Programming Language—Common Lisp

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*.

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

Figure 2–1. Readtable defined names

2.1.1.1 The Current Readtable

Several *readtables* describing different syntaxes can exist, but at any given time only one, called the ***current readtable***, affects the way in which *expressions*$_2$ are parsed into *objects* by the *Lisp reader*. The *current readtable* in a given *dynamic environment* is the *value* of ***readtable*** in that *environment*. To make a different *readtable* become the *current readtable*, ***readtable*** can be *assigned* or *bound*.

2.1.1.2 The Standard Readtable

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

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

2.1.1.3 The Initial Readtable

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

2.1.2 Variables that affect the Lisp Reader

The *Lisp reader* is influenced not only by the *current readtable*, but also by various *dynamic variables*. Figure 2–2 lists the *variables* that influence the behavior of the *Lisp reader*.

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

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

2.1.3 Standard Characters

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

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

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

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

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

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

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

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

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

2.1.4 Character Syntax Types

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

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

constituent	*macro character*	*single escape*
invalid	*multiple escape*	*whitespace*$_2$

Figure 2–6. Possible Character Syntax Types

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

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

<table>
<tr><th>character</th><th>syntax type</th><th>character</th><th>syntax type</th></tr>
<tr><td>Backspace</td><td>constituent</td><td>0–9</td><td>constituent</td></tr>
<tr><td>Tab</td><td>whitespace$_2$</td><td>:</td><td>constituent</td></tr>
<tr><td>Newline</td><td>whitespace$_2$</td><td>;</td><td>terminating macro char</td></tr>
<tr><td>Linefeed</td><td>whitespace$_2$</td><td><</td><td>constituent</td></tr>
<tr><td>Page</td><td>whitespace$_2$</td><td>=</td><td>constituent</td></tr>
<tr><td>Return</td><td>whitespace$_2$</td><td>></td><td>constituent</td></tr>
<tr><td>Space</td><td>whitespace$_2$</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>A–Z</td><td>constituent</td></tr>
<tr><td>#</td><td>non-terminating macro char</td><td>[</td><td>constituent*</td></tr>
<tr><td>$</td><td>constituent</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>'</td><td>terminating macro char</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>a–z</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>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>Rubout</td><td>constituent</td></tr>
<tr><td>/</td><td>constituent</td><td></td><td></td></tr>
</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 *alphabetic*$_2$ *characters*, but are not used in the names of any standard Common Lisp *defined names*.

Whitespace$_2$ characters serve as separators but are otherwise ignored. *Constituent* and *escape characters* are accumulated to make a *token*, which is then interpreted as a *number* or *symbol*. *Macro characters* trigger the invocation of *functions* (possibly user-supplied) that can perform arbitrary parsing actions. *Macro characters* are divided into two kinds, *terminating* and *non-terminating*, depending on whether or not they terminate a *token*. The following are descriptions of each kind of *syntax type*.

2.1.4.1 Constituent Characters

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

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

2.1.4.2 Constituent Traits

Every *character* has one or more *constituent traits* that define how the *character* is to be interpreted by the *Lisp reader* when the *character* is a *constituent character*. These *constituent traits* are *alphabetic*$_2$, digit, *package marker*, plus sign, minus sign, dot, decimal point, *ratio marker*, *exponent marker*, and *invalid*. Figure 2–8 shows the *constituent traits* of the *standard characters* and of certain *semi-standard characters*; no mechanism is provided for changing the *constituent trait* of a *character*. Any *character* with the alphadigit *constituent trait* in that figure is a digit if the *current input base* is greater than that character's digit value, otherwise the *character* is *alphabetic*$_2$. Any *character* quoted by a *single escape* is treated as an *alphabetic*$_2$ constituent, regardless of its normal syntax.

constituent character	traits	constituent character	traits
Backspace	*invalid*	{	*alphabetic*$_2$
Tab	*invalid**	}	*alphabetic*$_2$
Newline	*invalid**	+	*alphabetic*$_2$, plus sign
Linefeed	*invalid**	-	*alphabetic*$_2$, minus sign
Page	*invalid**	.	*alphabetic*$_2$, dot, decimal point
Return	*invalid**	/	*alphabetic*$_2$, *ratio marker*
Space	*invalid**	A, a	alphadigit
!	*alphabetic*$_2$	B, b	alphadigit
"	*alphabetic*$_2$*	C, c	alphadigit
#	*alphabetic*$_2$*	D, d	alphadigit, double-float *exponent marker*
$	*alphabetic*$_2$	E, e	alphadigit, float *exponent marker*
%	*alphabetic*$_2$	F, f	alphadigit, single-float *exponent marker*
&	*alphabetic*$_2$	G, g	alphadigit
'	*alphabetic*$_2$*	H, h	alphadigit
(	*alphabetic*$_2$*	I, i	alphadigit
)	*alphabetic*$_2$*	J, j	alphadigit
*	*alphabetic*$_2$	K, k	alphadigit
,	*alphabetic*$_2$*	L, l	alphadigit, long-float *exponent marker*
0-9	alphadigit	M, m	alphadigit
:	*package marker*	N, n	alphadigit
;	*alphabetic*$_2$*	O, o	alphadigit
<	*alphabetic*$_2$	P, p	alphadigit
=	*alphabetic*$_2$	Q, q	alphadigit
>	*alphabetic*$_2$	R, r	alphadigit
?	*alphabetic*$_2$	S, s	alphadigit, short-float *exponent marker*
@	*alphabetic*$_2$	T, t	alphadigit
[	*alphabetic*$_2$	U, u	alphadigit
\	*alphabetic*$_2$*	V, v	alphadigit
]	*alphabetic*$_2$	W, w	alphadigit
^	*alphabetic*$_2$	X, x	alphadigit
_	*alphabetic*$_2$	Y, y	alphadigit
`	*alphabetic*$_2$*	Z, z	alphadigit
|	*alphabetic*$_2$*	Rubout	*invalid*
~	*alphabetic*$_2$		

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

The interpretations in this table apply only to *characters* whose *syntax type* is *constituent*. Entries marked with an asterisk (*) are normally *shadowed*$_2$ because the indicated *characters* are of *syntax type whitespace*$_2$, *macro character*, *single escape*, or *multiple escape*; these *constituent traits* apply to them only if their *syntax types* are changed to *constituent*.

2.1.4.3 Invalid Characters

Characters with the *constituent trait invalid* cannot ever appear in a *token* except under the control of a *single escape character*. If an *invalid character* is encountered while an *object* is being read, an error of *type* **reader-error** is signaled. If an *invalid character* is preceded by a *single escape character*, it is treated as an *alphabetic*$_2$ *constituent* instead.

2.1.4.4 Macro Characters

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

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

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

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

If a *character* is a *dispatching macro character* C_1, its *reader macro function* is a *function* supplied by the *implementation*. This *function* reads decimal *digit characters* until a non-*digit* C_2 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* C_2 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* C_1. The *reader macro function* associated with the sub-character C_2 is invoked with three arguments: the *stream*, the sub-character C_2, and the infix parameter P. For more information about dispatch characters, see the *function* **set-dispatch-macro-character**.

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

2.1.4.5 Multiple Escape Characters

A pair of ***multiple escape*** *characters* is used to indicate that an enclosed sequence of characters, including possible *macro characters* and *whitespace*$_2$ *characters*, are to be treated as *alphabetic*$_2$ *characters* with *case* preserved. Any *single escape* and *multiple escape characters* that are to appear in the sequence must be preceded by a *single escape character*.

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

2.1.4.5.1 Examples of Multiple Escape Characters

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

2.1.4.6 Single Escape Character

A ***single escape*** is used to indicate that the next *character* is to be treated as an *alphabetic*$_2$ *character* with its *case* preserved, no matter what the *character* is or which *constituent traits* it has.

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

2.1.4.6.1 Examples of Single Escape Characters

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

2.1.4.7 Whitespace Characters

Whitespace$_2$ *characters* are used to separate *tokens*.

Space and *newline* are *whitespace*$_2$ *characters* in *standard syntax*.

2.1.4.7.1 Examples of Whitespace Characters

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

2.2 Reader Algorithm

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

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

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

1. If at end of file, end-of-file processing is performed as specified in **read**. Otherwise, one *character*, x, is read from the *input stream*, and dispatched according to the *syntax type* of x to one of steps 2 to 7.

2. If x is an *invalid character*, an error of *type* **reader-error** is signaled.

3. If x is a *whitespace*$_2$ *character*, then it is discarded and step 1 is re-entered.

4. If x is a *terminating* or *non-terminating macro character* then its associated *reader macro function* is called with two *arguments*, the *input stream* and x.

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

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

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

5. If x is a *single escape character* then the next *character*, y, is read, or an error of *type* **end-of-file** is signaled if at the end of file. y is treated as if it is a *constituent* whose only *constituent trait* is *alphabetic*$_2$. y is used to begin a *token*, and step 8 is entered.

6. If x is a *multiple escape character* then a *token* (initially containing no *characters*) is begun and step 9 is entered.

7. If x is a *constituent character*, then it begins a *token*. After the *token* is read in, it will be interpreted either as a Lisp *object* or as being of invalid syntax. If the *token* represents an

object, that *object* is returned as the result of the read operation. If the *token* is of invalid syntax, an error is signaled. If x is a *character* with *case*, it might be replaced with the corresponding *character* of the opposite *case*, depending on the *readtable case* of the *current readtable*, as outlined in Section 23.1.2 (Effect of Readtable Case on the Lisp Reader). X is used to begin a *token*, and step 8 is entered.

8. At this point a *token* is being accumulated, and an even number of *multiple escape characters* have been encountered. If at end of file, step 10 is entered. Otherwise, a *character*, y, is read, and one of the following actions is performed according to its *syntax type*:

   - If y is a *constituent* or *non-terminating macro character*:

     – If y is a *character* with *case*, it might be replaced with the corresponding *character* of the opposite *case*, depending on the *readtable case* of the *current readtable*, as outlined in Section 23.1.2 (Effect of Readtable Case on the Lisp Reader).

     – Y is appended to the *token* being built.

     – Step 8 is repeated.

   - If y is a *single escape character*, then the next *character*, z, is read, or an error of *type* **end-of-file** is signaled if at end of file. Z is treated as if it is a *constituent* whose only *constituent trait* is *alphabetic*$_2$. Z is appended to the *token* being built, and step 8 is repeated.

   - If y is a *multiple escape character*, then step 9 is entered.

   - If y is an *invalid character*, an error of *type* **reader-error** is signaled.

   - If y is a *terminating macro character*, then it terminates the *token*. First the *character* y is unread (see **unread-char**), and then step 10 is entered.

   - If y is a *whitespace*$_2$ *character*, then it terminates the *token*. First the *character* y is unread if appropriate (see **read-preserving-whitespace**), and then step 10 is entered.

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

   - If y is a *constituent*, macro, or *whitespace*$_2$ *character*, y is treated as a *constituent* whose only *constituent trait* is *alphabetic*$_2$. Y is appended to the *token* being built, and step 9 is repeated.

   - If y is a *single escape character*, then the next *character*, z, is read, or an error of *type* **end-of-file** is signaled if at end of file. Z is treated as a *constituent* whose only *constituent trait* is *alphabetic*$_2$. Z is appended to the *token* being built, and step 9 is repeated.

- If y is a *multiple escape character*, then step 8 is entered.

- If y is an *invalid character*, an error of *type* **reader-error** is signaled.

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

2.3 Interpretation of Tokens

2.3.1 Numbers as Tokens

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

2.3.1.1.1 Escape Characters and Potential Numbers

A *potential number* cannot contain any *escape characters*. An *escape character* robs the following *character* of all syntactic qualities, forcing it to be strictly *alphabetic*$_2$ and therefore unsuitable for use in a *potential number*. For example, all of the following representations are interpreted as *symbols*, not *numbers*:

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

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

2.3.1.1.2 Examples of Potential Numbers

As examples, the *tokens* in 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`	`777777q`	`1.7J`	`-3/4+6.7J`	`12/25/83`
`27^19`	`3^4/5`	`6//7`	`3.1.2.6`	`^-43^`
`3.141_592_653_589_793_238_4`	`-3.7+2.6i-6.17j+19.6k`			

Figure 2–10. Examples of reserved tokens

The *tokens* in 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^`

Figure 2–12. Examples of symbols or potential numbers

2.3.2 Constructing Numbers from Tokens

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

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

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

This section summarizes the full syntax for *numbers*.

2.3.2.1 Syntax of a Rational

2.3.2.1.1 Syntax of an Integer

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

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

2.3.2.1.2 Syntax of a Ratio

Ratios can be written as an optional *sign* followed by two non-empty sequences of *digits* separated by a *slash*; see Figure 2–9. The second sequence may not consist entirely of zeros. Examples of *ratios* are in Figure 2–13.

`2/3`	;This is in canonical form
`4/6`	;A non-canonical form for 2/3
`-17/23`	;A ratio preceded by a sign
`-30517578125/32768`	;This is $(-5/2)^{15}$
`10/5`	;The canonical form for this is `2`
`#o-101/75`	;Octal notation for $-65/61$
`#3r120/21`	;Ternary notation for 15/7
`#Xbc/ad`	;Hexadecimal notation for 188/173
`#xFADED/FACADE`	;Hexadecimal notation for 1027565/16435934

Figure 2–13. Examples of Ratios

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

2.3.2.2 Syntax of a Float

Floats can be written in either decimal fraction or computerized scientific notation: an optional sign, then a non-empty sequence of digits with an embedded decimal point, then an optional decimal exponent specification. If there is no exponent specifier, then the decimal point is required, and there must be digits after it. The exponent specifier consists of an *exponent marker*, an optional sign, and a non-empty sequence of digits. If no exponent specifier is present, or if the *exponent marker* `e` (or `E`) is used, then the format specified by ***read-default-float-format*** is used. See Figure 2–9.

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

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

Figure 2–14 contains examples of notations for *floats*:

`0.0`	;Floating-point zero in default format
`0E0`	;As input, this is also floating-point zero in default format. ;As output, this would appear as `0.0`.
`0e0`	;As input, this is also floating-point zero in default format. ;As output, this would appear as `0.0`.
`-.0`	;As input, this might be a zero or a minus zero, ; depending on whether the implementation supports ; a distinct minus zero. ;As output, `0.0` is zero and `-0.0` is minus zero.
`0.`	;On input, the integer zero—*not* a floating-point number! ;Whether this appears as `0` or `0.` on output depends ;on the *value* of ***print-radix***.
`0.0s0`	;A floating-point zero in short format
`0s0`	;As input, this is a floating-point zero in short format. ;As output, such a zero would appear as `0.0s0` ; (or as `0.0` if **short-float** was the default format).
`6.02E+23`	;Avogadro's number, in default format
`602E+21`	;Also Avogadro's number, in default format

Figure 2–14. Examples of Floating-point numbers

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

2.3.2.3 Syntax of a Complex

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

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

2.3.3 The Consing Dot

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

2.3.4 Symbols as Tokens

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

Examples of the printed representation of *symbols* are in Figure 2–15. For presentational simplicity, these examples assume that the *readtable case* of the *current readtable* is `:upcase`.

`FROBBOZ`	The *symbol* whose *name* is `FROBBOZ`.
`frobboz`	Another way to notate the same *symbol*.
`fRObBoz`	Yet another way to notate it.
`unwind-protect`	A *symbol* with a hyphen in its *name*.
`+$`	The *symbol* named `+$`.
`1+`	The *symbol* named `1+`.
`+1`	This is the *integer* `1`, not a *symbol*.
`pascal_style`	This *symbol* has an underscore in its *name*.
`file.rel.43`	This *symbol* has periods in its *name*.
`\(`	The *symbol* whose *name* is `(`.
`\+1`	The *symbol* whose *name* is `+1`.
`+\1`	Also the *symbol* whose *name* is `+1`.
`\frobboz`	The *symbol* whose *name* is `fROBBOZ`.
`3.14159265\s0`	The *symbol* whose *name* is `3.14159265s0`.
`3.14159265\S0`	A different *symbol*, whose *name* is `3.14159265S0`.
`3.14159265s0`	A possible *short float* approximation to π.

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

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

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

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

When parsing the syntax for a *symbol*, the *Lisp reader* looks up the *name* of that *symbol* in the *current package*. This lookup may involve looking in other *packages* whose *external symbols* are inherited by the *current package*. If the name is found, the corresponding *symbol* is returned. If the name is not found (that is, there is no *symbol* of that name *accessible* in the *current package*), a new *symbol* is created and is placed in the *current package* as an *internal symbol*. The *current package* becomes the owner (*home package*) of the *symbol*, and the *symbol* becomes interned in the *current package*. If the name is later read again while this same *package* is current, the same *symbol* will be found and returned.

2.3.5 Valid Patterns for Tokens

The valid patterns for *tokens* are summarized in Figure 2–17.

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

Figure 2–17. Valid patterns for tokens

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

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

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

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

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

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

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

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

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

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

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

For example, assuming the *readtable case* of the *current readtable* is `:upcase`, `editor:buffer` refers to the *external symbol* named `BUFFER` present in the *package* named `editor`, regardless of whether there is a *symbol* named `BUFFER` in the *current package*. If there is no *package* named `editor`, or if no *symbol* named `BUFFER` is present in `editor`, or if `BUFFER` is not exported by `editor`, the reader signals a correctable error. If `editor::buffer` is seen, the effect is exactly the same as reading `buffer` with the `EDITOR` *package* being the *current package*.

2.3.6 Package System Consistency Rules

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

Read-read consistency

Reading the same *symbol name* always results in the *same symbol*.

Print-read consistency

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

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

Print-print consistency

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

These rules are true regardless of any implicit interning. As long as the *current package* is not changed, results are reproducible regardless of the order of *loading* files or the exact history of what *symbols* were typed in when. If the *value* of ***package*** is changed and then changed back to the previous value, consistency is maintained. The rules can be violated by changing the *value* of ***package***, forcing a change to *symbols* or to *packages* or to both by continuing from an error, or calling one of the following *functions*: **unintern**, **unexport**, **shadow**, **shadowing-import**, or **unuse-package**.

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

2.4 Standard Macro Characters

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

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

2.4.1 Left-Parenthesis

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

```
(a b c)
```

is read as a *list* of three *objects* (the *symbols* `a`, `b`, and `c`). The right parenthesis need not immediately follow the printed representation of the last *object*; *whitespace*$_2$ characters and comments may precede it.

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

If a *token* that is just a dot not immediately preceded by an escape character is read after some *object* then exactly one more *object* must follow the dot, possibly preceded or followed by *whitespace*$_2$ or a comment, followed by the right parenthesis:

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

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

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

Similarly,

```
(cons 'this-one 'that-one) → (this-one . that-one)
```

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

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

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

2.4.2 Right-Parenthesis

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

2.4.3 Single-Quote

Syntax: `'`⟪*exp*⟫

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

2.4.3.1 Examples of Single-Quote

```
'foo → FOO
''foo → (QUOTE FOO)
(car ''foo) → QUOTE
```

2.4.4 Semicolon

Syntax: `;`⟪*text*⟫

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

2.4.4.1 Examples of Semicolon

```
 (+ 3 ; three
    4)
→ 7
```

2.4.4.2 Notes about Style for Semicolon

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

2.4.4.2.1 Use of Single Semicolon

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

2.4.4.2.2 Use of Double Semicolon

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

2.4.4.2.3 Use of Triple Semicolon

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

2.4.4.2.4 Use of Quadruple Semicolon

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

2.4.4.2.5 Examples of Style for Semicolon

```
;;;; Math Utilities

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

(defun fib (n)
  (check-type n integer)
  ;; At this point we're sure we have an integer argument.
  ;; Now we can get down to some serious computation.
  (cond ((< n 0)
         ;; Hey, this is just supposed to be a simple example.
         ;; Did you really expect me to handle the general case?
         (error "FIB got ~D as an argument." n))
        ((< n 2) n)             ;fib[0]=0 and fib[1]=1
        ;; The cheap cases didn't work.
        ;; Nothing more to do but recurse.
        (t (+ (fib (- n 1))     ;The traditional formula
              (fib (- n 2)))))) ; is fib[n-1]+fib[n-2].
```

2.4.5 Double-Quote

Syntax: `"`⟪*text*⟫`"`

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

Examples of the use of the *double-quote* character are in Figure 2–18.

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

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

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

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

2.4.6 Backquote

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

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

is roughly equivalent to writing

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

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

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

```
`(x ,x ,@x foo ,(cadr x) bar ,(cdr x) baz ,@(cdr x))
→ (x (a b c) a b c foo b bar (b c) baz b c)
```

The backquote syntax can be summarized formally as follows.

- `` `basic `` is the same as `'basic`, that is, `(quote basic)`, for any *expression* *basic* that is not a *list* or a general *vector*.

- `` `,form `` is the same as *form*, for any *form*, provided that the representation of *form* does not begin with *at-sign* or *dot*. (A similar caveat holds for all occurrences of a form after a *comma*.)

- `` `,@form `` has undefined consequences.

- `` `(x1 x2 x3 ... xn . atom) `` may be interpreted to mean

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

  where the brackets are used to indicate a transformation of an *xj* as follows:

  - `[form]` is interpreted as ``(list `form)``, which contains a backquoted form that must then be further interpreted.

  - `[,form]` is interpreted as `(list form)`.

  - `[,@form]` is interpreted as *form*.

- `` `(x1 x2 x3 ... xn) `` may be interpreted to mean the same as the backquoted form `` `(x1 x2 x3 ... xn . `` **nil**`)`, thereby reducing it to the previous case.

- `` `(x1 x2 x3 ... xn . ,form) `` may be interpreted to mean

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

  where the brackets indicate a transformation of an `xj` as described above.

- `` `(x1 x2 x3 ... xn . ,@form) `` has undefined consequences.

- `` `#(x1 x2 x3 ... xn) `` may be interpreted to mean ``(apply #'vector `(x1 x2 x3 ... xn))``.

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

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

An *implementation* is free to interpret a backquoted *form* F_1 as any *form* F_2 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

```
`((,a b) ,c ,@d)
```

will be interpreted as if it were

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

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

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

2.4.6.1 Notes about Backquote

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

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

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

2.4.7 Comma

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

2.4.8 Sharpsign

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

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

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

The *reader macros* associated with the *dispatching macro character* `#` are described later in this section and summarized in Figure 2–19.

<table>
<tr><th>dispatch char</th><th>purpose</th><th>dispatch char</th><th>purpose</th></tr>
<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>undefined</td><td>C, c</td><td>complex number</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>&</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><</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>></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>undefined</td><td>Y, y</td><td>undefined</td></tr>
<tr><td>|</td><td>balanced comment</td><td>Z, z</td><td>undefined</td></tr>
<tr><td>~</td><td>undefined</td><td>Rubout</td><td>undefined</td></tr>
</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: `#\`⟪*x*⟫

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

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

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

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

2.4.8.2 Sharpsign Single-Quote

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

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

2.4.8.3 Sharpsign Left-Parenthesis

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

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

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

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

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

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

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

2.4.8.4 Sharpsign Asterisk

Syntax: `#*`⟨⟨*bits*⟩⟩

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

Syntax: `#`⟨⟨*n*⟩⟩`*`⟨⟨*bits*⟩⟩

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

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

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

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

2.4.8.4.1 Examples of Sharpsign Asterisk

For example, `#*101111`

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

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

For example:

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

2.4.8.5 Sharpsign Colon

Syntax: `#:`⟨⟨*symbol-name*⟩⟩

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

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

2.4.8.6 Sharpsign Dot

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

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

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

2.4.8.7 Sharpsign B

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

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

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

2.4.8.8 Sharpsign O

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

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

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

2.4.8.9 Sharpsign X

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

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

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

2.4.8.10 Sharpsign R

`#`*n*`R`

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

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

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
`#xD5`	;Ditto, in hexadecimal radix
`#16r+D5`	;Ditto
`#o-300`	;Decimal `-192`, written in base 8
`#3r-21010`	;Same thing in base 3
`#25R-7H`	;Same thing in base 25
`#xACCEDED`	;`181202413`, in hexadecimal radix

Figure 2–20. Radix Indicator Example

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

2.4.8.11 Sharpsign C

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

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

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

`#`*n*`A`

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

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

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

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

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

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

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

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

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

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

2.4.8.13 Sharpsign S

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

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

where each *keywordj* is the result of computing

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

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

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

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

2.4.8.14 Sharpsign P

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

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

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

2.4.8.15 Sharpsign Equal-Sign

`#`*n*`=`

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

2.4.8.16 Sharpsign Sharpsign

#*n*#

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

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

could be represented in this way:

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

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

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

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

2.4.8.17 Sharpsign Plus

#+ provides a read-time conditionalization facility; the syntax is #+*test expression*. If the *feature expression test* succeeds, then this textual notation represents an *object* whose printed representation is *expression*. If the *feature expression test* fails, then this textual notation is treated as *whitespace*$_2$; that is, it is as if the "#+ *test expression*" did not appear and only a *space* appeared in its place.

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

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

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

2.4.8.18 Sharpsign Minus

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

#-*test expression* ≡ #+(not *test*) *expression*

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

2.4.8.19 Sharpsign Vertical-Bar

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

2.4.8.19.1 Examples of Sharpsign Vertical-Bar

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

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


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

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


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

```
     (format t "Don't use |\# unmatched or you'll get in trouble!") |#
 (fboundp 'mention-fun-fact-2b) → NIL


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

2.4.8.19.2 Notes about Style for Sharpsign Vertical-Bar

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

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

Such code is equivalent to:

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

2.4.8.20 Sharpsign Less-Than-Sign

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

2.4.8.21 Sharpsign Whitespace

`#` followed immediately by *whitespace*$_1$ is not valid reader syntax. The *Lisp reader* will signal an error of *type* **reader-error** if it encounters the reader macro notation `#`⟨*Newline*⟩ or `#`⟨*Space*⟩.

2.4.8.22 Sharpsign Right-Parenthesis

This is not valid reader syntax.

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

2.4.9 Re-Reading Abbreviated Expressions

Note that the *Lisp reader* will generally signal an error of *type* **reader-error** when reading an $expression_2$ that has been abbreviated because of length or level limits (see ***print-level***, ***print-length***, and ***print-lines***) due to restrictions on "..", "...", "#" followed by $whitespace_1$, and "#)".