Programming Language—Common Lisp

21. Streams

21.1 Stream Concepts

21.1.1 Introduction to Streams

A ***stream*** is an *object* that can be used with an input or output function to identify an appropriate source or sink of *characters* or *bytes* for that operation. A ***character stream*** is a source or sink of *characters*. A ***binary stream*** is a source or sink of *bytes*.

Some operations may be performed on any kind of *stream*; Figure 21–1 provides a list of *standardized* operations that are potentially useful with any kind of *stream*.

close	**stream-element-type**
input-stream-p	**streamp**
interactive-stream-p	**with-open-stream**
output-stream-p	

Figure 21–1. Some General-Purpose Stream Operations

Other operations are only meaningful on certain *stream types*. For example, **read-char** is only defined for *character streams* and **read-byte** is only defined for *binary streams*.

21.1.1.1 Abstract Classifications of Streams

21.1.1.1.1 Input, Output, and Bidirectional Streams

A *stream*, whether a *character stream* or a *binary stream*, can be an ***input stream*** (source of data), an ***output stream*** (sink for data), both, or (*e.g.*, when "`:direction :probe`" is given to **open**) neither.

Figure 21–2 shows *operators* relating to *input streams*.

clear-input	**read-byte**	**read-from-string**
listen	**read-char**	**read-line**
peek-char	**read-char-no-hang**	**read-preserving-whitespace**
read	**read-delimited-list**	**unread-char**

Figure 21–2. Operators relating to Input Streams.

Figure 21–3 shows *operators* relating to *output streams*.

clear-output	**prin1**	**write**
finish-output	**prin1-to-string**	**write-byte**
force-output	**princ**	**write-char**
format	**princ-to-string**	**write-line**
fresh-line	**print**	**write-string**
pprint	**terpri**	**write-to-string**

Figure 21–3. Operators relating to Output Streams.

A *stream* that is both an *input stream* and an *output stream* is called a ***bidirectional stream***. See the *functions* **input-stream-p** and **output-stream-p**.

Any of the *operators* listed in Figure 21–2 or Figure 21–3 can be used with *bidirectional streams*. In addition, Figure 21–4 shows a list of *operators* that relate specificaly to *bidirectional streams*.

y-or-n-p	**yes-or-no-p**

Figure 21–4. Operators relating to Bidirectional Streams.

21.1.1.1.2 Open and Closed Streams

Streams are either ***open*** or ***closed***.

Except as explicitly specified otherwise, operations that create and return *streams* return *open streams*.

The action of *closing* a *stream* marks the end of its use as a source or sink of data, permitting the *implementation* to reclaim its internal data structures, and to free any external resources which might have been locked by the *stream* when it was opened.

Except as explicitly specified otherwise, the consequences are undefined when a *closed stream* is used where a *stream* is called for.

Coercion of *streams* to *pathnames* is permissible for *closed streams*; in some situations, such as for a *truename* computation, the result might be different for an *open stream* and for that same *stream* once it has been *closed*.

21.1.1.1.3 Interactive Streams

An ***interactive stream*** is one on which it makes sense to perform interactive querying.

The precise meaning of an *interactive stream* is *implementation-defined*, and may depend on the underlying operating system. Some examples of the things that an *implementation* might choose to use as identifying characteristics of an *interactive stream* include:

- The *stream* is connected to a person (or equivalent) in such a way that the program can prompt for information and expect to receive different input depending on the prompt.

- The program is expected to prompt for input and support "normal input editing".

- **read-char** might wait for the user to type something before returning instead of immediately returning a character or end-of-file.

The general intent of having some *streams* be classified as *interactive streams* is to allow them to be distinguished from streams containing batch (or background or command-file) input. Output to batch streams is typically discarded or saved for later viewing, so interactive queries to such streams might not have the expected effect.

Terminal I/O might or might not be an *interactive stream*.

21.1.1.2 Abstract Classifications of Streams

21.1.1.2.1 File Streams

Some *streams*, called ***file streams***, provide access to *files*. An *object* of *class* **file-stream** is used to represent a *file stream*.

The basic operation for opening a *file* is **open**, which typically returns a *file stream* (see its dictionary entry for details). The basic operation for closing a *stream* is **close**. The macro **with-open-file** is useful to express the common idiom of opening a *file* for the duration of a given body of *code*, and assuring that the resulting *stream* is closed upon exit from that body.

21.1.1.3 Other Subclasses of Stream

The *class* **stream** has a number of *subclasses* defined by this specification. Figure 21–5 shows some information about these subclasses.

Class	Related Operators
broadcast-stream	**make-broadcast-stream**
	broadcast-stream-streams
concatenated-stream	**make-concatenated-stream**
	concatenated-stream-streams
echo-stream	**make-echo-stream**
	echo-stream-input-stream
	echo-stream-output-stream
string-stream	**make-string-input-stream**
	with-input-from-string
	make-string-output-stream
	with-output-to-string
	get-output-stream-string
synonym-stream	**make-synonym-stream**
	synonym-stream-symbol
two-way-stream	**make-two-way-stream**
	two-way-stream-input-stream
	two-way-stream-output-stream

Figure 21–5. Defined Names related to Specialized Streams

21.1.2 Stream Variables

Variables whose *values* must be *streams* are sometimes called ***stream variables***.

Certain *stream variables* are defined by this specification to be the proper source of input or output in various *situations* where no specific *stream* has been specified instead. A complete list of such *standardized stream variables* appears in Figure 21–6. The consequences are undefined if at any time the *value* of any of these *variables* is not an *open stream*.

Glossary Term	Variable Name
debug I/O	***debug-io***
error output	***error-output***
query I/O	***query-io***
standard input	***standard-input***
standard output	***standard-output***
terminal I/O	***terminal-io***
trace output	***trace-output***

Figure 21–6. Standardized Stream Variables

Note that, by convention, *standardized stream variables* have names ending in "`-input*`" if they must be *input streams*, ending in "`-output*`" if they must be *output streams*, or ending in "`-io*`" if they must be *bidirectional streams*.

User programs may *assign* or *bind* any *standardized stream variable* except ***terminal-io***.

21.1.3 Stream Arguments to Standardized Functions

The *operators* in Figure 21–7 accept *stream arguments* that might be either *open* or *closed streams*.

broadcast-stream-streams	**file-author**	**pathnamep**
close	**file-namestring**	**probe-file**
compile-file	**file-write-date**	**rename-file**
compile-file-pathname	**host-namestring**	**streamp**
concatenated-stream-streams	**load**	**synonym-stream-symbol**
delete-file	**logical-pathname**	**translate-logical-pathname**
directory	**merge-pathnames**	**translate-pathname**
directory-namestring	**namestring**	**truename**
dribble	**open**	**two-way-stream-input-stream**
echo-stream-input-stream	**open-stream-p**	**two-way-stream-output-stream**
echo-stream-ouput-stream	**parse-namestring**	**wild-pathname-p**
ed	**pathname**	**with-open-file**
enough-namestring	**pathname-match-p**	

Figure 21–7. Operators that accept either Open or Closed Streams

The *operators* in Figure 21–8 accept *stream arguments* that must be *open streams*.

clear-input	**output-stream-p**	**read-char-no-hang**
clear-output	**peek-char**	**read-delimited-list**
file-length	**pprint**	**read-line**
file-position	**pprint-fill**	**read-preserving-whitespace**
file-string-length	**pprint-indent**	**stream-element-type**
finish-output	**pprint-linear**	**stream-external-format**
force-output	**pprint-logical-block**	**terpri**
format	**pprint-newline**	**unread-char**
fresh-line	**pprint-tab**	**with-open-stream**
get-output-stream-string	**pprint-tabular**	**write**
input-stream-p	**prin1**	**write-byte**
interactive-stream-p	**princ**	**write-char**
listen	**print**	**write-line**
make-broadcast-stream	**print-object**	**write-string**
make-concatenated-stream	**print-unreadable-object**	**y-or-n-p**
make-echo-stream	**read**	**yes-or-no-p**
make-synonym-stream	**read-byte**	
make-two-way-stream	**read-char**	

Figure 21–8. Operators that accept Open Streams only

21.1.4 Restrictions on Composite Streams

The consequences are undefined if any *component* of a *composite stream* is *closed* before the *composite stream* is *closed*.

The consequences are undefined if the *synonym stream symbol* is not *bound* to an *open stream* from the time of the *synonym stream*'s creation until the time it is *closed*.

stream *System Class*

Class Precedence List:

stream, **t**

Description:

A *stream* is an *object* that can be used with an input or output function to identify an appropriate source or sink of *characters* or *bytes* for that operation.

For more complete information, see Section 21.1 (Stream Concepts).

See Also:

Section 21.1 (Stream Concepts), Section 22.1.3.16 (Printing Other Objects), Chapter 22 (Printer), Chapter 23 (Reader)

broadcast-stream *System Class*

Class Precedence List:

broadcast-stream, **stream**, **t**

Description:

A *broadcast stream* is an *output stream* which has associated with it a set of zero or more *output streams* such that any output sent to the *broadcast stream* gets passed on as output to each of the associated *output streams*. (If a *broadcast stream* has no *component streams*, then all output to the *broadcast stream* is discarded.)

The set of operations that may be performed on a *broadcast stream* is the intersection of those for its associated *output streams*.

Some output operations (*e.g.*, **fresh-line**) return *values* based on the state of the *stream* at the time of the operation. Since these *values* might differ for each of the *component streams*, it is defined that the *values* returned by such an output operation are the *values* resulting from performing the operation on the last of its *component streams*; the *values* resulting from performing the operation on all preceding *streams* are discarded. If there are no *component streams*, the value is *implementation-dependent*.

See Also:

broadcast-stream-streams, **make-broadcast-stream**

concatenated-stream *System Class*

Class Precedence List:

concatenated-stream, **stream**, **t**

Description:

A *concatenated stream* is an *input stream* which is a *composite stream* of zero or more other *input streams*, such that the sequence of data which can be read from the *concatenated stream* is the same as the concatenation of the sequences of data which could be read from each of the constituent *streams*.

Input from a *concatenated stream* is taken from the first of the associated *input streams* until it reaches *end of file*$_1$; then that *stream* is discarded, and subsequent input is taken from the next *input stream*, and so on. An *end of file* on the associated *input streams* is always managed invisibly by the *concatenated stream*—the only time a client of a *concatenated stream* sees an *end of file* is when an attempt is made to obtain data from the *concatenated stream* but it has no remaining *input streams* from which to obtain such data.

See Also:

concatenated-stream-streams, **make-concatenated-stream**

echo-stream *System Class*

Class Precedence List:

echo-stream, **stream**, **t**

Description:

An *echo stream* is a *bidirectional stream* that gets its input from an associated *input stream* and sends its output to an associated *output stream*.

All input taken from the *input stream* is echoed to the *output stream*. Whether the input is echoed immediately after it is encountered, or after it has been read from the *input stream* is *implementation-dependent*.

See Also:

echo-stream-input-stream, **echo-stream-output-stream**, **make-echo-stream**

file-stream *System Class*

Class Precedence List:

file-stream, **stream**, **t**

Description:

An *object* of *type* **file-stream** is a *stream* the direct source or sink of which is a *file*. Such a *stream* is created explicitly by **open** and **with-open-file**, and implicitly by *functions* such as **load** that process *files*.

See Also:

load, **open**, **with-open-file**

string-stream *System Class*

Class Precedence List:

string-stream, **stream**, **t**

Description:

A *string stream* is a *stream* which reads input from or writes output to an associated *string*.

The *stream element type* of a *string stream* is always a *subtype* of *type* **character**.

See Also:

make-string-input-stream, **make-string-output-stream**, **with-input-from-string**, **with-output-to-string**

synonym-stream *System Class*

Class Precedence List:

synonym-stream, **stream**, **t**

Description:

A *stream* that is an alias for another *stream*, which is the *value* of a *dynamic variable* whose *name* is the *synonym stream symbol* of the *synonym stream*.

Any operations on a *synonym stream* will be performed on the *stream* that is then the *value* of the *dynamic variable* named by the *synonym stream symbol*. If the *value* of the *variable* should change, or if the *variable* should be *bound*, then the *stream* will operate on the new *value* of the *variable*.

See Also:

make-synonym-stream, **synonym-stream-symbol**

two-way-stream *System Class*

Class Precedence List:

two-way-stream, **stream**, **t**

Description:

A *bidirectional composite stream* that receives its input from an associated *input stream* and sends its output to an associated *output stream*.

See Also:

make-two-way-stream, **two-way-stream-input-stream**, **two-way-stream-output-stream**

input-stream-p, output-stream-p *Function*

Syntax:

input-stream-p *stream* → *boolean*

output-stream-p *stream* → *boolean*

Arguments and Values:

stream—a *stream*.

boolean—a *boolean*.

Description:

input-stream-p returns *true* if *stream* is an *input stream*; otherwise, returns *false*.

output-stream-p returns *true* if *stream* is an *output stream*; otherwise, returns *false*.

Examples:

```
(input-stream-p *standard-input*) → true
(input-stream-p *terminal-io*) → true
(input-stream-p (make-string-output-stream)) → false
```

```
(output-stream-p *standard-output*) → true
(output-stream-p *terminal-io*) → true
(output-stream-p (make-string-input-stream "jr")) → false
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *stream* is not a *stream*.

interactive-stream-p *Function*

Syntax:

interactive-stream-p *stream* → *boolean*

Arguments and Values:

stream—a *stream*.

boolean—a *boolean*.

Description:

Returns *true* if *stream* is an *interactive stream*; otherwise, returns *false*.

Examples:

```
(when (> measured limit)
  (let ((error (round (* (- measured limit) 100)
                      limit)))
    (unless (if (interactive-stream-p *query-io*)
                (yes-or-no-p "The frammis is out of tolerance by ~D%.~@
                              Is it safe to proceed? " error)
                (< error 15))  ;15% is acceptable
      (error "The frammis is out of tolerance by ~D%." error))))
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *stream* is not a *stream*.

See Also:

Section 21.1 (Stream Concepts)

open-stream-p *Function*

Syntax:

open-stream-p *stream* → *boolean*

Arguments and Values:

stream—a *stream*.

boolean—a *boolean*.

Description:

Returns *true* if *stream* is an *open stream*; otherwise, returns *false*.

Streams are open until they have been explicitly closed with **close**, or until they are implicitly closed due to exit from a **with-output-to-string**, **with-open-file**, **with-input-from-string**, or **with-open-stream** *form*.

Examples:

```
(open-stream-p *standard-input*) → true
```

Affected By:

close.

Exceptional Situations:

Should signal an error of *type* **type-error** if *stream* is not a *stream*.

stream-element-type *Function*

Syntax:

stream-element-type *stream* → *typespec*

Arguments and Values:

stream—a *stream*.

typespec—a *type specifier*.

Description:

stream-element-type returns a *type specifier* that indicates the *types* of *objects* that may be read from or written to *stream*.

Streams created by **open** have an *element type* restricted to **integer** or a *subtype* of *type* **character**.

Examples:

```
;; Note that the stream must accomodate at least the specified type,
;; but might accomodate other types.  Further note that even if it does
;; accomodate exactly the specified type, the type might be specified in
;; any of several ways.
 (with-open-file (s "test" :element-type '(integer 0 1)
   :if-exists :error
   :direction :output)
   (stream-element-type s))
→ INTEGER
or→ (UNSIGNED-BYTE 16)
or→ (UNSIGNED-BYTE 8)
or→ BIT
or→ (UNSIGNED-BYTE 1)
or→ (INTEGER 0 1)
or→ (INTEGER 0 (2))
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *stream* is not a *stream*.

streamp *Function*

Syntax:

streamp *object* → *boolean*

Arguments and Values:

object—an *object*.

boolean—a *boolean*.

Description:

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

streamp is unaffected by whether *object*, if it is a *stream*, is *open* or closed.

Examples:

```
(streamp *terminal-io*) → true
(streamp 1) → false
```

Notes:

```
(streamp object) ≡ (typep object 'stream)
```

read-byte *Function*

Syntax:

read-byte *stream* &optional *eof-error-p eof-value* → *byte*

Arguments and Values:

stream—a *binary input stream*.

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

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

byte—an *integer*, or the *eof-value*.

Description:

read-byte reads and returns one byte from *stream*.

If an *end of file*$_2$ occurs and *eof-error-p* is *false*, the *eof-value* is returned.

Examples:

```
 (with-open-file (s "temp-bytes"
                     :direction :output
                     :element-type 'unsigned-byte)
    (write-byte 101 s)) → 101
 (with-open-file (s "temp-bytes" :element-type 'unsigned-byte)
    (format t "~S ~S" (read-byte s) (read-byte s nil 'eof)))
▷ 101 EOF
→ NIL
```

Side Effects:

Modifies *stream*.

Exceptional Situations:

Should signal an error of *type* **type-error** if *stream* is not a *stream*.

Should signal an error of *type* **error** if *stream* is not a *binary input stream*.

If there are no *bytes* remaining in the *stream* and *eof-error-p* is *true*, an error of *type* **end-of-file** is signaled.

See Also:

write-byte

write-byte *Function*

Syntax:

write-byte *byte stream* → *byte*

Arguments and Values:

byte—an *integer* of the *stream element type* of *stream*.

stream—a *binary output stream*.

Description:

write-byte writes one byte, *byte*, to *stream*.

Examples:

```
(with-open-file (s "temp-bytes"
                   :direction :output
                   :element-type 'unsigned-byte)
   (write-byte 101 s)) → 101
```

Side Effects:

stream is modified.

Affected By:

The *element type* of the *stream*.

Exceptional Situations:

Should signal an error of *type* **type-error** if *stream* is not a *stream*. Should signal an error of *type* **error** if *stream* is not a *binary output stream*.

Might signal an error of *type* **type-error** if *byte* is not an *integer* of the *stream element type* of *stream*.

See Also:

read-byte

peek-char

peek-char *Function*

Syntax:

peek-char &optional *peek-type input-stream eof-error-p eof-value recursive-p* → *char*

Arguments and Values:

peek-type—a *character* or **t** or **nil**.

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

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

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

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

char—a *character* or the *eof-value*.

Description:

peek-char obtains the next character in *input-stream* without actually reading it, thus leaving the character to be read at a later time. It can also be used to skip over and discard intervening characters in the *input-stream* until a particular character is found.

If *peek-type* is not supplied or **nil**, **peek-char** returns the next character to be read from *input-stream*, without actually removing it from *input-stream*. The next time input is done from *input-stream*, the character will still be there. If *peek-type* is **t**, then **peek-char** skips over *whitespace*$_2$ *characters*, but not comments, and then performs the peeking operation on the next character. The last character examined, the one that starts an *object*, is not removed from *input-stream*. If *peek-type* is a *character*, then **peek-char** skips over input characters until a character that is **char=** to that *character* is found; that character is left in *input-stream*.

If an *end of file*$_2$ occurs and *eof-error-p* is *false*, *eof-value* is returned.

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

When *input-stream* is an *echo stream*, characters that are only peeked at are not echoed. In the case that *peek-type* is not **nil**, the characters that are passed by **peek-char** are treated as if by **read-char**, and so are echoed unless they have been marked otherwise by **unread-char**.

Examples:

```
 (with-input-from-string (input-stream "    1 2 3 4 5")
   (format t "~S ~S ~S"
           (peek-char t input-stream)
```

```
              (peek-char #\4 input-stream)
              (peek-char nil input-stream)))
▷ #\1 #\4 #\4
→ NIL
```

Affected By:

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

Exceptional Situations:

If *eof-error-p* is *true* and an *end of file*$_2$ occurs an error of *type* **end-of-file** is signaled.

If *peek-type* is a *character*, an *end of file*$_2$ occurs, and *eof-error-p* is *true*, an error of *type* **end-of-file** is signaled.

If *recursive-p* is *true* and an *end of file*$_2$ occurs, an error of *type* **end-of-file** is signaled.

read-char *Function*

Syntax:

read-char &optional *input-stream eof-error-p eof-value recursive-p* → *char*

Arguments and Values:

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

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

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

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

char—a *character* or the *eof-value*.

Description:

read-char returns the next *character* from *input-stream*.

When *input-stream* is an *echo stream*, the character is echoed on *input-stream* the first time the character is seen. Characters that are not echoed by **read-char** are those that were put there by **unread-char** and hence are assumed to have been echoed already by a previous call to **read-char**.

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

If an *end of file*$_2$ occurs and *eof-error-p* is *false*, *eof-value* is returned.

Examples:

```
 (with-input-from-string (is "0123")
    (do ((c (read-char is) (read-char is nil 'the-end)))
        ((not (characterp c)))
     (format t "~S " c)))
▷ #\0 #\1 #\2 #\3
→ NIL
```

Affected By:

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

Exceptional Situations:

If an *end of file*$_2$ occurs before a character can be read, and *eof-error-p* is *true*, an error of *type* **end-of-file** is signaled.

See Also:

read

Notes:

The corresponding output function is **write-char**.

read-char-no-hang *Function*

Syntax:

read-char-no-hang &optional *input-stream eof-error-p eof-value recursive-p* → *char*

Arguments and Values:

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

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

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

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

char—a *character* or **nil** or the *eof-value*.

Description:

read-char-no-hang returns a character from *input-stream* if such a character is available. If no character is available, **read-char-no-hang** returns **nil**.

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

If an *end of file*$_2$ occurs and *eof-error-p* is *false*, *eof-value* is returned.

Examples:

```
;; This code assumes an implementation in which a newline is not
;; required to terminate input from the console.
 (defun test-it ()
   (unread-char (read-char))
   (list (read-char-no-hang)
         (read-char-no-hang)
         (read-char-no-hang)))
→ TEST-IT
;; Implementation A, where a Newline is not required to terminate
;; interactive input on the console.
 (test-it)
▷ a
→ (#\a NIL NIL)
;; Implementation B, where a Newline is required to terminate
;; interactive input on the console, and where that Newline remains
;; on the input stream.
 (test-it)
▷ a↩
→ (#\a #\Newline NIL)
```

Affected By:

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

Exceptional Situations:

If an *end of file*$_2$ occurs when *eof-error-p* is *true*, an error of *type* **end-of-file** is signaled .

See Also:

listen

Notes:

read-char-no-hang is exactly like **read-char**, except that if it would be necessary to wait in order to get a character (as from a keyboard), **nil** is immediately returned without waiting.

terpri, fresh-line

terpri, fresh-line *Function*

Syntax:

terpri &optional *output-stream* → **nil**

fresh-line &optional *output-stream* → *boolean*

Arguments and Values:

output-stream – an *output stream designator*. The default is *standard output*.

boolean—a *boolean*.

Description:

terpri outputs a *newline* to *output-stream*.

fresh-line is similar to **terpri** but outputs a *newline* only if the *output-stream* is not already at the start of a line. If for some reason this cannot be determined, then a *newline* is output anyway. **fresh-line** returns *true* if it outputs a *newline*; otherwise it returns *false*.

Examples:

```
 (with-output-to-string (s)
    (write-string "some text" s)
    (terpri s)
    (terpri s)
    (write-string "more text" s))
→ "some text

more text"
 (with-output-to-string (s)
    (write-string "some text" s)
    (fresh-line s)
    (fresh-line s)
    (write-string "more text" s))
→ "some text
more text"
```

Side Effects:

The *output-stream* is modified.

Affected By:

standard-output, ***terminal-io***.

Exceptional Situations:

None.

Notes:

terpri is identical in effect to

```
(write-char #\Newline output-stream)
```

unread-char *Function*

Syntax:

unread-char *character* &optional *input-stream* → **nil**

Arguments and Values:

character—a *character*; must be the last *character* that was read from *input-stream*.

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

Description:

unread-char places *character* back onto the front of *input-stream* so that it will again be the next character in *input-stream*.

When *input-stream* is an *echo stream*, no attempt is made to undo any echoing of the character that might already have been done on *input-stream*. However, characters placed on *input-stream* by **unread-char** are marked in such a way as to inhibit later re-echo by **read-char**.

It is an error to invoke **unread-char** twice consecutively on the same *stream* without an intervening call to **read-char** (or some other input operation which implicitly reads characters) on that *stream*.

Invoking **peek-char** or **read-char** commits all previous characters. The consequences of invoking **unread-char** on any character preceding that which is returned by **peek-char** (including those passed over by **peek-char** that has a *non-nil* *peek-type*) are unspecified. In particular, the consequences of invoking **unread-char** after **peek-char** are unspecified.

Examples:

```
 (with-input-from-string (is "0123")
    (dotimes (i 6)
      (let ((c (read-char is)))
        (if (evenp i) (format t "~&~S ~S~%" i c) (unread-char c is)))))
▷ 0 #\0
▷ 2 #\1
▷ 4 #\2
→ NIL
```

Affected By:

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

See Also:

peek-char, **read-char**, Section 21.1 (Stream Concepts)

Notes:

unread-char is intended to be an efficient mechanism for allowing the *Lisp reader* and other parsers to perform one-character lookahead in *input-stream*.

write-char *Function*

Syntax:

write-char *character* &optional *output-stream* → *character*

Arguments and Values:

character—a *character*.

output-stream – an *output stream designator*. The default is *standard output*.

Description:

write-char outputs *character* to *output-stream*.

Examples:

```
 (write-char #\a)
▷ a
→ #\a
 (with-output-to-string (s)
   (write-char #\a s)
   (write-char #\Space s)
   (write-char #\b s))
→ "a b"
```

Side Effects:

The *output-stream* is modified.

Affected By:

standard-output, ***terminal-io***.

read-line *Function*

Syntax:

read-line &optional *input-stream eof-error-p eof-value recursive-p*
→ *line, missing-newline-p*

Arguments and Values:

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

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

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

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

line—a *string* or the *eof-value*.

missing-newline-p—a *boolean*.

Description:

Reads from *input-stream* a line of text that is terminated by a *newline* or *end of file*.

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

The *primary value*, *line*, is the line that is read, represented as a *string* (without the trailing *newline*, if any). If *eof-error-p* is *false* and the *end of file* for *input-stream* is reached before any *characters* are read, *eof-value* is returned as the *line*.

The *secondary value*, *missing-newline-p*, is a *boolean* that is *false* if the *line* was terminated by a *newline*, or *true* if the *line* was terminated by the *end of file* for *input-stream* (or if the *line* is the *eof-value*).

Examples:

```
 (setq a "line 1
 line2")
→ "line 1
 line2"
 (read-line (setq input-stream (make-string-input-stream a)))
→ "line 1", false
 (read-line input-stream)
→ "line2", true
 (read-line input-stream nil nil)
→ NIL, true
```

Affected By:

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

Exceptional Situations:

If an *end of file*$_2$ occurs before any characters are read in the line, an error is signaled if *eof-error-p* is *true*.

See Also:

read

Notes:

The corresponding output function is **write-line**.

write-string, write-line *Function*

Syntax:

write-string *string* &optional *output-stream* &key *start end* → *string*

write-line *string* &optional *output-stream* &key *start end* → *string*

Arguments and Values:

string—a *string*.

output-stream – an *output stream designator*. The default is *standard output*.

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

Description:

write-string writes the *characters* of the subsequence of *string* *bounded* by *start* and *end* to *output-stream*. **write-line** does the same thing, but then outputs a newline afterwards.

Examples:

```
 (prog1 (write-string "books" nil :end 4) (write-string "worms"))
▷ bookworms
→ "books"
 (progn (write-char #\*)
        (write-line "test12" *standard-output* :end 5)
        (write-line "*test2")
        (write-char #\*)
        nil)
▷ *test1
```

```
▷ *test2
▷ *
→ NIL
```

Affected By:

standard-output, ***terminal-io***.

See Also:

read-line, **write-char**

Notes:

write-line and **write-string** return *string*, not the substring *bounded* by *start* and *end*.

```
 (write-string string)
≡ (dotimes (i (length string)
     (write-char (char string i)))

 (write-line string)
≡ (prog1 (write-string string) (terpri))
```

file-length *Function*

Syntax:

file-length *stream* → *length*

Arguments and Values:

stream—a *stream associated with a file*.

length—a non-negative *integer* or **nil**.

Description:

file-length returns the length of *stream*, or **nil** if the length cannot be determined.

For a binary file, the length is measured in units of the *element type* of the *stream*.

Examples:

```
 (with-open-file (s "decimal-digits.text"
                    :direction :output :if-exists :error)
   (princ "0123456789" s)
   (truename s))
→ #P"A:>Joe>decimal-digits.text.1"
 (with-open-file (s "decimal-digits.text")
```

```
   (file-length s))
→ 10
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *stream* is not a *stream associated with a file*.

See Also:

open

file-position *Function*

Syntax:

file-position *stream* → *position*

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

Arguments and Values:

stream—a *stream*.

position-spec—a *file position designator*.

position—a *file position* or **nil**.

boolean—a *boolean*.

Description:

Returns or changes the current position within a *stream*.

When *position-spec* is not supplied, **file-position** returns the current *file position* in the *stream*, or **nil** if this cannot be determined.

When *position-spec* is supplied, the *file position* in *stream* is set to that *file position* (if possible). **file-position** returns *true* if the repositioning is performed successfully, or *false* if it is not.

An *integer* returned by **file-position** of one argument should be acceptable as *position-spec* for use with the same file.

For a character file, performing a single **read-char** or **write-char** operation may cause the file position to be increased by more than 1 because of character-set translations (such as translating between the Common Lisp `#\Newline` character and an external ASCII carriage-return/line-feed sequence) and other aspects of the implementation. For a binary file, every **read-byte** or **write-byte** operation increases the file position by 1.

file-position

Examples:

```
 (defun tester ()
   (let ((noticed '()) file-written)
     (flet ((notice (x) (push x noticed) x))
       (with-open-file (s "test.bin"
                          :element-type '(unsigned-byte 8)
                          :direction :output
                          :if-exists :error)
          (notice (file-position s)) ;1
          (write-byte 5 s)
          (write-byte 6 s)
          (let ((p (file-position s)))
            (notice p) ;2
            (notice (when p (file-position s (1- p))))) ;3
          (write-byte 7 s)
          (notice (file-position s)) ;4
  (setq file-written (truename s)))
        (with-open-file (s file-written
           :element-type '(unsigned-byte 8)
           :direction :input)
          (notice (file-position s)) ;5
          (let ((length (file-length s)))
            (notice length) ;6
            (when length
      (dotimes (i length)
                (notice (read-byte s)))))) ;7,...
        (nreverse noticed))))
→ tester
 (tester)
→ (0 2 T 2 0 2 5 7)
or→ (0 2 NIL 3 0 3 5 6 7)
or→ (NIL NIL NIL NIL NIL NIL)
```

Side Effects:

When the *position-spec* argument is supplied, the *file position* in the *stream* might be moved.

Affected By:

The value returned by **file-position** increases monotonically as input or output operations are performed.

Exceptional Situations:

If *position-spec* is supplied, but is too large or otherwise inappropriate, an error is signaled.

See Also:

file-length, **file-string-length**, **open**

Notes:

Implementations that have character files represented as a sequence of records of bounded size might choose to encode the file position as, for example, ⟨⟨*record-number*⟩⟩*⟨⟨*max-record-size*⟩⟩+⟨⟨*character-within-record*⟩⟩. This is a valid encoding because it increases monotonically as each character is read or written, though not necessarily by 1 at each step. An *integer* might then be considered "inappropriate" as *position-spec* to **file-position** if, when decoded into record number and character number, it turned out that the supplied record was too short for the specified character number.

file-string-length *Function*

Syntax:

file-string-length *stream object* → *length*

Arguments and Values:

stream—an *output character file stream*.

object—a *string* or a *character*.

length—a non-negative *integer*, or **nil**.

Description:

file-string-length returns the difference between what `(file-position` *stream*`)` would be after writing *object* and its current value, or **nil** if this cannot be determined.

The returned value corresponds to the current state of *stream* at the time of the call and might not be the same if it is called again when the state of the *stream* has changed.

open *Function*

Syntax:

open *filespec* &key *direction element-type*
if-exists if-does-not-exist external-format

→ *stream*

Arguments and Values:

filespec—a *pathname designator*.

direction—one of :input, :output, :io, or :probe. The default is :input.

element-type—a *type specifier* for *recognizable subtype* of **character**; or a *type specifier* for a *finite recognizable subtype* of *integer*; or one of the *symbols* **signed-byte**, **unsigned-byte**, or :default. The default is **character**.

if-exists—one of :error, :new-version, :rename, :rename-and-delete, :overwrite, :append, :supersede, or **nil**.

if-does-not-exist—one of :error, :create, or **nil**.

external-format—an *external file format designator*. The default is :default.

stream—a *file stream* or **nil**.

Description:

open creates, opens, and returns a *file stream* that is connected to the file specified by *filespec*. *Filespec* is the name of the file to be opened. If the *filespec designator* is a *stream*, that *stream* is not closed first or otherwise affected.

The keyword arguments to **open** specify the characteristics of the *file stream* that is returned, and how to handle errors.

If *direction* is :input or :probe, or if *if-exists* is not :new-version and the version component of the *filespec* is :newest, then the file opened is that file already existing in the file system that has a version greater than that of any other file in the file system whose other pathname components are the same as those of *filespec*.

An implementation is required to recognize all of the **open** keyword options and to do something reasonable in the context of the host operating system. For example, if a file system does not support distinct file versions and does not distinguish the notions of deletion and expunging, :new-version might be treated the same as :rename or :supersede, and :rename-and-delete might be treated the same as :supersede.

:direction

open

Figure 21–9 lists the possible values for *direction* options and their meanings.

Direction	**Result with respect to the created *stream***
`:input`	Creates an *input file stream*.
`:output`	Creates an *output file stream*.
`:io`	Creates a *bidirectional file stream*.
`:probe`	Creates a no-directional *file stream*; the *file stream* is in effect created and then closed.

Figure 21–9. Options for open—1

:element-type

element-type specifies the unit of transaction for the *file stream*. If it is `:default`, the unit is determined by *file system*, possibly based on the *file*.

:if-exists

if-exists specifies the action to be taken if *direction* is `:output` or `:io` and a file of the name *filespec* already exists. If *direction* is `:input`, not supplied, or `:probe`, *if-exists* is ignored. Figure 21–10 gives the result of **open** as modified by *if-exists*.

open

If-exists	**Result with respect to file named** *filespec*
`:error` or not supplied when the version component of *filespec* is not `:newest`	An error of *type* **file-error** is signaled.
`:new-version` or not supplied when the version component of *filespec* is `:newest`	A new file is created with a larger version number.
`:rename`	The existing file is renamed to some other name and then a new file is created.
`:rename-and-delete`	The existing file is renamed to some other name, then it is deleted but not expunged, and then a new file is created.
`:overwrite`	Output operations on the *stream* destructively modify the existing file. If *direction* is `:io` the file is opened in a bidirectional mode that allows both reading and writing. The file pointer is initially positioned at the beginning of the file; however, the file is not truncated back to length zero when it is opened.
`:append`	Output operations on the *stream* destructively modify the existing file. The file pointer is initially positioned at the end of the file. If *direction* is `:io`, the file is opened in a bidirectional mode that allows both reading and writing.
`:supersede`	The existing file is superseded, *i.e.*, a new file with the same name as the old one is created. If possible, the implementation should not destroy the old file until the new *stream* is closed.
nil	No file or *stream* is created. **nil** is returned to indicate failure.

Figure 21–10. Options for open—3

open

:if-does-not-exist

if-does-not-exist specifies the action to be taken if a file of name *filespec* does not already exist. Figure 21–11 gives the result of **open** as modified by *if-does-not-exist*.

If-does-not-exist	Result with respect to file named *filespec*
`:error` or not supplied and *direction* is `:input`, or if *if-exists* is `:overwrite` or `:append`.	An error of *type* **file-error** is signaled.
`:create` or not supplied when *direction* is `:output` or `:io`, and *if-exists* is anything but `:overwrite` or `:append`.	An empty file is created. Processing continues as if the file had already existed but no processing as directed by *if-exists* is performed.
nil or not supplied when *direction* is `:probe`.	No file or *stream* is created. **nil** is returned to indicate failure.

Figure 21–11. Options for open—4

`:external-format`

This option selects an *external file format* for the *file*: The only *standardized* value for this option is `:default`, although *implementations* are permitted to define additional *external file formats* and *implementation-dependent* values returned by **stream-external-format** can also be used by *conforming programs*.

The *external-format* is meaningful for any kind of *file stream* whose *element type* is a *subtype* of *character*. This option is ignored for *streams* for which it is not meaningful; however, *implementations* may define other *element types* for which it is meaningful. The consequences are unspecified if a *character* is written that cannot be represented by the given *external file format*.

When a file is opened, a *file stream* is constructed to serve as the file system's ambassador to the Lisp environment; operations on the *file stream* are reflected by operations on the file in the file system.

A file can be deleted, renamed, or destructively modified by **open**.

Examples:

```
 (open filespec :direction :probe)  → #<Closed Probe File Stream...>
 (setq q (merge-pathnames (user-homedir-pathname) "test"))
→ #<PATHNAME :HOST NIL :DEVICE device-name :DIRECTORY directory-name
```

```
   :NAME "test" :TYPE NIL :VERSION :NEWEST>
(open filespec :if-does-not-exist :create) → #<Input File Stream...>
(setq s (open filespec :direction :probe)) → #<Closed Probe File Stream...>
(truename s) → #<PATHNAME :HOST NIL :DEVICE device-name :DIRECTORY
   directory-name :NAME filespec :TYPE extension :VERSION 1>
(open s :direction :output :if-exists nil) → NIL
```

Affected By:

The nature and state of the host computer's *file system*.

Exceptional Situations:

If *if-exists* is `:error`, (subject to the constraints on the meaning of *if-exists* listed above), an error of *type* **file-error** is signaled.

If *if-does-not-exist* is `:error` (subject to the constraints on the meaning of *if-does-not-exist* listed above), an error of *type* **file-error** is signaled.

If it is impossible for an implementation to handle some option in a manner close to what is specified here, an error of *type* **error** might be signaled.

An error of *type* **file-error** is signaled if `(wild-pathname-p` *filespec*`)` returns true.

An error of *type* **error** is signaled if the *external-format* is not understood by the *implementation*.

The various *file systems* in existence today have widely differing capabilities, and some aspects of the *file system* are beyond the scope of this specification to define. A given *implementation* might not be able to support all of these options in exactly the manner stated. An *implementation* is required to recognize all of these option keywords and to try to do something "reasonable" in the context of the host *file system*. Where necessary to accomodate the *file system*, an *implementation* deviate slightly from the semantics specified here without being disqualified for consideration as a *conforming implementation*. If it is utterly impossible for an *implementation* to handle some option in a manner similar to what is specified here, it may simply signal an error.

With regard to the `:element-type` option, if a *type* is requested that is not supported by the *file system*, a substitution of types such as that which goes on in *upgrading* is permissible. As a minimum requirement, it should be the case that opening an *output stream* to a *file* in a given *element type* and later opening an *input stream* to the same *file* in the same *element type* should work compatibly.

See Also:

with-open-file, **close**, **pathname**, **logical-pathname**

Notes:

Whether or not **open** recognizes *logical pathname namestrings* is *implementation-defined*.

open does not automatically close the file when an abnormal exit occurs.

When *element-type* is a *subtype* of **character**, **read-char** and/or **write-char** can be used on the resulting *file stream*.

When *element-type* is a *subtype* of *integer*, **read-byte** and/or **write-byte** can be used on the resulting *file stream*.

When *element-type* is `:default`, the *type* can be determined by using **stream-element-type**.

stream-external-format *Function*

Syntax:

stream-external-format *stream* → *format*

Arguments and Values:

stream—a *file stream*.

format—an *external file format*.

Description:

Returns an *external file format designator* for the *stream*.

Examples:

```
 (with-open-file (stream "test" :direction :output)
   (stream-external-format stream))
→ :DEFAULT
→(or) :ISO8859/1-1987
→(or) (:ASCII :SAIL)
→(or) ACME::PROPRIETARY-FILE-FORMAT-17
→(or) #<FILE-FORMAT :ISO646-1983 2343673>
```

See Also:

the `:external-format` *argument* to the *function* **open** and the **with-open-file** *macro*.

Notes:

The *format* returned is not necessarily meaningful to other *implementations*.

with-open-file *macro*

Syntax:

with-open-file (*stream filespec* {*options*}*) {*declaration*}* {*form*}*
→ *results*

Arguments and Values:

stream – a variable.

filespec—a *pathname designator*.

options – *forms*; evaluated.

declaration—a **declare** *expression*; not evaluated.

forms—an *implicit progn*.

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

Description:

with-open-file uses **open** to create a *file stream* to *file* named by *filespec*. *Filespec* is the name of the file to be opened. *Options* are used as keyword arguments to **open**.

The *stream object* to which the *stream variable* is *bound* has *dynamic extent*; its *extent* ends when the *form* is exited.

with-open-file evaluates the *forms* as an *implicit progn* with *stream* bound to the value returned by **open**.

When control leaves the body, either normally or abnormally (such as by use of **throw**), the file is automatically closed. If a new output file is being written, and control leaves abnormally, the file is aborted and the file system is left, so far as possible, as if the file had never been opened.

It is possible by the use of `:if-exists nil` or `:if-does-not-exist nil` for *stream* to be bound to **nil**. Users of `:if-does-not-exist nil` should check for a valid *stream*.

The consequences are undefined if an attempt is made to *assign* the *stream variable*. The compiler may choose to issue a warning if such an attempt is detected.

Examples:

```
 (setq p (merge-pathnames "test"))
→ #<PATHNAME :HOST NIL :DEVICE device-name :DIRECTORY directory-name
    :NAME "test" :TYPE NIL :VERSION :NEWEST>
 (with-open-file (s p :direction :output :if-exists :supersede)
    (format s "Here are a couple~%of test data lines~%")) → NIL
 (with-open-file (s p)
```

```
    (do ((l (read-line s) (read-line s nil 'eof)))
        ((eq l 'eof) "Reached end of file.")
     (format t "~&*** ~A~%" l)))
▷ *** Here are a couple
▷ *** of test data lines
→ "Reached end of file."

;; Normally one would not do this intentionally because it is
;; not perspicuous, but beware when using :IF-DOES-NOT-EXIST NIL
;; that this doesn't happen to you accidentally...
 (with-open-file (foo "no-such-file" :if-does-not-exist nil)
   (read foo))
▷ hello?
→ HELLO? ;This value was read from the terminal, not a file!

;; Here's another bug to avoid...
 (with-open-file (foo "no-such-file" :direction :output :if-does-not-exist nil)
   (format foo "Hello"))
→ "Hello" ;FORMAT got an argument of NIL!
```

Side Effects:

Creates a *stream* to the *file* named by *filename* (upon entry), and closes the *stream* (upon exit). In some *implementations*, the *file* might be locked in some way while it is open. If the *stream* is an *output stream*, a *file* might be created.

Affected By:

The host computer's file system.

Exceptional Situations:

An error of *type* **file-error** is signaled if `(wild-pathname-p` *filespec*`)` returns true.

See Also:

open, **close**, **pathname**, **logical-pathname**

Notes:

Whether or not **with-open-file** recognizes *logical pathname namestrings* is *implementation-defined*.

close *Function*

Syntax:

close *stream* &key *abort* → *result*

Arguments and Values:

stream—a *stream* (either *open* or *closed*).

abort—a *boolean*. The default is *false*.

result—**t** if the *stream* was *open* at the time it was received as an *argument*, or *implementation-dependent* otherwise.

Description:

close closes *stream*. Closing a *stream* means that it may no longer be used in input or output operations. The act of *closing* a *file stream* ends the association between the *stream* and its associated *file*; the transaction with the *file system* is terminated, and input/output may no longer be performed on the *stream*.

If *abort* is *true*, an attempt is made to clean up any side effects of having created *stream*. If *stream* performs output to a file that was created when the *stream* was created, the file is deleted and any previously existing file is not superseded.

It is permissible to close an already closed *stream*, but in that case the *result* is *implementation-dependent*.

After *stream* is closed, it is still possible to perform the following query operations upon it: **streamp**, **pathname**, **truename**, **merge-pathnames**, **pathname-host**, **pathname-device**, **pathname-directory**,**pathname-name**, **pathname-type**, **pathname-version**, **namestring**, **file-namestring**, **directory-namestring**, **host-namestring**, **enough-namestring**, **open**, **probe-file**, and **directory**.

The effect of **close** on a *constructed stream* is to close the argument *stream* only. There is no effect on the *constituents* of *composite streams*.

For a *stream* created with **make-string-output-stream**, the result of **get-output-stream-string** is unspecified after **close**.

Examples:

```
(setq s (make-broadcast-stream)) → #<BROADCAST-STREAM>
(close s) → T
(output-stream-p s) → true
```

Side Effects:

The *stream* is *closed* (if necessary). If *abort* is *true* and the *stream* is an *output file stream*, its associated *file* might be deleted.

See Also:

open

with-open-stream *Macro*

Syntax:

with-open-stream (*var stream*) {*declaration*}* {*form*}*
→ {*result*}*

Arguments and Values:

var—a *variable name*.

stream—a *form*; evaluated to produce a *stream*.

declaration—a **declare** *expression*; not evaluated.

forms—an *implicit progn*.

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

Description:

with-open-stream performs a series of operations on *stream*, returns a value, and then closes the *stream*.

Var is bound to the value of *stream*, and then *forms* are executed as an *implicit progn*. *stream* is automatically closed on exit from **with-open-stream**, no matter whether the exit is normal or abnormal. The *stream* has *dynamic extent*; its *extent* ends when the *form* is exited.

The consequences are undefined if an attempt is made to *assign* the the *variable var* with the *forms*.

Examples:

```
(with-open-stream (s (make-string-input-stream "1 2 3 4"))
   (+ (read s) (read s) (read s))) → 6
```

Side Effects:

The *stream* is closed (upon exit).

See Also:

close

listen *Function*

Syntax:

listen &optional *input-stream* → *boolean*

Arguments and Values:

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

boolean—a *boolean*.

Description:

Returns *true* if there is a character immediately available from *input-stream*; otherwise, returns *false*. On a non-interactive *input-stream*, **listen** returns *true* except when at *end of file*$_1$. If an *end of file* is encountered, **listen** returns *false*. **listen** is intended to be used when *input-stream* obtains characters from an interactive device such as a keyboard.

Examples:

```
 (progn (unread-char (read-char)) (list (listen) (read-char)))
▷ 1
→ (T #\1)
 (progn (clear-input) (listen))
→ NIL ;Unless you're a very fast typist!
```

Affected By:

standard-input

See Also:

interactive-stream-p, **read-char-no-hang**

clear-input

clear-input *Function*

Syntax:

clear-input &optional *input-stream* → **nil**

Arguments and Values:

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

Description:

Clears any available input from *input-stream*.

If **clear-input** does not make sense for *input-stream*, then **clear-input** does nothing.

Examples:

```
;; The exact I/O behavior of this example might vary from implementation
;; to implementation depending on the kind of interactive buffering that
;; occurs.  (The call to SLEEP here is intended to help even out the
;; differences in implementations which do not do line-at-a-time buffering.)

(defun read-sleepily (&optional (clear-p nil) (zzz 0))
  (list (progn (print '>) (read))
        ;; Note that input typed within the first ZZZ seconds
        ;; will be discarded.
        (progn (print '>)
               (if zzz (sleep zzz))
               (print '>>)
               (if clear-p (clear-input))
               (read))))

(read-sleepily)
▷ > 10
▷ >
▷ >> 20
→ (10 20)

(read-sleepily t)
▷ > 10
▷ >
▷ >> 20
→ (10 20)

(read-sleepily t 10)
▷ > 10
▷ > 20  ; Some implementations won't echo typeahead here.
```

```
▷ >> 30
→ (10 30)
```

Side Effects:

The *input-stream* is modified.

Affected By:

standard-input

Exceptional Situations:

Should signal an error of *type* **type-error** if *input-stream* is not a *stream designator*.

See Also:

clear-output

finish-output, force-output, clear-output *Function*

Syntax:

finish-output &optional *output-stream* → **nil**

force-output &optional *output-stream* → **nil**

clear-output &optional *output-stream* → **nil**

Arguments and Values:

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

Description:

finish-output, **force-output**, and **clear-output** exercise control over the internal handling of buffered stream output.

finish-output attempts to ensure that any buffered output sent to *output-stream* has reached its destination, and then returns.

force-output initiates the emptying of any internal buffers but does not wait for completion or acknowledgment to return.

clear-output attempts to abort any outstanding output operation in progress in order to allow as little output as possible to continue to the destination.

If any of these operations does not make sense for *output-stream*, then it does nothing. The precise actions of these *functions* are *implementation-dependent*.

Examples:

```
;; Implementation A
 (progn (princ "am i seen?") (clear-output))
→ NIL

;; Implementation B
 (progn (princ "am i seen?") (clear-output))
▷ am i seen?
→ NIL
```

Affected By:

standard-output

Exceptional Situations:

Should signal an error of *type* **type-error** if *output-stream* is not a *stream designator*.

See Also:

clear-input

y-or-n-p, yes-or-no-p *Function*

Syntax:

y-or-n-p &optional *control* &rest *arguments* → *boolean*

yes-or-no-p &optional *control* &rest *arguments* → *boolean*

Arguments and Values:

control—a *format control*.

arguments—*format arguments* for *control*.

boolean—a *boolean*.

Description:

These functions ask a question and parse a response from the user. They return *true* if the answer is affirmative, or *false* if the answer is negative.

y-or-n-p is for asking the user a question whose answer is either "yes" or "no." It is intended that the reply require the user to answer a yes-or-no question with a single character. **yes-or-no-p** is also for asking the user a question whose answer is either "Yes" or "No." It is intended that the reply require the user to take more action than just a single keystroke, such as typing the full word `yes` or `no` followed by a newline.

y-or-n-p types out a message (if supplied), reads an answer in some *implementation-dependent* manner (intended to be short and simple, such as reading a single character such as `Y` or `N`). **yes-or-no-p** types out a message (if supplied), attracts the user's attention (for example, by ringing the terminal's bell), and reads an answer in some *implementation-dependent* manner (intended to be multiple characters, such as `YES` or `NO`).

If *format-control* is supplied and not **nil**, then a **fresh-line** operation is performed; then a message is printed as if *format-control* and *arguments* were given to **format**. In any case, **yes-or-no-p** and **y-or-n-p** will provide a prompt such as "`(Y or N)`" or "`(Yes or No)`" if appropriate.

All input and output are performed using *query I/O*.

Examples:

```
 (y-or-n-p "(t or nil) given by")
▷ (t or nil) given by (Y or N) Y
→ true
 (yes-or-no-p "a ~S message" 'frightening)
▷ a FRIGHTENING message (Yes or No) no
→ false
 (y-or-n-p "Produce listing file?")
▷ Produce listing file?
▷ Please respond with Y or N. n
→ false
```

Side Effects:

Output to and input from *query I/O* will occur.

Affected By:

query-io.

See Also:

format

Notes:

yes-or-no-p and **yes-or-no-p** do not add question marks to the end of the prompt string, so any desired question mark or other punctuation should be explicitly included in the text query.

make-synonym-stream *Function*

Syntax:

make-synonym-stream *symbol* → *synonym-stream*

Arguments and Values:

symbol—a *symbol* that names a *dynamic variable*.

synonym-stream—a *synonym stream*.

Description:

Returns a *synonym stream* whose *synonym stream symbol* is *symbol*.

Examples:

```
 (setq a-stream (make-string-input-stream "a-stream")
        b-stream (make-string-input-stream "b-stream"))
→ #<String Input Stream>
 (setq s-stream (make-synonym-stream 'c-stream))
→ #<SYNONYM-STREAM for C-STREAM>
 (setq c-stream a-stream)
→ #<String Input Stream>
 (read s-stream) → A-STREAM
 (setq c-stream b-stream)
→ #<String Input Stream>
 (read s-stream) → B-STREAM
```

Exceptional Situations:

Should signal **type-error** if its argument is not a *symbol*.

See Also:

Section 21.1 (Stream Concepts)

synonym-stream-symbol *Function*

Syntax:

synonym-stream-symbol *synonym-stream* → *symbol*

Arguments and Values:

synonym-stream—a *synonym stream*.

symbol—a *symbol*.

Description:

Returns the *symbol* whose **symbol-value** the *synonym-stream* is using.

See Also:

make-synonym-stream

broadcast-stream-streams *Function*

Syntax:

broadcast-stream-streams *broadcast-stream* → *streams*

Arguments and Values:

broadcast-stream—a *broadcast stream*.

streams—a *list* of *streams*.

Description:

Returns a *list* of output *streams* that constitute all the *streams* to which the *broadcast-stream* is broadcasting.

make-broadcast-stream *Function*

Syntax:

make-broadcast-stream `&rest` *streams* → *broadcast-stream*

Arguments and Values:

stream—an *output stream*.

broadcast-stream—a *broadcast stream*.

Description:

Returns a *broadcast stream*.

Examples:

```
(setq a-stream (make-string-output-stream)
       b-stream (make-string-output-stream)) → #<String Output Stream>
(format (make-broadcast-stream a-stream b-stream)
         "this will go to both streams") → NIL
(get-output-stream-string a-stream) → "this will go to both streams"
(get-output-stream-string b-stream) → "this will go to both streams"
```

Exceptional Situations:

Should signal an error of *type* **type-error** if any *stream* is not an *output stream*.

See Also:

broadcast-stream-streams

make-two-way-stream *Function*

Syntax:

make-two-way-stream *input-stream output-stream* → *two-way-stream*

Arguments and Values:

input-stream—a *stream*.

output-stream—a *stream*.

two-way-stream—a *two-way stream*.

Description:

Returns a *two-way stream* that gets its input from *input-stream* and sends its output to *output-stream*.

Examples:

```
(with-output-to-string (out)
   (with-input-from-string (in "input...")
     (let ((two (make-two-way-stream in out)))
       (format two "output...")
       (setq what-is-read (read two))))) → "output..."
what-is-read → INPUT...
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *input-stream* is not an *input stream*. Should signal an error of *type* **type-error** if *output-stream* is not an *output stream*.

two-way-stream-input-stream, two-way-stream-output-stream *Function*

Syntax:

two-way-stream-input-stream *two-way-stream* → *input-stream*

two-way-stream-output-stream *two-way-stream* → *output-stream*

Arguments and Values:

two-way-stream—a *two-way stream*.

input-stream—an *input stream*.

output-stream—an *output stream*.

Description:

two-way-stream-input-stream returns the *stream* from which *two-way-stream* receives input.

two-way-stream-output-stream returns the *stream* to which *two-way-stream* sends output.

echo-stream-input-stream, echo-stream-output-stream *Function*

Syntax:

echo-stream-input-stream *echo-stream* → *input-stream*

echo-stream-output-stream *echo-stream* → *output-stream*

Arguments and Values:

echo-stream—an *echo stream*.

input-stream—an *input stream*.

output-stream—an *output stream*.

Description:

echo-stream-input-stream returns the *input stream* from which *echo-stream* receives input.

echo-stream-output-stream returns the *output stream* to which *echo-stream* sends output.

make-echo-stream *Function*

Syntax:

make-echo-stream *input-stream output-stream* → *echo-stream*

Arguments and Values:

input-stream—an *input stream*.

output-stream—an *output stream*.

echo-stream—an *echo stream*.

Description:

Creates and returns an *echo stream* that takes input from *input-stream* and sends output to *output-stream*.

Examples:

```
(let ((out (make-string-output-stream)))
   (with-open-stream
       (s (make-echo-stream
           (make-string-input-stream "this-is-read-and-echoed")
           out))
     (read s)
     (format s " * this-is-direct-output")
     (get-output-stream-string out)))
→ "this-is-read-and-echoed * this-is-direct-output"
```

See Also:

echo-stream-input-stream, **echo-stream-output-stream**, **make-two-way-stream**

concatenated-stream-streams *Function*

Syntax:

concatenated-stream-streams *concatenated-stream* → *streams*

Arguments and Values:

concatenated-stream – a *concatenated stream*.

streams—a *list* of *input streams*.

Description:

Returns a *list* of *input streams* that constitute the ordered set of *streams* the *concatenated-stream* still has to read from, starting with the current one it is reading from. The list may be *empty* if no more *streams* remain to be read.

The consequences are undefined if the *list structure* of the *streams* is ever modified.

make-concatenated-stream *Function*

Syntax:

make-concatenated-stream &rest *input-streams* → *concatenated-stream*

Arguments and Values:

input-stream—an *input stream*.

concatenated-stream—a *concatenated stream*.

Description:

Returns a *concatenated stream* that has the indicated *input-streams* initially associated with it.

Examples:

```
(read (make-concatenated-stream
        (make-string-input-stream "1")
        (make-string-input-stream "2"))) → 12
```

Exceptional Situations:

Should signal **type-error** if any argument is not an *input stream*.

See Also:

concatenated-stream-streams

get-output-stream-string *Function*

Syntax:

get-output-stream-string *string-output-stream* → *string*

Arguments and Values:

string-output-stream—a *stream*.

string—a *string*.

Description:

Returns a *string* containing, in order, all the *characters* that have been output to *string-output-stream*. This operation clears any *characters* on *string-output-stream*, so the *string* contains only those *characters* which have been output since the last call to **get-output-stream-string** or since the creation of the *string-output-stream*, whichever occurred most recently.

Examples:

```
(setq a-stream (make-string-output-stream)
      a-string "abcdefghijklm") → "abcdefghijklm"
(write-string a-string a-stream) → "abcdefghijklm"
(get-output-stream-string a-stream) → "abcdefghijklm"
(get-output-stream-string a-stream) → ""
```

Side Effects:

The *string-output-stream* is cleared.

Exceptional Situations:

The consequences are undefined if *stream-output-string* is *closed*.

The consequences are undefined if *string-output-stream* is a *stream* that was not produced by **make-string-output-stream**. The consequences are undefined if *string-output-stream* was created implicitly by **with-output-to-string** or **format**.

See Also:

make-string-output-stream

make-string-input-stream *Function*

Syntax:

make-string-input-stream *string* &optional *start end* → *string-stream*

Arguments and Values:

string—a *string*.

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

string-stream—an *input string stream*.

Description:

Returns an *input string stream*. This *stream* will supply, in order, the *characters* in the substring of *string bounded* by *start* and *end*. After the last *character* has been supplied, the *string stream* will then be at *end of file*.

Examples:

```
 (let ((string-stream (make-string-input-stream "1 one ")))
   (list (read string-stream nil nil)
 (read string-stream nil nil)
 (read string-stream nil nil)))
→ (1 ONE NIL)

 (read (make-string-input-stream "prefixtargetsuffix" 6 12)) → TARGET
```

See Also:

with-input-from-string

make-string-output-stream *Function*

Syntax:

make-string-output-stream &key *element-type* → *string-stream*

Arguments and Values:

element-type—a *type specifier*. The default is **character**.

string-stream—an *output string stream*.

Description:

Returns an *output string stream* that accepts *characters* and makes available (via **get-output-stream-string**) a *string* that contains the *characters* that were actually output.

The *element-type* names the *type* of the *elements* of the *string*; a *string* is constructed of the most specialized *type* that can accommodate *elements* of that *element-type*.

Examples:

```
 (let ((s (make-string-output-stream)))
   (write-string "testing... " s)
   (prin1 1234 s)
   (get-output-stream-string s))
→ "testing... 1234"
```

None..

See Also:

get-output-stream-string, **with-output-to-string**

with-input-from-string *Macro*

Syntax:

with-input-from-string (*var string* `&key` *index start end*) {*declaration*}* {*form*}*
→ {*result*}*

Arguments and Values:

var—a *variable name*.

string—a *form*; evaluated to produce a *string*.

index—a *place*.

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

declaration—a **declare** *expression*; not evaluated.

forms—an *implicit progn*.

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

Description:

Creates an *input string stream*, provides an opportunity to perform operations on the *stream* (returning zero or more *values*), and then closes the *string stream*.

String is evaluated first, and *var* is bound to a character *input string stream* that supplies *characters* from the subsequence of the resulting *string bounded* by *start* and *end*. The body is executed as an *implicit progn*.

The *input string stream* is automatically closed on exit from **with-input-from-string**, no matter whether the exit is normal or abnormal. The *input string stream* to which the *variable var* is *bound* has *dynamic extent*; its *extent* ends when the *form* is exited.

The *index* is a pointer within the *string* to be advanced. If **with-input-from-string** is exited normally, then *index* will have as its *value* the index into the *string* indicating the first character not read which is `(length` *string*`)` if all characters were used. The place specified by *index* is not updated as reading progresses, but only at the end of the operation.

start and *index* may both specify the same variable, which is a pointer within the *string* to be advanced, perhaps repeatedly by some containing loop.

The consequences are undefined if an attempt is made to *assign* the *variable var*.

Examples:

```
(with-input-from-string (s "XXX1 2 3 4xxx"
                            :index ind
                            :start 3 :end 10)
   (+ (read s) (read s) (read s))) → 6
ind → 9
(with-input-from-string (s "Animal Crackers" :index j :start 6)
  (read s)) → CRACKERS
```

The variable `j` is set to `15`.

Side Effects:

The *value* of the *place* named by *index*, if any, is modified.

See Also:

make-string-input-stream, Section 3.6 (Traversal Rules and Side Effects)

with-output-to-string *Macro*

Syntax:

with-output-to-string (*var* `&optional` *string-form* `&key` *element-type*) {*declaration*}* {*form*}*
→ {*result*}*

Arguments and Values:

var—a *variable name*.

with-output-to-string

string-form—a *form* or **nil**; if *non-nil*, evaluated to produce *string*.

string—a *string* that has a *fill pointer*.

element-type—a *type specifier*; evaluated. The default is **character**.

declaration—a **declare** *expression*; not evaluated.

forms—an *implicit progn*.

results—If a *string-form* is not supplied or **nil**, a *string*; otherwise, the *values* returned by the *forms*.

Description:

with-output-to-string creates a character *output stream*, performs a series of operations that may send results to this *stream*, and then closes the *stream*.

The *element-type* names the *type* of the elements of the *stream*; a *stream* is constructed of the most specialized *type* that can accommodate elements of the given *type*.

The body is executed as an *implicit progn* with *var* bound to an *output string stream*. All output to that *string stream* is saved in a *string*.

If *string* is supplied, *element-type* is ignored, and the output is incrementally appended to *string* as if by use of **vector-push-extend**.

The *output stream* is automatically closed on exit from **with-output-from-string**, no matter whether the exit is normal or abnormal. The *output string stream* to which the *variable var* is *bound* has *dynamic extent*; its *extent* ends when the *form* is exited.

If no *string* is provided, then **with-output-from-string** produces a *stream* that accepts characters and returns a *string* of the indicated *element-type*. If *string* is provided, **with-output-to-string** returns the results of evaluating the last *form*.

The consequences are undefined if an attempt is made to *assign* the *variable var*.

Examples:

```
(setq fstr (make-array '(0) :element-type 'base-char
                             :fill-pointer 0 :adjustable t)) → ""
(with-output-to-string (s fstr)
   (format s "here's some output")
   (input-stream-p s)) → false
fstr → "here's some output"
```

Side Effects:

The *string* is modified.

Exceptional Situations:

The consequences are undefined if destructive modifications are performed directly on the *string* during the *dynamic extent* of the call.

See Also:

make-string-output-stream, **vector-push-extend**, Section 3.6 (Traversal Rules and Side Effects)

debug-io, *error-output*, *query-io*, *standard-input*, *standard-output*, *trace-output* *Variable*

Value Type:

For ***standard-input***: an *input stream*

For ***error-output***, ***standard-output***, and ***trace-output***: an *output stream*.

For ***debug-io***, ***query-io***: a *bidirectional stream*.

Initial Value:

implementation-dependent, but it must be an *open stream* that is not a *generalized synonym stream* to an *I/O customization variables* but that might be a *generalized synonym stream* to the value of some *I/O customization variable*. The initial value might also be a *generalized synonym stream* to either the *symbol* ***terminal-io*** or to the *stream* that is its *value*.

Description:

These *variables* are collectively called the *standardized I/O customization variables*. They can be *bound* or *assigned* in order to change the default destinations for input and/or output used by various *standardized operators* and facilities.

The *value* of ***debug-io***, called *debug I/O*, is a *stream* to be used for interactive debugging purposes.

The *value* of ***error-output***, called *error output*, is a *stream* to which warnings and non-interactive error messages should be sent.

The *value* of ***query-io***, called *query I/O*, is a *bidirectional stream* to be used when asking questions of the user. The question should be output to this *stream*, and the answer read from it.

The *value* of ***standard-input***, called *standard input*, is a *stream* that is used by many *operators* as a default source of input when no specific *input stream* is explicitly supplied.

The *value* of ***standard-output***, called *standard output*, is a *stream* that is used by many *operators* as a default destination for output when no specific *output stream* is explicitly supplied.

∗debug-io∗, ∗error-output∗, ∗query-io∗, ...

The *value* of ***trace-output***, called *trace output*, is the *stream* on which traced functions (see **trace**) and the **time** *macro* print their output.

Examples:

```
 (with-output-to-string (*error-output*)
   (warn "this string is sent to *error-output*"))
 → "Warning: this string is sent to *error-output*
" ;The exact format of this string is implementation-dependent.



 (with-input-from-string (*standard-input* "1001")
    (+ 990 (read))) → 1991



 (progn (setq out (with-output-to-string (*standard-output*)
                     (print "print and format t send things to")
                     (format t "*standard-output* now going to a string")))
        :done)
→ :DONE
 out
→ "
\"print and format t send things to\" *standard-output* now going to a string"



 (defun fact (n) (if (< n 2) 1 (* n (fact (- n 1)))))
→ FACT
 (trace fact)
→ (FACT)
;; Of course, the format of traced output is implementation-dependent.
 (with-output-to-string (*trace-output*)
   (fact 3))
→ "
1 Enter FACT 3
| 2 Enter FACT 2
|   3 Enter FACT 1
|   3 Exit FACT 1
| 2 Exit FACT 2
1 Exit FACT 6"
```

See Also:

terminal-io, **synonym-stream**, **time**, **trace**, Chapter 9 (Conditions), Chapter 23 (Reader), Chapter 22 (Printer)

Notes:

The intent of the constraints on the initial *value* of the *I/O customization variables* is to ensure that it is always safe to *bind* or *assign* such a *variable* to the *value* of another *I/O customization variable*, without unduly restricting *implementation* flexibility.

It is common for an *implementation* to make the initial *values* of ***debug-io*** and ***query-io*** be the *same stream*, and to make the initial *values* of ***error-output*** and ***standard-output*** be the *same stream*.

The functions **y-or-n-p** and **yes-or-no-p** use *query I/O* for their input and output.

In the normal *Lisp read-eval-print loop*, input is read from *standard input*. Many input functions, including **read** and **read-char**, take a *stream* argument that defaults to *standard input*.

In the normal *Lisp read-eval-print loop*, output is sent to *standard output*. Many output functions, including **print** and **write-char**, take a *stream* argument that defaults to *standard output*.

A program that wants, for example, to divert output to a file should do so by *binding* ***standard-output***; that way error messages sent to ***error-output*** can still get to the user by going through ***terminal-io*** (if ***error-output*** is bound to ***terminal-io***), which is usually what is desired.

terminal-io *Variable*

Value Type:

a *bidirectional stream*.

Initial Value:

implementation-dependent, but it must be an *open stream* that is not a *generalized synonym stream* to an *I/O customization variables* but that might be a *generalized synonym stream* to the *value* of some *I/O customization variable*.

Description:

The *value* of ***terminal-io***, called *terminal I/O*, is ordinarily a *bidirectional stream* that connects to the user's console. Typically, writing to this *stream* would cause the output to appear on a display screen, for example, and reading from the *stream* would accept input from a keyboard. It is intended that standard input functions such as **read** and **read-char**, when used with this *stream*, cause echoing of the input into the output side of the *stream*. The means by which this is accomplished are *implementation-dependent*.

The effect of changing the *value* of ***terminal-io***, either by *binding* or *assignment*, is *implementation-defined*.

Examples:

```
 (progn (prin1 'foo) (prin1 'bar *terminal-io*))
▷ FOOBAR
→ BAR
 (with-output-to-string (*standard-output*)
   (prin1 'foo)
   (prin1 'bar *terminal-io*))
▷ BAR
→ "FOO"
```

See Also:

debug-io, ***error-output***, ***query-io***, ***standard-input***, ***standard-output***, ***trace-output***

stream-error *Condition Type*

Class Precedence List:

stream-error, **error**, **serious-condition**, **condition**, **t**

Description:

The *type* **stream-error** consists of error conditions that are related to receiving input from or sending output to a *stream*. The "offending stream" is initialized by the `:stream` initialization argument to **make-condition**, and is *accessed* by the *function* **stream-error-stream**.

See Also:

stream-error-stream

stream-error-stream *Function*

Syntax:

stream-error-stream *condition* → *stream*

Arguments and Values:

condition—a *condition* of *type* **stream-error**.

stream—a *stream*.

Description:

Returns the offending *stream* of a *condition* of *type* **stream-error**.

Examples:

```
(with-input-from-string (s "(FOO")
  (handler-case (read s)
    (end-of-file (c)
      (format nil "~&End of file on ~S." (stream-error-stream c)))))
"End of file on #<String Stream>."
```

See Also:

stream-error, Chapter 9 (Conditions)

end-of-file *Condition Type*

Class Precedence List:

end-of-file, **stream-error**, **error**, **serious-condition**, **condition**, **t**

Description:

The *type* **end-of-file** consists of error conditions related to read operations that are done on *streams* that have no more data.

See Also:

stream-error-stream

Table of Contents