

Programming Language—Common Lisp

23. Reader

Version 15.17R, X3J13/94-101R.
Fri 12-Aug-1994 6:35pm EDT

23.1 Reader Concepts

23.1.1 Dynamic Control of the Lisp Reader

Various aspects of the *Lisp reader* can be controlled dynamically. See Section 2.1.1 (Readtables) and Section 2.1.2 (Variables that affect the Lisp Reader).

23.1.2 Effect of Readtable Case on the Lisp Reader

The *readtable case* of the *current readtable* affects the *Lisp reader* in the following ways:

:upcase

When the *readtable case* is **:upcase**, unescaped constituent *characters* are converted to *uppercase*, as specified in Section 2.2 (Reader Algorithm).

:downcase

When the *readtable case* is **:downcase**, unescaped constituent *characters* are converted to *lowercase*.

:preserve

When the *readtable case* is **:preserve**, the case of all *characters* remains unchanged.

:invert

When the *readtable case* is **:invert**, then if all of the unescaped letters in the extended token are of the same *case*, those (unescaped) letters are converted to the opposite *case*.

23.1.2.1 Examples of Effect of Readtable Case on the Lisp Reader

```
(defun test-readtable-case-reading ()
  (let ((*readtable* (copy-readtable nil)))
    (format t "READTABLE-CASE  Input  Symbol-name~
              ~%-----~
              ~%")
    (dolist (readtable-case '(:upcase :downcase :preserve :invert))
      (setf (readtable-case *readtable*) readtable-case)
      (dolist (input '("ZEBRA" "Zebra" "zebra"))
        (format t "~&:~A~16T~A~24T~A"
                  (string-upcase readtable-case)
                  input
                  (symbol-name (read-from-string input)))))))
```

The output from (test-readtable-case-reading) should be as follows:

READTABLE-CASE	Input	Symbol-name

:UPCASE	ZEBRA	ZEBRA
:UPCASE	Zebra	ZEBRA
:UPCASE	zebra	ZEBRA
:DOWNCASE	ZEBRA	zebra
:DOWNCASE	Zebra	zebra
:DOWNCASE	zebra	zebra
:PRESERVE	ZEBRA	ZEBRA
:PRESERVE	Zebra	Zebra
:PRESERVE	zebra	zebra
:INVERT	ZEBRA	zebra
:INVERT	Zebra	Zebra
:INVERT	zebra	ZEBRA

23.1.3 Argument Conventions of Some Reader Functions

23.1.3.1 The EOF-ERROR-P argument

Eof-error-p in input function calls controls what happens if input is from a file (or any other input source that has a definite end) and the end of the file is reached. If *eof-error-p* is *true* (the default), an error of *type* **end-of-file** is signaled at end of file. If it is *false*, then no error is signaled, and instead the function returns *eof-value*.

Functions such as **read** that read the representation of an *object* rather than a single character always signals an error, regardless of *eof-error-p*, if the file ends in the middle of an object representation. For example, if a file does not contain enough right parentheses to balance the left parentheses in it, **read** signals an error. If a file ends in a *symbol* or a *number* immediately followed by end-of-file, **read** reads the *symbol* or *number* successfully and when called again will act according to *eof-error-p*. Similarly, the *function* **read-line** successfully reads the last line of a file even if that line is terminated by end-of-file rather than the newline character. Ignorable text, such as lines containing only *whitespace₂* or comments, are not considered to begin an *object*; if **read** begins to read an *expression* but sees only such ignorable text, it does not consider the file to end in the middle of an *object*. Thus an *eof-error-p* argument controls what happens when the file ends between *objects*.

23.1.3.2 The RECURSIVE-P argument

If *recursive-p* is supplied and not **nil**, it specifies that this function call is not an outermost call to **read** but an embedded call, typically from a *reader macro function*. It is important to distinguish such recursive calls for three reasons.

1. An outermost call establishes the context within which the **#n=** and **#n#** syntax is scoped. Consider, for example, the expression

```
(cons '#3=(p q r) '(x y . #3#))
```

If the *single-quote reader macro* were defined in this way:

```
(set-macro-character #'      ;incorrect
 #'(lambda (stream char)
      (declare (ignore char))
      (list 'quote (read stream)))))
```

then each call to the *single-quote reader macro function* would establish independent contexts for the scope of **read** information, including the scope of identifications between markers like “**#3=**” and “**#3#**”. However, for this expression, the scope was clearly intended to be determined by the outer set of parentheses, so such a definition would be incorrect. The correct way to define the *single-quote reader macro* uses *recursive-p*:

```
(set-macro-character #'      ;correct
 #'(lambda (stream char)
      (declare (ignore char))
      (list 'quote (read stream t nil t)))))
```

2. A recursive call does not alter whether the reading process is to preserve *whitespace₂* or not (as determined by whether the outermost call was to **read** or **read-preserving-whitespace**). Suppose again that *single-quote* were to be defined as shown above in the incorrect definition. Then a call to **read-preserving-whitespace** that read the expression **'foo**(*Space*) would fail to preserve the space character following the symbol **foo** because the *single-quote reader macro function* calls **read**, not **read-preserving-whitespace**, to read the following expression (in this case **foo**). The correct definition, which passes the value *true* for *recursive-p* to **read**, allows the outermost call to determine whether *whitespace₂* is preserved.
3. When end-of-file is encountered and the *eof-error-p* argument is not **nil**, the kind of error that is signaled may depend on the value of *recursive-p*. If *recursive-p* is *true*, then the end-of-file is deemed to have occurred within the middle of a printed representation; if *recursive-p* is *false*, then the end-of-file may be deemed to have occurred between *objects* rather than within the middle of one.

readtable

System Class

Class Precedence List:

readtable, t

Description:

A *readtable* maps *characters* into *syntax types* for the *Lisp reader*; see Chapter 2 (Syntax). A *readtable* also contains associations between *macro characters* and their *reader macro functions*, and records information about the case conversion rules to be used by the *Lisp reader* when parsing *symbols*.

Each *simple character* must be representable in the *readtable*. It is *implementation-defined* whether *non-simple characters* can have syntax descriptions in the *readtable*.

See Also:

Section 2.1.1 (Readtables), Section 22.1.3.13 (Printing Other Objects)

copy-readtable

Function

Syntax:

`copy-readtable &optional from-readtable to-readtable` → *readtable*

Arguments and Values:

from-readtable—a *readtable designator*. The default is the *current readtable*.

to-readtable—a *readtable* or `nil`. The default is `nil`.

readtable—the *to-readtable* if it is *non-nil*, or else a *fresh readtable*.

Description:

`copy-readtable` copies *from-readtable*.

If *to-readtable* is `nil`, a new *readtable* is created and returned. Otherwise the *readtable* specified by *to-readtable* is modified and returned.

`copy-readtable` copies the setting of `readtable-case`.

Examples:

```
(setq zvar 123) → 123
(set-syntax-from-char #\z #'(setq table2 (copy-readtable))) → T
zvar → 123
(copy-readtable table2 *readtable*) → #<READTABLE 614000277>
```

```
zvar → VAR
(setq *readtable* (copy-readtable)) → #<READTABLE 46210223>
zvar → VAR
(setq *readtable* (copy-readtable nil)) → #<READTABLE 46302670>
zvar → 123
```

See Also:

`readtable`, `*readtable*`

Notes:

```
(setq *readtable* (copy-readtable nil))
```

restores the input syntax to standard Common Lisp syntax, even if the *initial readtable* has been clobbered (assuming it is not so badly clobbered that you cannot type in the above expression).

On the other hand,

```
(setq *readtable* (copy-readtable))
```

replaces the current *readtable* with a copy of itself. This is useful if you want to save a copy of a *readtable* for later use, protected from alteration in the meantime. It is also useful if you want to locally bind the *readtable* to a copy of itself, as in:

```
(let ((*readtable* (copy-readtable))) ...)
```

make-dispatch-macro-character

Function

Syntax:

```
make-dispatch-macro-character char &optional non-terminating-p readtable → t
```

Arguments and Values:

char—a *character*.

non-terminating-p—a *generalized boolean*. The default is *false*.

readtable—a *readtable*. The default is the *current readtable*.

Description:

`make-dispatch-macro-character` makes *char* be a *dispatching macro character* in *readtable*.

Initially, every *character* in the dispatch table associated with the *char* has an associated function that signals an error of *type* `reader-error`.

If *non-terminating-p* is *true*, the *dispatching macro character* is made a *non-terminating macro character*; if *non-terminating-p* is *false*, the *dispatching macro character* is made a *terminating macro character*.

Examples:

```
(get-macro-character #\{} → NIL, false
(make-dispatch-macro-character #\{} → T
(not (get-macro-character #\{})) → false
```

The *readtable* is altered.

See Also:

readtable, set-dispatch-macro-character

read, read-preserving-whitespace

Function

Syntax:

```
read &optional input-stream eof-error-p eof-value recursive-p → object

read-preserving-whitespace &optional input-stream eof-error-p
                                     eof-value recursive-p

→ object
```

Arguments and Values:

input-stream—an *input stream designator*.

eof-error-p—a *generalized boolean*. The default is *true*.

eof-value—an *object*. The default is *nil*.

recursive-p—a *generalized boolean*. The default is *false*.

object—an *object* (parsed by the *Lisp reader*) or the *eof-value*.

Description:

read parses the printed representation of an *object* from *input-stream* and builds such an *object*.

read-preserving-whitespace is like **read** but preserves any *whitespace₂ character* that delimits the printed representation of the *object*. **read-preserving-whitespace** is exactly like **read** when the *recursive-p* argument to **read-preserving-whitespace** is *true*.

read, read-preserving-whitespace

When ***read-suppress*** is *false*, **read** throws away the delimiting *character* required by certain printed representations if it is a *whitespace₂ character*; but **read** preserves the character (using **unread-char**) if it is syntactically meaningful, because it could be the start of the next expression.

If a file ends in a *symbol* or a *number* immediately followed by an *end of file₁*, **read** reads the *symbol* or *number* successfully; when called again, it sees the *end of file₁* and only then acts according to *eof-error-p*. If a file contains ignorable text at the end, such as blank lines and comments, **read** does not consider it to end in the middle of an *object*.

If *recursive-p* is *true*, the call to **read** is expected to be made from within some function that itself has been called from **read** or from a similar input function, rather than from the top level.

Both functions return the *object* read from *input-stream*. *Eof-value* is returned if *eof-error-p* is *false* and end of file is reached before the beginning of an *object*.

Examples:

```
(read)
▷ 'a
→ (QUOTE A)
(with-input-from-string (is " ") (read is nil 'the-end)) → THE-END
(defun skip-then-read-char (s c n)
  (if (char= c #\{) (read s t nil t) (read-preserving-whitespace s))
  (read-char-no-hang s)) → SKIP-THEN-READ-CHAR
(let ((*readtable* (copy-readtable nil)))
  (set-dispatch-macro-character #\# #\{ #'skip-then-read-char)
  (set-dispatch-macro-character #\# #\} #'skip-then-read-char)
  (with-input-from-string (is "#{123 x #}123 y")
    (format t "~S ~S" (read is) (read is)))) → #\x, #\Space, NIL
```

As an example, consider this *reader macro* definition:

```
(defun slash-reader (stream char)
  (declare (ignore char))
  '(path . ,(loop for dir = (read-preserving-whitespace stream t nil t)
    then (progn (read-char stream t nil t)
      (read-preserving-whitespace stream t nil t))
    collect dir
    while (eql (peek-char nil stream nil nil t) #\(/))))
(set-macro-character #\ / #'slash-reader)
```

Consider now calling **read** on this expression:

```
(zyedh /usr/games/zork /usr/games/boggle)
```

The **/** macro reads objects separated by more **/** characters; thus **/usr/games/zork** is intended to read as **(path usr games zork)**. The entire example expression should therefore be read as

```
(zyedh (path usr games zork) (path usr games boggle))
```

However, if `read` had been used instead of **read-preserving-whitespace**, then after the reading of the symbol `zork`, the following space would be discarded; the next call to **peek-char** would see the following `/`, and the loop would continue, producing this interpretation:

```
(zyedh (path usr games zork usr games boggle))
```

There are times when *whitespace₂* should be discarded. If a command interpreter takes single-character commands, but occasionally reads an *object* then if the *whitespace₂* after a *symbol* is not discarded it might be interpreted as a command some time later after the *symbol* had been read.

Affected By:

`*standard-input*`, `*terminal-io*`, `*readtable*`, `*read-default-float-format*`, `*read-base*`,
`*read-suppress*`, `*package*`, `*read-eval*`.

Exceptional Situations:

`read` signals an error of *type* **end-of-file**, regardless of *eof-error-p*, if the file ends in the middle of an *object* representation. For example, if a file does not contain enough right parentheses to balance the left parentheses in it, `read` signals an error. This is detected when `read` or **read-preserving-whitespace** is called with *recursive-p* and *eof-error-p non-nil*, and end-of-file is reached before the beginning of an *object*.

If *eof-error-p* is *true*, an error of *type* **end-of-file** is signaled at the end of file.

See Also:

`peek-char`, `read-char`, `unread-char`, `read-from-string`, `read-delimited-list`, `parse-integer`,
Chapter 2 (Syntax), Section 23.1 (Reader Concepts)

read-delimited-list

Function

Syntax:

```
read-delimited-list char &optional input-stream recursive-p → list
```

Arguments and Values:

char—a *character*.

input-stream—an *input stream designator*. The default is *standard input*.

recursive-p—a *generalized boolean*. The default is *false*.

list—a *list* of the *objects* read.

read-delimited-list

Description:

read-delimited-list reads *objects* from *input-stream* until the next character after an *object*'s representation (ignoring *whitespace₂* characters and comments) is *char*.

read-delimited-list looks ahead at each step for the next non-*whitespace₂* character and peeks at it as if with **peek-char**. If it is *char*, then the *character* is consumed and the *list* of *objects* is returned. If it is a *constituent* or *escape character*, then **read** is used to read an *object*, which is added to the end of the *list*. If it is a *macro character*, its *reader macro function* is called; if the function returns a *value*, that *value* is added to the *list*. The peek-ahead process is then repeated.

If *recursive-p* is *true*, this call is expected to be embedded in a higher-level call to **read** or a similar function.

It is an error to reach end-of-file during the operation of **read-delimited-list**.

The consequences are undefined if *char* has a *syntax type* of *whitespace₂* in the *current readtable*.

Examples:

```
(read-delimited-list #\]) 1 2 3 4 5 6 ]  
→ (1 2 3 4 5 6)
```

Suppose you wanted `#{a b c ... z}` to read as a list of all pairs of the elements *a*, *b*, *c*, ..., *z*, for example.

```
#{p q z a} reads as ((p q) (p z) (p a) (q z) (q a) (z a))
```

This can be done by specifying a macro-character definition for `#{` that does two things: reads in all the items up to the `}`, and constructs the pairs. **read-delimited-list** performs the first task.

```
(defun |#{-reader| (stream char arg)  
  (declare (ignore char arg))  
  (mapcon #'(lambda (x)  
    (mapcar #'(lambda (y) (list (car x) y)) (cdr x)))  
    (read-delimited-list #\} stream t))) → |#{-reader|  
  
(set-dispatch-macro-character #\# #\{ #'|#{-reader|) → T  
(set-macro-character #\} (get-macro-character #\) nil))
```

Note that *true* is supplied for the *recursive-p* argument.

It is necessary here to give a definition to the character `}` as well to prevent it from being a constituent. If the line

```
(set-macro-character #\} (get-macro-character #\) nil))
```

shown above were not included, then the `}` in

```
#{ p q z a}
```

would be considered a constituent character, part of the symbol named `a}`. This could be corrected by putting a space before the `}`, but it is better to call `set-macro-character`.

Giving `}` the same definition as the standard definition of the character `}` has the twin benefit of making it terminate tokens for use with **read-delimited-list** and also making it invalid for use in any other context. Attempting to read a stray `}` will signal an error.

Affected By:

***standard-input*, *readtable*, *terminal-io*.**

See Also:

read, peek-char, read-char, unread-char.

Notes:

read-delimited-list is intended for use in implementing *reader macros*. Usually it is desirable for *char* to be a *terminating macro character* so that it can be used to delimit tokens; however, **read-delimited-list** makes no attempt to alter the syntax specified for *char* by the current readtable. The caller must make any necessary changes to the readtable syntax explicitly.

read-from-string

Function

Syntax:

```
read-from-string string &optional eof-error-p eof-value
                  &key start end preserve-whitespace
```

→ *object, position*

Arguments and Values:

string—a *string*.

eof-error-p—a *generalized boolean*. The default is *true*.

eof-value—an *object*. The default is **nil**.

start, *end*—bounding index designators of *string*. The defaults for *start* and *end* are 0 and **nil**, respectively.

preserve-whitespace—a *generalized boolean*. The default is *false*.

object—an *object* (parsed by the *Lisp reader*) or the *eof-value*.

position—an *integer* greater than or equal to zero, and less than or equal to one more than the *length* of the *string*.

Description:

Parses the printed representation of an *object* from the subsequence of *string* bounded by *start* and *end*, as if **read** had been called on an *input stream* containing those same *characters*.

If *preserve-whitespace* is *true*, the operation will preserve *whitespace₂* as **read-preserving-whitespace** would do.

If an *object* is successfully parsed, the *primary value*, *object*, is the *object* that was parsed. If *eof-error-p* is *false* and if the end of the *substring* is reached, *eof-value* is returned.

The *secondary value*, *position*, is the index of the first *character* in the bounded *string* that was not read. The *position* may depend upon the value of *preserve-whitespace*. If the entire *string* was read, the *position* returned is either the *length* of the *string* or one greater than the *length* of the *string*.

Examples:

```
(read-from-string " 1 3 5" t nil :start 2) → 3, 5  
(read-from-string "(a b c)") → (A B C), 7
```

Exceptional Situations:

If the end of the supplied substring occurs before an *object* can be read, an error is signaled if *eof-error-p* is *true*. An error is signaled if the end of the *substring* occurs in the middle of an incomplete *object*.

See Also:

read, **read-preserving-whitespace**

Notes:

The reason that *position* is allowed to be beyond the *length* of the *string* is to permit (but not require) the *implementation* to work by simulating the effect of a trailing delimiter at the end of the bounded *string*. When *preserve-whitespace* is *true*, the *position* might count the simulated delimiter.

readtable-case

Accessor

Syntax:

```
readtable-case readtable → mode  
(setf (readtable-case readtable) mode)
```

Arguments and Values:

readtable—a *readtable*.

mode—a *case sensitivity mode*.

Description:

Accesses the *readtable case* of *readtable*, which affects the way in which the *Lisp Reader* reads *symbols* and the way in which the *Lisp Printer* writes *symbols*.

Examples:

See Section 23.1.2.1 (Examples of Effect of Readtable Case on the Lisp Reader) and Section 22.1.3.3.2.1 (Examples of Effect of Readtable Case on the Lisp Printer).

Exceptional Situations:

Should signal an error of *type* **type-error** if *readtable* is not a *readtable*. Should signal an error of *type* **type-error** if *mode* is not a *case sensitivity mode*.

See Also:

readtable, ***print-escape***, Section 2.2 (Reader Algorithm), Section 23.1.2 (Effect of Readtable Case on the Lisp Reader), Section 22.1.3.3.2 (Effect of Readtable Case on the Lisp Printer)

Notes:

copy-readtable copies the *readtable case* of the *readtable*.

readtablep

Function

Syntax:

readtablep *object* → *generalized-boolean*

Arguments and Values:

object—an *object*.

generalized-boolean—a *generalized boolean*.

Description:

Returns *true* if *object* is of *type* **readtable**; otherwise, returns *false*.

Examples:

```
(readtablep *readtable*) → true
(readtablep (copy-readtable)) → true
(readtablep 'readtable) → false
```

Notes:

```
(readtablep object) ≡ (typep object 'readtable)
```

set-dispatch-macro-character, get-dispatch-macro-character

Function

Syntax:

`get-dispatch-macro-character` *disp-char sub-char* &optional *readtable* → *function*

`set-dispatch-macro-character` *disp-char sub-char new-function* &optional *readtable* → *t*

Arguments and Values:

disp-char—a *character*.

sub-char—a *character*.

readtable—a *readtable designator*. The default is the *current readtable*.

function—a *function designator* or `nil`.

new-function—a *function designator*.

Description:

`set-dispatch-macro-character` causes *new-function* to be called when *disp-char* followed by *sub-char* is read. If *sub-char* is a lowercase letter, it is converted to its uppercase equivalent. It is an error if *sub-char* is one of the ten decimal digits.

`set-dispatch-macro-character` installs a *new-function* to be called when a particular *dispatching macro character* pair is read. *New-function* is installed as the dispatch function to be called when *readtable* is in use and when *disp-char* is followed by *sub-char*.

For more information about how the *new-function* is invoked, see Section 2.1.4.4 (Macro Characters).

`get-dispatch-macro-character` retrieves the dispatch function associated with *disp-char* and *sub-char* in *readtable*.

`get-dispatch-macro-character` returns the macro-character function for *sub-char* under *disp-char*, or `nil` if there is no function associated with *sub-char*. If *sub-char* is a decimal digit, `get-dispatch-macro-character` returns `nil`.

Examples:

```
(get-dispatch-macro-character #\# #\{) → NIL
(set-dispatch-macro-character #\# #\{          ;dispatch on #{
  #'(lambda(s c n)
    (let ((list (read s nil (values) t)))    ;list is object after #{
```

```
(when (consp list)                                ;return nth element of list
      (unless (and n (< 0 n (length list))) (setq n 0))
      (setq list (nth n list)))
list))) → T
#{(1 2 3 4) → 1
#3{(0 1 2 3) → 3
#{123 → 123
```

If it is desired that `#$foo`: as if it were (dollars *foo*).

```
(defun |#$_reader| (stream subchar arg)
  (declare (ignore subchar arg))
  (list 'dollars (read stream t nil t))) → |#$_reader|
(set-dispatch-macro-character #\# #\$ #'|#$_reader|) → T
```

See Also:

Section 2.1.4.4 (Macro Characters)

Side Effects:

The *readtable* is modified.

Affected By:

`*readtable*`.

Exceptional Situations:

For either function, an error is signaled if *disp-char* is not a *dispatching macro character* in *readtable*.

See Also:

`*readtable*`

Notes:

It is necessary to use **make-dispatch-macro-character** to set up the dispatch character before specifying its sub-characters.

set-macro-character, get-macro-character *Function*

Syntax:

```
get-macro-character char &optional readtable → function, non-terminating-p
set-macro-character char new-function &optional non-terminating-p readtable → t
```


set-macro-character, get-macro-character

Arguments and Values:

char—a *character*.

non-terminating-p—a *generalized boolean*. The default is *false*.

readtable—a *readtable designator*. The default is the *current readtable*.

function—*nil*, or a *designator* for a *function* of two *arguments*.

new-function—a *function designator*.

Description:

get-macro-character returns as its *primary value*, *function*, the *reader macro function* associated with *char* in *readtable* (if any), or else *nil* if *char* is not a *macro character* in *readtable*. The *secondary value*, *non-terminating-p*, is *true* if *char* is a *non-terminating macro character*; otherwise, it is *false*.

set-macro-character causes *char* to be a *macro character* associated with the *reader macro function* *new-function* (or the *designator* for *new-function*) in *readtable*. If *non-terminating-p* is *true*, *char* becomes a *non-terminating macro character*; otherwise it becomes a *terminating macro character*.

Examples:

```
(get-macro-character #\{) → NIL, false
(not (get-macro-character #\;)) → false
```

The following is a possible definition for the *single-quote reader macro* in *standard syntax*:

```
(defun single-quote-reader (stream char)
  (declare (ignore char))
  (list 'quote (read stream t nil t))) → SINGLE-QUOTE-READER
(set-macro-character #\' #'single-quote-reader) → T
```

Here *single-quote-reader* reads an *object* following the *single-quote* and returns a *list* of **quote** and that *object*. The *char* argument is ignored.

The following is a possible definition for the *semicolon reader macro* in *standard syntax*:

```
(defun semicolon-reader (stream char)
  (declare (ignore char))
  ;; First swallow the rest of the current input line.
  ;; End-of-file is acceptable for terminating the comment.
  (do () ((char= (read-char stream nil #\Newline t) #\Newline)))
  ;; Return zero values.
  (values)) → SEMICOLON-READER
(set-macro-character #\; #'semicolon-reader) → T
```

Side Effects:

The *readtable* is modified.

See Also:

readtable

set-syntax-from-char

Function

Syntax:

`set-syntax-from-char to-char from-char &optional to-readtable from-readtable` → *t*

Arguments and Values:

to-char—a character.

from-char—a character.

to-readtable—a *readtable*. The default is the *current readtable*.

from-readtable—a *readtable designator*. The default is the *standard readtable*.

Description:

set-syntax-from-char makes the syntax of *to-char* in *to-readtable* be the same as the syntax of *from-char* in *from-readtable*.

set-syntax-from-char copies the *syntax types* of *from-char*. If *from-char* is a *macro character*, its *reader macro function* is copied also. If the character is a *dispatching macro character*, its entire dispatch table of *reader macro functions* is copied. The *constituent traits* of *from-char* are not copied.

A macro definition from a character such as " can be copied to another character; the standard definition for " looks for another character that is the same as the character that invoked it. The definition of (can not be meaningfully copied to {, on the other hand. The result is that *lists* are of the form {a b c), not {a b c}, because the definition always looks for a closing parenthesis, not a closing brace.

Examples:

```
(set-syntax-from-char #\7 #\;) → T
123579 → 1235
```

Side Effects:

The *to-readtable* is modified.

Affected By:

The existing values in the *from-readtable*.

See Also:

set-macro-character, **make-dispatch-macro-character**, Section 2.1.4 (Character Syntax Types)

Notes:

The *constituent traits* of a *character* are “hard wired” into the parser for extended *tokens*. For example, if the definition of **S** is copied to *****, then ***** will become a *constituent* that is *alphabetic₂* but that cannot be used as a *short float exponent marker*. For further information, see Section 2.1.4.2 (Constituent Traits).

with-standard-io-syntax

Macro

Syntax:

with-standard-io-syntax {*form*}^{*} → {*result*}^{*}

Arguments and Values:

forms—an *implicit progn.*

results—the *values* returned by the *forms*.

Description:

Within the dynamic extent of the body of *forms*, all reader/prINTER control variables, including any *implementation-defined* ones not specified by this standard, are bound to values that produce standard read/print behavior. The values for the variables specified by this standard are listed in Figure 23–1.

Variable	Value
package	The CL-USER <i>package</i>
print-array	t
print-base	10
print-case	:upcase
print-circle	nil
print-escape	t
print-gensym	t
print-length	nil
print-level	nil
print-lines	nil
print-miser-width	nil
print-pprint-dispatch	The <i>standard pprint dispatch table</i>
print-pretty	nil
print-radix	nil
print-readably	t
print-right-margin	nil
read-base	10
read-default-float-format	single-float
read-eval	t
read-suppress	nil
readtable	The <i>standard readtable</i>

Figure 23–1. Values of standard control variables

Examples:

```
(with-open-file (file pathname :direction :output)
  (with-standard-io-syntax
    (print data file)))
```

;;; ... Later, in another Lisp:

```
(with-open-file (file pathname :direction :input)
  (with-standard-io-syntax
    (setq data (read file))))
```

read-base

Variable

Value Type:

a *radix*.

Initial Value:

10.

Description:

Controls the interpretation of tokens by **read** as being *integers* or *ratios*.

The *value* of ***read-base***, called the **current input base**, is the radix in which *integers* and *ratios* are to be read by the *Lisp reader*. The parsing of other numeric *types* (e.g., *floats*) is not affected by this option.

The effect of ***read-base*** on the reading of any particular *rational* number can be locally overridden by explicit use of the **#0**, **#X**, **#B**, or **#nR** syntax or by a trailing decimal point.

Examples:

```
(dotimes (i 6)
  (let ((*read-base* (+ 10. i)))
    (let ((object (read-from-string "\\DAD DAD |BEE| BEE 123. 123)"))
      (print (list *read-base* object)))))
> (10 (DAD DAD BEE BEE 123 123))
> (11 (DAD DAD BEE BEE 123 146))
> (12 (DAD DAD BEE BEE 123 171))
> (13 (DAD DAD BEE BEE 123 198))
> (14 (DAD 2701 BEE BEE 123 227))
> (15 (DAD 3088 BEE 2699 123 258))
→ NIL
```

Notes:

Altering the input radix can be useful when reading data files in special formats.

read-default-float-format

Variable

Value Type:

one of the *atomic type specifiers* **short-float**, **single-float**, **double-float**, or **long-float**, or else some other *type specifier* defined by the *implementation* to be acceptable.

Initial Value:

The *symbol* **single-float**.

Description:

Controls the floating-point format that is to be used when reading a floating-point number that has no *exponent marker* or that has **e** or **E** for an *exponent marker*. Other *exponent markers* explicitly prescribe the floating-point format to be used.

The printer uses ***read-default-float-format*** to guide the choice of *exponent markers* when printing floating-point numbers.

Examples:

```
(let ((*read-default-float-format* 'double-float))
  (read-from-string "(1.0 1.0e0 1.0s0 1.0f0 1.0d0 1.0L0)"))
→ (1.0 1.0 1.0 1.0 1.0 1.0) ;Implementation has float format F.
→ (1.0 1.0 1.0s0 1.0 1.0 1.0) ;Implementation has float formats S and F.
→ (1.0d0 1.0d0 1.0 1.0 1.0d0 1.0d0) ;Implementation has float formats F and D.
→ (1.0d0 1.0d0 1.0s0 1.0 1.0d0 1.0d0) ;Implementation has float formats S, F, D.
→ (1.0d0 1.0d0 1.0 1.0 1.0d0 1.0L0) ;Implementation has float formats F, D, L.
→ (1.0d0 1.0d0 1.0s0 1.0 1.0d0 1.0L0) ;Implementation has formats S, F, D, L.
```

read-eval

Variable

Value Type:

a *generalized boolean*.

Initial Value:

true.

Description:

If it is *true*, the **#.** *reader macro* has its normal effect. Otherwise, that *reader macro* signals an error of *type* **reader-error**.

See Also:

print-readably

Notes:

If ***read-eval*** is *false* and ***print-readably*** is *true*, any *method* for **print-object** that would output a reference to the **#.** *reader macro* either outputs something different or signals an error of *type* **print-not-readable**.

read-suppress

read-suppress

Variable

Value Type:

a generalized boolean.

Initial Value:

false.

Description:

This variable is intended primarily to support the operation of the read-time conditional notations **#+** and **#-**. It is important for the *reader macros* which implement these notations to be able to skip over the printed representation of an *expression* despite the possibility that the syntax of the skipped *expression* may not be entirely valid for the current implementation, since **#+** and **#-** exist in order to allow the same program to be shared among several Lisp implementations (including dialects other than Common Lisp) despite small incompatibilities of syntax.

If it is *false*, the *Lisp reader* operates normally.

If the *value* of ***read-suppress*** is *true*, **read**, **read-preserving-whitespace**, **read-delimited-list**, and **read-from-string** all return a *primary value* of **nil** when they complete successfully; however, they continue to parse the representation of an *object* in the normal way, in order to skip over the *object*, and continue to indicate *end of file* in the normal way. Except as noted below, any *standardized reader macro*₂ that is defined to *read*₂ a following *object* or *token* will do so, but not signal an error if the *object* read is not of an appropriate type or syntax. The *standard syntax* and its associated *reader macros* will not construct any new *objects* (e.g., when reading the representation of a *symbol*, no *symbol* will be constructed or interned).

Extended tokens

All extended tokens are completely uninterpreted. Errors such as those that might otherwise be signaled due to detection of invalid *potential numbers*, invalid patterns of *package markers*, and invalid uses of the *dot* character are suppressed.

Dispatching macro characters (including *sharpsign*)

Dispatching macro characters continue to parse an infix numerical argument, and invoke the dispatch function. The *standardized sharpsign reader macros* do not enforce any constraints on either the presence of or the value of the numerical argument.

#=

The **#=** notation is totally ignored. It does not read a following *object*. It produces no *object*, but is treated as *whitespace*₂.

##

The `##` notation always produces `nil`.

No matter what the *value* of `*read-suppress*`, parentheses still continue to delimit and construct *lists*; the `#(` notation continues to delimit *vectors*; and comments, *strings*, and the *single-quote* and *backquote* notations continue to be interpreted properly. Such situations as `'`), `#<`, `#)`, and `#(Space)` continue to signal errors.

Examples:

```
(let ((*read-suppress* t))
  (mapcar #'read-from-string
    '("#(foo bar baz)" "#P(:type :lisp)" "#c1.2"
      "#.(PRINT 'FOO)" "#3AHELLO" "#S(INTEGER)"
      "**ABC" "#\GARBAGE" "#RALPHA" "#3R444")))
→ (NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
```

See Also:

`read`, Chapter 2 (Syntax)

Notes:

Programmers and *implementations* that define additional *macro characters* are strongly encouraged to make them respect `*read-suppress*` just as *standardized macro characters* do. That is, when the *value* of `*read-suppress*` is *true*, they should ignore type errors when reading a following *object* and the *functions* that implement *dispatching macro characters* should tolerate `nil` as their infix *parameter* value even if a numeric value would ordinarily be required.

`*readtable*`

Variable

Value Type:

a *readtable*.

Initial Value:

A *readtable* that conforms to the description of Common Lisp syntax in Chapter 2 (Syntax).

Description:

The *value* of `*readtable*` is called the *current readtable*. It controls the parsing behavior of the *Lisp reader*, and can also influence the *Lisp printer* (e.g., see the *function* `readtable-case`).

Examples:

```
(readtablep *readtable*) → true
(setq zvar 123) → 123
```

```
(set-syntax-from-char #\z #\' (setq table2 (copy-readtable))) → T
zvar → 123
(setq *readtable* table2) → #<READTABLE>
zvar → VAR
(setq *readtable* (copy-readtable nil)) → #<READTABLE>
zvar → 123
```

Affected By:

compile-file, load

See Also:

compile-file, load, readtable, Section 2.1.1.1 (The Current Readtable)

reader-error

Condition Type

Class Precedence List:

reader-error, parse-error, stream-error, error, serious-condition, condition, t

Description:

The *type* **reader-error** consists of error conditions that are related to tokenization and parsing done by the *Lisp reader*.

See Also:

read, stream-error-stream, Section 23.1 (Reader Concepts)

Version 15.17R, X3J13/94-101R.
Fri 12-Aug-1994 6:35pm EDT
