

```lisp
(with-open-file (s p :direction :output :if-exists :supersede)
  (format s "Here are a couple\% of test data lines\%") → NIL
(with-open-file (s p)
  (do ((l (read-line s) (read-line s nil 'eof)))
      ((eq l 'eof) "Reached end of file."
       (format t "\%*** \% \%A\%" l))
  )
> *** Here are a couple
> *** of test data lines
→ "Reached end of file."

;; Normally one would not do this intentionally because it is
;; not perspicuous, but beware when using :IF-DOES-NOT-EXIST NIL
;; that this doesn’t happen to you accidentally...
(with-open-file (foo "no-such-file" :if-does-not-exist nil)
  (read foo))
> hello?
→ HELLO? ;This value was read from the terminal, not a file!

;; Here’s another bug to avoid...
(with-open-file (foo "no-such-file" :direction :output :if-does-not-exist nil)
  (format foo "Hello"))
→ "Hello" ;FORMAT got an argument of NIL!
```

Side Effects:

Creates a stream to the file named by filename (upon entry), and closes the stream (upon exit). In some implementations, the file might be locked in some way while it is open. If the stream is an output stream, a file might be created.

Affected By:

The host computer's file system.

Exceptional Situations:

See the function open.
close

Function

Syntax:

\texttt{close stream &key abort \rightarrow result}

Arguments and Values:

- \texttt{stream}—a stream (either open or closed).
- \texttt{abort}—a generalized boolean. The default is false.
- \texttt{result}—\texttt{t} if the \texttt{stream} was open at the time it was received as an argument, or implementation-dependent otherwise.

Description:

\texttt{close} closes \texttt{stream}. Closing a \texttt{stream} means that it may no longer be used in input or output operations. The act of closing a file \texttt{stream} ends the association between the \texttt{stream} and its associated \texttt{file}; the transaction with the \texttt{file system} is terminated, and input/output may no longer be performed on the \texttt{stream}.

If \texttt{abort} is \texttt{true}, an attempt is made to clean up any side effects of having created \texttt{stream}. If \texttt{stream} performs output to a file that was created when the \texttt{stream} was created, the file is deleted and any previously existing file is not superseded.

It is permissible to close an already closed \texttt{stream}, but in that case the \texttt{result} is implementation-dependent.

After \texttt{stream} is closed, it is still possible to perform the following query operations upon it: \texttt{stream}, \texttt{pathname}, \texttt{true-name}, \texttt{merge-pathnames}, \texttt{pathname-host}, \texttt{pathname-device}, \texttt{pathname-directory}, \texttt{pathname-name}, \texttt{pathname-type}, \texttt{pathname-version}, \texttt{namestring}, \texttt{file-namestring}, \texttt{directory-namestring}, \texttt{host-namestring}, \texttt{enough-namestring}, \texttt{open}, \texttt{probe-file}, and \texttt{directory}.

The effect of \texttt{close} on a \texttt{constructed stream} is to close the argument \texttt{stream} only. There is no effect on the \texttt{constituents} of \texttt{composite streams}.

For a \texttt{stream} created with \texttt{make-string-output-stream}, the result of \texttt{get-output-stream-string} is unspecified after \texttt{close}.

Examples:

\begin{verbatim}
(setq s (make-broadcast-stream)) \rightarrow #<BROADCAST-STREAM>
\end{verbatim}
(close s) → T
(output-stream-p s) → true

Side Effects:
The stream is closed (if necessary). If abort is true and the stream is an output file stream, its associated file might be deleted.

See Also:
open

**with-open-stream**

Macro

Syntax:

```
(with-open-stream (var stream) {declaration}*[form]* → {result}*)
```

Arguments and Values:

- **var**—a variable name.
- **stream**—a form; evaluated to produce a stream.
- **declaration**—a declare expression; not evaluated.
- **forms**—an implicit progn.
- **results**—the values returned by the forms.

Description:

with-open-stream performs a series of operations on stream, returns a value, and then closes the stream.

Var is bound to the value of stream, and then forms are executed as an implicit progn. stream is automatically closed on exit from with-open-stream, no matter whether the exit is normal or abnormal. The stream has dynamic extent; its extent ends when the form is exited.

The consequences are undefined if an attempt is made to assign the the variable var with the forms.

Examples:

```
(with-open-stream (s (make-string-input-stream "1 2 3 4"))
  (+ (read s) (read s) (read s))) → 6
```
Side Effects:
The stream is closed (upon exit).

See Also:
close

listen  

Function

Syntax:
listen &optional input-stream → generalized-boolean

Arguments and Values:
input-stream—an input stream designator. The default is standard input.
generalized-boolean—a generalized boolean.

Description:
Returns true if there is a character immediately available from input-stream; otherwise, returns false. On a non-interactive input-stream, listen returns true except when at end of file. If an end of file is encountered, listen returns false. listen is intended to be used when input-stream obtains characters from an interactive device such as a keyboard.

Examples:
(progn (unread-char (read-char)) (list (listen) (read-char)))
▷ 1
→ (T #\1)
(progn (clear-input) (listen))
→ NIL ; Unless you’re a very fast typist!

Affected By:
*standard-input*

See Also:
interactive-stream-p, read-char-no-hang
clear-input

Function

Syntax:

clear-input &optional input-stream → nil

Arguments and Values:

input-stream—an input stream designator. The default is standard input.

Description:

Clears any available input from input-stream.

If clear-input does not make sense for input-stream, then clear-input does nothing.

Examples:

;;; The exact I/O behavior of this example might vary from implementation
;;; to implementation depending on the kind of interactive buffering that
;;; occurs. (The call to SLEEP here is intended to help even out the
;;; differences in implementations which do not do line-at-a-time buffering.)

(defun read-sleepily (&optional (clear-p nil) (zzz 0))
  (list (progn (print '>) (read))
        ;; Note that input typed within the first ZZZ seconds
        ;; will be discarded.
        (progn (print '>
                        (if zzz (sleep zzz))
                        (print '>>)
                        (if clear-p (clear-input))
                        (read))))

(read-sleepily)
▷ > 10
▷ >
▷ >> 20
→ (10 20)

(read-sleepily t)
▷ > 10
▷ >
▷ >> 20
→ (10 20)

(read-sleepily t 10)
▷ > 10
▷ > 20

; Some implementations won't echo typeahead here.
Side Effects:
The input-stream is modified.

Affected By:
*standard-input*

Exceptional Situations:
Should signal an error of type type-error if input-stream is not a stream designator.

See Also:
clear-output

---

**finish-output, force-output, clear-output**  
*Function*

Syntax:
- `finish-output &optional output-stream → nil`
- `force-output &optional output-stream → nil`
- `clear-output &optional output-stream → nil`

Arguments and Values:
- `output-stream`—an output stream designator. The default is standard output.

Description:
- `finish-output, force-output, and clear-output` exercise control over the internal handling of buffered stream output.
- `finish-output` attempts to ensure that any buffered output sent to output-stream has reached its destination, and then returns.
- `force-output` initiates the emptying of any internal buffers but does not wait for completion or acknowledgment to return.
- `clear-output` attempts to abort any outstanding output operation in progress in order to allow as little output as possible to continue to the destination.

If any of these operations does not make sense for output-stream, then it does nothing. The precise actions of these functions are implementation-dependent.
Examples:

;; Implementation A
(progn (princ "am i seen?") (clear-output))
→ NIL

;; Implementation B
(progn (princ "am i seen?") (clear-output))
▷ am i seen?
→ NIL

Affected By:
*standard-output*

Exceptional Situations:
Should signal an error of type type-error if output-stream is not a stream designator.

See Also:
clear-input

y-or-n-p, yes-or-no-p

Function

Syntax:

\[
\text{y-or-n-p \&optional \ control \&rest \ arguments} \rightarrow \text{generalized-boolean}
\]

\[
\text{yes-or-no-p \&optional \ control \&rest \ arguments} \rightarrow \text{generalized-boolean}
\]

Arguments and Values:

control—a format control.

arguments—format arguments for control.

generalized-boolean—a generalized boolean.

Description:

These functions ask a question and parse a response from the user. They return \text{true} if the answer is affirmative, or \text{false} if the answer is negative.

\text{y-or-n-p} is for asking the user a question whose answer is either “yes” or “no.” It is intended that the reply require the user to answer a yes-or-no question with a single character. \text{yes-or-no-p} is also for asking the user a question whose answer is either “Yes” or “No.” It is intended that the reply require the user to take more action than just a single keystroke, such as typing the full word \text{yes} or \text{no} followed by a newline.

21–44 Programming Language—Common Lisp
y-or-n-p types out a message (if supplied), reads an answer in some implementation-dependent manner (intended to be short and simple, such as reading a single character such as Y or N).

yes-or-no-p types out a message (if supplied), attracts the user’s attention (for example, by ringing the terminal’s bell), and reads an answer in some implementation-dependent manner (intended to be multiple characters, such as YES or NO).

If format-control is supplied and not nil, then a fresh-line operation is performed; then a message is printed as if format-control and arguments were given to format. In any case, yes-or-no-p and y-or-n-p will provide a prompt such as “(Y or N)” or “(Yes or No)” if appropriate.

All input and output are performed using query I/O.

Examples:

```
(y-or-n-p "(t or nil) given by")
▷ (t or nil) given by (Y or N) Y
  → true

(yes-or-no-p "a "$ message" 'frightening)
▷ a FRIGHTENING message (Yes or No) no
  → false

(y-or-n-p "Produce listing file?")
▷ Please respond with Y or N. n
  → false
```

Side Effects:
Output to and input from query I/O will occur.

Affected By:
*query-io*.

See Also:
format

Notes:
yes-or-no-p and yes-or-no-p do not add question marks to the end of the prompt string, so any desired question mark or other punctuation should be explicitly included in the text query.
**make-synonym-stream**  
*Function*

**Syntax:**

\[
\text{make-synonym-stream symbol} \rightarrow \text{synonym-stream}
\]

**Arguments and Values:**

- **symbol** — a symbol that names a dynamic variable.
- **synonym-stream** — a synonym stream.

**Description:**

Returns a synonym stream whose synonym stream symbol is `symbol`.

**Examples:**

\[
\begin{align*}
\text{(setq a-stream (make-string-input-stream "a-stream"))} \\
\text{b-stream (make-string-input-stream "b-stream"))} \\
\rightarrow \ #\lt\text{String Input Stream}\gt \\
\text{(setq s-stream (make-synonym-stream 'c-stream))} \\
\rightarrow \ #\lt\text{SYNONYM-STREAM for C-STREAM}\gt \\
\text{(setq c-stream a-stream)} \\
\rightarrow \ #\lt\text{String Input Stream}\gt \\
\text{(read s-stream)} \rightarrow \ \text{A-STREAM} \\
\text{(setq c-stream b-stream)} \\
\rightarrow \ #\lt\text{String Input Stream}\gt \\
\text{(read s-stream)} \rightarrow \ \text{B-STREAM}
\end{align*}
\]

**Exceptional Situations:**

Should signal `type-error` if its argument is not a `symbol`.

**See Also:**

Section 21.1 (Stream Concepts)

---

**synonym-stream-symbol**  
*Function*

**Syntax:**

\[
\text{synonym-stream-symbol synonym-stream} \rightarrow \text{symbol}
\]

**Arguments and Values:**

- **synonym-stream** — a synonym stream.
- **symbol** — a symbol.
Description:
Returns the symbol whose symbol-value the synonym-stream is using.

See Also:
make-synonym-stream

broadcast-stream-streams  Function

Syntax:
broadcast-stream-streams broadcast-stream  \(\rightarrow\) streams

Arguments and Values:
broadcast-stream—a broadcast stream.
streams—a list of streams.

Description:
Returns a list of output streams that constitute all the streams to which the broadcast-stream is broadcasting.

make-broadcast-stream  Function

Syntax:
make-broadcast-stream &rest streams  \(\rightarrow\) broadcast-stream

Arguments and Values:
stream—an output stream.
broadcast-stream—a broadcast stream.

Description:
Returns a broadcast stream.

Examples:
(setq a-stream (make-string-output-stream)
b-stream (make-string-output-stream)) \(\rightarrow\) #<String Output Stream>
(format (make-broadcast-stream a-stream b-stream)
"this will go to both streams") \(\rightarrow\) NIL
(get-output-stream-string a-stream) \(\rightarrow\) "this will go to both streams"
(get-output-stream-string b-stream) \(\rightarrow\) "this will go to both streams"
make-two-way-stream

Function

Syntax:

make-two-way-stream input-stream output-stream → two-way-stream

Arguments and Values:
input-stream—a stream.
output-stream—a stream.
two-way-stream—a two-way stream.

Description:
Returns a two-way stream that gets its input from input-stream and sends its output to output-stream.

Examples:

(with-output-to-string (out)
  (with-input-from-string (in "input...")
    (let ((two (make-two-way-stream in out)))
      (format two "output...")
      (setq what-is-read (read two))))) → "output...

what-is-read → INPUT...

Exceptional Situations:
Should signal an error of type type-error if input-stream is not an input stream. Should signal an error of type type-error if output-stream is not an output stream.
two-way-stream-input-stream, two-way-stream-output-stream

Function

Syntax:

two-way-stream-input-stream two-way-stream → input-stream

two-way-stream-output-stream two-way-stream → output-stream

Arguments and Values:

two-way-stream—a two-way stream.

input-stream—an input stream.

output-stream—an output stream.

Description:

two-way-stream-input-stream returns the stream from which two-way-stream receives input.

two-way-stream-output-stream returns the stream to which two-way-stream sends output.

echo-stream-input-stream, echo-stream-output-stream

Function

Syntax:

echo-stream-input-stream echo-stream → input-stream

echo-stream-output-stream echo-stream → output-stream

Arguments and Values:

echo-stream—an echo stream.

input-stream—an input stream.

output-stream—an output stream.

Description:

echo-stream-input-stream returns the input stream from which echo-stream receives input.

echo-stream-output-stream returns the output stream to which echo-stream sends output.
### make-echo-stream

**Function**

**Syntax:**

```
make-echo-stream input-stream output-stream  → echo-stream
```

**Arguments and Values:**

- `input-stream`—an input stream.
- `output-stream`—an output stream.
- `echo-stream`—an echo stream.

**Description:**

Creates and returns an echo stream that takes input from `input-stream` and sends output to `output-stream`.

**Examples:**

```
(let ((out (make-string-output-stream)))
 (with-open-stream
   (s (make-echo-stream
        (make-string-input-stream "this-is-read-and-echoed")
        out))
   (read s)
   (format s " * this-is-direct-output")
   (get-output-stream-string out)))
→ "this-is-read-and-echoed * this-is-direct-output"
```

**See Also:**

- echo-stream-input-stream
- echo-stream-output-stream
- make-two-way-stream

### concatenated-stream-streams

**Function**

**Syntax:**

```
concatenated-stream-streams concatenated-stream  → streams
```

**Arguments and Values:**

- `concatenated-stream`—a concatenated stream.
- `streams`—a list of input streams.
Description:

Returns a list of input streams that constitute the ordered set of streams the concatenated-stream still has to read from, starting with the current one it is reading from. The list may be empty if no more streams remain to be read.

The consequences are undefined if the list structure of the streams is ever modified.

make-concatenated-stream

Function

Syntax:

make-concatenated-stream &rest input-streams → concatenated-stream

Arguments and Values:

input-stream—an input stream.
concatenated-stream—a concatenated stream.

Description:

Returns a concatenated stream that has the indicated input-streams initially associated with it.

Examples:

(read (make-concatenated-stream
 (make-string-input-stream "1")
 (make-string-input-stream "2"))) → 12

Exceptional Situations:

Should signal type-error if any argument is not an input stream.

See Also:

concatenated-stream-streams
get-output-stream-string

Function

Syntax:

get-output-stream-string string-output-stream → string

Arguments and Values:

string-output-stream—a stream.

string—a string.

Description:

Returns a string containing, in order, all the characters that have been output to string-output-stream. This operation clears any characters on string-output-stream, so the string contains only those characters which have been output since the last call to get-output-stream-string or since the creation of the string-output-stream, whichever occurred most recently.

Examples:

(setq a-stream (make-string-output-stream)
   a-string "abcdefgijklm") → "abcdefgijklm"
(write-string a-string a-stream) → "abcdefgijklm"
(get-output-stream-string a-stream) → "abcdefgijklm"
(get-output-stream-string a-stream) → ""

Side Effects:

The string-output-stream is cleared.

Exceptional Situations:

The consequences are undefined if stream-output-string is closed.

The consequences are undefined if string-output-stream is a stream that was not produced by make-string-output-stream. The consequences are undefined if string-output-stream was created implicitly by with-output-to-string or format.

See Also:

make-string-output-stream
make-string-input-stream

Function

Syntax:
```lisp
make-string-input-stream string koptional start end  → string-stream
```

Arguments and Values:
- `string`—a string.
- `start`, `end`—bounding index designators of `string`. The defaults for `start` and `end` are 0 and `nil`, respectively.
- `string-stream`—an input string stream.

Description:
Returns an input string stream. This stream will supply, in order, the characters in the substring of `string` bounded by `start` and `end`. After the last character has been supplied, the string stream will then be at end of file.

Examples:
```lisp
(let ((string-stream (make-string-input-stream "1 one "))
  (list (read string-stream nil nil)
        (read string-stream nil nil)
        (read string-stream nil nil)))
  → (1 ONE NIL))

(read (make-string-input-stream "prefixtargetsuffix" 6 12)) → TARGET
```

See Also:
- `with-input-from-string`

make-string-output-stream

Function

Syntax:
```lisp
make-string-output-stream &key element-type  → string-stream
```

Arguments and Values:
- `element-type`—a type specifier. The default is `character`.
- `string-stream`—an output string stream.
Description:

Returns an output string stream that accepts characters and makes available (via get-output-stream-string) a string that contains the characters that were actually output.

The element-type names the type of the elements of the string; a string is constructed of the most specialized type that can accommodate elements of that element-type.

Examples:

(let ((s (make-string-output-stream)))
  (write-string "testing... " s)
  (prinl 1234 s)
  (get-output-stream-string s))
→ "testing... 1234"

None..

See Also:

get-output-stream-string, with-output-to-string

with-input-from-string

Macro

Syntax:

```
with-input-from-string (var string &key index start end) {declaration}* {form}* → {result}*```

Arguments and Values:

- `var`—a variable name.
- `string`—a form; evaluated to produce a string.
- `index`—a place.
- `start, end`—bounding index designators of string. The defaults for start and end are 0 and nil, respectively.
- `declaration`—a declare expression; not evaluated.
- `forms`—an implicit progn.
- `result`—the values returned by the forms.

Description:

Creates an input string stream, provides an opportunity to perform operations on the stream (returning zero or more values), and then closes the string stream.
String is evaluated first, and var is bound to a character input string stream that supplies characters from the subsequence of the resulting string bounded by start and end. The body is executed as an implicit progn.

The input string stream is automatically closed on exit from with-input-from-string, no matter whether the exit is normal or abnormal. The input string stream to which the variable var is bound has dynamic extent; its extent ends when the form is exited.

The index is a pointer within the string to be advanced. If with-input-from-string is exited normally, then index will have as its value the index into the string indicating the first character not read which is (length string) if all characters were used. The place specified by index is not updated as reading progresses, but only at the end of the operation.

start and index may both specify the same variable, which is a pointer within the string to be advanced, perhaps repeatedly by some containing loop.

The consequences are undefined if an attempt is made to assign the variable var.

Examples:

(with-input-from-string (s "XXX1 2 3 4xxx" :index ind :start 3 :end 10)
 (+ (read s) (read s) (read s))) → 6

ind → 9
(with-input-from-string (s "Animal Crackers" :index j :start 6)
 (read s)) → CRACKERS

The variable j is set to 15.

Side Effects:

The value of the place named by index, if any, is modified.

See Also:

make-string-input-stream, Section 3.6 (Traversal Rules and Side Effects)

with-output-to-string  
Macro

Syntax:

with-output-to-string (var &optional string-form &key element-type) {declaration}* {form}* → {result}*  

Arguments and Values:

var—a variable name.
with-output-to-string

string-form—a form or nil; if non-nil, evaluated to produce string.

string—a string that has a fill pointer.

element-type—a type specifier; evaluated. The default is character.

declaration—a declare expression; not evaluated.

forms—an implicit progn.

results—If a string-form is not supplied or nil, a string; otherwise, the values returned by the forms.

Description:
with-output-to-string creates a character output stream, performs a series of operations that may send results to this stream, and then closes the stream.

The element-type names the type of the elements of the stream; a stream is constructed of the most specialized type that can accommodate elements of the given type.

The body is executed as an implicit progn with var bound to an output string stream. All output to that string stream is saved in a string.

If string is supplied, element-type is ignored, and the output is incrementally appended to string as if by use of vector-push-extend.

The output stream is automatically closed on exit from with-output-from-string, no matter whether the exit is normal or abnormal. The output string stream to which the variable var is bound has dynamic extent; its extent ends when the form is exited.

If no string is provided, then with-output-from-string produces a stream that accepts characters and returns a string of the indicated element-type. If string is provided, with-output-to-string returns the results of evaluating the last form.

The consequences are undefined if an attempt is made to assign the variable var.

Examples:

(setq fstr (make-array '(0) :element-type 'base-char  
 :fill-pointer 0 :adjustable t)) → ""
(with-output-to-string (s fstr)  
 (format s "here’s some output")  
 (input-stream-p s)) → false
fstr → "here’s some output"

Side Effects:

The string is modified.
Exceptional Situations:

The consequences are undefined if destructive modifications are performed directly on the string during the dynamic extent of the call.

See Also:

make-string-output-stream, vector-push-extend, Section 3.6 (Traversal Rules and Side Effects)

*debug-io*, *error-output*, *query-io*, *standard-input*, *standard-output*, *trace-output* Variable

Value Type:

For *standard-input*: an input stream

For *error-output*, *standard-output*, and *trace-output*: an output stream.

For *debug-io*, *query-io*: a bidirectional stream.

Initial Value:

implementation-dependent, but it must be an open stream that is not a generalized synonym stream to an I/O customization variables but that might be a generalized synonym stream to the value of some I/O customization variable. The initial value might also be a generalized synonym stream to either the symbol *terminal-io* or to the stream that is its value.

Description:

These variables are collectively called the standardized I/O customization variables. They can be bound or assigned in order to change the default destinations for input and/or output used by various standardized operators and facilities.

The value of *debug-io*, called debug I/O, is a stream to be used for interactive debugging purposes.

The value of *error-output*, called error output, is a stream to which warnings and non-interactive error messages should be sent.

The value of *query-io*, called query I/O, is a bidirectional stream to be used when asking questions of the user. The question should be output to this stream, and the answer read from it.

The value of *standard-input*, called standard input, is a stream that is used by many operators as a default source of input when no specific input stream is explicitly supplied.

The value of *standard-output*, called standard output, is a stream that is used by many operators as a default destination for output when no specific output stream is explicitly supplied.
*debug-io*, *error-output*, *query-io*, …

The value of *trace-output*, called trace output, is the stream on which traced functions (see trace) and the time macro print their output.

Examples:

```
(with-output-to-string (*error-output*)
  (warn "this string is sent to *error-output*")
→ "Warning: this string is sent to *error-output*
" ;The exact format of this string is implementation-dependent.
```

```
(with-input-from-string (*standard-input* "1001")
  (+ 990 (read))) → 1991
```

```
(progn (setq out (with-output-to-string (*standard-output*)
  (print "print and format t send things to")
    (format t "*standard-output* now going to a string")))
  :done)
→ :DONE
out
→ "
"print and format t send things to" *standard-output* now going to a string"
```

```
(defun fact (n) (if (< n 2) 1 (* n (fact (- n 1)))))
→ FACT
(trace fact)
→ (FACT)
;;;; Of course, the format of traced output is implementation-dependent.
(with-output-to-string (*trace-output*)
  (fact 3))
→ "
1 Enter FACT 3
 | 2 Enter FACT 2
 | 3 Enter FACT 1
 | 3 Exit FACT 1
 | 2 Exit FACT 2
1 Exit FACT 6"
```

See Also:

*terminal-io*, synonym-stream, time, trace, Chapter 9 (Conditions), Chapter 23 (Reader), Chapter 22 (Printer)

21–58 Programming Language—Common Lisp
Notes:
The intent of the constraints on the initial value of the I/O customization variables is to ensure that it is always safe to bind or assign such a variable to the value of another I/O customization variable, without unduly restricting implementation flexibility.

It is common for an implementation to make the initial values of *debug-io* and *query-io* be the same stream, and to make the initial values of *error-output* and *standard-output* be the same stream.

The functions y-or-n-p and yes-or-no-p use query I/O for their input and output.

In the normal Lisp read-eval-print loop, input is read from standard input. Many input functions, including read and read-char, take a stream argument that defaults to standard input.

In the normal Lisp read-eval-print loop, output is sent to standard output. Many output functions, including print and write-char, take a stream argument that defaults to standard output.

A program that wants, for example, to divert output to a file should do so by binding *standard-output*; that way error messages sent to *error-output* can still get to the user by going through *terminal-io* (if *error-output* is bound to *terminal-io*), which is usually what is desired.

*terminal-io*

Value Type:
a bidirectional stream.

Initial Value:
implementation-dependent, but it must be an open stream that is not a generalized synonym stream to an I/O customization variables but that might be a generalized synonym stream to the value of some I/O customization variable.

Description:
The value of *terminal-io*, called terminal I/O, is ordinarily a bidirectional stream that connects to the user’s console. Typically, writing to this stream would cause the output to appear on a display screen, for example, and reading from the stream would accept input from a keyboard. It is intended that standard input functions such as read and read-char, when used with this stream, cause echoing of the input into the output side of the stream. The means by which this is accomplished are implementation-dependent.

The effect of changing the value of *terminal-io*, either by binding or assignment, is implementation-defined.
Examples:

```lisp
(progn (prin1 'foo) (prin1 'bar *terminal-io*))
▷ FOOBAR
  ▷ BAR
(with-output-to-string (*standard-output*)
  (prin1 'foo)
  (prin1 'bar *terminal-io*))
▷ BAR
  ▷ "FOO"
```

See Also:

*debug-io*, *error-output*, *query-io*, *standard-input*, *standard-output*, *trace-output*

---

**stream-error**

*Condition Type*

Class Precedence List:

stream-error, error, serious-condition, condition, t

Description:

The type stream-error consists of error conditions that are related to receiving input from or sending output to a stream. The “offending stream” is initialized by the :stream initialization argument to make-condition, and is accessed by the function stream-error-stream.

See Also:

stream-error-stream

---

**stream-error-stream**

*Function*

Syntax:

```
stream-error-stream condition → stream
```

Arguments and Values:

- `condition`—a condition of type stream-error.
- `stream`—a stream.

Description:

Returns the offending stream of a condition of type stream-error.
Examples:

```lisp
(with-input-from-string (s "(FOO")
  (handler-case (read s)
    (end-of-file (c)
      (format nil "&End of file on ~S." (stream-error-stream c))))
  "End of file on #<String Stream>."
```

See Also:
stream-error, Chapter 9 (Conditions)

---

**end-of-file**

*Condition Type*

**Class Precedence List:**
end-of-file, stream-error, error, serious-condition, condition, t

**Description:**
The type `end-of-file` consists of error conditions related to read operations that are done on `streams` that have no more data.

See Also:
stream-error-stream
Programming Language—Common Lisp

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:

\[ 27 \quad 27. \quad \#o33 \quad \#x1B \quad \#b11011 \quad \#.(\ast 3 3 3) \quad 81/3 \]

A list containing the two symbols \(A\) and \(B\) can also be textually expressed in a variety of ways:

\[(A \ B) \quad (a \ b) \quad (\ a \ b \ ) \quad (\ A \ |B|) \quad (|A|\ B)\]

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. Figure 22–1 shows the standardized printer control variables; there might also be implementation-defined printer control variables.

<table>
<thead>
<tr>
<th>Option Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>print-array</em></td>
</tr>
<tr>
<td><em>print-base</em></td>
</tr>
<tr>
<td><em>print-case</em></td>
</tr>
<tr>
<td><em>print-circle</em></td>
</tr>
<tr>
<td><em>print-escape</em></td>
</tr>
<tr>
<td><em>print-gensym</em></td>
</tr>
<tr>
<td><em>print-length</em></td>
</tr>
<tr>
<td><em>print-level</em></td>
</tr>
<tr>
<td><em>print-lines</em></td>
</tr>
<tr>
<td><em>print-miser-width</em></td>
</tr>
<tr>
<td><em>print-pprint-dispatch</em></td>
</tr>
<tr>
<td><em>print-pretty</em></td>
</tr>
<tr>
<td><em>print-radix</em></td>
</tr>
<tr>
<td><em>print-readably</em></td>
</tr>
<tr>
<td><em>print-right-margin</em></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th><em>package</em></th>
<th><em>read-eval</em></th>
<th>readtable-case</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>read-default-float-format</em></td>
<td><em>readtable</em></td>
<td></td>
</tr>
</tbody>
</table>

Figure 22–2. Additional Influences on the Lisp printer.

### 22.1.1.1 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, 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.

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 this section 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

```scheme
(let ((x (make-symbol "FOO"))) (list x x))
```

would be printed as (#1=#:foo #1#) 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:

<table>
<thead>
<tr>
<th>READTABLE-CASE</th>
<th><em>PRINT-CASE</em></th>
<th>Symbol-name</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>:UPCASE</td>
<td>:UPCASE</td>
<td>ZEBRA</td>
<td>ZEBRA</td>
</tr>
<tr>
<td>:UPCASE</td>
<td>:UPCASE</td>
<td>Zebra</td>
<td></td>
</tr>
<tr>
<td>:UPCASE</td>
<td>:UPCASE</td>
<td>zebra</td>
<td></td>
</tr>
<tr>
<td>:UPCASE</td>
<td>:DOWNCASE</td>
<td>ZEBRA</td>
<td>zebra</td>
</tr>
<tr>
<td>:UPCASE</td>
<td>:DOWNCASE</td>
<td>Zebra</td>
<td></td>
</tr>
<tr>
<td>:UPCASE</td>
<td>:DOWNCASE</td>
<td>zebra</td>
<td></td>
</tr>
<tr>
<td>:UPCASE</td>
<td>:CAPITALIZE</td>
<td>ZEBRA</td>
<td>Zebra</td>
</tr>
<tr>
<td>:UPCASE</td>
<td>:CAPITALIZE</td>
<td>Zebra</td>
<td></td>
</tr>
<tr>
<td>:UPCASE</td>
<td>:CAPITALIZE</td>
<td>zebra</td>
<td></td>
</tr>
<tr>
<td>:DOWNCASE</td>
<td>:UPCASE</td>
<td>ZEBRA</td>
<td></td>
</tr>
<tr>
<td>:DOWNCASE</td>
<td>:UPCASE</td>
<td>Zebra</td>
<td></td>
</tr>
<tr>
<td>:DOWNCASE</td>
<td>:UPCASE</td>
<td>zebra</td>
<td></td>
</tr>
<tr>
<td>:DOWNCASE</td>
<td>:DOWNCASE</td>
<td>ZEBRA</td>
<td></td>
</tr>
<tr>
<td>:DOWNCASE</td>
<td>:DOWNCASE</td>
<td>Zebra</td>
<td></td>
</tr>
<tr>
<td>:DOWNCASE</td>
<td>:DOWNCASE</td>
<td>zebra</td>
<td></td>
</tr>
<tr>
<td>:DOWNCASE</td>
<td>:CAPITALIZE</td>
<td>ZEBRA</td>
<td></td>
</tr>
<tr>
<td>:DOWNCASE</td>
<td>:CAPITALIZE</td>
<td>Zebra</td>
<td></td>
</tr>
<tr>
<td>:DOWNCASE</td>
<td>:CAPITALIZE</td>
<td>zebra</td>
<td></td>
</tr>
<tr>
<td>:PREVENT</td>
<td>:UPCASE</td>
<td>ZEBRA</td>
<td>ZEBRA</td>
</tr>
<tr>
<td>:PREVENT</td>
<td>:UPCASE</td>
<td>Zebra</td>
<td>Zebra</td>
</tr>
<tr>
<td>:PREVENT</td>
<td>:UPCASE</td>
<td>zebra</td>
<td>zebra</td>
</tr>
<tr>
<td>:PREVENT</td>
<td>:DOWNCASE</td>
<td>ZEBRA</td>
<td>ZEBRA</td>
</tr>
<tr>
<td>:PREVENT</td>
<td>:DOWNCASE</td>
<td>Zebra</td>
<td>Zebra</td>
</tr>
<tr>
<td>:PREVENT</td>
<td>:DOWNCASE</td>
<td>zebra</td>
<td>zebra</td>
</tr>
</tbody>
</table>
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 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:

\[(a . b) ; A dotted pair of a and b\]
\[(a . b) ; A list of two elements a and b\]
\[(a b . c) ; A dotted list of a and b with c at the end; two conses\]
\[.iot ; The symbol whose name is .iot\]
\[. (b) ; Invalid -- an error is signaled if an attempt is made to read \]
\[; this syntax.\]
\[(a ..) ; Invalid -- an error is signaled.\]
\[(a .. b) ; Invalid -- an error is signaled.\]
\[(a .. b) ; Invalid -- an error is signaled.\]
\[(a b c ...) ; Invalid -- an error is signaled.\]
\[(a \ . b) ; A list of three elements a, ., and b\]
\[(a . \ b) ; A list of three elements a, ., and b\]
\[(a \ ... b) ; A list of three elements a, ..., and b\]
\[(a \ ... b) ; A list of three elements a, ..., and b\]

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, 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 (Sharp-sign 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. 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.
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

```lisp
(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

```lisp
#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 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))
▷
(progn (fresh-line) (princ "\n"))

(a)

(dolist (val '(t nil))
  (let ((*print-escape* val) (*print-readably* val))
    (print "\n")
    (prinl "\n")
    (write-char \Space)
    (princ "\n")
    (writechar \Space)
    (write "\n")))

NIL

(progn (fresh-line) (write '(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))

(progn (fresh-line)
  (write '(let ((a 1) (b 2)) (+ a b)) :pretty t))

LET ((A 1) (B 2)) (+ A B))

(with-output-to-string (s)
  (write 'write :stream s)
  (prinl 'prin1 s))

"WRITEPRINT"
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 Figure 22–3 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 Figure 22–3, 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--->
000000000000000000000000
11 11111111111111111111111
 22 222
     333
     4444444444444444
```

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. Figure 22–4 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
*print-right-margin*    pprint-indent
copy-pprint-dispatch    pprint-linear
format    pprint-logical-block
formatter    pprint-newline
```

Figure 22–4. Defined names related to pretty printing.

Figure 22–5 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"    "M"    "<...>:>
"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
)"
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.

```lisp
(pprint-logical-block (*standard-output* nil :per-line-prefix ";;; ")
  (simple-pprint-defun *standard-output* '(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.

```
(defvar *indent* 1)
(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-logical-block (nil list :prefix "(" :suffix ")")
        (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)))
```

22–18 Programming Language—Common Lisp
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)
  (ppprint-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~∧
           ~}~:>~
           ~:\~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)"
       (~ - (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,-4:0s" ;This is an S directive with two parameters, 3 and -4,
"~,+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,

```lisp
(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,

```lisp
(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"
(format nil "~:@C" #\Control-Space) → "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,

```lisp
(format nil "~:@C" #\Control-Partial) → "Control-∂ (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.

```
\% prints the character in a way that the Lisp reader can understand, using \#\ syntax.
```

```
\% 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

```
\%nR 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-intervalR.
```

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 "", ',1,2:R" #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.

"mincolD 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,padcharD 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-intervalD.

"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-intervalB.

"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-interval0.

“D 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-intervalX.

“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,padcharF. 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 `~wD`, 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 \(\sim8.2E\) may correctly produce either 6.37E+2 or 6.38E+2.

Following the digit sequence, the exponent is printed. First the character parameter \(\text{exponentchar}\) is printed; if this parameter is omitted, then the exponent marker that \texttt{prin1} would use is printed, as determined from the type of the \texttt{float} and the current value of \texttt{*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 \(\text{overflowchar}\) is supplied, then \(w\) copies of that parameter are printed instead of the scaled value of \(arg\). If the \(\text{overflowchar}\) 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; \texttt{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 \(\sim\) directive always prints a plus or minus sign in front of the exponent, while \texttt{prin1} omits the plus sign if the exponent is non-negative.

If \(arg\) is a \textit{rational} number, then it is coerced to be a \texttt{single float} and then printed. Alternatively, an implementation is permitted to process a \texttt{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 \textit{complex} number or some non-numeric \textit{object}, then it is printed using the format directive \(\sim wD\), thereby printing it in decimal radix and a minimum field width of \(w\).
22.3.3.3 Tilde G: General Floating-Point

The next \textit{arg} is printed as a \textit{float} in either fixed-format or exponential notation as appropriate. The full form is \~\texttt{w, d, e, k, overflowchar, padchar, exponentchar}G. The format in which to print \textit{arg} depends on the magnitude (absolute value) of the \textit{arg}. Let \( n \) be an integer such that \( 10^{n-1} \leq |\textit{arg}| < 10^n \). Let \( ee \) equal \( e+2 \), or 4 if \( e \) is omitted. Let \( ww \) equal \( w-ee \), or \texttt{nil} if \( w \) is omitted. If \( d \) is omitted, first let \( q \) be the number of digits needed to print \textit{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 \( 0 \leq dd \leq d \), then \textit{arg} is printed as if by the format directives
\~\texttt{ww, dd, overflowchar, padchar, exponentchar}F\texttt{exT}

Note that the scale factor \( k \) is not passed to the \~\texttt{F} directive. For all other values of \( dd \), \textit{arg} is printed as if by the format directive
\~\texttt{w, d, e, k, overflowchar, padchar, exponentchar}E

In either case, an \texttt{@} modifier is supplied to the \~\texttt{F} or \~\texttt{E} directive if and only if one was supplied to the \~\texttt{G} directive.

\~\texttt{G} binds *\texttt{print-escape*} to \textit{false} and *\texttt{print-readably*} to \textit{false}.

22.3.3.4 Tilde Dollarsign: Monetary Floating-Point

The next \textit{arg} is printed as a \textit{float} in fixed-format notation. The full form is \~\texttt{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 \textit{arg} is negative, then a minus sign is printed; if the \textit{arg} is not negative, then a plus sign is printed if and only if the \texttt{@} modifier was supplied. If the \texttt{:} 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 \textit{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 \textit{arg}, with leading zeros if necessary; then a decimal point; then \( d \) digits of fraction, properly rounded.

If the magnitude of \textit{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 \~\texttt{w, q, , padchar}E, where \( w \) and \textit{padchar} are present or omitted according to whether they were present or omitted in the \~\texttt{G} directive, and where \( q=d+n-1 \), where \( d \) and \( n \) are the (possibly default) values given to the \~\texttt{G} directive.
If \( \text{arg} \) 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 \( \text{arg} \) is a complex number or some non-numeric object, then it is printed using the format directive \( \sim wD \), thereby printing it in decimal radix and a minimum field width of \( w \).

\( \sim S \) binds \(*\text{print-escape}*\) to false and \(*\text{print-readably}*\) to false.

### 22.3.4 FORMAT Printer Operations

#### 22.3.4.1 Tilde A: Aesthetic

An \( \text{arg} \), any object, is printed without escape characters (as by \texttt{princ}). If \( \text{arg} \) is a string, its characters will be output verbatim. If \( \text{arg} \) is \texttt{nil} it will be printed as \texttt{nil}; the colon modifier (\( \sim:A \)) will cause an \( \text{arg} \) of \texttt{nil} to be printed as \texttt{()}, but if \( \text{arg} \) is a composite structure, such as a list or vector, any contained occurrences of \texttt{nil} will still be printed as \texttt{nil}.

\( \sim \text{mincol}A \) inserts spaces on the right, if necessary, to make the width at least \( \text{mincol} \) columns. The \( \# \) modifier causes the spaces to be inserted on the left rather than the right.

\( \sim \text{mincol}, \text{colinc}, \text{minpad}, \text{padchar}A \) is the full form of \( \sim A \), 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 \( \text{minpad} \) copies of \texttt{padchar}; padding characters are then inserted \( \text{colinc} \) characters at a time until the total width is at least \( \text{mincol} \). The defaults are 0 for \( \text{mincol} \) and \( \text{minpad} \), 1 for \( \text{colinc} \), and the space character for \texttt{padchar}.

\( \sim A \) binds \(*\text{print-escape}*\) to false, and \(*\text{print-readably}*\) to false.

#### 22.3.4.2 Tilde S: Standard

This is just like \( \sim A \), but \( \text{arg} \) is printed with escape characters (as by \texttt{prin1} rather than \texttt{princ}). The output is therefore suitable for input to \texttt{read}. \( \sim S \) accepts all the arguments and modifiers that \( \sim A \) does.

\( \sim S \) binds \(*\text{print-escape}*\) to \texttt{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 # 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`, `~<...~:@>`, `~T`. As a result, an error is signaled if any of these directives is nested within `~<...~>`.

See also Section 22.3.6.2 (Tilde Less-Than-Sign: Justification).

### 22.3.5.3 Tilde I: Indent

```
~nI 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 an 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, colincT 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.

“colrel, colinc@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 `~;` 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.
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. `~*` 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.
```

22–36  Programming Language—Common Lisp
"@ 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:

```lisp
(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

```lisp
(format stream "...@[str...]..." ...) ≡ (format stream "...";";::*str";...) ...)
```

The combination of "[ and # is useful, for example, for dealing with English conventions for printing lists:

```lisp
(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:

```lisp
(format nil "The winners are:"{ "S")."}
```
'(fred harry jill))
→ "The winners are: FRED HARRY JILL."
(format nil "Pairs:{} {<~,S>,~S}>." 
'[(a b c))
→ "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 b c))
→ "Pairs: <A,1> <B,2> <C,3>.

`:str`) is similar, 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 arg-
uments 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 b c))
→ "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 "" string arguments)

This will use string as a formatting string. The `:t:` says it will be processed at most once,
and the `:t:` 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 `:t:` directive, but this
general feature of `:t:` is more powerful than `:t:`.
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) 7) → "<~A \D> 7"

(format nil "~? \D" ~(~A \D) 14 7) → "<~A \D> 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" ~(~@R~) "Foo" 5 7) → "<~A \D> 7"

(format nil "~@? \D" ~(~@R~) "Foo" 5 14 7) → "<~A \D> 7"

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-capitalise).

~:@ (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 (~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. ":^ 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 characters. With a : , the newline is ignored, but any following whitespace is left in place. With an @, the newline is left in place, but any following whitespace 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)) → F00
(foo 3.14159) → " 3.14| 31.42| 3.14| 3.1416| 3.14| 3.14159"
(foo 100.0) → "100.00|********|100.00| 100.0|100.00|100.00"
(foo 1234.0) → "1234.00|********|*******|1234.0|1234.00|1234.00"
(foo 0.006) → " 0.01| 0.06| 0.01| 0.006| 0.010| 0.006"

(defun foo (x)
 (format nil
 "~9,2,1,,'*E|~10,3,2,2,'?,,'$E|~9,3,2,-2,'%@E|~9,2E"
 x x x x x))
(foo 3.14159) → " 3.14E+0| 31.42$-01|+.003E+03| 3.14E+0"
(foo -3.14159) → "-3.14E+0|-31.42$-01|-3.003E+03|-3.14E+0"
(foo 100.0) → " 1.00E+3|11.00$+02|+.001E+06| 1.00E+3"
(foo 1100.0) → " 1.00L+3|11.00$+02|+.001L+06| 1.00L+3"
(foo 1.1E13) → "********|||11.00$+12|+.001E+16| 1.00E+13"
(foo 1.1L120) → "********|||********|11.00L+120"n
(foo 1.1L1200) → "********|||********|11.00L+120"

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|
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).
copy-pprint-dispatch

Function

Syntax:

copy-pprint-dispatch &optional table → new-table

Arguments and Values:

- **table**—a pprint dispatch table, or nil.
- **new-table**—a fresh pprint dispatch table.

Description:

Creates and returns a copy of the specified `table`, or of the value of `*print-pprint-dispatch*` if no `table` is specified, or of the initial value of `*print-pprint-dispatch*` if nil is specified.

Exceptional Situations:

Should signal an error of type `type-error` if `table` is not a pprint dispatch table.

formatter

Macro

Syntax:

formatter control-string → function

Arguments and Values:

- **control-string**—a format string; not evaluated.
- **function**—a function.

Description:

Returns a function which has behavior equivalent to:

```
(lambda (*standard-output* &rest arguments)
  (apply #'format t control-string arguments)
  arguments-tail)
```

where `arguments-tail` is either the tail of `arguments` which has as its car the argument that would be processed next if there were more format directives in the `control-string`, or else `nil` if no more `arguments` follow the most recently processed argument.

Examples:

```
(funcall (formatter "~&~A~A") *standard-output* 'a 'b 'c)
> AB
→ (C)
```
(format t (formatter ""&A"A") 'a 'b 'c)
▷ AB
→ NIL

Exceptional Situations:
Might signal an error (at macro expansion time or at run time) if the argument is not a valid
format string.

See Also:
format

pprint-dispatch  Function

Syntax:
    pprint-dispatch object &optional table  → function, found-p

Arguments and Values:
    object—an object.
    table—a pprint dispatch table, or nil. The default is the value of *print-pprint-dispatch*.
    function—a function designator.
    found-p—a generalized boolean.

Description:
Retrieves the highest priority function in table that is associated with a type specifier that
matches object. The function is chosen by finding all of the type specifiers in table that match
the object and selecting the highest priority function associated with any of these type speci-
fiers. If there is more than one highest priority function, an arbitrary choice is made. If no type
specifiers match the object, a function is returned that prints object using print-object.

The secondary value, found-p, is true if a matching type specifier was found in table, or false
otherwise.

If table is nil, retrieval is done in the initial pprint dispatch table.

Affected By:
The state of the table.

Exceptional Situations:
Should signal an error of type type-error if table is neither a pprint-dispatch-table nor nil.
Notes:

(let ((*print-pretty* t))
  (write object :stream s))
≡ (funcall (pprint-dispatch object) s object)

**pprint-exit-if-list-exhausted**

*Local Macro*

**Syntax:**

```
pprint-exit-if-list-exhausted (no arguments) → nil
```

**Description:**

Tests whether or not the *list* passed to the *lexically current logical block* has been exhausted; see Section 22.2.1.1 (Dynamic Control of the Arrangement of Output). If this *list* has been reduced to *nil*, **pprint-exit-if-list-exhausted** terminates the execution of the *lexically current logical block* except for the printing of the suffix. Otherwise **pprint-exit-if-list-exhausted** returns *nil*.

Whether or not **pprint-exit-if-list-exhausted** is *fbound* in the *global environment* is *implementation-dependent*; however, the restrictions on redefinition and shadowing of **pprint-exit-if-list-exhausted** are the same as for *symbols* in the **COMMON-LISP** package which are *fbound* in the *global environment*. The consequences of attempting to use **pprint-exit-if-list-exhausted** outside of **pprint-logical-block** are undefined.

**Exceptional Situations:**

An error is signaled (at macro expansion time or at run time) if **pprint-exit-if-list-exhausted** is used anywhere other than lexically within a call on **pprint-logical-block**. Also, the consequences of executing **pprint-if-list-exhausted** outside of the dynamic extent of the **pprint-logical-block** which lexically contains it are undefined.

**See Also:**

**pprint-logical-block**, **pprint-pop**.

---

22–48 Programming Language—Common Lisp
pprint-fill, pprint-linear, pprint-tabular

Syntax:

pprint-fill stream object &optional colon-p at-sign-p → nil
pprint-linear stream object &optional colon-p at-sign-p → nil
pprint-tabular stream object &optional colon-p at-sign-p tabsize → nil

Arguments and Values:

stream—an output stream designator.
object—an object.
colon-p—a generalized boolean. The default is true.
at-sign-p—a generalized boolean. The default is implementation-dependent.
tabsize—a non-negative integer. The default is 16.

Description:

The functions pprint-fill, pprint-linear, and pprint-tabular specify particular ways of pretty printing a list to stream. Each function prints parentheses around the output if and only if colon-p is true. Each function ignores its at-sign-p argument. (Both arguments are included even though only one is needed so that these functions can be used via "~/.../ and as set-pprint-dispatch functions, as well as directly.) Each function handles abbreviation and the detection of circularity and sharing correctly, and uses write to print object when it is a non-list.

If object is a list and if the value of *print-pretty* is false, each of these functions prints object using a minimum of whitespace, as described in Section 22.1.3.5 (Printing Lists and Conses). Otherwise (if object is a list and if the value of *print-pretty* is true):

- The function pprint-linear prints a list either all on one line, or with each element on a separate line.
- The function pprint-fill prints a list with as many elements as possible on each line.
- The function pprint-tabular is the same as pprint-fill except that it prints the elements so that they line up in columns. The tabsize specifies the column spacing in ems, which is the total spacing from the leading edge of one column to the leading edge of the next.

Examples:

Evaluating the following with a line length of 25 produces the output shown.

(progn (princ "Roads ")

Printer 22–49
(pprint-tabular *standard-output* '(elm main maple center) nil nil 8))
Roads ELM MAIN
MAPLE CENTER

Side Effects:
Performs output to the indicated stream.

Affected By:
The cursor position on the indicated stream, if it can be determined.

Notes:
The function pprint-tabular could be defined as follows:
(defun pprint-tabular (s list &optional (colon-p t) at-sign-p (tabsize nil))
  (declare (ignore at-sign-p))
  (when (null tabsize) (setq tabsize 16))
  (pprint-logical-block (s list :prefix (if colon-p "(" "")
                           :suffix (if colon-p ")" ""))
    (pprint-exit-if-list-exhausted)
    (loop (write (pprint-pop) :stream s)
      (pprint-exit-if-list-exhausted)
      (write-char #\Space s)
      (pprint-tab :section-relative 0 tabsize s)
      (pprint-newline :fill s))))

Note that it would have been inconvenient to specify this function using format, because of the need to pass its tabsize argument through to a ~:T format directive nested within an iteration over a list.

pprint-indent

Function

Syntax:
  pprint-indent relative-to n &optional stream → nil

Arguments and Values:
  relative-to—either :block or :current.
  n—a real.
  stream—an output stream designator. The default is standard output.
Description:

`pprint-indent` specifies the indentation to use in a logical block on `stream`. If `stream` is a pretty printing stream and the value of `*print-pretty*` is true, `pprint-indent` sets the indentation in the innermost dynamically enclosing logical block; otherwise, `pprint-indent` has no effect.

`N` specifies the indentation in ems. If `relative-to` is `:block`, the indentation is set to the horizontal position of the first character in the dynamically current logical block plus `n` ems. If `relative-to` is `:current`, the indentation is set to the current output position plus `n` ems. (For robustness in the face of variable-width fonts, it is advisable to use `:current` with an `n` of zero whenever possible.)

`N` can be negative; however, the total indentation cannot be moved left of the beginning of the line or left of the end of the rightmost per-line prefix—an attempt to move beyond one of these limits is treated the same as an attempt to move to that limit. Changes in indentation caused by `pprint-indent` do not take effect until after the next line break. In addition, in miser mode all calls to `pprint-indent` are ignored, forcing the lines corresponding to the logical block to line up under the first character in the block.

Exceptional Situations:

An error is signaled if `relative-to` is any object other than `:block` or `:current`.

See Also:

Section 22.3.5.3 (Tilde I: Indent)

**pprint-logical-block**

Macro

Syntax:

```
pprint-logical-block (stream-symbol object &key prefix per-line-prefix suffix)
  {declaration}*[form]*
→ nil
```

Arguments and Values:

- `stream-symbol`—a stream variable designator.
- `object`—an object; evaluated.
- `:prefix`—a string; evaluated. Complicated defaulting behavior; see below.
- `:per-line-prefix`—a string; evaluated. Complicated defaulting behavior; see below.
- `:suffix`—a string; evaluated. The default is the null string.
- `declaration`—a declare expression; not evaluated.
- `forms`—an implicit progn.
pprint-logical-block

Description:
Causes printing to be grouped into a logical block.

The logical block is printed to the stream that is the value of the variable denoted by stream-symbol. During the execution of the forms, that variable is bound to a pretty printing stream that supports decisions about the arrangement of output and then forwards the output to the destination stream. All the standard printing functions (e.g., write, prin, and terpri) can be used to print output to the pretty printing stream. All and only the output sent to this pretty printing stream is treated as being in the logical block.

The prefix specifies a prefix to be printed before the beginning of the logical block. The per-line-prefix specifies a prefix that is printed before the block and at the beginning of each new line in the block. The :prefix and :pre-line-prefix arguments are mutually exclusive. If neither :prefix nor :pre-line-prefix is specified, a prefix of the null string is assumed.

The suffix specifies a suffix that is printed just after the logical block.

The object is normally a list that the body forms are responsible for printing. If object is not a list, it is printed using write. (This makes it easier to write printing functions that are robust in the face of malformed arguments.) If *print-circle* is non-nil and object is a circular (or shared) reference to a cons, then an appropriate "#N" marker is printed. (This makes it easy to write printing functions that provide full support for circularity and sharing abbreviation.) If *print-level* is not nil and the logical block is at a dynamic nesting depth of greater than *print-level* in logical blocks, "#" is printed. (This makes easy to write printing functions that provide full support for depth abbreviation.)

If either of the three conditions above occurs, the indicated output is printed on stream-symbol and the body forms are skipped along with the printing of the :prefix and :suffix. (If the body forms are not to be responsible for printing a list, then the first two tests above can be turned off by supplying nil for the object argument.)

In addition to the object argument of pprint-logical-block, the arguments of the standard printing functions (such as write, print, prin1, and pprint, as well as the arguments of the standard format directives such as ~A, ~S, (and ~W)) are all checked (when necessary) for circularity and sharing. However, such checking is not applied to the arguments of the functions write-line, write-string, and write-char or to the literal text output by format. A consequence of this is that you must use one of the latter functions if you want to print some literal text in the output that is not supposed to be checked for circularity or sharing.

The body forms of a pprint-logical-block form must not perform any side-effects on the surrounding environment; for example, no variables must be assigned which have not been bound within its scope.

The pprint-logical-block macro may be used regardless of the value of *print-pretty*.

Affected By:

*print-circle*, *print-level*.
Exceptional Situations:

An error of type **type-error** is signaled if any of the :suffix, :prefix, or :per-line-prefix is supplied but does not evaluate to a string.

An error is signaled if :prefix and :pre-line-prefix are both used.

**pprint-logical-block** and the pretty printing stream it creates have dynamic extent. The consequences are undefined if, outside of this extent, output is attempted to the pretty printing stream it creates.

It is also unspecified what happens if, within this extent, any output is sent directly to the underlying destination stream.

See Also:

pprint-pop, pprint-exit-if-list-exhausted, Section 22.3.5.2 (Tilde Less-Than-Sign: Logical Block)

Notes:

One reason for using the **pprint-logical-block** macro when the value of *print-pretty* is nil would be to allow it to perform checking for dotted lists, as well as (in conjunction with pprint-pop) checking for *print-level* or *print-length* being exceeded.

Detection of circularity and sharing is supported by the pretty printer by in essence performing requested output twice. On the first pass, circularities and sharing are detected and the actual outputting of characters is suppressed. On the second pass, the appropriate “#n” and “#n#” markers are inserted and characters are output. This is why the restriction on side-effects is necessary. Obeying this restriction is facilitated by using pprint-pop, instead of an ordinary pop when traversing a list being printed by the body forms of the **pprint-logical-block** form.)

---

**pprint-newline**

*Function*

Syntax:

`pprint-newline kind &optional stream → nil`

Arguments and Values:

kind—one of :linear, :fill, :miser, or :mandatory.

stream—a stream designator. The default is standard output.

Description:

If stream is a pretty printing stream and the value of *print-pretty* is true, a line break is inserted in the output when the appropriate condition below is satisfied; otherwise, pprint-newline has no effect.

Kind specifies the style of conditional newline. This parameter is treated as follows:
pprint-newline

:linear

This specifies a “linear-style” conditional newline. A line break is inserted if and only if the immediately containing section cannot be printed on one line. The effect of this is that line breaks are either inserted at every linear-style conditional newline in a logical block or at none of them.

:miser

This specifies a “miser-style” conditional newline. A line break is inserted if and only if the immediately containing section cannot be printed on one line and miser style is in effect in the immediately containing logical block. The effect of this is that miser-style conditional newlines act like linear-style conditional newlines, but only when miser style is in effect. Miser style is in effect for a logical block if and only if the starting position of the logical block is less than or equal to *print-miser-width* ems from the right margin.

:fill

This specifies a “fill-style” conditional newline. A line break is inserted if and only if either (a) the following section cannot be printed on the end of the current line, (b) the preceding section was not printed on a single line, or (c) the immediately containing section cannot be printed on one line and miser style is in effect in the immediately containing logical block. If a logical block is broken up into a number of subsections by fill-style conditional newlines, the basic effect is that the logical block is printed with as many subsections as possible on each line. However, if miser style is in effect, fill-style conditional newlines act like linear-style conditional newlines.

:mandatory

This specifies a “mandatory-style” conditional newline. A line break is always inserted. This implies that none of the containing sections can be printed on a single line and will therefore trigger the insertion of line breaks at linear-style conditional newlines in these sections.

When a line break is inserted by any type of conditional newline, any blanks that immediately precede the conditional newline are omitted from the output and indentation is introduced at the beginning of the next line. By default, the indentation causes the following line to begin in the same horizontal position as the first character in the immediately containing logical block. (The indentation can be changed via pprint-indent.)

There are a variety of ways unconditional newlines can be introduced into the output (i.e., via terpri or by printing a string containing a newline character). As with mandatory conditional newlines, this prevents any of the containing sections from being printed on one line. In general, when an unconditional newline is encountered, it is printed out without suppression of the preceding blanks and without any indentation following it. However, if a per-line prefix has been specified (see pprint-logical-block), this prefix will always be printed no matter how a newline originates.
Examples:

See Section 22.2.2 (Examples of using the Pretty Printer).

Side Effects:

Output to stream.

Affected By:

*print-pretty*, *print-miser*. The presence of containing logical blocks. The placement of newlines and conditional newlines.

Exceptional Situations:

An error of type type-error is signaled if kind is not one of :linear, :fill, :miser, or :mandatory.

See Also:

Section 22.3.5.1 (Tilde Underscore: Conditional Newline), Section 22.2.2 (Examples of using the Pretty Printer)

---

**pprint-pop**

Local Macro

Syntax:

pprint-pop (no arguments) → object

Arguments and Values:

object—an element of the list being printed in the lexically current logical block, or nil.

Description:

Pops one element from the list being printed in the lexically current logical block, obeying *print-length* and *print-circle* as described below.

Each time pprint-pop is called, it pops the next value off the list passed to the lexically current logical block and returns it. However, before doing this, it performs three tests:

- If the remaining ‘list’ is not a list, “. ” is printed followed by the remaining ‘list.’ (This makes it easier to write printing functions that are robust in the face of malformed arguments.)

- If *print-length* is non-nil, and pprint-pop has already been called *print-length* times within the immediately containing logical block, “...” is printed. (This makes it easy to write printing functions that properly handle *print-length*.)
• If *print-circle* is non-nil, and the remaining list is a circular (or shared) reference, then “. . .” is printed followed by an appropriate “#*n*” marker. (This catches instances of cdr circularity and sharing in lists.)

If either of the three conditions above occurs, the indicated output is printed on the pretty printing stream created by the immediately containing `pprint-logical-block` and the execution of the immediately containing `pprint-logical-block` is terminated except for the printing of the suffix.

If `pprint-logical-block` is given a ‘list’ argument of nil—because it is not processing a list—`pprint-pop` can still be used to obtain support for *print-length*. In this situation, the first and third tests above are disabled and `pprint-pop` always returns nil. See Section 22.2.2 (Examples of using the Pretty Printer)—specifically, the `pprint-vector` example.

Whether or not `pprint-pop` is `fbound` in the global environment is implementation-dependent; however, the restrictions on redefinition and shadowing of `pprint-pop` are the same as for symbols in the Common-Lisp package which are `fbound` in the global environment. The consequences of attempting to use `pprint-pop` outside of `pprint-logical-block` are undefined.

Side Effects:

Might cause output to the pretty printing stream associated with the lexically current logical block.

Affected By:

*print-length*, *print-circle*.

Exceptional Situations:

An error is signaled (either at macro expansion time or at run time) if a usage of `pprint-pop` occurs where there is no lexically containing `pprint-logical-block` form.

The consequences are undefined if `pprint-pop` is executed outside of the dynamic extent of this `pprint-logical-block`.

See Also:

`pprint-exit-if-list-exhausted`, `pprint-logical-block`.

Notes:

It is frequently a good idea to call `pprint-exit-if-list-exhausted` before calling `pprint-pop`. 
pprint-tab

Syntax:

```lisp
(pprint-tab kind colnum colinc &optional stream) → nil
```

Arguments and Values:

- `kind`: one of :line, :section, :line-relative, or :section-relative.
- `colnum`: a non-negative integer.
- `colinc`: a non-negative integer.
- `stream`: an output stream designator.

Description:

Specifies tabbing to `stream` as performed by the standard `~T` format directive. If `stream` is a pretty printing stream and the value of `*print-pretty*` is true, tabbing is performed; otherwise, `pprint-tab` has no effect.

The arguments `colnum` and `colinc` correspond to the two parameters to `~T` and are in terms of ems. The `kind` argument specifies the style of tabbing. It must be one of :line (tab as by `~T`), :section (tab as by `~:T`, but measuring horizontal positions relative to the start of the dynamically enclosing section), :line-relative (tab as by `~@T`), or :section-relative (tab as by `~:@T`, but measuring horizontal positions relative to the start of the dynamically enclosing section).

Exceptional Situations:

An error is signaled if `kind` is not one of :line, :section, :line-relative, or :section-relative.

See Also:

- `pprint-logical-block`

print-object

Standard Generic Function

Syntax:

```lisp
(print-object object stream) → object
```

Method Signatures:

```
(print-object (object standard-object) stream)
(print-object (object structure-object) stream)
```
Arguments and Values:

*object*—an object.

*stream*—a stream.

Description:

The generic function print-object writes the printed representation of object to stream. The function print-object is called by the Lisp printer; it should not be called by the user.

Each implementation is required to provide a method on the class standard-object and on the class structure-object. In addition, each implementation must provide methods on enough other classes so as to ensure that there is always an applicable method. Implementations are free to add methods for other classes. Users may write methods for print-object for their own classes if they do not wish to inherit an implementation-dependent method.

The method on the class structure-object prints the object in the default #S notation; see Section 22.1.3.12 (Printing Structures).

Methods on print-object are responsible for implementing their part of the semantics of the printer control variables, as follows:

*print-readable*

All methods for print-object must obey *print-readable*. This includes both user-defined methods and implementation-defined methods. Readable printing of structures and standard objects is controlled by their print-object method, not by their make-load-form method. Similarity for these objects is application dependent and hence is defined to be whatever these methods do; see Section 3.2.4.2 (Similarity of Literal Objects).

*print-escape*

Each method must implement *print-escape*.

*print-pretty*

The method may wish to perform specialized line breaking or other output conditional on the value of *print-pretty*. For further information, see (for example) the macro pprint-fill. See also Section 22.2.1.4 (Pretty Print Dispatch Tables) and Section 22.2.2 (Examples of using the Pretty Printer).

*print-length*

Methods that produce output of indefinite length must obey *print-length*. For further information, see (for example) the macros pprint-logical-block and pprint-pop. See also Section 22.2.1.4 (Pretty Print Dispatch Tables) and Section 22.2.2 (Examples of using the Pretty Printer).
The printer takes care of *print-level* automatically, provided that each *method* handles exactly one level of structure and calls `write` (or an equivalent function) recursively if there are more structural levels. The printer’s decision of whether an object has components (and therefore should not be printed when the printing depth is not less than *print-level*) is implementation-dependent. In some implementations its print-object method is not called; in others the method is called, and the determination that the object has components is based on what it tries to write to the *stream*.

When the value of *print-circle* is true, a user-defined print-object method can print objects to the supplied stream using `write`, `prin1`, `princ`, or `format` and expect circularities to be detected and printed using the `#n#` syntax. If a user-defined print-object method prints to a stream other than the one that was supplied, then circularity detection starts over for that stream. See *print-circle*.

These printer control variables apply to specific types of objects and are handled by the methods for those objects.

If these rules are not obeyed, the results are undefined.

In general, the printer and the print-object methods should not rebind the print control variables as they operate recursively through the structure, but this is implementation-dependent.

In some implementations the stream argument passed to a print-object method is not the original stream, but is an intermediate stream that implements part of the printer. Methods should therefore not depend on the identity of this stream.

See Also:
pprint-fill, pprint-logical-block, pprint-pop, write, *print-readably*, *print-escape*, *print-pretty*, *print-length*, Section 22.1.3 (Default Print-Object Methods), Section 22.1.3.12 (Printing Structures), Section 22.2.1.4 (Pretty Print Dispatch Tables), Section 22.2.2 (Examples of using the Pretty Printer)

---

**print-unreadable-object**

Macro

**Syntax:**
```lisp
print-unreadable-object (object stream &key type identity) {form}*  → nil
```
Arguments and Values:

- object—an object; evaluated.
- stream—a stream designator; evaluated.
- type—a generalized boolean; evaluated.
- identity—a generalized boolean; evaluated.
- forms—an implicit progn.

Description:

Outputs a printed representation of object on stream, beginning with “#<” and ending with “>”. Everything output to stream by the body forms is enclosed in the the angle brackets. If type is true, the output from forms is preceded by a brief description of the object’s type and a space character. If identity is true, the output from forms is followed by a space character and a representation of the object’s identity, typically a storage address.

If either type or identity is not supplied, its value is false. It is valid to omit the body forms. If type and identity are both true and there are no body forms, only one space character separates the type and the identity.

Examples:

;; Note that in this example, the precise form of the output ;; is implementation-dependent.

(defun print-object ((obj airplane) stream)
  (print-unreadable-object (obj stream :type t :identity t)
    (princ (tail-number obj) stream)))

(prin1-to-string my-airplane)
→ "#<Airplane NW0773 36000123135>
or "#<FAA:AIRPLANE NW0773 17>"

Exceptional Situations:

If *print-readably* is true, print-unreadable-object signals an error of type print-not-readable without printing anything.

set-pprint-dispatch

Function

Syntax:

set-pprint-dispatch type-specifier function &optional priority table → nil

Arguments and Values:

- type-specifier—a type specifier.
**function**—a function, a function name, or nil.

**priority**—a real. The default is 0.

**table**—a pprint dispatch table. The default is the value of *print-pprint-dispatch*.

**Description:**
Installs an entry into the pprint dispatch table which is table.

**Type-specifier** is the key of the entry. The first action of set-pprint-dispatch is to remove any pre-existing entry associated with type-specifier. This guarantees that there will never be two entries associated with the same type specifier in a given pprint dispatch table. Equality of type specifiers is tested by equal.

Two values are associated with each type specifier in a pprint dispatch table: a function and a priority. The function must accept two arguments: the stream to which output is sent and the object to be printed. The function should pretty print the object to the stream. The function can assume that object satisfies the type given by type-specifier. The function must obey *print-readably*. Any values returned by the function are ignored.

Priority is a priority to resolve conflicts when an object matches more than one entry.

It is permissible for function to be nil. In this situation, there will be no type-specifier entry in table after set-pprint-dispatch returns.

**Exceptional Situations:**
An error is signaled if priority is not a real.

**Notes:**
Since pprint dispatch tables are often used to control the pretty printing of Lisp code, it is common for the type-specifier to be an expression of the form

(cons car-type cdr-type)

This signifies that the corresponding object must be a cons cell whose car matches the type specifier car-type and whose cdr matches the type specifier cdr-type. The cdr-type can be omitted in which case it defaults to t.

---

**write, prin1, print, pprint, princ**

**Function**

**Syntax:**

```
write object &key array base case circle escape gensym
  length level lines miser-width pprint-dispatch
  pretty radix readably right-margin stream
```
write, prin1, print, pprint, princ

→ object

prin1 object &optional output-stream → object
princ object &optional output-stream → object
print object &optional output-stream → object
pprint object &optional output-stream → ⟨ ⟨no values⟩ ⟩

Arguments and Values:
object—an object.
output-stream—an output stream designator. The default is standard output.
array—a generalized boolean.
base—a radix.
case—a symbol of type (member :upcase :downcase :capitalize).
circle—a generalized boolean.
escape—a generalized boolean.
gensym—a generalized boolean.
length—a non-negative integer, or nil.
level—a non-negative integer, or nil.
lines—a non-negative integer, or nil.
miser-width—a non-negative integer, or nil.
pprint-dispatch—a pprint dispatch table.
pretty—a generalized boolean.
radius—a generalized boolean.
readably—a generalized boolean.
right-margin—a non-negative integer, or nil.
stream—an output stream designator. The default is standard output.

Description:
write, prin1, prin1, print, and pprint write the printed representation of object to output-stream.
write is the general entry point to the Lisp printer. For each explicitly supplied keyword param-
write, prin1, print, pprint, princ

eter named in Figure 22–7, the corresponding printer control variable is dynamically bound to its value while printing goes on; for each keyword parameter in Figure 22–7 that is not explicitly supplied, the value of the corresponding printer control variable is the same as it was at the time write was invoked. Once the appropriate bindings are established, the object is output by the Lisp printer.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Corresponding Dynamic Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td><em>print-array</em></td>
</tr>
<tr>
<td>base</td>
<td><em>print-base</em></td>
</tr>
<tr>
<td>case</td>
<td><em>print-case</em></td>
</tr>
<tr>
<td>circle</td>
<td><em>print-circle</em></td>
</tr>
<tr>
<td>escape</td>
<td><em>print-escape</em></td>
</tr>
<tr>
<td>gensym</td>
<td><em>print-gensym</em></td>
</tr>
<tr>
<td>length</td>
<td><em>print-length</em></td>
</tr>
<tr>
<td>level</td>
<td><em>print-level</em></td>
</tr>
<tr>
<td>lines</td>
<td><em>print-lines</em></td>
</tr>
<tr>
<td>miser-width</td>
<td><em>print-miser-width</em></td>
</tr>
<tr>
<td>pprint-dispatch</td>
<td><em>print-pprint-dispatch</em></td>
</tr>
<tr>
<td>pretty</td>
<td><em>print-pretty</em></td>
</tr>
<tr>
<td>radix</td>
<td><em>print-radix</em></td>
</tr>
<tr>
<td>readably</td>
<td><em>print-readably</em></td>
</tr>
<tr>
<td>right-margin</td>
<td><em>print-right-margin</em></td>
</tr>
</tbody>
</table>

Figure 22–7. Argument correspondences for the WRITE function.

prin1, princ, print, and pprint implicitly bind certain print parameters to particular values. The remaining parameter values are taken from *print-array*, *print-base*, *print-case*, *print-circle*, *print-escape*, *print-gensym*, *print-length*, *print-level*, *print-lines*, *print-miser-width*, *print-pprint-dispatch*, *print-pretty*, *print-radix*, and *print-right-margin*.

prin1 produces output suitable for input to read. It binds *print-escape* to true.

princ is just like prin1 except that the output has no escape characters. It binds *print-escape* to false and *print-readably* to false. The general rule is that output from princ is intended to look good to people, while output from prin1 is intended to be acceptable to read.

print is just like prin1 except that the printed representation of object is preceded by a newline and followed by a space.

pprint is just like print except that the trailing space is omitted and object is printed with the *print-pretty* flag non-nil to produce pretty output.

Output-stream specifies the stream to which output is to be sent.

Printer 22–63
Affected By:
*standard-output*, *terminal-io*, *print-escape*, *print-radix*, *print-base*, *print-circle*, *print-pretty*, *print-level*, *print-length*, *print-case*, *print-gensym*, *print-array*, *read-default-float-format*.

See Also:
readtable-case, Section 22.3.4 (FORMAT Printer Operations)

Notes:
The functions prin1 and print do not bind *print-readably*.

\[(prin1 object output-stream) ≡ (write object :stream output-stream :escape t)\]

\[(princ object output-stream) ≡ (write object :stream output-stream :escape nil :readably nil)\]

\[(print object output-stream) \equiv (progn (terpri output-stream)
 (write object :stream output-stream :escape t)
 (write-char #\space output-stream))\]

\[(pprint object output-stream) \equiv (write object :stream output-stream :escape t :pretty t)\]

---

**write-to-string, prin1-to-string, princ-to-string**

*Function*

**Syntax:**

\[
\text{write-to-string} \quad \text{object} \quad \&\text{key} \quad \text{array base case circle escape gensym length level lines miser-width pprint-dispatch pretty radix readably right-margin}
\]

\[\rightarrow \text{string}\]

\[
\text{prin1-to-string} \quad \text{object} \quad \rightarrow \text{string}\]

\[
\text{princ-to-string} \quad \text{object} \quad \rightarrow \text{string}\]

22–64  Programming Language—Common Lisp
write-to-string, prin1-to-string, princ-to-string

Arguments and Values:

object—an object.
array—a generalized boolean.
base—a radix.
case—a symbol of type (member :upcase :downcase :capitalize).
circle—a generalized boolean.
escape—a generalized boolean.
gensym—a generalized boolean.
length—a non-negative integer, or nil.
level—a non-negative integer, or nil.
lines—a non-negative integer, or nil.
miser-width—a non-negative integer, or nil.
print-dispatch—a pprint dispatch table.
pretty—a generalized boolean.
radix—a generalized boolean.
readably—a generalized boolean.
right-margin—a non-negative integer, or nil.
string—a string.

Description:

write-to-string, prin1-to-string, and princ-to-string are used to create a string consisting of the printed representation of object. Object is effectively printed as if by write, prin1, or princ, respectively, and the characters that would be output are made into a string.

write-to-string is the general output function. It has the ability to specify all the parameters applicable to the printing of object.

prin1-to-string acts like write-to-string with :escape t, that is, escape characters are written where appropriate.

princ-to-string acts like write-to-string with :escape nil :readably nil. Thus no escape characters are written.

All other keywords that would be specified to write-to-string are default values when prin1-to-string or princ-to-string is invoked.
The meanings and defaults for the keyword arguments to write-to-string are the same as those for write.

Examples:

```lisp
(prin1-to-string "abc") -> "\"abc\"
(princ-to-string "abc") -> "abc"
```

Affected By:

*print-escape*, *print-radix*, *print-base*, *print-circle*, *print-pretty*, *print-level*, *print-length*, *print-case*, *print-gensym*, *print-array*, *read-default-float-format*.

See Also:

write

Notes:

```lisp
(write-to-string object {key argument}*)
≡ (with-output-to-string (#i=#:string-stream)
   (write object :stream #1# {key argument}*))

(princ-to-string object)
≡ (with-output-to-string (string-stream)
   (princ object string-stream))

(prin1-to-string object)
≡ (with-output-to-string (string-stream)
   (prin1 object string-stream))
```

### *print-array*

**Variable**

**Value Type:**

a generalized boolean.

**Initial Value:**

implementation-dependent.

**Description:**

Controls the format in which arrays are printed. If it is false, the contents of arrays other than strings are never printed. Instead, arrays are printed in a concise form using #< that gives enough information for the user to be able to identify the array, but does not include the entire array contents. If it is true, non-string arrays are printed using #(...), #*, or #nA syntax.
**print-base**, **print-radix**

**Value Type:**

*print-base*—a radix. *print-radix*—a generalized boolean.

**Initial Value:**

The initial value of *print-base* is 10. The initial value of *print-radix* is false.

**Description:**

*print-base* and *print-radix* control the printing of rationals. The value of *print-base* is called the current output base.

The value of *print-base* is the radix in which the printer will print rationals. For radices above 10, letters of the alphabet are used to represent digits above 9.

If the value of *print-radix* is true, the printer will print a radix specifier to indicate the radix in which it is printing a rational number. The radix specifier is always printed using lowercase letters. If *print-base* is 2, 8, or 16, then the radix specifier used is #b, #o, or #x, respectively.

For integers, base ten is indicated by a trailing decimal point instead of a leading radix specifier; for ratios, #10r is used.

**Examples:**

```
(let ((print-base 24.) (print-radix t))
  (print 23.))
▷ #24rN
23
```

```
(setq *print-base* 10) → 10
(setq *print-radix* nil) → NIL
(dotimes (i 35)
  (let ((print-base (+ i 2))) ;print the decimal number 40
    (write 40) ;in each base from 2 to 36
    (if (zerop (mod i 10)) (terpri) (format t " "))))
▷ 101000
▷ 1111 220 130 104 55 50 44 40 37 34
▷ 31 2C 2A 28 26 24 22 20 1J 1I
▷ 1H 1G 1F 1E 1D 1C 1B 1A 19 18
```
(dolist (pb '(2 3 8 10 16))
  (let ((*print-radix* t) (*print-base* pb))
    (format t "~&~S ~S~%" 10 (1/10))))

→ #b1010 #b1/1010
→ #3r101 #3r1/101
→ #o12 #o1/12
→ 10. #10r1/10
→ #xA #x1/A
→ NIL

Affected By:
Might be bound by format, and write, write-to-string.

See Also:
format, write, write-to-string

*print-case*  Variable

Value Type:
One of the symbols :upcase, :downcase, or :capitalize.

Initial Value:
The symbol :upcase.

Description:
The value of *print-case* controls the case (upper, lower, or mixed) in which to print any uppercase characters in the names of symbols when vertical-bar syntax is not used.

*print-case* has an effect at all times when the value of *print-escape* is false. *print-case* also has an effect when the value of *print-escape* is true unless inside an escape context (i.e., unless between vertical-bars or after a slash).

Examples:
(defun test-print-case ()
  (dolist (*print-case* '(:upcase :downcase :capitalize))
    (format t "~&" "~S "~S" "~%" 'this-and-that '|And-something-else|)))
→ TEST-PC

; Although the choice of which characters to escape is specified by
; *PRINT-CASE*, the choice of how to escape those characters

22–68  Programming Language—Common Lisp
(i.e., whether single escapes or multiple escapes are used) is implementation-dependent. The examples here show two of the many valid ways in which escaping might appear.

(test-print-case) ;Implementation A
⊿ THIS-AND-THAT |And-something-else|
⊿ this-and-that a\n\d-\s\l\o\m\e\t\h\i\n\g-\e\l\se
⊿ This-And-That A\n\d-\s\l\o\m\e\t\h\i\n\g-\e\l\se
→ NIL

(test-print-case) ;Implementation B
⊿ THIS-AND-THAT |And-something-else|
⊿ this-and-that a\n\d-\s\l\o\m\e\t\h\i\n\g-\e\l\se
⊿ This-And-That A\n\d-\s\l\o\m\e\t\h\i\n\g-\e\l\se
→ NIL

See Also:
write

Notes:
read normally converts lowercase characters appearing in symbols to corresponding uppercase characters, so that internally print names normally contain only uppercase characters.

If *print-escape* is true, lowercase characters in the name of a symbol are always printed in lowercase, and are preceded by a single escape character or enclosed by multiple escape characters; uppercase characters in the name of a symbol are printed in upper case, in lower case, or in mixed case so as to capitalize words, according to the value of *print-case*. The convention for what constitutes a “word” is the same as for string-capitalize.

*print-circle*

Variable

Value Type:
a generalized boolean.

Initial Value:
false.

Description:
Controls the attempt to detect circularity and sharing in an object being printed.

If false, the printing process merely proceeds by recursive descent without attempting to detect circularity and sharing.

If true, the printer will endeavor to detect cycles and sharing in the structure to be printed, and to use #\# syntax to indicate the circularities or shared components.
If true, a user-defined print-object method can print objects to the supplied stream using write, prin1, princ, or format and expect circularities and sharing to be detected and printed using the #\# syntax. If a user-defined print-object method prints to a stream other than the one that was supplied, then circularity detection starts over for that stream.

Note that implementations should not use #\# notation when the Lisp reader would automatically assure sharing without it (e.g., as happens with interned symbols).

Examples:

(let ((a (list 1 2 3)))
 (setf (cdddr a) a)
 (let ((*print-circle* t))
   (write a)
   :done))
▷ #1=(1 2 3 . #1#)
→ :DONE

See Also:
write

Notes:
An attempt to print a circular structure with *print-circle* set to nil may lead to looping behavior and failure to terminate.

*print-escape*  

Variable

Value Type:
a generalized boolean.

Initial Value:
true.

Description:
If false, escape characters and package prefixes are not output when an expression is printed.

If true, an attempt is made to print an expression in such a way that it can be read again to produce an equal expression. (This is only a guideline; not a requirement. See *print-readably*.)

For more specific details of how the value of *print-escape* affects the printing of certain types, see Section 22.1.3 (Default Print-Object Methods).
Examples:

(let ((*print-escape* t)) (write #\a))
▷ #\a
→ #\a

(let ((*print-escape* nil)) (write #\a))
▷ a
→ #\a

Affected By:
princ, prin1, format

See Also:
write, readtable-case

Notes:
princ effectively binds *print-escape* to false. prin1 effectively binds *print-escape* to true.

*print-gensym*  

Value Type:
a generalized boolean.

Initial Value:
true.

Description:
Controls whether the prefix “#:” is printed before apparently uninterned symbols. The prefix is printed before such symbols if and only if the value of *print-gensym* is true.

Examples:

(let ((*print-gensym* nil))
 (print (gensym)))
▷ G6040
→ #:G6040

See Also:
write, *print-escape*
**print-level*, **print-length**

### Value Type:
A non-negative integer, or nil.

### Initial Value:
Nil.

### Description:
**print-level** controls how many levels deep a nested object will print. If it is false, then no control is exercised. Otherwise, it is an integer indicating the maximum level to be printed. An object to be printed is at level 0; its components (as of a list or vector) are at level 1; and so on. If an object to be recursively printed has components and is at a level equal to or greater than the value of **print-level**, then the object is printed as “#”.

**print-length** controls how many elements at a given level are printed. If it is false, there is no limit to the number of components printed. Otherwise, it is an integer indicating the maximum number of elements of an object to be printed. If exceeded, the printer will print “...” in place of the other elements. In the case of a dotted list, if the list contains exactly as many elements as the value of **print-length**, the terminating atom is printed rather than printing “...”

**print-level** and **print-length** affect the printing of any object printed with a list-like syntax. They do not affect the printing of symbols, strings, and bit vectors.

### Examples:

```lisp
(setq a '(1 (2 (3 (4 (5 (6))))))) → (1 (2 (3 (4 (5 (6))))))
(dotimes (i 8)
   (let ((**print-level** i))
     (format t "~D -- ~S~%" i a))
   ∆ 0 -- #
   ∆ 1 -- (1 #)
   ∆ 2 -- (1 (2 #))
   ∆ 3 -- (1 (2 (3 #)))
   ∆ 4 -- (1 (2 (3 (4 #))))
   ∆ 5 -- (1 (2 (3 (4 (5 #)))))
   ∆ 6 -- (1 (2 (3 (4 (5 (6)))))
   ∆ 7 -- (1 (2 (3 (4 (5 (6))))))
   → NIL)

(setq a '(1 2 3 4 5 6)) → (1 2 3 4 5 6)
(dotimes (i 7)
   (let ((**print-length** i))
     (format t "~D -- ~S~%" i a))
   ∆ 0 -- #
   ∆ 1 -- (1 #)
   ∆ 2 -- (1 2 #)
   ∆ 3 -- (1 2 3 #)
   ∆ 4 -- (1 2 3 4 #)
   ∆ 5 -- (1 2 3 4 5 #)
   ∆ 6 -- (1 2 3 4 5 6 #)
   ∆ 7 -- (1 2 3 4 5 6 #)
   → NIL)
```

22–72 Programming Language—Common Lisp
(format t "^&D -- "&"%" i a)))
▷ 0 -- (....)
▷ 1 -- (1 ....)
▷ 2 -- (1 2 ....)
▷ 3 -- (1 2 3 ....)
▷ 4 -- (1 2 3 4 ....)
▷ 5 -- (1 2 3 4 5 6)
▷ 6 -- (1 2 3 4 5 6)
→ NIL

(dolist (level-length '((0 1) (1 1) (1 2) (1 3) (1 4)
                      (2 1) (2 2) (2 3) (3 2) (3 3) (3 4)))
  (let ((*print-level* (first level-length))
       (*print-length* (second level-length)))
    (format t "^&D "&"D -- "&"%"^"%"^"%"
       *print-level* *print-length* ^
                '(if (member x y) (+ (car x) 3) '(foo . #(a b c d "Baz"))))))
▷ 0 1 -- #
▷ 1 1 -- (IF ....)
▷ 1 2 -- (IF # ....)
▷ 1 3 -- (IF # # ....)
▷ 1 4 -- (IF # # #)
▷ 2 1 -- (IF ....)
▷ 2 2 -- (IF (MEMBER X ...) ....)
▷ 2 3 -- (IF (MEMBER X Y) (+ # 3) ....)
▷ 3 2 -- (IF (MEMBER X ...) ....)
▷ 3 3 -- (IF (MEMBER X Y) (+ (CAR X) 3) ....)
▷ 3 4 -- (IF (MEMBER X Y) (+ (CAR X) 3) '(FOO . #(A B C D ...)))
→ NIL

See Also:
write

*print-lines*  Variable

Value Type:
a non-negative integer, or nil.

Initial Value:
nil.
Description:
When the value of *print-lines* is other than nil, it is a limit on the number of output lines produced when something is pretty printed. If an attempt is made to go beyond that many lines, “..” is printed at the end of the last line followed by all of the suffixes (closing delimiters) that are pending to be printed.

Examples:

```lisp
(let ((*print-right-margin* 25) (*print-lines* 3))
  (pprint '(progn (setq a 1 b 2 c 3 d 4))))
▷ (PROGN (SETQ A 1
▷ B 2
▷ C 3 ..))
→ (no values)
```

Notes:
The “..” notation is intentionally different than the “...” notation used for level abbreviation, so that the two different situations can be visually distinguished.

This notation is used to increase the likelihood that the Lisp reader will signal an error if an attempt is later made to read the abbreviated output. Note however that if the truncation occurs in a string, as in "This string has been trunc..", the problem situation cannot be detected later and no such error will be signaled.

*print-miser-width*

Variable

Value Type:
a non-negative integer, or nil.

Initial Value:
implementation-dependent

Description:
If it is not nil, the pretty printer switches to a compact style of output (called miser style) whenever the width available for printing a substructure is less than or equal to this many ems.
*print-pprint-dispatch*  

Value Type:  
a pprint dispatch table.

Initial Value:  
implementation-dependent, but the initial entries all use a special class of priorities that have the property that they are less than every priority that can be specified using set-pprint-dispatch, so that the initial contents of any entry can be overridden.

Description:  
The pprint dispatch table which currently controls the pretty printer.

See Also:  
*print-pretty*, Section 22.2.1.4 (Pretty Print Dispatch Tables)

Notes:  
The intent is that the initial value of this variable should cause ‘traditional’ pretty printing of code. In general, however, you can put a value in *print-pprint-dispatch* that makes pretty-printed output look exactly like non-pretty-printed output. Setting *print-pretty* to true just causes the functions contained in the current pprint dispatch table to have priority over normal print-object methods; it has no magic way of enforcing that those functions actually produce pretty output. For details, see Section 22.2.1.4 (Pretty Print Dispatch Tables).

---

*print-pretty*  

Value Type:  
a generalized boolean.

Initial Value:  
implementation-dependent.

Description:  
Controls whether the Lisp printer calls the pretty printer.

If it is false, the pretty printer is not used and a minimum of whitespace is output when printing an expression.

If it is true, the pretty printer is used, and the Lisp printer will endeavor to insert extra whitespace where appropriate to make expressions more readable.

*print-pretty* has an effect even when the value of *print-escape* is false.
Examples:

```
(setq *print-pretty* 'nil) → NIL
(progn (write '(let ((a 1) (b 2) (c 3)) (+ a b c))) nil)
▷ (LET ((A 1) (B 2) (C 3)) (+ A B C))
→ NIL
(let ((*print-pretty* t))
  (progn (write '(let ((a 1) (b 2) (c 3)) (+ a b c))) nil))
▷ (LET ((A 1)
  ▷ (B 2)
  ▷ (C 3))
  ▷ (+ A B C))
→ NIL
```

;; Note that the first two expressions printed by this next form
;; differ from the second two only in whether escape characters are printed.
;; In all four cases, extra whitespace is inserted by the pretty printer.

```
(let ((test (x))
  (let ((*print-pretty* t))
    (print x)
    (format t "%S " x)
    (terpri) (princ x) (princ " ")
    (format t "%A " x)))
  (test '#(lambda () (list "a" '#c '#d)))))
▷ '#(LAMBDA () (list "a" '#C '#D))
▷ '#(LAMBDA () (LAMBDA () (LIST "a" '#C '#D))
▷ '#(LAMBDA () (LAMBDA () (LIST a b '#C '#D))
▷ '#(LAMBDA () (LAMBDA () (LIST a b '#C '#D))
→ NIL
```

See Also:

write

*print-readably*  

Value Type:  
a generalized boolean.
**Initial Value:**

*false*.

**Description:**

If *print-readably* is true, some special rules for printing objects go into effect. Specifically, printing any object $O_1$ produces a printed representation that, when seen by the Lisp reader while the standard readable is in effect, will produce an object $O_2$ that is similar to $O_1$. The printed representation produced might or might not be the same as the printed representation produced when *print-readably* is false. If printing an object readable is not possible, an error of type print-not-readable is signaled rather than using a syntax (e.g., the “#:” syntax) that would not be readable by the same implementation. If the value of some other printer control variable is such that these requirements would be violated, the value of that other variable is ignored.

Specifically, if *print-readably* is true, printing proceeds as if *print-escape*, *print-array*, and *print-gensym* were also true, and as if *print-length*, *print-level*, and *print-lines* were false.

If *print-readably* is false, the normal rules for printing and the normal interpretations of other printer control variables are in effect.

Individual methods for print-object, including user-defined methods, are responsible for implementing these requirements.

If *read-eval* is false and *print-readably* is true, any such method that would output a reference to the “#:” reader macro will either output something else or will signal an error (as described above).

**Examples:**

```
(let ((x (list "a" '\a (gensym) '((a (b (c))) d e f g)))
  (*print-escape* nil)
  (*print-gensym* nil)
  (*print-level* 3)
  (*print-length* 3))
  (write x)
  (let ((*print-readably* t))
    (terpri)
    (write x)
    :done))
```

▷ (a a G4581 ((A #) D E ...))
▷ ("a" |a| #:G4581 ((A (B (C))) D E F G))
→ :DONE

;; This is setup code is shared between the examples
;; of three hypothetical implementations which follow.
(setq table (make-hash-table)) → #<HASH-TABLE EQL 0/120 32005763>
(setf (gethash table 1) 'one) → ONE
(setf (gethash table 2) 'two) → TWO

;; Implementation A
(let ((*print-readably* t)) (print table))
Error: Can’t print #<HASH-TABLE EQL 0/120 32005763> readably.

;; Implementation B
;; No standardized #S notation for hash tables is defined,
;; but there might be an implementation-defined notation.
(let ((*print-readably* t)) (print table))
> #S(HASH-TABLE :TEST EQL :SIZE 120 :CONTENTS (1 ONE 2 TWO))
→ #<HASH-TABLE EQL 0/120 32005763>

;; Implementation C
;; Note that #. notation can only be used if *READ-EVAL* is true.
;; If *READ-EVAL* were false, this same implementation might have to
;; signal an error unless it had yet another printing strategy to fall
;; back on.
(let ((*print-readably* t)) (print table))
> #.((LET ((HASH-TABLE (MAKE-HASH-TABLE)))
    (SETF (GETHASH 1 HASH-TABLE) ONE)
    (SETF (GETHASH 2 HASH-TABLE) TWO)
    HASH-TABLE))
→ #<HASH-TABLE EQL 0/120 32005763>

See Also:
write, print-unreadable-object

Notes:
The rules for “similarity” imply that #A or #( syntax cannot be used for arrays of element type
other than t. An implementation will have to use another syntax or signal an error of type
print-not-readable.

*print-right-margin*

Variable

Value Type:
a non-negative integer, or nil.

Initial Value:
nil.
Description:
If it is `non-nil`, it specifies the right margin (as `integer` number of `ems`) to use when the `pretty printer` is making layout decisions.

If it is `nil`, the right margin is taken to be the maximum line length such that output can be displayed without wraparound or truncation. If this cannot be determined, an `implementation-dependent` value is used.

Notes:
This measure is in units of `ems` in order to be compatible with `implementation-defined` variable-width fonts while still not requiring the language to provide support for fonts.

print-not-readable

Condition Type

Class Precedence List:
print-not-readable, error, serious-condition, condition, t

Description:
The type `print-not-readable` consists of error conditions that occur during output while *
`print-readably`* is `true`, as a result of attempting to write a printed representation with the `Lisp printer` that would not be correctly read back with the `Lisp reader`. The object which could not be printed is initialized by the `:object` initialization argument to `make-condition`, and is accessed by the function `print-not-readable-object`.

See Also:
print-not-readable-object

print-not-readable-object

Function

Syntax:
`print-not-readable-object condition` → `object`

Arguments and Values:
`condition`—a condition of type `print-not-readable`.
`object`—an object.

Description:
Returns the `object` that could not be printed readably in the situation represented by `condition`. 
See Also:

print-not-readable, Chapter 9 (Conditions)

format Function

Syntax:

format destination control-string &rest args → result

Arguments and Values:

destination—nil, t, a stream, or a string with a fill pointer.

table-string—a format control.

args—format arguments for control-string.

result—if destination is non-nil, then nil; otherwise, a string.

Description:

format produces formatted output by outputting the characters of control-string and observing that a tilde introduces a directive. The character after the tilde, possibly preceded by prefix parameters and modifiers, specifies what kind of formatting is desired. Most directives use one or more elements of args to create their output.

If destination is a string, a stream, or t, then the result is nil. Otherwise, the result is a string containing the ‘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.

For details on how the control-string is interpreted, see Section 22.3 (Formatted Output).

Affected By:

*standard-output*, *print-escape*, *print-radix*, *print-base*, *print-circle*,

*print-pretty*, *print-level*, *print-length*, *print-case*, *print-gensym*, *print-array*.

Exceptional Situations:

If destination is a string with a fill pointer, the consequences are undefined if destructive modifications are performed directly on the string during the dynamic extent of the call.

See Also:

write, Section 13.1.10 (Documentation of Implementation-Defined Scripts)
Programming Language—Common Lisp

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 (unescape) 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 \texttt{(test-readtable-case-reading)} should be as follows:

<table>
<thead>
<tr>
<th>READTABLE-CASE</th>
<th>Input Symbol-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>:UPCASE</td>
<td>ZEBRA ZEBRA</td>
</tr>
<tr>
<td>:UPCASE</td>
<td>Zebra ZEBRA</td>
</tr>
<tr>
<td>:UPCASE</td>
<td>zebra ZEBRA</td>
</tr>
<tr>
<td>:DOWNCASE</td>
<td>ZEBRA zebra</td>
</tr>
<tr>
<td>:DOWNCASE</td>
<td>Zebra zebra</td>
</tr>
<tr>
<td>:DOWNCASE</td>
<td>zebra zebra</td>
</tr>
<tr>
<td>:PRESERVE</td>
<td>ZEBRA ZEBRA</td>
</tr>
<tr>
<td>:PRESERVE</td>
<td>Zebra Zebra</td>
</tr>
<tr>
<td>:PRESERVE</td>
<td>zebra zebra</td>
</tr>
<tr>
<td>:INVERT</td>
<td>ZEBRA zebra</td>
</tr>
<tr>
<td>:INVERT</td>
<td>Zebra Zebra</td>
</tr>
<tr>
<td>:INVERT</td>
<td>zebra ZEBRA</td>
</tr>
</tbody>
</table>

### 23.1.3 Argument Conventions of Some Reader Functions

#### 23.1.3.1 The \texttt{EOF-ERROR-P} argument

\textit{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 \texttt{eof-error-p} is \texttt{true} (the default), an error of \texttt{type end-of-file} is signaled at end of file. If it is \texttt{false}, then no error is signaled, and instead the function returns \texttt{eof-value}.

Functions such as \texttt{read} that read the representation of an \textit{object} rather than a single character always signals an error, regardless of \texttt{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, \texttt{read} signals an error. If a file ends in a \texttt{symbol} or a \texttt{number} immediately followed by end-of-file, \texttt{read} reads the \texttt{symbol} or \texttt{number} successfully and when called again will \textit{act according to \texttt{eof-error-p}}. Similarly, the function \texttt{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 \texttt{whitespace} or comments, are not considered to begin an \textit{object}; if \texttt{read} begins to read an \texttt{expression} but sees only such ignorable text, it does not consider the file to end in the middle of an \textit{object}. Thus an \texttt{eof-error-p} argument controls what happens when the file ends between \textit{objects}.  

23–2 Programming Language—Common Lisp
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 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 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.
readtable

Class Precedence List:
readtable, t

Description:
A *readtable* maps characters into syntax types for the Lisp reader; see Chapter 2 (Syntax). A
*readtable* also contains associations between *macro characters* and their *reader macro functions*,
and records information about the case conversion rules to be used by the *Lisp reader* when
parsing *symbols*.

Each *simple character* must be representable in the *readtable*. It is *implementation-defined* whether
*non-simple characters* can have syntax descriptions in the *readtable*.

See Also:
Section 2.1.1 (Readtables), Section 22.1.3.13 (Printing Other Objects)

copy-readtable

Syntax:
copy-readtable &optional from-readtable to-readtable → readtable

Arguments and Values:
from-readtable—a *readtable* designator. The default is the *current readtable*.
to-readtable—a *readtable* or nil. The default is nil.
readtable—the *to-readtable* if it is non-nil, or else a *fresh readtable*.

Description:
copy-readtable copies from-readtable.

If to-readtable is nil, a new *readtable* is created and returned. Otherwise the *readtable* specified by
to-readtable is modified and returned.

copy-readtable copies the setting of readtable-case.

Examples:

(setq zvar 123) → 123
(set-syntax-from-char #\z #\' (setq table2 (copy-readtable))) → T
zvar → 123
(cpy-readtable table2 *readtable*) → #<READTABLE 614000277>
zvar → VAR
(setq *readtable* (copy-readtable)) → #<READTABLE 46210223>
zvar → VAR
(setq *readtable* (copy-readtable nil)) → #<READTABLE 46302670>
zvar → 123

See Also:
readtable, *readtable*

Notes:

(setq *readtable* (copy-readtable nil))
restores the input syntax to standard Common Lisp syntax, even if the initial readtable has been clobbered (assuming it is not so badly clobbered that you cannot type in the above expression).

On the other hand,
(setq *readtable* (copy-readtable))
replaces the current readtable with a copy of itself. This is useful if you want to save a copy of a readtable for later use, protected from alteration in the meantime. It is also useful if you want to locally bind the readtable to a copy of itself, as in:
(let ((*readtable* (copy-readtable))) ...)

---

**make-dispatch-macro-character**  
*Function*

**Syntax:**

```
make-dispatch-macro-character char &optional non-terminating-p readtable → t
```

**Arguments and Values:**

- `char`—a character.
- `non-terminating-p`—a generalized boolean. The default is false.
- `readtable`—a readtable. The default is the current readtable.

**Description:**

`make-dispatch-macro-character` makes `char` be a dispatching macro character in `readtable`.

Initially, every character in the dispatch table associated with the `char` has an associated function that signals an error of type `reader-error`. 
If non-terminating-p is true, the dispatching macro character is made a non-terminating macro character; if non-terminating-p is false, the dispatching macro character is made a terminating macro character.

Examples:

\begin{verbatim}
(get-macro-character #\{) → NIL, false
(make-dispatch-macro-character #\{) → T
(not (get-macro-character #\{)) → false
\end{verbatim}

The readtable is altered.

See Also:

*readtable*, set-dispatch-macro-character

---

**read, read-preserving-whitespace**

*Function*

Syntax:

\begin{verbatim}
read &optional input-stream eof-error-p eof-value recursive-p → object
read-preserving-whitespace &optional input-stream eof-error-p eof-value recursive-p → object
\end{verbatim}

Arguments and Values:

- **input-stream**—an input stream designator.
- **eof-error-p**—a generalized boolean. The default is true.
- **eof-value**—an object. The default is nil.
- **recursive-p**—a generalized boolean. The default is false.
- **object**—an object (parsed by the Lisp reader) or the eof-value.

Description:

*read* parses the printed representation of an object from input-stream and builds such an object.

*read-preserving-whitespace* is like *read* but preserves any whitespace character that delimits the printed representation of the object. *read-preserving-whitespace* is exactly like *read* when the recursive-p argument to *read-preserving-whitespace* is true.
read, read-preserving-whitespace

When *read-suppress* is false, read throws away the delimiting character required by certain printed representations if it is a whitespace character; but read preserves the character (using unread-char) if it is syntactically meaningful, because it could be the start of the next expression.

If a file ends in a symbol or a number immediately followed by an end of file, read reads the symbol or number successfully; when called again, it sees the end of file and only then acts according to eof-error-p. If a file contains ignorable text at the end, such as blank lines and comments, read does not consider it to end in the middle of an object.

If recursive-p is true, the call to read is expected to be made from within some function that itself has been called from read or from a similar input function, rather than from the top level.

Both functions return the object read from input-stream. Eof-value is returned if eof-error-p is false and end of file is reached before the beginning of an object.

Examples:

```lisp
(read)
⊿
'(a
→ (QUOTE A)
(with-input-from-string (is " ") (read is nil 'the-end)) → THE-END
(defun skip-then-read-char (s c n)
  (if (char= c #\{) (read-preserving-whitespace s))
  (read-char-no-hang s)) → SKIP-THEN-READ-CHAR
(let ((*readtable* (copy-readtable nil)))
  (set-dispatch-macro-character #\# #\{ #'skip-then-read-char)
  (set-dispatch-macro-character #\# #\} #'skip-then-read-char)
  (with-input-from-string (is "#{123 x #}123 y")
    (format t "~S ~S" (read is) (read is)))) → #\x, #\Space, NIL
```

As an example, consider this reader macro definition:

```lisp
(defun slash-reader (stream char)
  (declare (ignore char))
  '(path ,,(loop for dir = (read-preserving-whitespace stream t nil t)
    then (progn (read-char stream t nil t)
      (read-preserving-whitespace stream t nil t)))
    collect dir
    while (eql (peek-char nil stream nil nil t) #\/)))))
  (set-macro-character #\/ #'slash-reader)
```

Consider now calling read on this expression:

```lisp
(zyedh /usr/games/zork /usr/games/boggle)
```

The / macro reads objects separated by more / characters; thus /usr/games/zork is intended to read as (path user games zork). The entire example expression should therefore be read as
However, if `read` had been used instead of `read-preserving-whitespace`, then after the reading of the symbol `zork`, the following space would be discarded; the next call to `peek-char` would see the following `/`, and the loop would continue, producing this interpretation:

```
(zyedh (path usr games zork (path usr games boggle)))
```

There are times when whitespace should be discarded. If a command interpreter takes single-character commands, but occasionally reads an `object` then if the `whitespace`2 after a `symbol` is not discarded it might be interpreted as a command some time later after the `symbol` had been read.

**Affected By:**
- *standard-input*, *terminal-io*, *readtable*, *read-default-float-format*, *read-base*,
- *read-suppress*, *package*, *read-eval*.

**Exceptional Situations:**

`read` signals an error of type `end-of-file`, 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. This is detected when `read` or `read-preserving-whitespace` is called with `recursive-p` and `eof-error-p` non-nil, and end-of-file is reached before the beginning of an `object`.

If `eof-error-p` is `true`, an error of type `end-of-file` is signaled at the end of file.

**See Also:**
- `peek-char`, `read-char`, `unread-char`, `read-from-string`, `read-delimited-list`, `parse-integer`,
- Chapter 2 (Syntax), Section 23.1 (Reader Concepts)

---

### read-delimited-list

**Function**

**Syntax:**

```
read-delimited-list char &optional input-stream recursive-p → list
```

**Arguments and Values:**

- `char`—a character.
- `input-stream`—an input stream designator. The default is `standard input`.
- `recursive-p`—a generalized boolean. The default is `false`.
- `list`—a list of the `objects` read.
read-delimited-list

Description:

read-delimited-list reads objects from input-stream until the next character after an object’s representation (ignoring whitespace characters and comments) is char.

read-delimited-list looks ahead at each step for the next non-whitespace character and peeks at it as if with peek-char. If it is char, then the character is consumed and the list of objects is returned. If it is a constituent or escape character, then read is used to read an object, which is added to the end of the list. If it is a macro character, its reader macro function is called; if the function returns a value, that value is added to the list. The peek-ahead process is then repeated.

If recursive-p is true, this call is expected to be embedded in a higher-level call to read or a similar function.

It is an error to reach end-of-file during the operation of read-delimited-list.

The consequences are undefined if char has a syntax type of whitespace in the current readtable.

Examples:

(read-delimited-list #\]) 1 2 3 4 5 6 ]
→ (1 2 3 4 5 6)

Suppose you wanted #{a b c ... z} to read as a list of all pairs of the elements a, b, c, ..., z, for example.

#{p q z a} reads as ((p q) (p z) (p a) (q z) (q a) (z a))

This can be done by specifying a macro-character definition for #{ that does two things: reads in all the items up to the }, and constructs the pairs. read-delimited-list performs the first task.

(defun |#{-reader| (stream char arg)
  (declare (ignore char arg))
  (mapcon #'(lambda (x)
    (mapcar #'(lambda (y) (list (car x) y)) (cdr x)))
    (read-delimited-list #\} stream t)))
  → |#{-reader|

(set-dispatch-macro-character #\ #\ #{#{-reader|} #\ #\ #{#{-reader|} } (get-macro-character #\) nil))

Note that true is supplied for the recursive-p argument.

It is necessary here to give a definition to the character } as well to prevent it from being a constituent. If the line

(set-macro-character #\} (get-macro-character #\) nil))

shown above were not included, then the } in

#{ p q z a}
would be considered a constituent character, part of the symbol named \texttt{a}). This could be corrected by putting a space before the \texttt{)}, but it is better to call \texttt{set-macro-character}.

Giving \texttt{)} the same definition as the standard definition of the character \texttt{)} has the twin benefit of making it terminate tokens for use with \texttt{read-delimited-list} and also making it invalid for use in any other context. Attempting to read a stray \texttt{)} will signal an error.

**Affected By:**
- \texttt{*standard-input*}, \texttt{*readtable*}, \texttt{*terminal-io*}.

**See Also:**
read, peek-char, read-char, unread-char.

**Notes:**
\texttt{read-delimited-list} is intended for use in implementing reader macros. Usually it is desirable for \texttt{char} to be a terminating macro character so that it can be used to delimit tokens; however, \texttt{read-delimited-list} makes no attempt to alter the syntax specified for \texttt{char} by the current readtable. The caller must make any necessary changes to the readtable syntax explicitly.

---

**read-from-string**

*Function*

**Syntax:**

\[
\text{read-from-string} \quad \text{string} \quad \text{&optional} \quad \text{eof-error-p} \quad \text{eof-value} \\
\quad \text{&key} \quad \text{start} \quad \text{end} \quad \text{preserve-whitespace} \\
\rightarrow \quad \text{object}, \quad \text{position}
\]

**Arguments and Values:**

- \texttt{string}—a string.
- \texttt{eof-error-p}—a generalized boolean. The default is \texttt{true}.
- \texttt{eof-value}—an object. The default is \texttt{nil}.
- \texttt{start}, \texttt{end}—bounding index designators of \texttt{string}. The defaults for \texttt{start} and \texttt{end} are 0 and \texttt{nil}, respectively.
- \texttt{preserve-whitespace}—a generalized boolean. The default is \texttt{false}.
- \texttt{object}—an object (parsed by the Lisp reader) or the \texttt{eof-value}.
- \texttt{position}—an integer greater than or equal to zero, and less than or equal to one more than the length of the \texttt{string}.
Description:

Parses the printed representation of an object from the subsequence of string bounded by start and end, as if read had been called on an input stream containing those same characters.

If preserve-whitespace is true, the operation will preserve whitespace as read-preserving-whitespace would do.

If an object is successfully parsed, the primary value, object, is the object that was parsed. If eof-error-p is false and if the end of the substring is reached, eof-value is returned.

The secondary value, position, is the index of the first character in the bounded string that was not read. The position may depend upon the value of preserve-whitespace. If the entire string was read, the position returned is either the length of the string or one greater than the length of the string.

Examples:

(read-from-string " 1 3 5" t nil :start 2) \rightarrow 3, 5
(read-from-string "(a b c)") \rightarrow (A B C), 7

Exceptional Situations:

If the end of the supplied substring occurs before an object can be read, an error is signaled if eof-error-p is true. An error is signaled if the end of the substring occurs in the middle of an incomplete object.

See Also:
read, read-preserving-whitespace

Notes:

The reason that position is allowed to be beyond the length of the string is to permit (but not require) the implementation to work by simulating the effect of a trailing delimiter at the end of the bounded string. When preserve-whitespace is true, the position might count the simulated delimiter.

---

**readtable-case**

Accessor

Syntax:

readtable-case readtable \rightarrow mode

(setf (readtable-case readtable) mode)

Arguments and Values:

readtable—a readtable.
mode—a case sensitivity mode.

Description:
Accesses the readtable case of readtable, which affects the way in which the Lisp Reader reads symbols and the way in which the Lisp Printer writes symbols.

Examples:
See Section 23.1.2.1 (Examples of Effect of Readtable Case on the Lisp Reader) and Section 22.1.3.3.2.1 (Examples of Effect of Readtable Case on the Lisp Printer).

Exceptional Situations:
Should signal an error of type type-error if readtable is not a readtable. Should signal an error of type type-error if mode is not a case sensitivity mode.

See Also:
*readtable*, *print-escape*, Section 2.2 (Reader Algorithm), Section 23.1.2 (Effect of Readtable Case on the Lisp Reader), Section 22.1.3.3.2 (Effect of Readtable Case on the Lisp Printer)

Notes:
copy-readtable copies the readtable case of the readtable.

---

readtablep

Function

Syntax:
readtablep object → generalized-boolean

Arguments and Values:
object—an object.

generalized-boolean—a generalized boolean.

Description:
Returns true if object is of type readtable; otherwise, returns false.

Examples:
(readtablep *readtable*) → true
(readtablep (copy-readtable)) → true
(readtablep '*readtable*) → false

Notes:
(readtablep object) ≡ (typep object 'readtable)
set-dispatch-macro-character, get-dispatch-macro-character

Function

Syntax:
get-dispatch-macro-character disp-char sub-char &optional readtable → function
set-dispatch-macro-character disp-char sub-char new-function &optional readtable → t

Arguments and Values:
disp-char—a character.
sub-char—a character.
readtable—a readtable designator. The default is the current readtable.
function—a function designator or nil.
new-function—a function designator.

Description:
set-dispatch-macro-character causes new-function to be called when disp-char followed by sub-char is read. If sub-char is a lowercase letter, it is converted to its uppercase equivalent. It is an error if sub-char is one of the ten decimal digits.

set-dispatch-macro-character installs a new-function to be called when a particular dispatching macro character pair is read. New-function is installed as the dispatch function to be called when readable is in use and when disp-char is followed by sub-char.

For more information about how the new-function is invoked, see Section 2.1.4.4 (Macro Characters).

get-dispatch-macro-character retrieves the dispatch function associated with disp-char and sub-char in readable.

get-dispatch-macro-character returns the macro-character function for sub-char under disp-char, or nil if there is no function associated with sub-char. If sub-char is a decimal digit, get-dispatch-macro-character returns nil.

Examples:
(get-dispatch-macro-character #\# #\{} → NIL
(set-dispatch-macro-character #\# #\{} ;dispatch on #(
  #'(lambda(s c n)
      (let ((list (read s nil (values) t))) ;list is object after #n
(when (consp list) ; return nth element of list
  (unless (and n (< 0 n (length list))) (setq n 0))
  (setq list (nth n list)))

→ T

{ (1 2 3 4) → 1
  #3( (0 1 2 3) → 3
  #((123 → 123

If it is desired that #$foo : as if it were (dollars $foo).

(defun |#$-reader| (stream subchar arg)
  (declare (ignore subchar arg))
  (list 'dollars (read stream t nil t))) → |#$-reader|
  (set-dispatch-macro-character #\# #\$ #\' |#$-reader|) → T

See Also:
  Section 2.1.4.4 (Macro Characters)

Side Effects:
  The readtable is modified.

Affected By:
  *readtable*.

Exceptional Situations:
  For either function, an error is signaled if disp-char is not a dispatching macro character in readtable.

See Also:
  *readtable*

Notes:
  It is necessary to use make-dispatch-macro-character to set up the dispatch character before specifying its sub-characters.

set-macro-character, get-macro-character Function

Syntax:
  get-macro-character char &optional readtable → function, non-terminating-p
  set-macro-character char new-function &optional non-terminating-p readtable → t

23–14 Programming Language—Common Lisp
set-macro-character, get-macro-character

Arguments and Values:

char—a character.

non-terminating-p—a generalized boolean. The default is false.

readtable—a readtable designator. The default is the current readtable.

function—nil, or a designator for a function of two arguments.

new-function—a function designator.

Description:

get-macro-character returns as its primary value, function, the reader macro function associated with char in readtable (if any), or else nil if char is not a macro character in readtable. The secondary value, non-terminating-p, is true if char is a non-terminating macro character; otherwise, it is false.

set-macro-character causes char to be a macro character associated with the reader macro function new-function (or the designator for new-function) in readtable. If non-terminating-p is true, char becomes a non-terminating macro character; otherwise it becomes a terminating macro character.

Examples:

(get-macro-character #\{) → NIL, false
(not (get-macro-character #\\)) → false

The following is a possible definition for the single-quote reader macro in standard syntax:

(defun single-quote-reader (stream char)
  (declare (ignore char))
  (list 'quote (read stream t nil t)) → SINGLE-QUOTE-READER
  (set-macro-character \' #'single-quote-reader) → T

Here single-quote-reader reads an object following the single-quote and returns a list of quote and that object. The char argument is ignored.

The following is a possible definition for the semicolon reader macro in standard syntax:

(defun semicolon-reader (stream char)
  (declare (ignore char))
  ;; First swallow the rest of the current input line.
  ;; End-of-file is acceptable for terminating the comment.
  (do () ((char= (read-char stream nil #\Newline t) #\Newline)))
  ;; Return zero values.
  (values)) → SEMICOLON-READER
  (set-macro-character #\; #'semicolon-reader) → T

Reader 23–15
**Side Effects:**

The *readtable* is modified.

**See Also:**

*readtable*

---

**set-syntax-from-char**

*Function*

**Syntax:**

```
(set-syntax-from-char to-char from-char &optional to-readtable from-readtable) → t
```

**Arguments and Values:**

- **to-char**—a character.
- **from-char**—a character.
- **to-readtable**—a readtable. The default is the current readtable.
- **from-readtable**—a readtable designator. The default is the standard readtable.

**Description:**

*set-syntax-from-char* makes the syntax of *to-char* in *to-readtable* be the same as the syntax of *from-char* in *from-readtable*.

*set-syntax-from-char* copies the syntax types of *from-char*. If *from-char* is a macro character, its reader macro function is copied also. If the character is a dispatching macro character, its entire dispatch table of reader macro functions is copied. The constituent traits of *from-char* are not copied.

A macro definition from a character such as " can be copied to another character; the standard definition for " looks for another character that is the same as the character that invoked it. The definition of ( can not be meaningfully copied to [, on the other hand. The result is that lists are of the form (a b c), not {a b c}, because the definition always looks for a closing parenthesis, not a closing brace.

**Examples:**

```
(set-syntax-from-char #\7 #\;)
123579 → T
```

**Side Effects:**

The *to-readtable* is modified.

---

23–16  Programming Language—Common Lisp
Affected By:
The existing values in the from-readtable.

See Also:
set-macro-character, make-dispatch-macro-character, Section 2.1.4 (Character Syntax Types)

Notes:
The constituent traits of a character are “hard wired” into the parser for extended tokens. For example, if the definition of $ is copied to *, then * will become a constituent that is alphabetic but that cannot be used as a short float exponent marker. For further information, see Section 2.1.4.2 (Constituent Traits).

with-standard-io-syntax  Macro

Syntax:
with-standard-io-syntax \{form\} \rightarrow \{result\}

Arguments and Values:
forms—an implicit progn.
results—the values returned by the forms.

Description:
Within the dynamic extent of the body of forms, all reader/printer control variables, including any implementation-defined ones not specified by this standard, are bound to values that produce standard read/print behavior. The values for the variables specified by this standard are listed in Figure 23–1.
<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>package</em></td>
<td>The CL-USER package</td>
</tr>
<tr>
<td><em>print-array</em></td>
<td>t</td>
</tr>
<tr>
<td><em>print-base</em></td>
<td>10</td>
</tr>
<tr>
<td><em>print-case</em></td>
<td>:upcase</td>
</tr>
<tr>
<td><em>print-circle</em></td>
<td>nil</td>
</tr>
<tr>
<td><em>print-escape</em></td>
<td>t</td>
</tr>
<tr>
<td><em>print-gensym</em></td>
<td>t</td>
</tr>
<tr>
<td><em>print-length</em></td>
<td>nil</td>
</tr>
<tr>
<td><em>print-level</em></td>
<td>nil</td>
</tr>
<tr>
<td><em>print-lines</em></td>
<td>nil</td>
</tr>
<tr>
<td><em>print-miser-width</em></td>
<td>nil</td>
</tr>
<tr>
<td><em>print-pretty</em></td>
<td>nil</td>
</tr>
<tr>
<td><em>print-radix</em></td>
<td>nil</td>
</tr>
<tr>
<td><em>print-readably</em></td>
<td>t</td>
</tr>
<tr>
<td><em>print-right-margin</em></td>
<td>nil</td>
</tr>
<tr>
<td><em>read-base</em></td>
<td>10</td>
</tr>
<tr>
<td><em>read-default-float-format</em></td>
<td>single-float</td>
</tr>
<tr>
<td><em>read-eval</em></td>
<td>t</td>
</tr>
<tr>
<td><em>read-suppress</em></td>
<td>nil</td>
</tr>
<tr>
<td><em>readtable</em></td>
<td>The standard readtable</td>
</tr>
</tbody>
</table>

Figure 23–1. Values of standard control variables

Examples:

```lisp
(with-open-file (file pathname :direction :output)
  (with-standard-io-syntax
    (print data file)))

;;; ... Later, in another Lisp:

(with-open-file (file pathname :direction :input)
  (with-standard-io-syntax
    (setq data (read file)))
```

23–18 Programming Language—Common Lisp
*read-base*  

Variable

Value Type:

a radix.

Initial Value:

10.

Description:

Controls the interpretation of tokens by read as being integers or ratios.

The value of *read-base*, called the current input base, is the radix in which integers and ratios are to be read by the Lisp reader. The parsing of other numeric types (e.g., floats) is not affected by this option.

The effect of *read-base* on the reading of any particular rational number can be locally overridden by explicit use of the #O, #X, #B, or #nR syntax or by a trailing decimal point.

Examples:

```lisp
(dotimes (i 6)
  (let ((*read-base* (+ 10. i)))
    (let ((object (read-from-string "(\DAD DAD |BEE| BEE 123. 123)")))
      (print (list *read-base* object))))
  > (10 (DAD DAD BEE BEE 123 123))
  > (11 (DAD DAD BEE BEE 123 146))
  > (12 (DAD DAD BEE BEE 123 171))
  > (13 (DAD DAD BEE BEE 123 198))
  > (14 (DAD 2701 BEE BEE 123 227))
  > (15 (DAD 3088 BEE 2699 123 258))
  → NIL)
```

Notes:

Altering the input radix can be useful when reading data files in special formats.

*read-default-float-format*  

Variable

Value Type:

one of the atomic type specifiers short-float, single-float, double-float, or long-float, or else some other type specifier defined by the implementation to be acceptable.
Initial Value:
The symbol `single-float`.

Description:
Controls the floating-point format that is to be used when reading a floating-point number that has no exponent marker or that has `e` or `E` for an exponent marker. Other exponent markers explicitly prescribe the floating-point format to be used.

The printer uses `*read-default-float-format*` to guide the choice of exponent markers when printing floating-point numbers.

Examples:

```lisp
(let ((*read-default-float-format* 'double-float))
  (read-from-string "(1.0 1.0e0 1.0s0 1.0f0 1.0d0 1.0L0)"))
→ (1.0 1.0 1.0 1.0 1.0 1.0) ; Implementation has float format F.
→ (1.0d0 1.0d0 1.0 1.0 1.0d0 1.0d0) ; Implementation has float formats S and F.
→ (1.0d0 1.0d0 1.0 1.0 1.0d0 1.0d0) ; Implementation has float formats F and D.
→ (1.0d0 1.0d0 1.0 1.0 1.0d0 1.0L0) ; Implementation has float formats F, D, L.
→ (1.0d0 1.0d0 1.0s0 1.0 1.0d0 1.0L0) ; Implementation has formats S, F, D, L.
```

*read-eval*

Variable

Value Type:
a generalized boolean.

Initial Value:
true.

Description:
If it is true, the `#.` reader macro has its normal effect. Otherwise, that reader macro signals an error of type `reader-error`.

See Also:
`*print-readably*`

Notes:
If `*read-eval*` is false and `*print-readably*` is true, any method for `print-object` that would output a reference to the `#.` reader macro either outputs something different or signals an error of type `print-not-readable`.

23–20 Programming Language—Common Lisp
*read-suppress*

**Variable**

**Value Type:**

A generalized boolean.

**Initial Value:**

False.

**Description:**

This variable is intended primarily to support the operation of the read-time conditional notations `#+` and `#-`. It is important for the reader macros which implement these notations to be able to skip over the printed representation of an expression despite the possibility that the syntax of the skipped expression may not be entirely valid for the current implementation, since `#+` and `#-` exist in order to allow the same program to be shared among several Lisp implementations (including dialects other than Common Lisp) despite small incompatibilities of syntax.

If it is false, the Lisp reader operates normally.

If the value of *read-suppress* is true, read, read-preserving-whitespace, read-delimited-list, and read-from-string all return a primary value of nil when they complete successfully; however, they continue to parse the representation of an object in the normal way, in order to skip over the object, and continue to indicate end of file in the normal way. Except as noted below, any standardized reader macro that is defined to read a following object or token will do so, but not signal an error if the object read is not of an appropriate type or syntax. The standard syntax and its associated reader macros will not construct any new objects (e.g., when reading the representation of a symbol, no symbol will be constructed or interned).

**Extended tokens**

All extended tokens are completely uninterpreted. Errors such as those that might otherwise be signaled due to detection of invalid potential numbers, invalid patterns of package markers, and invalid uses of the dot character are suppressed.

**Dispatching macro characters (including sharpsign)**

Dispatching macro characters continue to parse an infix numerical argument, and invoke the dispatch function. The standardized sharpsign reader macros do not enforce any constraints on either the presence of or the value of the numerical argument.

`#=`

The `#=` notation is totally ignored. It does not read a following object. It produces no object, but is treated as whitespace.

`##`

**Reader** 23–21
The ## notation always produces nil.

No matter what the value of *read-suppress*, parentheses still continue to delimit and construct lists; the # notation continues to delimit vectors; and comments, strings, and the single-quote and backquote notations continue to be interpreted properly. Such situations as ’', #<, #), and #⟨Space⟩ continue to signal errors.

Examples:

(\let ((*read-suppress* t))
 (mapcar #\'read-from-string
   '("(foo bar baz)" "#P(:type :lisp) "#c1.2"
   "#.(PRINT 'FOO)" "#3HELLO" "#S(INTEGER)"
   "#*ABC" "#\GARBAGE" "#RALPHA" "#3R444")))
→ (NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)

See Also:
read, Chapter 2 (Syntax)

Notes:

Programmers and implementations that define additional macro characters are strongly encouraged to make them respect *read-suppress* just as standardized macro characters do. That is, when the value of *read-suppress* is true, they should ignore type errors when reading a following object and the functions that implement dispatching macro characters should tolerate nil as their infix parameter value even if a numeric value would ordinarily be required.

### *readtable*

**Variable**

**Value Type:**
a readtable.

**Initial Value:**
A readtable that conforms to the description of Common Lisp syntax in Chapter 2 (Syntax).

**Description:**
The value of *readtable* is called the current readtable. It controls the parsing behavior of the Lisp reader, and can also influence the Lisp printer (e.g., see the function readtable-case).

**Examples:**

(readtablep *readtable*) → true
(setq zvar 123) → 123

23–22 Programming Language—Common Lisp
(set-syntax-from-char #\z #\' (setq table2 (copy-readtable))) → T
zvar → 123
(setq *readtable* table2) → #<READTABLE>
zvar → VAR
(setq *readtable* (copy-readtable nil)) → #<READTABLE>
zvar → 123

Affected By:
compile-file, load

See Also:
compile-file, load, readable, Section 2.1.1.1 (The Current Readtable)

---

reader-error

Class Precedence List:
reader-error, parse-error, stream-error, error, serious-condition, condition, t

Description:
The type reader-error consists of error conditions that are related to tokenization and parsing done by the Lisp reader.

See Also:
read, stream-error-stream, Section 23.1 (Reader Concepts)
Programming Language—Common Lisp

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 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. Figure 24–1 shows some sample expressions, and how they would be read 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)
```

Figure 24–1. Features examples
**Syntax:**

```
compile-file input-file &key output-file verbose
  print external-format

→ output-truename, warnings-p, failure-p
```

**Arguments and Values:**

- `input-file`—a pathname designator. (Default fillers for unspecified components are taken from `*default-pathname-defaults*`.)
- `output-file`—a pathname designator. The default is implementation-defined.
- `verbose`—a generalized boolean. The default is the value of `*compile-verbose*`.
- `print`—a generalized boolean. The default is the value of `*compile-print*`.
- `external-format`—an external file format designator. The default is `:default`.
- `output-truename`—a pathname (the truename of the output file), or `nil`.
- `warnings-p`—a generalized boolean.
- `failure-p`—a generalized boolean.

**Description:**

The `compile-file` function transforms the contents of the file specified by `input-file` into implementation-dependent binary data which are placed in the file specified by `output-file`.

The file to which `input-file` refers should be a source file. `output-file` can be used to specify an output pathname; the actual pathname of the compiled file to which compiled code will be output is computed as if by calling `compile-file-pathname`.

If `input-file` or `output-file` is a logical pathname, it is translated into a physical pathname as if by calling `translate-logical-pathname`.

If `verbose` is `true`, `compile-file` prints a message in the form of a comment (i.e., with a leading `semicolon`) to standard output indicating what file is being compiled and other useful information. If `verbose` is `false`, `compile-file` does not print this information.

If `print` is `true`, information about top level forms in the file being compiled is printed to standard output. Exactly what is printed is implementation-dependent, but nevertheless some information is printed. If `print` is `nil`, no information is printed.

The `external-format` specifies the external file format to be used when opening the file; see the
**compile-file**

`function open, compile-file` and `load` must cooperate in such a way that the resulting *compiled file* can be *loaded* without specifying an *external file format* anew; see the `function load`.

`compile-file` binds `*readtable*` and `*package*` to the values they held before processing the file.

`*compile-file-truename*` is bound by `compile-file` to hold the *truename* of the *pathname* of the file being compiled.

`*compile-file-pathname*` is bound by `compile-file` to hold a *pathname* denoted by the first argument to `compile-file`, merged against the defaults; that is, `(pathname (merge-pathnames input-file))`.

The compiled *functions* contained in the *compiled file* become available for use when the *compiled file* is *loaded* into Lisp. Any function definition that is processed by the compiler, including `#'(lambda ...)` forms and local function definitions made by `flet`, `labels` and `defun` forms, result in an object of type *compiled-function*.

The primary value returned by `compile-file`, `output-truename`, is the *truename* of the output file, or `nil` if the file could not be created.

The secondary value, `warnings-p`, is `false` if no conditions of type *error* or *warning* were detected by the compiler, and `true` otherwise.

The tertiary value, `failure-p`, is `false` if no conditions of type *error* or *warning* (other than *style-warning*) were detected by the compiler, and `true` otherwise.

For general information about how *files* are processed by the *file compiler*, see Section 3.2.3 (File Compilation).

Programs to be compiled by the *file compiler* must only contain *externalizable objects*; for details on such objects, see Section 3.2.4 (Literal Objects in Compiled Files). For information on how to extend the set of *externalizable objects*, see the `function make-load-form` and Section 3.2.4.4 (Additional Constraints on Externalizable Objects).

**Affected By:**

- `*error-output*`, `*standard-output*`, `*compile-verbose*`, `*compile-print*`

  The computer’s file system.

**Exceptional Situations:**

For information about errors detected during the compilation process, see Section 3.2.5 (Exceptional Situations in the Compiler).

An error of type *file-error* might be signaled if `(wild-pathname-p input-file)` returns true.

If either the attempt to open the *source file* for input or the attempt to open the *compiled file* for output fails, an error of type *file-error* is signaled.

---

24–4 Programming Language—Common Lisp
compile-file-pathname

**Function**

**Syntax:**

```
compile-file-pathname input-file &key output-file &allow-other-keys → pathname
```

**Arguments and Values:**

- `input-file`—a pathname designator. (Default fillers for unspecified components are taken from `*default-pathname-defaults*.)
- `output-file`—a pathname designator. The default is implementation-defined.
- `pathname`—a pathname.

**Description:**

Returns the `pathname` that `compile-file` would write into, if given the same arguments.

The defaults for the `output-file` are taken from the `pathname` that results from merging the `input-file` with the value of `*default-pathname-defaults*`, except that the type component should default to the appropriate implementation-defined default type for compiled files.

If `input-file` is a logical pathname and `output-file` is unsupplied, the result is a logical pathname. If `input-file` is a logical pathname, it is translated into a physical pathname as if by calling `translate-logical-pathname`. If `input-file` is a stream, the stream can be either open or closed. `compile-file-pathname` returns the same `pathname` after a file is closed as it did when the file was open. It is an error if `input-file` is a stream that is created with `make-two-way-stream`, `make-echo-stream`, `make-broadcast-stream`, `make-concatenated-stream`, `make-string-input-stream`, `make-string-output-stream`.

If an implementation supports additional keyword arguments to `compile-file`, `compile-file-pathname` must accept the same arguments.

**Examples:**

See `logical-pathname-translations`.

**Exceptional Situations:**

An error of type `file-error` might be signaled if either `input-file` or `output-file` is wild.

**See Also:**
**load**  
*Function*

**Syntax:**

```
load filespec &key verbose print  
  if-does-not-exist external-format  
→ generalized-boolean
```

**Arguments and Values:**

- **filespec**—a stream, or a pathname designator. The default is taken from *default-pathname-defaults*.
- **verbose**—a generalized boolean. The default is the value of *load-verbose*.
- **print**—a generalized boolean. The default is the value of *load-print*.
- **if-does-not-exist**—a generalized boolean. The default is true.
- **external-format**—an external file format designator. The default is :default.
- **generalized-boolean**—a generalized boolean.

**Description:**

`load` loads the file named by `filespec` into the Lisp environment.

The manner in which a `source file` is distinguished from a `compiled file` is implementation-dependent. If the file specification is not complete and both a `source file` and a `compiled file` exist which might match, then which of those files `load` selects is implementation-dependent.

If `filespec` is a `stream`, `load` determines what kind of `stream` it is and loads directly from the `stream`. If `filespec` is a `logical pathname`, it is translated into a `physical pathname` as if by calling `translate-logical-pathname`.

`load` sequentially executes each `form` it encounters in the file named by `filespec`. If the file is a `source file` and the implementation chooses to perform implicit compilation, `load` must recognize `top level forms` as described in Section 3.2.3.1 (Processing of Top Level Forms) and arrange for each `top level form` to be executed before beginning implicit compilation of the next. (Note, however, that processing of `eval-when forms` by `load` is controlled by the `:execute` situation.)

If `verbose` is true, `load` prints a message in the form of a comment (i.e., with a leading `semicolon`) to standard output indicating what file is being loaded and other useful information. If `verbose` is false, `load` does not print this information.
If `print` is `true`, `load` incrementally prints information to standard output showing the progress of the loading process. For a `source file`, this information might mean printing the `values yielded by each form` in the file as soon as those values are returned. For a `compiled file`, what is printed might not reflect precisely the contents of the `source file`, but some information is generally printed. If `print` is `false`, `load` does not print this information.

If the file named by `filespec` is successfully loaded, `load` returns `true`. If the file does not exist, the specific action taken depends on `if-does-not-exist`: if it is `nil`, `load` returns `nil`; otherwise, `load` signals an error.

The `external-format` specifies the `external file format` to be used when opening the file (see the function `open`), except that when the file named by `filespec` is a `compiled file`, the `external-format` is ignored. `compile-file` and `load` cooperate in an `implementation-dependent` way to assure the preservation of the `similarity of characters` referred to in the source file at the time the source file was processed by the file compiler under a given `external file format`, regardless of the value of `external-format` at the time the compiled file is `loaded`.

`load` binds `*readtable*` and `*package*` to the values they held before `loading` the file.

`*load-truename*` is `bound` by `load` to hold the `truename` of the `pathname` of the file being loaded.

`*load-pathname*` is `bound` by `load` to hold a `pathname` that represents `filespec` merged against the defaults. That is, `(pathname (merge-pathnames filespec))`.

Examples:

```lisp
;; Establish a data file...
(with-open-file (str "data.in" :direction :output :if-exists :error)
  (print 1 str) (print '(setq a 888) str) t)
→ T
a → 888
(load "data.in") → true
(load (setq p (merge-pathnames "data.in")) :verbose t) ; Loading contents of file /fred/data.in ; Finished loading /fred/data.in ; true
(merge-pathnames "data.in") → true
(load p :print t) ; Loading contents of file /fred/data.in ; 1 ; 888 ; Finished loading /fred/data.in ; true

;;----[Begin file SETUP]----
(in-package "MY-STUFF")
```

System Construction 24–7
(defmacro compile-truename () '',*compile-file-truename*)
(defvar *my-compile-truename* (compile-truename) "Just for debugging.")
(defvar *my-load-pathname* *load-pathname*)
(defun load-my-system ()
  (dolist (module-name '("FOO" "BAR" "BAZ")
    (load (merge-pathnames module-name *my-load-pathname*))))

;;----[End of file SETUP]----

(load "SETUP")
(load-my-system)

Affected By:
The implementation, and the host computer’s file system.

Exceptional Situations:
If :if-does-not-exist is supplied and is true, or is not supplied, load signals an error of type file-error if the file named by filespec does not exist, or if the file system cannot perform the requested operation.

An error of type file-error might be signaled if (wild-pathname-p filespec) returns true.

See Also:
error, merge-pathnames, *load-verbose*, *default-pathname-defaults*, pathname, logical-pathname, Section 20.1 (File System Concepts), Section 19.1.2 (Pathnames as Filenames)

---

with-compilation-unit

with-compilation-unit (\ [\ option\ ]\ \ \ {\ form\ }* \ → \ \ {\ result\ }*\n
option::=\ :override\ \ override

Arguments and Values:
override—a generalized boolean; evaluated. The default is nil.
forms—an implicit progn.
results—the values returned by the forms.

24–8 Programming Language—Common Lisp
Description:

Executes forms from left to right. Within the dynamic environment of with-compilation-unit, actions deferred by the compiler until the end of compilation will be deferred until the end of the outermost call to with-compilation-unit.

The set of options permitted may be extended by the implementation, but the only standardized keyword is :override.

If nested dynamically only the outer call to with-compilation-unit has any effect unless the value associated with :override is true, in which case warnings are deferred only to the end of the innermost call for which override is true.

The function compile-file provides the effect of

(with-compilation-unit (:override nil) ...)

around its code.

Any implementation-dependent extensions can only be provided as the result of an explicit programmer request by use of an implementation-dependent keyword. Implementations are forbidden from attaching additional meaning to a use of this macro which involves either no keywords or just the keyword :override.

Examples:

If an implementation would normally defer certain kinds of warnings, such as warnings about undefined functions, to the end of a compilation unit (such as a file), the following example shows how to cause those warnings to be deferred to the end of the compilation of several files.

(defun compile-files (&rest files)
  (with-compilation-unit ()
    (mapcar #'(lambda (file) (compile-file file)) files)))

(compile-files "A" "B" "C")

Note however that if the implementation does not normally defer any warnings, use of with-compilation-unit might not have any effect.

See Also:

compile, compile-file
*features*

**Variable**

**Value Type:**

a proper list.

**Initial Value:**

implementation-dependent.

**Description:**

The value of *features* is called the features list. It is a list of symbols, called features, that correspond to some aspect of the implementation or environment.

Most features have implementation-dependent meanings; The following meanings have been assigned to feature names:

:cltl1

If present, indicates that the LISP package purports to conform to the 1984 specification Common Lisp: The Language. It is possible, but not required, for a conforming implementation to have this feature because this specification specifies that its symbols are to be in the COMMON-LISP package, not the LISP package.

:cltl2

If present, indicates that the implementation purports to conform to Common Lisp: The Language, Second Edition. This feature must not be present in any conforming implementation, since conformance to that document is not compatible with conformance to this specification. The name, however, is reserved by this specification in order to help programs distinguish implementations which conform to that document from implementations which conform to this specification.

:ieee-floating-point

If present, indicates that the implementation purports to conform to the requirements of IEEE Standard for Binary Floating-Point Arithmetic.

:x3j13

If present, indicates that the implementation conforms to some particular working draft of this specification, or to some subset of features that approximates a belief about what this specification might turn out to contain. A conforming implementation might or might not contain such a feature. (This feature is intended primarily as a stopgap in order to provide implementors something to use prior to the availability of a draft standard, in order to discourage them from introducing the :draft-ansi-cl and :ansi-cl features prematurely.)
:draft-ansi-cl

If present, indicates that the implementation purports to conform to the first full draft of this specification, which went to public review in 1992. A conforming implementation which has the :draft-ansi-cl-2 or :ansi-cl feature is not permitted to retain the :draft-ansi-cl feature since incompatible changes were made subsequent to the first draft.

:draft-ansi-cl-2

If present, indicates that a second full draft of this specification has gone to public review, and that the implementation purports to conform to that specification. (If additional public review drafts are produced, this keyword will continue to refer to the second draft, and additional keywords will be added to identify conformance with such later drafts. As such, the meaning of this keyword can be relied upon not to change over time.) A conforming implementation which has the :ansi-cl feature is only permitted to retain the :draft-ansi-cl feature if the finally approved standard is not incompatible with the draft standard.

:ansi-cl

If present, indicates that this specification has been adopted by ANSI as an official standard, and that the implementation purports to conform.

:common-lisp

This feature must appear in *features* for any implementation that has one or more of the features :x3j13, :draft-ansi-cl, or :ansi-cl. It is intended that it should also appear in implementations which have the features :cltl1 or :cltl2, but this specification cannot force such behavior. The intent is that this feature should identify the language family named “Common Lisp,” rather than some specific dialect within that family.

See Also:
Section 1.5.2.1.1 (Use of Read-Time Conditionals), Section 2.4 (Standard Macro Characters)

Notes:
The value of *features* is used by the #+ and #- reader syntax.

Symbols in the features list may be in any package, but in practice they are generally in the KEYWORD package. This is because KEYWORD is the package used by default when reading feature expressions in the #+ and #- reader macros. Code that needs to name a feature in a package P (other than KEYWORD) can do so by making explicit use of a package prefix for P, but note that such code must also assure that the package P exists in order for the feature expression to be read—even in cases where the feature expression is expected to fail.

It is generally considered wise for an implementation to include one or more features identifying
the specific implementation, so that conditional expressions can be written which distinguish idiosyncrasies of one implementation from those of another. Since features are normally symbols in the KEYWORD package where name collisions might easily result, and since no uniquely defined mechanism is designated for deciding who has the right to use which symbol for what reason, a conservative strategy is to prefer names derived from one’s own company or product name, since those names are often trademarked and are hence less likely to be used unwittingly by another implementation.

*compile-file-pathname*, *compile-file-truename*

Variable

Value Type:
The value of *compile-file-pathname* must always be a pathname or nil. The value of *compile-file-truename* must always be a physical pathname or nil.

Initial Value:
nil.

Description:
During a call to compile-file, *compile-file-pathname* is bound to the pathname denoted by the first argument to compile-file, merged against the defaults; that is, it is bound to (pathname (merge-pathnames input-file)). During the same time interval, *compile-file-truename* is bound to the truename of the file being compiled.

At other times, the value of these variables is nil.

If a break loop is entered while compile-file is ongoing, it is implementation-dependent whether these variables retain the values they had just prior to entering the break loop or whether they are bound to nil.

The consequences are unspecified if an attempt is made to assign or bind either of these variables.

Affected By:
The file system.

See Also:
compile-file

24–12 Programming Language—Common Lisp
**load-pathname**, **load-truename**

**Variable**

**Value Type:**
The value of **load-pathname** must always be a pathname or nil. The value of **load-truename** must always be a physical pathname or nil.

**Initial Value:**
nil.

**Description:**
During a call to load, **load-pathname** is bound to the pathname denoted by the first argument to load, merged against the defaults; that is, it is bound to (pathname (merge-pathnames filespec)). During the same time interval, **load-truename** is bound to the truename of the file being loaded.

At other times, the value of these variables is nil.

If a break loop is entered while load is ongoing, it is implementation-dependent whether these variables retain the values they had just prior to entering the break loop or whether they are bound to nil.

The consequences are unspecified if an attempt is made to assign or bind either of these variables.

**Affected By:**
The file system.

**See Also:**
load

**compile-print**, **compile-verbose**

**Variable**

**Value Type:**
a generalized boolean.

**Initial Value:**
implementation-dependent.

**Description:**
The value of **compile-print** is the default value of the :print argument to compile-file. The value of **compile-verbose** is the default value of the :verbose argument to compile-file.
See Also:

compile-file

*load-print*, *load-verbose*

Variable

Value Type:
a generalized boolean.

Initial Value:
The initial value of *load-print* is false. The initial value of *load-verbose* is implementation-dependent.

Description:
The value of *load-print* is the default value of the :print argument to load. The value of *load-verbose* is the default value of the :verbose argument to load.

See Also:

load

*modules*

Variable

Value Type:
a list of strings.

Initial Value:
implementation-dependent.

Description:
The value of *modules* is a list of names of the modules that have been loaded into the current Lisp image.

Affected By:
provide

See Also:
provide, require

Notes:
The variable *modules* is deprecated.
provide, require

Function

Syntax:

provide module-name → implementation-dependent
require module-name &optional pathname-list → implementation-dependent

Arguments and Values:

module-name—a string designator.

pathname-list—nil, or a designator for a non-empty list of pathname designators. The default is nil.

Description:

provide adds the module-name to the list held by *modules*, if such a name is not already present.

require tests for the presence of the module-name in the list held by *modules*. If it is present, require immediately returns. Otherwise, an attempt is made to load an appropriate set of files as follows: The pathname-list argument, if non-nil, specifies a list of pathnames to be loaded in order, from left to right. If the pathname-list is nil, an implementation-dependent mechanism will be invoked in an attempt to load the module named module-name; if no such module can be loaded, an error of type error is signaled.

Both functions use string= to test for the presence of a module-name.

Examples:

;;; This illustrates a nonportable use of REQUIRE, because it
;;; depends on the implementation-dependent file-loading mechanism.

(require "CALCULUS")

;;; This use of REQUIRE is nonportable because of the literal
;;; physical pathname.

(require "CALCULUS" "/usr/lib/lisp/calculus")

;;; One form of portable usage involves supplying a logical pathname,
;;; with appropriate translations defined elsewhere.

(require "CALCULUS" "lib:calculus")

System Construction 24–15
provide, require

;;; Another form of portable usage involves using a variable or
;;; table lookup function to determine the pathname, which again
;;; must be initialized elsewhere.

(require "CALCULUS" *calculus-module-pathname*)

Side Effects:
provide modifies *modules*.

Affected By:
The specific action taken by require is affected by calls to provide (or, in general, any changes to the value of *modules*).

Exceptional Situations:
Should signal an error of type type-error if module-name is not a string designator.

If require fails to perform the requested operation due to a problem while interacting with the file system, an error of type file-error is signaled.

An error of type file-error might be signaled if any pathname in pathname-list is a designator for a wild pathname.

See Also:
*modules*, Section 19.1.2 (Pathnames as Filenames)

Notes:
The functions provide and require are deprecated.

If a module consists of a single package, it is customary for the package and module names to be the same.
Programming Language—Common Lisp

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. Figure 25–1 lists variables that are maintained by the Lisp read-eval-print loop.

<table>
<thead>
<tr>
<th>*</th>
<th>+</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>++</td>
<td>//</td>
</tr>
<tr>
<td>***</td>
<td>+++</td>
<td>///</td>
</tr>
</tbody>
</table>

Figure 25–1. Variables maintained by the Read-Eval-Print Loop

25.1.2 Debugging Utilities

Figure 25–2 shows defined names relating to debugging.

<table>
<thead>
<tr>
<th><em>debugger-hook</em></th>
<th>documentation</th>
<th>step</th>
</tr>
</thead>
<tbody>
<tr>
<td>apropos</td>
<td>dribble</td>
<td>time</td>
</tr>
<tr>
<td>apropos-list</td>
<td>ed</td>
<td>trace</td>
</tr>
<tr>
<td>break</td>
<td>inspect</td>
<td>untrace</td>
</tr>
<tr>
<td>describe</td>
<td>invoke-debugger</td>
<td></td>
</tr>
</tbody>
</table>

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.

Figure 25–3 shows 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.

Figure 25–4 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 |

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.

Figure 25–5 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.
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.

<table>
<thead>
<tr>
<th>decode-universal-time</th>
<th>get-universal-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>encode-universal-time</td>
<td></td>
</tr>
</tbody>
</table>

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.

Figure 25–7 shows defined names related to internal time.

<table>
<thead>
<tr>
<th>get-internal-real-time</th>
<th>internal-time-units-per-second</th>
</tr>
</thead>
<tbody>
<tr>
<td>get-internal-run-time</td>
<td></td>
</tr>
</tbody>
</table>

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.
decode-universal-time

Syntax:

\[ \text{decode-universal-time} \ \text{universal-time} \ \&\text{optional} \ \text{time-zone} \rightarrow \text{second, minute, hour, date, month, year, day, daylight-p, zone} \]

Arguments and Values:

- \text{universal-time}—a \text{universal time}.
- \text{time-zone}—a \text{time zone}.
- \text{second, minute, hour, date, month, year, day, daylight-p, zone}—a \text{decoded time}.

Description:

Returns the \text{decoded time} represented by the given \text{universal time}.

If \text{time-zone} is not supplied, it defaults to the current time zone adjusted for daylight saving time. If \text{time-zone} is supplied, daylight saving time information is ignored. The daylight saving time flag is \text{nil} if \text{time-zone} is supplied.

Examples:

\[
\begin{align*}
\text{(decode-universal-time 0 0)} & \rightarrow 0, 0, 0, 1, 1, 1900, 0, \text{false}, 0 \\
\text{(decode-universal-time 2414296800 5)} & \rightarrow 0, 0, 1, 4, 7, 1976, 6, \text{false}, 5 \\
\text{(decode-universal-time 2414293200)} & \rightarrow 0, 0, 1, 4, 7, 1976, 6, \text{true}, 5 \\
\end{align*}
\]

\[
\begin{align*}
\text{;; This example assumes that the time zone is Eastern Daylight Time} \\
\text{;; (and that the time zone is constant throughout the example).} \\
\text{(let* ((here (nth 8 (multiple-value-list (get-decoded-time)))) ; Time zone} \\
\text{ (recently (get-universal-time)))} \\
\text{ (a (nthcdr 7 (multiple-value-list (decode-universal-time recently))))} \\
\text{ (b (nthcdr 7 (multiple-value-list (decode-universal-time recently here))))} \\
\text{ (list a b (equal a b)))} & \rightarrow ((\text{T 5}) (\text{NIL 5}) \text{NIL})
\end{align*}
\]

Affected By:

\text{Implementation-dependent} mechanisms for calculating when or if daylight savings time is in effect for any given session.

See Also:

encode-universal-time, get-universal-time, Section 25.1.4 (Time)
encode-universal-time

Syntax:

\[
\text{encode-universal-time} \ second \ minute \ hour \ date \ month \ year \ \text{optional} \ time-zone \\
\rightarrow \ universal-time
\]

Arguments and Values:

\text{second, minute, hour, date, month, year, time-zone}—the corresponding parts of a \textit{decoded time}.
(Note that some of the nine values in a full \textit{decoded time} are redundant, and so are not used as inputs to this function.)

\text{universal-time}—a \textit{universal time}.

Description:

\text{encode-universal-time} converts a time from Decoded Time format to a \textit{universal time}.

If \textit{time-zone} is supplied, no adjustment for daylight savings time is performed.

Examples:

\begin{align*}
(\text{encode-universal-time} \ 0 \ 0 \ 0 \ 1 \ 1 \ 1900 \ 0) & \rightarrow \ 0 \\
(\text{encode-universal-time} \ 0 \ 0 \ 1 \ 4 \ 7 \ 1976 \ 5) & \rightarrow \ 2414296800 \\
\text{;; The next example assumes Eastern Daylight Time.}
(\text{encode-universal-time} \ 0 \ 0 \ 1 \ 4 \ 7 \ 1976) & \rightarrow \ 2414293200
\end{align*}

See Also:

decode-universal-time, get-decoded-time

get-universal-time, get-decoded-time

Syntax:

\[
\text{get-universal-time} \ \langle \text{no arguments} \rangle \rightarrow \ universal-time \\
\text{get-decoded-time} \ \langle \text{no arguments} \rangle \\
\rightarrow \ second, minute, hour, date, month, year, \text{daylight-p, zone}
\]

Arguments and Values:

\text{universal-time}—a \textit{universal time}.

\text{second, minute, hour, date, month, year, \text{daylight-p, zone}}—a \textit{decoded time}.

25–6 Programming Language—Common Lisp
Description:

get-universal-time returns the current time, represented as a *universal time*.

get-decoded-time returns the current time, represented as a *decoded time*.

Examples:

```lisp
;; At noon on July 4, 1976 in Eastern Daylight Time.
(get-decoded-time) → 0, 0, 12, 4, 7, 1976, 6, true, 5

;; At exactly the same instant.
(get-universal-time) → 2414332800

;; Exactly five minutes later.
(get-universal-time) → 2414333100

;; The difference is 300 seconds (five minutes)
(+ **) → 300
```

Affected By:

The time of day (*i.e.*, the passage of time), the system clock’s ability to keep accurate time, and the accuracy of the system clock’s initial setting.

Exceptional Situations:

An error of *type error* might be signaled if the current time cannot be determined.

See Also:

decode-universal-time, encode-universal-time, Section 25.1.4 (Time)

Notes:

\[
\text{(get-decoded-time)} \equiv \text{(decode-universal-time (get-universal-time))}
\]

No *implementation* is required to have a way to verify that the time returned is correct. However, if an *implementation* provides a validity check (*e.g.*, the failure to have properly initialized the system clock can be reliably detected) and that validity check fails, the *implementation* is strongly encouraged (but not required) to signal an error of *type error* (rather than, for example, returning a known-to-be-wrong value) that is *correctable* by allowing the user to interactively set the correct time.

---

**sleep**

*Function*

Syntax:

\[
\text{sleep} \; \text{seconds} \; \rightarrow \; \text{nil}
\]

Arguments and Values:

\[
\text{seconds} \; - \; \text{a non-negative real.}
\]
Description:
Causes execution to cease and become dormant for approximately the seconds of real time indicated by seconds, whereupon execution is resumed.

Examples:
(sleep 1) → NIL

;; Actually, since SLEEP is permitted to use approximate timing,
;; this might not always yield true, but it will often enough that
;; we felt it to be a productive example of the intent.
(let ((then (get-universal-time))
   (now (progn (sleep 10) (get-universal-time)))))
(>= (~ now then) 10))
→ true

Side Effects:
Causes processing to pause.

Affected By:
The granularity of the scheduler.

Exceptional Situations:
Should signal an error of type type-error if seconds is not a non-negative real.

apropos, apropos-list  Function

Syntax:
apropos string &optional package  → ⟨no values⟩
apropos-list string &optional package  → symbols

Arguments and Values:
string—a string designator.
package—a package designator or nil. The default is nil.
symbols—a list of symbols.

Description:
These functions search for interned symbols whose names contain the substring string.
For **apropos**, as each such **symbol** is found, its name is printed on **standard output**. In addition, if such a **symbol** is defined as a **function** or **dynamic variable**, information about those definitions might also be printed.

For **apropos-list**, no output occurs as the search proceeds; instead a list of the matching **symbols** is returned when the search is complete.

If **package** is non-nil, only the **symbols accessible** in that **package** are searched; otherwise all **symbols accessible in any package** are searched.

Because a **symbol** might be available by way of more than one inheritance path, **apropos** might print information about the **same symbol** more than once, or **apropos-list** might return a **list containing duplicate symbols**.

Whether or not the search is case-sensitive is **implementation-defined**.

**Affected By:**

The set of **symbols** which are currently **interned** in any **packages** being searched.

**apropos** is also affected by ***standard-output***.

---

### describe

**Function**

**Syntax:**

\[
\text{describe } \text{object} \text{ optional stream } \rightarrow \text{(no values)}
\]

**Arguments and Values:**

- **object**—an object.
- **stream**—an output stream designator. The default is **standard output**.

**Description:**

**describe** displays information about **object** to **stream**.

For example, **describe** of a **symbol** might show the **symbol’s value**, its definition, and each of its properties. **describe** of a **float** might show the number’s internal representation in a way that is useful for tracking down round-off errors. In all cases, however, the nature and format of the output of **describe** is **implementation-dependent**.

**describe** can describe something that it finds inside the **object**; in such cases, a notational device such as increased indentation or positioning in a table is typically used in order to visually distinguish such recursive descriptions from descriptions of the argument **object**.

---

Environment  25–9
The actual act of describing the object is implemented by `describe-object`. `describe` exists as an interface primarily to manage argument defaulting (including conversion of arguments `t` and `nil` into stream objects) and to inhibit any return values from `describe-object`.

`describe` is not intended to be an interactive function. In a conforming implementation, `describe` must not, by default, prompt for user input. User-defined methods for `describe-object` are likewise restricted.

**Side Effects:**
Output to *standard output* or *terminal I/O*.

**Affected By:**
*standard-output* and *terminal-io*, methods on `describe-object` and `print-object` for objects having user-defined classes.

**See Also:**
inspect, `describe-object`

---

**describe-object**

*Standard Generic Function*

**Syntax:**
```
describe-object object stream → implementation-dependent
```

**Method Signatures:**
```
describe-object (object standard-object) stream
```

**Arguments and Values:**
- `object`—an object.
- `stream`—a stream.

**Description:**
The generic function `describe-object` prints a description of `object` to a `stream`. `describe-object` is called by `describe`; it must not be called by the user.

Each implementation is required to provide a method on the class `standard-object` and methods on enough other classes so as to ensure that there is always an applicable method. Implementations are free to add methods for other classes. Users can write methods for `describe-object` for their own classes if they do not wish to inherit an implementation-supplied method.

Methods on `describe-object` can recursively call `describe`. Indentation, depth limits, and circularity detection are all taken care of automatically, provided that each method handles exactly one level of structure and calls `describe` recursively if there are more structural levels. The consequences are undefined if this rule is not obeyed.
In some implementations the stream argument passed to a \texttt{describe-object} method is not the original stream, but is an intermediate stream that implements parts of \texttt{describe}. Methods should therefore not depend on the identity of this stream.

\textbf{Examples:}

\begin{verbatim}
(defun class spaceship ()
  ((captain :initarg :captain :accessor spaceship-captain)
   (serial# :initarg :serial-number :accessor spaceship-serial-number)))

(defun class federation-starship (spaceship) ())

(defun method describe-object ((s spaceship) stream)
  (with-slots (captain serial#) s
    (format stream "~&~S is a spaceship of type ~S,~
      with ~A at the helm ~
      and with serial number ~D."
    s (type-of s) captain serial#)))

(make-instance 'federation-starship
  :captain "Rachel Garrett"
  :serial-number "NCC-1701-C")
-> #<FEDERATION-STARSHIP 26312465>

(describe *)
> #<FEDERATION-STARSHIP 26312465> is a spaceship of type FEDERATION-STARSHIP,
> with Rachel Garrett at the helm and with serial number NCC-1701-C.
> (no values)
\end{verbatim}

\textbf{See Also:}

\begin{itemize}
\item \texttt{describe}
\end{itemize}

\textbf{Notes:}

The same implementation techniques that are applicable to \texttt{print-object} are applicable to \texttt{describe-object}.

The reason for making the return values for \texttt{describe-object} unspecified is to avoid forcing users to include explicit (\texttt{values}) in all of their methods. \texttt{describe} takes care of that.
trace, untrace

Macro

Syntax:

\[
\text{trace } \{\text{function-name}\}^* \rightarrow \text{trace-result}
\]

\[
\text{untrace } \{\text{function-name}\}^* \rightarrow \text{untrace-result}
\]

Arguments and Values:

- \text{function-name} — a function name.
- \text{trace-result} — implementation-dependent, unless no \text{function-names} are supplied, in which case \text{trace-result} is a list of function names.
- \text{untrace-result} — implementation-dependent.

Description:

\text{trace} and \text{untrace} control the invocation of the trace facility.

Invoking \text{trace} with one or more \text{function-names} causes the denoted \text{functions} to be “traced.” Whenever a traced \text{function} is invoked, information about the call, about the arguments passed, and about any eventually returned values is printed to \text{trace output}. If \text{trace} is used with no \text{function-names}, no tracing action is performed; instead, a list of the \text{functions} currently being traced is returned.

Invoking \text{untrace} with one or more function names causes those functions to be “untraced” (i.e., no longer traced). If \text{untrace} is used with no \text{function-names}, all \text{functions} currently being traced are untraced.

If a \text{function} to be traced has been open-coded (e.g., because it was declared \text{inline}), a call to that \text{function} might not produce trace output.

Examples:

\[
\begin{align*}
\text{(defun fact (n)} & \text{ (if (zerop n) 1 (* n (fact (- n 1))))) return FACT} \\
& \text{(trace fact)} \\
& \text{(FACT)} \\
& \text{; Of course, the format of traced output is implementation-dependent.} \\
& \text{(fact 3)} \\
& \text{> 1 Enter FACT 3} \\
& \text{> 2 Enter FACT 2} \\
& \text{> 3 Enter FACT 1} \\
& \text{> 4 Enter FACT 0} \\
& \text{> 4 Exit FACT 1} \\
& \text{> 3 Exit FACT 1} \\
& \text{> 2 Exit FACT 2}
\end{align*}
\]
Side Effects:
Might change the definitions of the functions named by function-names.

Affected By:
Whether the functions named are defined or already being traced.

Exceptional Situations:
Tracing an already traced function, or untracing a function not currently being traced, should produce no harmful effects, but might signal a warning.

See Also:
*trace-output*, step

Notes:
trace and untrace may also accept additional implementation-dependent argument formats. The format of the trace output is implementation-dependent.

Although trace can be extended to permit non-standard options, implementations are nevertheless encouraged (but not required) to warn about the use of syntax or options that are neither specified by this standard nor added as an extension by the implementation, since they could be symptomatic of typographical errors or of reliance on features supported in implementations other than the current implementation.

**step**

Macro

Syntax:
step form → {result}*

Arguments and Values:
form—a form; evaluated as described below.
results—the values returned by the form.

Description:
step implements a debugging paradigm wherein the programmer is allowed to step through the evaluation of a form. The specific nature of the interaction, including which I/O streams are used and whether the stepping has lexical or dynamic scope, is implementation-defined.
**step** evaluates *form* in the current *environment*. A call to **step** can be compiled, but it is acceptable for an implementation to interactively step through only those parts of the computation that are interpreted.

It is technically permissible for a *conforming implementation* to take no action at all other than normal execution of the *form*. In such a situation, (**step form**) is equivalent to, for example, (**(let () form)**). In implementations where this is the case, the associated documentation should mention that fact.

**See Also:**
- **trace**

**Notes:**

*Implementations* are encouraged to respond to the typing of ? or the pressing of a “help key” by providing help including a list of commands.

---

**time**  
*Macro*

**Syntax:**

```
time form  \rightarrow  \{result\}*
```

**Arguments and Values:**

- *form*—a *form*; evaluated as described below.
- *results*—the *values* returned by the *form*.

**Description:**

**time** evaluates *form* in the current *environment* (lexical and dynamic). A call to **time** can be compiled.

**time** prints various timing data and other information to *trace output*. The nature and format of the printed information is *implementation-defined*. Implementations are encouraged to provide such information as elapsed real time, machine run time, and storage management statistics.

**Affected By:**

The accuracy of the results depends, among other things, on the accuracy of the corresponding functions provided by the underlying operating system.

The magnitude of the results may depend on the hardware, the operating system, the lisp implementation, and the state of the global environment. Some specific issues which frequently affect the outcome are hardware speed, nature of the scheduler (if any), number of competing processes (if any), system paging, whether the call is interpreted or compiled, whether functions called are compiled, the kind of garbage collector involved and whether it runs, whether internal data structures (e.g., hash tables) are implicitly reorganized, *etc*.
internal-time-units-per-second

Constant Variable

Constant Value:
A positive integer, the magnitude of which is implementation-dependent.

Description:
The number of internal time units in one second.

See Also:
get-internal-run-time, get-internal-real-time

Notes:
These units form the basis of the Internal Time format representation.

get-internal-real-time

Function

Syntax:
get-internal-real-time ⟨no arguments⟩ → internal-time

Arguments and Values:
internal-time—a non-negative integer.

Description:
get-internal-real-time returns as an integer the current time in internal time units, relative to an arbitrary time base. The difference between the values of two calls to this function is the amount of elapsed real time (i.e., clock time) between the two calls.

Affected By:
Time of day (i.e., the passage of time). The time base affects the result magnitude.
get-internal-run-time

Function

Syntax:

get-internal-run-time (no arguments) → internal-time

Arguments and Values:

internal-time—a non-negative integer.

Description:

Returns as an integer the current run time in internal time units. The precise meaning of this quantity is implementation-defined; it may measure real time, run time, CPU cycles, or some other quantity. The intent is that the difference between the values of two calls to this function be the amount of time between the two calls during which computational effort was expended on behalf of the executing program.

Affected By:

The implementation, the time of day (i.e., the passage of time).

See Also:

internal-time-units-per-second

Notes:

Depending on the implementation, paging time and garbage collection time might be included in this measurement. Also, in a multitasking environment, it might not be possible to show the time for just the running process, so in some implementations, time taken by other processes during the same time interval might be included in this measurement as well.
disassemble

Function

Syntax:

disassemble fn → nil

Arguments and Values:

fn—an extended function designator or a lambda expression.

Description:

The function disassemble is a debugging aid that composes symbolic instructions or expressions in some implementation-dependent language which represent the code used to produce the function which is or is named by the argument fn. The result is displayed to standard output in an implementation-dependent format.

If fn is a lambda expression or interpreted function, it is compiled first and the result is disassembled.

If the fn designator is a function name, the function that it names is disassembled. (If that function is an interpreted function, it is first compiled but the result of this implicit compilation is not installed.)

Examples:

(\(defun f (a) (1+ a)) → F
(eq (symbol-function 'f)
  (progn (disassemble 'f)
    (symbol-function 'f))) → true

Affected By:

*standard-output*.

Exceptional Situations:

Should signal an error of type type-error if fn is not an extended function designator or a lambda expression.

documentation, (setf documentation)

Standard Generic

Function

Syntax:

documentation x doc-type → documentation

Environment 25–17
documentation, (setf documentation)

(setf documentation) new-value x doc-type → new-value

Argument Precedence Order:
doc-type, object

Method Signatures:
Functions, Macros, and Special Forms:
documentation (x function) (doc-type (eql 't))
documentation (x function) (doc-type (eql 'function))
documentation (x list) (doc-type (eql 'function))
documentation (x list) (doc-type (eql 'compiler-macro))
documentation (x symbol) (doc-type (eql 'function))
documentation (x symbol) (doc-type (eql 'compiler-macro))
documentation (x symbol) (doc-type (eql 'setf))
(setf documentation) new-value (x function) (doc-type (eql 't))
(setf documentation) new-value (x function) (doc-type (eql 'function))
(setf documentation) new-value (x list) (doc-type (eql 'function))
(setf documentation) new-value (x list) (doc-type (eql 'compiler-macro))
(setf documentation) new-value (x symbol) (doc-type (eql 'function))
(setf documentation) new-value (x symbol) (doc-type (eql 'compiler-macro))
(setf documentation) new-value (x symbol) (doc-type (eql 'setf))

Method Combinations:
documentation (x method-combination) (doc-type (eql 't))
documentation (x method-combination) (doc-type (eql 'method-combination))
documentation (x symbol) (doc-type (eql 'method-combination))
(setf documentation) new-value (x method-combination) (doc-type (eql 't))
(setf documentation) new-value (x method-combination) (doc-type (eql 'method-combination))
(setf documentation) new-value (x symbol) (doc-type (eql 'method-combination))
documentation, (setf documentation)

Methods:

documentation (x standard-method) (doc-type (eql 't))
(setf documentation) new-value (x standard-method) (doc-type (eql 't))

Packages:

documentation (x package) (doc-type (eql 't))
(setf documentation) new-value (x package) (doc-type (eql 't))

Types, Classes, and Structure Names:

documentation (x standard-class) (doc-type (eql 't))
documentation (x standard-class) (doc-type (eql 'type))
documentation (x structure-class) (doc-type (eql 't))
documentation (x structure-class) (doc-type (eql 'type))
documentation (x symbol) (doc-type (eql 'type))
documentation (x symbol) (doc-type (eql 'structure))
(setf documentation) new-value (x standard-class) (doc-type (eql 't))
(setf documentation) new-value (x standard-class) (doc-type (eql 'type))
(setf documentation) new-value (x structure-class) (doc-type (eql 't))
(setf documentation) new-value (x structure-class) (doc-type (eql 'type))
(setf documentation) new-value (x symbol) (doc-type (eql 'type))
(setf documentation) new-value (x symbol) (doc-type (eql 'structure))

Variables:

documentation (x symbol) (doc-type (eql 'variable))
(setf documentation) new-value (x symbol) (doc-type (eql 'variable))

Arguments and Values:

x—an object.
doc-type—a symbol.
documentation—a string, or nil.
documentation, (setf documentation)

*new-value*—a string.

**Description:**

The *generic function* **documentation** returns the *documentation string* associated with the given *object* if it is available; otherwise it returns **nil**.

The *generic function* **(setf documentation)** updates the *documentation string* associated with *x* to *new-value*. If *x* is a *list*, it must be of the form **(setf symbol)**.

_Documentation strings_ are made available for debugging purposes. _Conforming programs_ are permitted to use _documentation strings_ when they are present, but should not depend for their correct behavior on the presence of those _documentation strings_. An _implementation_ is permitted to discard _documentation strings_ at any time for _implementation-defined_ reasons.

The nature of the _documentation string_ returned depends on the _doc-type_, as follows:

- **compiler-macro**
  
  Returns the *documentation string* of the _compiler macro_ whose _name_ is the _function_ _name_ *x*.

- **function**
  
  If *x* is a _function name_, returns the *documentation string* of the _function_, _macro_, or _special operator_ whose _name_ is *x*.

  If *x* is a _function_, returns the *documentation string* associated with *x*.

- **method-combination**
  
  If *x* is a _symbol_, returns the *documentation string* of the _method combination_ whose _name_ is *x*.

  If *x* is a _method combination_, returns the *documentation string* associated with *x*.

- **setf**
  
  Returns the *documentation string* of the _setf expander_ whose _name_ is the _symbol_ *x*.

- **structure**
  
  Returns the *documentation string* associated with the _structure name_ *x*.

- **t**
  
  Returns a _documentation string_ specialized on the _class_ of the argument *x* itself. For example, if *x* is a _function_, the *documentation string* associated with the _function_ *x* is returned.
type

If \( x \) is a symbol, returns the documentation string of the class whose name is the symbol \( x \), if there is such a class. Otherwise, it returns the documentation string of the type which is the type specifier symbol \( x \).

If \( x \) is a structure class or standard class, returns the documentation string associated with the class \( x \).

variable

Returns the documentation string of the dynamic variable or constant variable whose name is the symbol \( x \).

A conforming implementation or a conforming program may extend the set of symbols that are acceptable as the doc-type.

Notes:

This standard prescribes no means to retrieve the documentation strings for individual slots specified in a defclass form, but implementations might still provide debugging tools and/or programming language extensions which manipulate this information. Implementors wishing to provide such support are encouraged to consult the Metaobject Protocol for suggestions about how this might be done.

---

**room**

**Function**

**Syntax:**

```
room &optional x  → implementation-dependent
```

**Arguments and Values:**

- \( x \) — one of \( t \), \( nil \), or \( :default \).

**Description:**

*room* prints, to standard output, information about the state of internal storage and its management. This might include descriptions of the amount of memory in use and the degree of memory compaction, possibly broken down by internal data type if that is appropriate. The nature and format of the printed information is implementation-dependent. The intent is to provide information that a programmer might use to tune a program for a particular implementation.

*room nil* prints out a minimal amount of information. *room t* prints out a maximal amount of information. *room* or *room :default* prints out an intermediate amount of information that is likely to be useful.
Side Effects:
Output to standard output.

Affected By:
*standard-output*.

---

**ed**

*Function*

**Syntax:**

\[
\text{ed} \quad \text{optional} \ x \quad \rightarrow \quad \text{implementation-dependent}
\]

**Arguments and Values:**

\(x\)—nil, a pathname, a string, or a function name. The default is nil.

**Description:**

ed invokes the editor if the implementation provides a resident editor.

If \(x\) is nil, the editor is entered. If the editor had been previously entered, its prior state is resumed, if possible.

If \(x\) is a pathname or string, it is taken as the pathname designator for a file to be edited.

If \(x\) is a function name, the text of its definition is edited. The means by which the function text is obtained is implementation-defined.

**Exceptional Situations:**

The consequences are undefined if the implementation does not provide a resident editor.

Might signal type-error if its argument is supplied but is not a symbol, a pathname, or nil.

If a failure occurs when performing some operation on the file system while attempting to edit a file, an error of type file-error is signaled.

An error of type file-error might be signaled if \(x\) is a designator for a wild pathname.

Implementation-dependent additional conditions might be signaled as well.

**See Also:**

pathname, logical-pathname, compile-file, load, Section 19.1.2 (Pathnames as Filenames)
inspect

Function

Syntax:
inspect object  → implementation-dependent

Arguments and Values:
object—an object.

Description:
inspect is an interactive version of describe. The nature of the interaction is implementation-dependent, but the purpose of inspect is to make it easy to wander through a data structure, examining and modifying parts of it.

Side Effects:
implementation-dependent.

Affected By:
implementation-dependent.

Exceptional Situations:
implementation-dependent.

See Also:
describe

Notes:
Implementations are encouraged to respond to the typing of ? or a “help key” by providing help, including a list of commands.

dribble

Function

Syntax:
dribble &optional pathname  → implementation-dependent

Arguments and Values:
pathname—a pathname designator.
Description:

Either binds *standard-input* and *standard-output* or takes other appropriate action, so as to send a record of the input/output interaction to a file named by pathname. dribble is intended to create a readable record of an interactive session.

If pathname is a logical pathname, it is translated into a physical pathname as if by calling translate-logical-pathname.

(dribble) terminates the recording of input and output and closes the dribble file.

If dribble is called while a stream to a “dribble file” is still open from a previous call to dribble, the effect is implementation-defined. For example, the already-open stream might be closed, or dribbling might occur both to the old stream and to a new one, or the old stream might stay open but not receive any further output, or the new request might be ignored, or some other action might be taken.

Affected By:

The implementation.

Exceptional Situations:

If a failure occurs when performing some operation on the file system while creating the dribble file, an error of type file-error is signaled.

An error of type file-error might be signaled if pathname is a designator for a wild pathname.

See Also:

Section 19.1.2 (Pathnames as Filenames)

Notes:

dribble can return before subsequent forms are executed. It also can enter a recursive interaction loop, returning only when (dribble) is done.

dribble is intended primarily for interactive debugging; its effect cannot be relied upon when used in a program.

—

Variable

Value Type:

a form.

Initial Value:

implementation-dependent.
Description:
The value of - is the form that is currently being evaluated by the Lisp read-eval-print loop.

Examples:

```lisp
(format t "~&Evaluating ~S~%")
> Evaluating (FORMAT T "~&Evaluating ~S~%")
→ NIL
```

Affected By:
Lisp read-eval-print loop.

See Also:
+ (variable), * (variable), / (variable), Section 25.1.1 (Top level loop)

### +, ++, +++

**Variable**

**Value Type:**
an object.

**Initial Value:**
implementation-dependent.

**Description:**
The variables +, ++, and +++ are maintained by the Lisp read-eval-print loop to save forms that were recently evaluated.

The value of + is the last form that was evaluated, the value of ++ is the previous value of +, and the value of +++ is the previous value of ++.

**Examples:**

```lisp
(+ 0 1) → 1
(- 4 2) → 2
(/ 9 3) → 3
(list + ++ +++) → ((/ 9 3) (- 4 2) (+ 0 1))
(setq a 1 b 2 c 3 d (list a b c)) → (1 2 3)
(setq a 4 b 5 c 6 d (list a b c)) → (4 5 6)
(list a b c) → (4 5 6)
(eval +++) → (1 2 3)
#:(',(#(d) 1 2 3)) → (1 2 3 (1 2 3))
```

Environment 25–25
Affected By:

*Lisp read-eval-print loop.*

See Also:

- (variable), * (variable), / (variable), Section 25.1.1 (Top level loop)

<table>
<thead>
<tr>
<th>*, **, ***</th>
<th>Variable</th>
</tr>
</thead>
</table>

**Value Type:**

an object.

**Initial Value:**

implementation-dependent.

**Description:**

The variables *, **, and *** are maintained by the *Lisp read-eval-print loop* to save the values of results that are printed each time through the loop.

The *value* of * is the most recent primary value that was printed, the *value* of ** is the previous value of *, and the *value* of *** is the previous value of **.

If several values are produced, * contains the first value only; * contains nil if zero values are produced.

The *values* of *, **, and *** are updated immediately prior to printing the return value of a top-level form by the *Lisp read-eval-print loop*. If the *evaluation* of such a form is aborted prior to its normal return, the values of *, **, and *** are not updated.

**Examples:**

```lisp
(values 'a1 'a2) → A1, A2
'b → B
(values 'c1 'c2 'c3) → C1, C2, C3
(list * ** ***) → (C1 B A1)
```

```lisp
(defun cube-root (x) (expt x 1/3)) → CUBE-ROOT
(compile *) → CUBE-ROOT
(setq a (cube-root 27.0)) → 3.0
(* a 9.0) → 27.0
```

**Affected By:**

*Lisp read-eval-print loop.*
See Also:
- (variable), + (variable), / (variable), Section 25.1.1 (Top level loop)

Notes:

* ≡ (car /)
** ≡ (car //)
*** ≡ (car ///)

/ , // , ///

Value Type:
a proper list.

Initial Value:
implementation-dependent.

Description:
The variables /, //, and /// are maintained by the Lisp read-eval-print loop to save the values of results that were printed at the end of the loop.

The value of / is a list of the most recent values that were printed, the value of // is the previous value of /, and the value of /// is the previous value of //.

The values of /, //, and /// are updated immediately prior to printing the return value of a top-level form by the Lisp read-eval-print loop. If the evaluation of such a form is aborted prior to its normal return, the values of /, //, and /// are not updated.

Examples:

(floor 22 7) → 3, 1
(+ (* (car /) 7) (cadr /)) → 22

Affected By:
Lisp read-eval-print loop.

See Also:
- (variable), + (variable), * (variable), Section 25.1.1 (Top level loop)
lisp-implementation-type, lisp-implementation-version

Syntax:

lisp-implementation-type (no arguments) → description
lisp-implementation-version (no arguments) → description

Arguments and Values:

description—a string or nil.

Description:

lisp-implementation-type and lisp-implementation-version identify the current implementation of Common Lisp.

lisp-implementation-type returns a string that identifies the generic name of the particular Common Lisp implementation.

lisp-implementation-version returns a string that identifies the version of the particular Common Lisp implementation.

If no appropriate and relevant result can be produced, nil is returned instead of a string.

Examples:

(lisp-implementation-type)
→ "ACME Lisp"
or "Joe’s Common Lisp"
(lisp-implementation-version)
→ "1.3a"
→ "V2"
or "Release 17.3, ECO #6"

short-site-name, long-site-name

Syntax:

short-site-name (no arguments) → description
long-site-name (no arguments) → description
Arguments and Values:

description—a string or nil.

Description:

short-site-name and long-site-name return a string that identifies the physical location of the computer hardware, or nil if no appropriate description can be produced.

Examples:

(short-site-name)
→ "MIT AI Lab"
or
→ "CMU-CSD"

(long-site-name)
→ "MIT Artificial Intelligence Laboratory"
or
→ "CMU Computer Science Department"

Affected By:
The implementation, the location of the computer hardware, and the installation/configuration process.

machine-instance

Function

Syntax:
machine-instance (no arguments) → description

Arguments and Values:
description—a string or nil.

Description:

Returns a string that identifies the particular instance of the computer hardware on which Common Lisp is running, or nil if no such string can be computed.

Examples:

(machine-instance)
→ "ACME.COM"
or
→ "S/N 123231"
or
→ "18.26.0.179"
or
→ "AA-00-04-00-A7-A4"

Affected By:
The machine instance, and the implementation.
See Also:
  machine-type, machine-version

machine-type  Function

Syntax:
  machine-type (no arguments) → description

Arguments and Values:
  description—a string or nil.

Description:
  Returns a string that identifies the generic name of the computer hardware on which Common Lisp is running.

Examples:
  (machine-type)
  → "DEC PDP-10"
  or
  → "Symbolics LM-2"

Affected By:
  The machine type. The implementation.

See Also:
  machine-version

machine-version  Function

Syntax:
  machine-version (no arguments) → description

Arguments and Values:
  description—a string or nil.

Description:
  Returns a string that identifies the version of the computer hardware on which Common Lisp is running, or nil if no such value can be computed.
Examples:

(machine-version) → "KL-10, microcode 9"

Affected By:
The machine version, and the implementation.

See Also:
machine-type, machine-instance

software-type, software-version  Function

Syntax:
software-type (no arguments) → description
software-version (no arguments) → description

Arguments and Values:
description—a string or nil.

Description:
software-type returns a string that identifies the generic name of any relevant supporting software, or nil if no appropriate or relevant result can be produced.

software-version returns a string that identifies the version of any relevant supporting software, or nil if no appropriate or relevant result can be produced.

Examples:

(software-type) → "Multics"
(software-version) → "1.3x"

Affected By:
Operating system environment.

Notes:
This information should be of use to maintainers of the implementation.
user-homedir-pathname

Syntax:

user-homedir-pathname &optional host → pathname

Arguments and Values:

host—a string, a list of strings, or :unspecific.

pathname—a pathname, or nil.

Description:

user-homedir-pathname determines the pathname that corresponds to the user’s home directory on host. If host is not supplied, its value is implementation-dependent. For a description of :unspecific, see Section 19.2.1 (Pathname Components).

The definition of home directory is implementation-dependent, but defined in Common Lisp to mean the directory where the user keeps personal files such as initialization files and mail.

user-homedir-pathname returns a pathname without any name, type, or version component (those components are all nil) for the user’s home directory on host.

If it is impossible to determine the user’s home directory on host, then nil is returned.

user-homedir-pathname never returns nil if host is not supplied.

Examples:

(pathnamep (user-homedir-pathname)) → true

Affected By:

The host computer’s file system, and the implementation.
Programming Language—Common Lisp

26. Glossary
26.1 Glossary

Each entry in this glossary has the following parts:

- the term being defined, set in boldface.

- optional pronunciation, enclosed in square brackets and set in boldface, as in the following example: [a, list]. The pronunciation key follows Webster’s Third New International Dictionary the English Language, Unabridged, except that “e” is used to notate the schwa (upside-down “e”) character.

- the part or parts of speech, set in italics. If a term can be used as several parts of speech, there is a separate definition for each part of speech.

- one or more definitions, organized as follows:
  - an optional number, present if there are several definitions. Lowercase letters might also be used in cases where subdefinitions of a numbered definition are necessary.
  - an optional part of speech, set in italics, present if the term is one of several parts of speech.
  - an optional discipline, set in italics, present if the term has a standard definition being repeated. For example, “Math.”
  - an optional context, present if this definition is meaningful only in that context. For example, “(of a symbol)”.
  - the definition.
  - an optional example sentence. For example, “This is an example of an example.”
  - optional cross references.

In addition, some terms have idiomatic usage in the Common Lisp community which is not shared by other communities, or which is not technically correct. Definitions labeled “Idiom.” represent such idiomatic usage; these definitions are sometimes followed by an explanatory note.

Words in this font are words with entries in the glossary. Words in example sentences do not follow this convention.

When an ambiguity arises, the longest matching substring has precedence. For example, “complex float” refers to a single glossary entry for “complex float” rather than the combined meaning of the glossary terms “complex” and “float.”
Subscript notation, as in “somethingₙ,” means that the nth definition of “something” is intended. This notation is used only in situations where the context might be insufficient to disambiguate.

The following are abbreviations used in the glossary:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>adj.</td>
<td>adjective</td>
</tr>
<tr>
<td>adv.</td>
<td>adverb</td>
</tr>
<tr>
<td>ANSI</td>
<td>compatible with one or more ANSI standards</td>
</tr>
<tr>
<td>Comp.</td>
<td>computers</td>
</tr>
<tr>
<td>Idiom.</td>
<td>idiomatic</td>
</tr>
<tr>
<td>IEEE</td>
<td>compatible with one or more IEEE standards</td>
</tr>
<tr>
<td>ISO</td>
<td>compatible with one or more ISO standards</td>
</tr>
<tr>
<td>Math.</td>
<td>mathematics</td>
</tr>
<tr>
<td>Trad.</td>
<td>traditional</td>
</tr>
<tr>
<td>n.</td>
<td>noun</td>
</tr>
<tr>
<td>v.</td>
<td>verb</td>
</tr>
<tr>
<td>v.t.</td>
<td>transitive verb</td>
</tr>
</tbody>
</table>

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 read₁ or write₁ the value of the place or an element of the array. 2. n. (of a place) an attempt to access₁ 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 [ ¯ a 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. 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. See Figure 2–8.

alphanumeric adj. (of a character) being either an alphabetic 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 operatorsetq.

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, 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).
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 transferred 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 “by order of argument” 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. adv. having a local binding which shadows another. “The variable *print-escape* is bound while in the prin 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 $0 \leq i_{\text{start}} \leq i_{\text{end}} \leq n$, and which therefore delimit a subrange of the sequence bounded by $i_{\text{start}}$ and $i_{\text{end}}$.

**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.

**Cadr** [ˈkaɪdər], *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. “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.)

cddr [ 'kùdr] or [ 'kçdüdr], n. (of an object) the cdr of the cdr of that object.

cdr [ 'kùdr], 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. “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.
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, 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 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*↓ 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-competition-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*. 

Glossary 26–11
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 adherence 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 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 syntax type. c. n. a constituent 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 $A_i$) A fresh list $B$ with elements $B_i$, each of which is nil if $A_i$ is nil, or else a copy of the cons $A_i$. 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) `correctable1` 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 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 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.

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.
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 environment1 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.

Glossary  26–21
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 readable are shown in Figure 26–1. 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.”

<table>
<thead>
<tr>
<th>Marker</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>double-float</td>
</tr>
<tr>
<td>d</td>
<td>double-float</td>
</tr>
<tr>
<td>E</td>
<td>float (see <em>read-default-float-format</em>)</td>
</tr>
<tr>
<td>e</td>
<td>float (see <em>read-default-float-format</em>)</td>
</tr>
<tr>
<td>F</td>
<td>single-float</td>
</tr>
<tr>
<td>f</td>
<td>single-float</td>
</tr>
<tr>
<td>L</td>
<td>long-float</td>
</tr>
<tr>
<td>l</td>
<td>long-float</td>
</tr>
<tr>
<td>S</td>
<td>short-float</td>
</tr>
<tr>
<td>s</td>
<td>short-float</td>
</tr>
</tbody>
</table>

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. \("\text{defmacro}\) 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 accommodate 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 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.

**false n.** the symbol nil, used to represent the failure of a predicate test.

**fbound [ˈefˈbaʊnd] 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 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. 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 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 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 [ˈefunbaʊnd] 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 \( \text{fbound symbol} \ S \)) 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 \( \text{fbound} \). But if \( S \) is such a name and is \( \text{fbound} \) 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.”

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 \textit{nil} represents \textit{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 Figure 26–2, or some other (implementation-defined) stream variable that is defined by the implementation to be an I/O customization variable.
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 21.1.1.3 (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 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 repetitions) 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. See Figure 2–8.

iteration form n. a compound form whose operator is named in Figure 26–3, or a compound form that has an implementation-defined operator and that is defined by the implementation to be an iteration form.
<table>
<thead>
<tr>
<th>do</th>
<th>do-external-symbols</th>
<th>dotimes</th>
</tr>
</thead>
<tbody>
<tr>
<td>do*</td>
<td>do-symbols</td>
<td>loop</td>
</tr>
<tr>
<td>do-all-symbols</td>
<td>dolist</td>
<td></td>
</tr>
</tbody>
</table>

**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. 2. a terminal node of a tree.

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 a form, evaluates it, and prints (i.e., *writes*) 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 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 of each cons in the list) are not themselves part of its list structure, even if they are conses, except in the (circular) 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. (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 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 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 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.”
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 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, 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 cdr of the function special form, which might be either an operator or a lambda expression. 4. (of a compound form) the car of the compound form, which might be either an operator 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 ‘\0 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 (eq1 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 (eq1 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`.

**plist** [ˈpleɪlist] 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 $O_1$ subject to constraints) an object $O_2$ that if the specified constraints are satisfied by $O_1$ without any modification might or might not be identical to $O_1$, or else that must be a fresh object that resembles a copy of $O_1$ 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.

**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.
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 \( a.d.v. \) (of a manner of printing an object \( O_1 \)) in such a way as to permit the Lisp Reader to later parse the printed output into an object \( O_2 \) that is similar to \( O_1 \).

reader \( n. \) 1. a function that reads 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 macro1; 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 macro2. 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. Figure 26–4 shows a list of the *standardized restart functions*.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>transfer control and values from the block</td>
</tr>
<tr>
<td>continue</td>
<td>cause the block to yield the values immediately</td>
</tr>
<tr>
<td>muffle-warning</td>
<td>yield the values</td>
</tr>
<tr>
<td>use-value</td>
<td>yield the values</td>
</tr>
<tr>
<td>store-value</td>
<td>yield the values</td>
</tr>
</tbody>
</table>

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*

**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*.

**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* 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, 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).


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 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 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 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 \( L_1 \) of length \( n \), whose last element is a list \( L_2 \) of length \( m \) (denoting a list \( L_3 \) of length \( m+n-1 \) whose elements are \( L_1_i \) for \( i < n-1 \) followed by \( L_2_j \) 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-allocate 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 readable n. A readable that is different from the initial readable, that implements the expression syntax defined in this specification, and that, unlike other readables, must never be modified by any program. (Although the definite reference “the standard readable” is generally used within this document, it is actually implementation-dependent whether a single object fills the role of the standard readable, or whether there might be multiple such objects, any one of which could be used on any given occasion where “the standard readable” 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 readable 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 receives 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 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.

26–58 Programming Language—Common Lisp
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) the set of conses that make up the tree. Note that while the car1b component of each such cons is part of the tree structure, the objects that are the cars2 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 t1.

truename n. 1. the canonical filename of a file in the file system. See Section 20.1.3 (Truenames). 2. a pathname representing a truename1.

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 $S_n$” naturally implies that “X is of type $S_m$” if $S_n$ is a subtype of $S_m$.) 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*.

**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.

**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 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) 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**

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**

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.

**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 syntax type. c. n. a whitespace 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 a variable or slot.
Y

**yield** *v.t.* *(values)* to produce the *values* as the result of *evaluation*. “The form

\((+ 2 3)\) yields 5.”
Programming Language—Common Lisp

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.
A–2 Programming Language—Common Lisp