Programming Language—Common Lisp

1. Introduction

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 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. Between 1984 and 1989, Common Lisp became a de facto standard.

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 1.8 (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

- *The Anatomy of Lisp*, John Allen, McGraw-Hill, Inc., 1978.

- *The Art of Computer Programming, Volume 3*, Donald E. Knuth, Addison-Wesley Company (Reading, MA), 1973.

- *The Art of the Metaobject Protocol*, Kiczales et al., MIT Press (Cambridge, MA), 1991.

- "Common Lisp Object System Specification," D. Bobrow, L. DiMichiel, R. Gabriel, S. Keene, G. Kiczales, D. Moon, *SIGPLAN Notices* V23, September, 1988.

- *Common Lisp: The Language*, Guy L. Steele, Jr., Digital Press (Burlington, MA), 1984.

- *Common Lisp: The Language, Second Edition*, Guy L. Steele, Jr., Digital Press (Bedford, MA), 1990.

- *Exceptional Situations in Lisp*, Kent M. Pitman, *Proceedings of the First European Conference on the Practical Application of LISP* (EUROPAL '90), Churchill College, Cambridge, England, March 27-29, 1990.

- *Flavors: A Non-Hierarchical Approach to Object-Oriented Programming*, Howard I. Cannon, 1982.

- *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985, Institute of Electrical and Electronics Engineers, Inc. (New York), 1985.

- *IEEE Standard for the Scheme Programming Language*, IEEE Std 1178-1990, Institute of Electrical and Electronic Engineers, Inc. (New York), 1991.

- *Interlisp Reference Manual*, Third Revision, Teitelman, Warren, et al, Xerox Palo Alto Research Center (Palo Alto, CA), 1978.

- ISO 6937/2, *Information processing—Coded character sets for text communication—Part 2: Latin alphabetic and non-alphabetic graphic characters*, ISO, 1983.

- *Lisp 1.5 Programmer's Manual*, John McCarthy, MIT Press (Cambridge, MA), August, 1962.

- *Lisp Machine Manual*, D.L. Weinreb and D.A. Moon, Artificial Intelligence Laboratory, MIT (Cambridge, MA), July, 1981.

- *Maclisp Reference Manual, Revision 0*, David A. Moon, Project MAC (Laboratory for Computer Science), MIT (Cambridge, MA), March, 1974.

- "NIL—A Perspective," JonL White, *Macsyma User's Conference*, 1979.

- *Performance and Evaluation of Lisp Programs*, Richard P. Gabriel, MIT Press (Cambridge, MA), 1985.

- "Principal Values and Branch Cuts in Complex APL," Paul Penfield Jr., *APL 81 Conference Proceedings*, ACM SIGAPL (San Francisco, September 1981), 248-256. Proceedings published as *APL Quote Quad 12*, 1 (September 1981).

- *The Revised Maclisp Manual*, Kent M. Pitman, Technical Report 295, Laboratory for Computer Science, MIT (Cambridge, MA), May 1983.

- "Revised^3 Report on the Algorithmic Language Scheme," Jonathan Rees and William Clinger (editors), *SIGPLAN Notices* V21, #12, December, 1986.

- "S-1 Common Lisp Implementation," R.A. Brooks, R.P. Gabriel, and G.L. Steele, *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, 108-113, 1982.

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

- "Standard LISP Report," J.B. Marti, A.C. Hearn, M.L. Griss, and C. Griss, *SIGPLAN Notices* V14, #10, October, 1979.

- *Webster's Third New International Dictionary the English Language, Unabridged*, Merriam Webster (Springfield, MA), 1986.

- *XP: A Common Lisp Pretty Printing System*, R.C. Waters, Memo 1102a, Artificial Intelligence Laboratory, MIT (Cambridge, MA), September 1989.

1.4 Definitions

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

1.4.1 Notational Conventions

The following notational conventions are used throughout this document.

1.4.1.1 Font Key

Fonts are used in this document to convey information.

name

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

Sometimes a glossary term appears subscripted, as in "*whitespace*$_2$." Such a notation selects one particular Glossary definition out of several, in this case the second. The subscript notation for Glossary terms is generally used where the context might be insufficient to disambiguate among the available definitions.

name

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

name

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

`name`

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

This font is also used for certain *standardized* names that are not names of *external symbols* of the `COMMON-LISP` *package*, such as *keywords*$_1$, *package names*, and *loop keywords*.

name

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

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

1.4.1.2 Modified BNF Syntax

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

1.4.1.2.1 Splicing in Modified BNF Syntax

The primary extension used is the following:

$$[\![O]\!]$$

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

$$O_1 \mid \ldots \mid O_l$$

where each O_i can be either of the form S or of the form S*. The expression $[\![O]\!]$ means that a list of the form

$$(O_{i_1} \ldots O_{i_j}) \quad 1 \le j$$

is spliced into the enclosing expression, such that if $n \ne m$ and $1 \le n, m \le j$, then either $O_{i_n} \ne O_{i_m}$ or $O_{i_n} = O_{i_m} = Q_k$, where for some $1 \le k \le n$, O_k is of the form Q_k*.

For example, the expression

```
(x ⟦A | B* | C⟧ y)
```

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

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

but not any of these:

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

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

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 | B* | C

1.4.1.2.3 Additional Uses for Indirect Definitions in Modified BNF Syntax

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

case *keyform* {↓*normal-clause*}* [↓*otherwise-clause*] → {*result*}*

ccase *keyplace* {↓*normal-clause*}* → {*result*}*

ecase *keyform* {↓*normal-clause*}* → {*result*}*

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

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

1.4.1.3 Special Symbols

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

→

This indicates evaluation. For example:

```
(+ 4 5) → 9
```

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

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

```
 (truncate 7 5)
→ 1 2
 (truncate 7 5)
→ 1
  2
 (truncate 7 5)
→ 1, 2
```

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

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

$\xrightarrow{or}$

The notation "$\xrightarrow{or}$" is used to denote one of several possible alternate results. The example

```
 (char-name #\a)
→ NIL
or→ "LOWERCASE-a"
or→ "Small-A"
or→ "LA01"
```

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

`(char-name #\a)` → *implementation-dependent*

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

$\xrightarrow{not}$

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

```
 (function-lambda-expression
```

```
 (funcall #'(lambda (x) #'(lambda () x)) nil))
```
$\rightarrow$ NIL, *true*, NIL
$\xrightarrow{or}$ (LAMBDA () X), *true*, NIL
$\xrightarrow{not}$ NIL, *false*, NIL
$\xrightarrow{not}$ (LAMBDA () X), *false*, NIL

$\equiv$

This indicates code equivalence. For example:

```
(gcd x (gcd y z)) ≡ (gcd (gcd x y) z)
```

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

▷

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

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

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

For example, the notation

```
(+ 1 (print (+ (sqrt (read)) (sqrt (read)))))
▷ 9 16 
▷ 7
→ 8
```

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

The use of the 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 false*, it is the *empty list*, and it is the *name* of the *empty type* (a *subtype* of all *types*).

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

For Evaluation?	Notation	Typically Implied Role
Yes	`nil`	use as a *boolean*.
Yes	`'nil`	use as a *symbol*.
Yes	`'()`	use as an *empty list*
No	`nil`	use as a *symbol* or *boolean*.
No	`()`	use as an *empty list*.

Figure 1–1. Notations for NIL

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

For example:

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

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

1.4.1.5 Designators

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

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

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

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

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

```
 (defun add-some (x)
   (defun add-some (x) (+ x 2))
   (+ x 1)) → ADD-SOME
 (mapcar 'add-some '(1 2 3 4))
→ (2 3 4 5)
or
→ (2 4 5 6)
```

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

1.4.1.6 Nonsense Words

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

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

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

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

1.4.2 Error Terminology

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

Safe code

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

Unsafe code

This is code processed with lower safety levels.

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

An error is signaled

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

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

An error should be signaled

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

Should be prepared to signal an error

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

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

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

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

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

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

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

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

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

The consequences are unspecified

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

The consequences are undefined

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

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

An error might be signaled

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

The return values are unspecified

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

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

Implementations may be extended to cover this situation

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

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

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

Implementations are free to extend the syntax

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

A warning might be issued

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

1.4.3 Sections Not Formally Part Of This Standard

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

Within the concept sections, subsections whose names begin with the words "Notes" 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*.

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 "Method Signature" Section of a Dictionary Entry

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

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

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

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

1.4.4.13 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*.

1.4.4.14 The "Notes" Section of a Dictionary Entry

Information not found elsewhere in this description which retains 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.

1.4.4.15 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*.

1.4.4.16 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.

1.4.4.17 The "Side Effects" Section of a Dictionary Entry

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

1.4.4.18 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.

1.4.4.19 The "Syntax" Section of a Dictionary Entry

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

For example, an *operator* description might say:

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

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

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

1.4.4.19.1 Special "Syntax" Notations for Overloaded Operators

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

file-position *stream* → *position*

file-position *stream position-spec* → *boolean*

is operationally equivalent to this line:

file-position *stream* `&optional` *position-spec* → *result*

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

1.4.4.19.2 Naming Conventions for Rest Parameters

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

For example, given a syntax description such as:

F `&rest` *arguments*

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

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

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

F `&rest` *arguments*$^{+}$

means the same as

F `&rest` *arguments*

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

1.4.4.19.4 Return values in the "Syntax" Section

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

F *a b c* $\rightarrow$ *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* $\rightarrow$ *x, y, z*

1.4.4.19.4.1 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 $\langle$*no arguments*$\rangle$ $\rightarrow$ $\langle$*no values*$\rangle$

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

1.4.4.19.4.2 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* $\rightarrow$|

1.4.4.20 The "Valid Context" Section of a Dictionary Entry

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

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

1.4.4.21 The "Value Type" Section of a Dictionary Entry

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

1.5 Conformance

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

1.5.1 Conforming Implementations

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

1.5.1.1 Required Language Features

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

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

1.5.1.2 Documentation of Implementation-Dependent Features

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

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

1.5.1.3 Documentation of Extensions

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

1.5.1.4 Treatment of Exceptional Situations

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

1.5.1.4.1 Resolution of Apparent Conflicts in Exceptional Situations

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

1.5.1.4.1.1 Examples of Resolution of Apparent Conflicts in Exceptional Situations

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

1.5.1.5 Conformance Statement

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

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

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

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

1.5.2 Conforming Programs

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

1. *Conforming code* shall use only those features of the language syntax and semantics that are either specified in this standard or defined using the extension mechanisms specified in the standard.

2. *Conforming code* shall not rely on any particular interpretation of *implementation-dependent* features.

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.

Portable code is written using only *standard characters*.

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

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

1.5.2.1.1 Use of Read-Time Conditionals

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

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

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

1.6 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*⟩⟩."

1.7.1 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.

1.7.2 Deprecated Functions

The *functions* in Figure 1–2 are deprecated.

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

Figure 1–2. Deprecated Functions

1.7.3 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.

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

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

1.7.4 Deprecated Variables

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

1.7.5 Deprecated Reader Syntax

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

1.8 Symbols in the COMMON-LISP Package

The figures on the next twelve pages contain a complete enumeration of the 973 *external symbols* in the `COMMON-LISP` *package*.

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

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

adjustable-array-p	**bignum**
allocate-instance	**bit**
alpha-char-p	**bit-and**
alphanumericp	**bit-andc1**
and	**bit-andc2**
append	**bit-eqv**
apply	**bit-ior**
apropos	**bit-nand**
apropos-list	**bit-nor**
aref	**bit-not**
arithmetic-error	**bit-orc1**
arithmetic-error-operands	**bit-orc2**
arithmetic-error-operation	**bit-vector**
array	**bit-vector-p**
array-dimension	**bit-xor**
array-dimension-limit	**block**
array-dimensions	**boole**
array-displacement	**boole-1**
array-element-type	**boole-2**
array-has-fill-pointer-p	**boole-and**
array-in-bounds-p	**boole-andc1**
array-rank	**boole-andc2**
array-rank-limit	**boole-c1**
array-row-major-index	**boole-c2**
array-total-size	**boole-clr**
array-total-size-limit	**boole-eqv**
arrayp	**boole-ior**
ash	**boole-nand**
asin	**boole-nor**
asinh	**boole-orc1**
assert	**boole-orc2**
assoc	**boole-set**
assoc-if	**boole-xor**
assoc-if-not	**both-case-p**
atan	**boundp**
atanh	**break**
atom	**broadcast-stream**
base-char	**broadcast-stream-streams**
base-string	**built-in-class**

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

butlast	**cdr**	**compile**
byte	**ceiling**	**compile-file**
byte-position	**cell-error**	**compile-file-pathname**
byte-size	**cell-error-name**	**compiled-function**
caaaar	**cerror**	**compiled-function-p**
caaadr	**change-class**	**compiler-macro**
caaar	**char**	**compiler-macro-function**
caadar	**char-code**	**complement**
caaddr	**char-code-limit**	**complex**
caadr	**char-downcase**	**complexp**
caar	**char-equal**	**compute-applicable-methods**
cadaar	**char-greaterp**	**compute-restarts**
cadadr	**char-int**	**concatenate**
cadar	**char-lessp**	**concatenated-stream**
caddar	**char-name**	**concatenated-stream-streams**
cadddr	**char-not-equal**	**cond**
caddr	**char-not-greaterp**	**condition**
cadr	**char-not-lessp**	**conjugate**
call-arguments-limit	**char-upcase**	**cons**
call-method	**char/=**	**consp**
call-next-method	**char<**	**constantly**
car	**char<=**	**constantp**
case	**char=**	**continue**
catch	**char>**	**control-error**
ccase	**char>=**	**copy-alist**
cdaaar	**character**	**copy-list**
cdaadr	**characterp**	**copy-pprint-dispatch**
cdaar	**check-type**	**copy-readtable**
cdadar	**cis**	**copy-seq**
cdaddr	**class**	**copy-structure**
cdadr	**class-name**	**copy-symbol**
cdar	**class-of**	**copy-tree**
cddaar	**clear-input**	**cos**
cddadr	**clear-output**	**cosh**
cddar	**close**	**count**
cdddar	**clrhash**	**count-if**
cddddr	**code-char**	**count-if-not**
cdddr	**coerce**	**ctypecase**
cddr	**compilation-speed**	**debug**

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

decf	**division-by-zero**
declaim	**do**
declaration	**do***
declare	**do-all-symbols**
decode-float	**do-external-symbols**
decode-universal-time	**do-symbols**
defclass	**documentation**
defconstant	**dolist**
defgeneric	**dotimes**
define-compiler-macro	**double-float**
define-condition	**double-float-epsilon**
define-method-combination	**double-float-negative-epsilon**
define-modify-macro	**dpb**
define-setf-expander	**dribble**
defmacro	**dynamic-extent**
defmethod	**ecase**
defpackage	**echo-stream**
defparameter	**echo-stream-input-stream**
defsetf	**echo-stream-output-stream**
defstruct	**ed**
deftype	**eighth**
defun	**elt**
defvar	**encode-universal-time**
delete	**end-of-file**
delete-duplicates	**endp**
delete-file	**enough-namestring**
delete-if	**ensure-generic-function**
delete-if-not	**eq**
delete-package	**eql**
denominator	**equal**
deposit-field	**equalp**
describe	**error**
describe-object	**etypecase**
destructuring-bind	**eval**
digit-char	**eval-when**
digit-char-p	**evenp**
directory	**every**
directory-namestring	**exp**
disassemble	**export**

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

expt	**floating-point-underflow**
extended-char	**floatp**
fboundp	**floor**
fceiling	**fmakunbound**
fdefinition	**force-output**
ffloor	**format**
fifth	**formatter**
file-author	**fourth**
file-error	**fresh-line**
file-error-pathname	**fround**
file-length	**ftruncate**
file-namestring	**ftype**
file-position	**funcall**
file-stream	**function**
file-string-length	**function-keywords**
file-write-date	**function-lambda-expression**
fill	**functionp**
fill-pointer	**gcd**
find	**generic-function**
find-all-symbols	**gensym**
find-class	**gentemp**
find-if	**get**
find-if-not	**get-decoded-time**
find-method	**get-dispatch-macro-character**
find-package	**get-internal-real-time**
find-restart	**get-internal-run-time**
find-symbol	**get-macro-character**
finish-output	**get-output-stream-string**
first	**get-properties**
fixnum	**get-setf-expansion**
flet	**get-universal-time**
float	**getf**
float-digits	**gethash**
float-precision	**go**
float-radix	**graphic-char-p**
float-sign	**handler-bind**
floating-point-inexact	**handler-case**
floating-point-invalid-operation	**hash-table**
floating-point-overflow	**hash-table-count**

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

hash-table-p	**lcm**
hash-table-rehash-size	**ldb**
hash-table-rehash-threshold	**ldb-test**
hash-table-size	**ldiff**
hash-table-test	**least-negative-double-float**
host-namestring	**least-negative-long-float**
identity	**least-negative-normalized-double-float**
if	**least-negative-normalized-long-float**
ignorable	**least-negative-normalized-short-float**
ignore	**least-negative-normalized-single-float**
ignore-errors	**least-negative-short-float**
imagpart	**least-negative-single-float**
import	**least-positive-double-float**
in-package	**least-positive-long-float**
incf	**least-positive-normalized-double-float**
initialize-instance	**least-positive-normalized-long-float**
inline	**least-positive-normalized-short-float**
input-stream-p	**least-positive-normalized-single-float**
inspect	**least-positive-short-float**
integer	**least-positive-single-float**
integer-decode-float	**length**
integer-length	**let**
integerp	**let***
interactive-stream-p	**lisp-implementation-type**
intern	**lisp-implementation-version**
internal-time-units-per-second	**list**
intersection	**list***
invalid-method-error	**list-all-packages**
invoke-debugger	**list-length**
invoke-restart	**listen**
invoke-restart-interactively	**listp**
isqrt	**load**
keyword	**load-logical-pathname-translations**
keywordp	**load-time-value**
labels	**locally**
lambda	**log**
lambda-list-keywords	**logand**
lambda-parameters-limit	**logandc1**
last	**logandc2**

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

logbitp	**make-method**
logcount	**make-package**
logeqv	**make-pathname**
logical-pathname	**make-random-state**
logical-pathname-translations	**make-sequence**
logior	**make-string**
lognand	**make-string-input-stream**
lognor	**make-string-output-stream**
lognot	**make-symbol**
logorc1	**make-synonym-stream**
logorc2	**make-two-way-stream**
logtest	**makunbound**
logxor	**map**
long-float	**map-into**
long-float-epsilon	**mapc**
long-float-negative-epsilon	**mapcan**
long-site-name	**mapcar**
loop	**mapcon**
loop-finish	**maphash**
lower-case-p	**mapl**
machine-instance	**maplist**
machine-type	**mask-field**
machine-version	**max**
macro-function	**member**
macroexpand	**member-if**
macroexpand-1	**member-if-not**
macrolet	**merge**
make-array	**merge-pathnames**
make-broadcast-stream	**method**
make-concatenated-stream	**method-combination**
make-condition	**method-combination-error**
make-dispatch-macro-character	**method-qualifiers**
make-echo-stream	**min**
make-hash-table	**minusp**
make-instance	**mismatch**
make-instances-obsolete	**mod**
make-list	**most-negative-double-float**
make-load-form	**most-negative-fixnum**
make-load-form-saving-slots	**most-negative-long-float**

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

most-negative-short-float	**nsubstitute**
most-negative-single-float	**nsubstitute-if**
most-positive-double-float	**nsubstitute-if-not**
most-positive-fixnum	**nth**
most-positive-long-float	**nth-value**
most-positive-short-float	**nthcdr**
most-positive-single-float	**null**
muffle-warning	**number**
multiple-value-bind	**numberp**
multiple-value-call	**numerator**
multiple-value-list	**nunion**
multiple-value-prog1	**oddp**
multiple-value-setq	**open**
multiple-values-limit	**open-stream-p**
name-char	**optimize**
namestring	**or**
nbutlast	**otherwise**
nconc	**output-stream-p**
next-method-p	**package**
nil	**package-error**
nintersection	**package-error-package**
ninth	**package-name**
no-applicable-method	**package-nicknames**
no-next-method	**package-shadowing-symbols**
not	**package-use-list**
notany	**package-used-by-list**
notevery	**packagep**
notinline	**pairlis**
nreconc	**parse-error**
nreverse	**parse-integer**
nset-difference	**parse-namestring**
nset-exclusive-or	**pathname**
nstring-capitalize	**pathname-device**
nstring-downcase	**pathname-directory**
nstring-upcase	**pathname-host**
nsublis	**pathname-match-p**
nsubst	**pathname-name**
nsubst-if	**pathname-type**
nsubst-if-not	**pathname-version**

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

pathnamep	**psetf**
peek-char	**psetq**
phase	**push**
pi	**pushnew**
plusp	**quote**
pop	**random**
position	**random-state**
position-if	**random-state-p**
position-if-not	**rassoc**
pprint	**rassoc-if**
pprint-dispatch	**rassoc-if-not**
pprint-exit-if-list-exhausted	**ratio**
pprint-fill	**rational**
pprint-indent	**rationalize**
pprint-linear	**rationalp**
pprint-logical-block	**read**
pprint-newline	**read-byte**
pprint-pop	**read-char**
pprint-tab	**read-char-no-hang**
pprint-tabular	**read-delimited-list**
prin1	**read-from-string**
prin1-to-string	**read-line**
princ	**read-preserving-whitespace**
princ-to-string	**reader-error**
print	**readtable**
print-not-readable	**readtable-case**
print-not-readable-object	**readtablep**
print-object	**real**
print-unreadable-object	**realp**
probe-file	**realpart**
proclaim	**reduce**
prog	**reinitialize-instance**
prog*	**rem**
prog1	**remf**
prog2	**remhash**
progn	**remove**
program-error	**remove-duplicates**
progv	**remove-if**
provide	**remove-if-not**

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

remove-method	**seventh**
remprop	**shadow**
rename-file	**shadowing-import**
rename-package	**shared-initialize**
replace	**shiftf**
require	**short-float**
rest	**short-float-epsilon**
restart	**short-float-negative-epsilon**
restart-bind	**short-site-name**
restart-case	**signal**
restart-name	**signed-byte**
return	**signum**
return-from	**simple-array**
revappend	**simple-base-string**
reverse	**simple-bit-vector**
room	**simple-bit-vector-p**
rotatef	**simple-condition**
round	**simple-condition-format-arguments**
row-major-aref	**simple-condition-format-control**
rplaca	**simple-error**
rplacd	**simple-string**
safety	**simple-string-p**
satisfies	**simple-type-error**
sbit	**simple-vector**
scale-float	**simple-vector-p**
schar	**simple-warning**
search	**sin**
second	**single-float**
sequence	**single-float-epsilon**
serious-condition	**single-float-negative-epsilon**
set	**sinh**
set-difference	**sixth**
set-dispatch-macro-character	**sleep**
set-exclusive-or	**slot-boundp**
set-macro-character	**slot-exists-p**
set-pprint-dispatch	**slot-makunbound**
set-syntax-from-char	**slot-missing**
setf	**slot-unbound**
setq	**slot-value**

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

software-type	**string-upcase**
software-version	**string/=**
some	**string<**
sort	**string<=**
space	**string=**
special	**string>**
special-operator-p	**string>=**
speed	**stringp**
sqrt	**structure**
stable-sort	**structure-class**
standard	**structure-object**
standard-char	**style-warning**
standard-char-p	**sublis**
standard-class	**subseq**
standard-generic-function	**subsetp**
standard-method	**subst**
standard-object	**subst-if**
step	**subst-if-not**
storage-condition	**substitute**
store-value	**substitute-if**
stream	**substitute-if-not**
stream-element-type	**subtypep**
stream-error	**svref**
stream-error-stream	**sxhash**
stream-external-format	**symbol**
streamp	**symbol-function**
string	**symbol-macrolet**
string-capitalize	**symbol-name**
string-downcase	**symbol-package**
string-equal	**symbol-plist**
string-greaterp	**symbol-value**
string-left-trim	**symbolp**
string-lessp	**synonym-stream**
string-not-equal	**synonym-stream-symbol**
string-not-greaterp	**t**
string-not-lessp	**tagbody**
string-right-trim	**tailp**
string-stream	**tan**
string-trim	**tanh**

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

tenth	**upgraded-complex-part-type**
terpri	**upper-case-p**
the	**use-package**
third	**use-value**
throw	**user-homedir-pathname**
time	**values**
trace	**values-list**
translate-logical-pathname	**variable**
translate-pathname	**vector**
tree-equal	**vector-pop**
truename	**vector-push**
truncate	**vector-push-extend**
two-way-stream	**vectorp**
two-way-stream-input-stream	**warn**
two-way-stream-output-stream	**warning**
type	**when**
type-error	**wild-pathname-p**
type-error-datum	**with-accessors**
type-error-expected-type	**with-compilation-unit**
type-of	**with-condition-restarts**
typecase	**with-hash-table-iterator**
typep	**with-input-from-string**
unbound-slot	**with-open-file**
unbound-slot-instance	**with-open-stream**
unbound-variable	**with-output-to-string**
undefined-function	**with-package-iterator**
unexport	**with-simple-restart**
unintern	**with-slots**
union	**with-standard-io-syntax**
unless	**write**
unread-char	**write-byte**
unsigned-byte	**write-char**
untrace	**write-line**
unuse-package	**write-string**
unwind-protect	**write-to-string**
update-instance-for-different-class	**y-or-n-p**
update-instance-for-redefined-class	**yes-or-no-p**
upgraded-array-element-type	**zerop**

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

Table of Contents