22. The I/O System

Zetalisp provides a powerful and flexible system for performing input and output to peripheral devices. Device independent I/O is generalized in the concept of an *I/O stream*. A stream is a source or sink for data in the form of characters or integers; sources are called *input streams* and sinks are called *output streams*. A stream may be capable of use in either direction, in which case it is a *bidirectional* stream. In a few unusual cases, it is useful to have a 'stream' which supports neither input nor output; for example, opening a file with direction :probe returns one (page 583). Streams on which characters are transferred are called *character streams*, and are used more often than *binary streams*, which usually transfer integers of type (unsigned-byte *n*) for some *n*.

Streams automatically provide a modular separation between the program which implements the stream and the program which uses it, because streams obey a standard protocol. The stream protocol is a special case is based on the general message passing protocol: a stream operation is invoked by calling the stream as a function, with a first argument that is a keyword and identifies the I/O operation desired (such as, :tyi to read a character) and additional arguments as that operation calls for them. The stream protocol consists of a particular set of operation names and calling conventions for them. It is documented in section 22.3, page 459.

Many programs do not invoke the stream operations directly; instead, they call standard I/O functions which then invoke stream operations. This is done for two reasons: the functions may provide useful services. and they may be transportable to Common Lisp or Maclisp. Programs that use stream operations directly are not transportable outside Zetalisp. The I/O functions are documented in the first sections of this chapter.

The generality of the Zetalisp I/O stream comes from the fact that I/O operations on it can invoke arbitrary Lisp code. For example, it would be very simple to implement a "morse code" stream that accepted character output and used beep with appropriate pauses to 'display' it. How to implement a stream is documented in section 22.3.12, page 474, and the following sections.

The most commonly used streams are windows, which read input from the keyboard and dispose of output by drawing on the screen, file streams, editor buffer streams which get input from the text in a buffer and insert output into the buffer, and string streams which do likewise with the contents of a string.

Another unusual aspect of Lisp I/O is the ability to input and output general Lisp objects, represented as text. These are done using the read and related functions and using print and related functions. They are documented in chapter 23.

22.1 Input Functions

The input functions read characters, lines, or bytes from an input stream. This argument is called *stream*. If omitted or nil, the current value of *standard-input*. This is the "default input stream", which in simple use reads from the terminal keyboard. If the argument is t, the current value of *terminal-io* is used; this is conventionally supposed to access "the user's terminal" and nearly always reads from the keyboard in processes belonging to windows.

If the stream is an interactive one, such as the terminal, the input is echoed, and functions which read more than a single character allow editing as well. peek-char echoes all of the characters that were skipped over if read-char would have echoed them; the character not removed from the stream is not echoed either.

When an input stream has no more data to return, it reports end of file. Each stream input operation has a convention for how to do this. The input functions accept an argument *eof-option* or two arguments *eof-error* and *eof-value* to tell them what to do if end of file is encountered instead of any input. The functions that take two *eof-* arguments are the Common Lisp ones. For them, end of file is an error if *eof-error* is non-nil or if it is unsupplied. If *eof-error* is nil, then the function returns *eof-value* at end of file.

The functions which have one argument called *eof-option* are from Maclisp. End of file causes an error if the argument is not supplied. Otherwise, end of file causes the function to return the argument's value. Note that an *eof-option* of nil means to return nil if the end of the file is reached; it is *not* equivalent to supplying no *eof-option*.

sys:end-of-file (error) *Condition*

All errors signaled to report end of file possess this condition name.

The :stream operation on the condition instance returns the stream on which end of file was reached.

22.1.1 String Input Functions

read-line &optional *stream* (*eof-errorp* t) *eof-value* *ignore* *options*

Reads a line of text, terminated by a Return. It returns the line as a character string, *without* the Return character that ended the line. The argument *ignore* must be accepted for the sake of the Common Lisp specifications but it is not used.

This function is usually used to get a line of input from the user. If rubout processing is happening, then *options* is passed as the list of options to the rubout handler (see section 22.5, page 500).

There is a second value, t if the line was terminated by end of file.

readline &optional *stream eof-option options*
Like read-line but uses the Maclisp convention for specifying what to do about end of file. This function can take its first two arguments in the other order, for Maclisp compatibility only; see the note in section 22.1.3, page 451.

readline-trim &optional *stream eof-option options*
This is like readline except that leading and trailing spaces and tabs are discarded from the value before it is returned.

readline-or-nil &optional *stream eof-option options*
Like readline-trim except that nil is returned if the line is empty or all blank.

read-delimited-string &optional *delimiter stream eof rubout-handler-options buffer-size*
Reads input from *stream* until a delimiter character is reached, then returns as a string all the input up to but not including the delimiter. *delimiter* is either a character or a list of characters which all serve as delimiters. It defaults to the character End. *stream* defaults to the value of *standard-input*.

If *eof* is non-nil, then end of file on attempting to read the first character is an error. Otherwise it just causes an empty string to be returned. End of file once at least one character has been read is never an error but it does cause the function to return all the input so far.

Input is done using rubout handling and echoing if stream supports the :rubout-handler operation. In this case, *rubout-handler-options* are passed as the options argument to that operation.

buffer-size specifies the size of string buffer to allocate initially.

The second value returned is t if input ended due to end of file.

The third value is the delimiter character which terminated input, or nil if input terminated due to end of file. This character is currently represented as a fixnum, but perhaps someday will be a character object instead.

22.1.2 Character-Level Input Functions

read-char &optional *stream* (*eof-errorp* t) *eof-value*
Reads a character from *stream* and returns it as a character object. End of file is an error if *eof-errorp* is non-nil; otherwise, it causes read-char to return *eof-value*. This uses the :tyi stream operation.

read-byte *stream* &optional (*eof-errorp* t) *eof-value*
Like read-char but returns an integer rather than a character object. In strict Common Lisp, only read-char can be used on character streams and only read-byte can be used on binary streams.

read-char-no-hang &optional *stream* (*eof-errorp* **t**) *eof-value*

Similar but returns **nil** immediately when no input is available on an interactive stream. Uses the **:tyi-no-hang** stream operation (page 466).

unread-char *char* &optional *stream*

Puts *char* back into *stream* so that it will be read again as the next input character. *char* must be the same character that was read from stream most recently. It may not work to unread two characters in a row before reading again. Uses the **:untyi** stream operation (page 461).

peek-char *peek-type* &optional *stream* (*eof-errorp* **t**) *eof-value*

If *peek-type* is **nil**, this is like **read-char** except leaves the character to be read again by the next input operation.

If *peek-type* is **t**, skips whitespace characters and peeks at the first nonwhitespace character. That character is the value, and is also left to be reread.

If *peek-type* is a character, reads input until that character is seen. That character is unread and also returned.

listen &optional *stream*

t if input is now available on *stream*. Uses the **:listen** operation (page 466).

clear-input &optional *stream*

Discards any input now available on *stream*, if it is an interactive stream. Uses the **:clear-input** stream operation (page 469).

22.1.3 Maclisp Compatibility Input Functions

These functions accept an argument *eof-option* to tell them what to do if end of file is encountered instead of any input. End of file signals an error if the argument is not supplied. Otherwise, end of file causes the function to return the argument's value. Note that an *eof-option* of **nil** means to return **nil** if the end of the file is reached; it is *not* equivalent to supplying no *eof-option*.

The arguments *stream* and *eof-option* can also be given in the reverse order for compatibility with old Maclisp programs. The functions attempt to figure out which way they were called by seeing whether each argument is a plausible stream. Unfortunately, there is an ambiguity with symbols: a symbol might be a stream and it might be an eof-option. If there are two arguments, one being a symbol and the other being something that is a valid stream, or only one argument, which is a symbol, then these functions interpret the symbol as an eof-option instead of as a stream. To force them to interpret a symbol as a stream, give the symbol an **si:io-stream-p** property whose value is **t**.

tyi &optional *stream eof-option*

Reads one character from *stream* and returns it. The character is echoed if *stream* is interactive, except that **Rubout** is not echoed. The **Control**, **Meta**, etc. shifts echo as C-, M-, etc.

The :tyi stream operation is preferred over the tyi function for some purposes. Note that it does not echo. See page 461.

(This function can take its arguments in the other order, for Maclisp compatibility only; see the note above.)

readch &optional *stream eof-option*

Like tyi except that instead of returning a fixnum character, it returns a symbol whose print name is the character. The symbol is interned in the current package. This is just Maclisp's version of character object. (This function can take its arguments in the other order, for Maclisp compatibility only; see the note above.)

This function is provided only for Maclisp compatibility, since in Zetalisp never uses symbols to represent characters in this way.

tyipeek &optional *peek-type stream eof-option*

This function is provided mainly for Maclisp compatibility; the :tyipeek stream operation is usually clearer (see page 461).

What tyipeek does depends on the *peek-type*, which defaults to nil. With a *peek-type* of nil, tyipeek returns the next character to be read from *stream*, without actually removing it from the input stream. The next time input is done from *stream* the character will still be there; in general, (= (tyipeek) (tyi)) is t.

If *peek-type* is a fixnum less than 1000 octal, then tyipeek reads characters from *stream* until it gets one equal to *peek-type*. That character is not removed from the input stream.

If *peek-type* is t, then tyipeek skips over input characters until the start of the printed representation of a Lisp object is reached. As above, the last character (the one that starts an object) is not removed from the input stream.

The form of tyipeek supported by Maclisp in which *peek-type* is a fixnum not less than 1000 octal is not supported, since the readtable formats of the Maclisp reader and the Zetalisp reader are quite different.

Characters passed over by tyipeek are echoed if *stream* is interactive.

22.1.4 Interactive Input with Prompting

prompt-and-read *type-of-parsing format-string* &rest *format-args*

Reads some sort of object from ***query-io***, parsing it according to *type-of-parsing*, and prompting by calling **format** using *format-string* and *format-args*.

type-of-parsing is either a keyword or a list starting with a keyword and continuing with a list of options and values, whose meanings depend on the keyword used.

Most keywords specify reading a line of input and parsing it in some way. The line can be terminated with **Return** or **End**. Sometimes typing just **End** has a special meaning.

The keywords defined are

:eval-sexp
:eval-form This keyword directs **prompt-and-read** to accept a Lisp expression. It is evaluated, and the value is returned by **prompt-and-read**.

If the Lisp expression is not a constant or quoted, the user is asked to confirm the value it evaluated to.

A default value can be specified with an option, as in

```
(:eval-sexp :default default)
```

Then, if the user types **Space**, **prompt-and-read** returns the *default* as the first value and **:default** as the second value.

:eval-sexp-or-end
:eval-form-or-end
Synonymously direct **prompt-and-read** to accept a Lisp expression or just the character **End**. If **End** is typed, **prompt-and-read** returns **nil** as its first value and **:end** as its second value. Otherwise, things proceed as for **:eval-sexp**.

A default value is allowed, as in **:eval-sexp**.

:read
:expression Synonymously direct **prompt-and-read** to read an object and return it, with no evaluation.

:expression-or-end
Is like **:expression** except that the user is also allowed to type just **End**. If he does so, **prompt-and-read** returns the two values **nil** and **:end**.

:number Directs **prompt-and-read** to read and return a number. It insists on getting a number, forcing the user to rub out anything else. Additional features can be specified with options:

```
(:number :input-radix radix :or-nil nil-ok-flag)
```

parses the number using radix *radix* if the number is a rational. (By default, the ambient radix is used). If *nil-ok-flag* is non-nil, then the user is also permitted to type just **Return** or **End**, and then **nil** is returned.

:decimal-number
:number-or-nil
:decimal-number-or-nil

Abbreviations for
```
(:number :input-radix 10)
(:number :or-nil t)
(:number :input-radix 10 :or-nil t)
```

:date — Directs prompt-and-read to read a date and time, terminated with *Return* or End, and return it as a universal time (see page 777). It allows several options:
```
(:date :never-p never-ok :past-p past-required)
```
If *past-required* is non-nil, the date must be before the present time, or the user must rub out and use a different date. If *never-ok* is non-nil, the user may also type "never"; then nil is returned.

:date-or-never
:past-date
:past-date-or-never

Abbreviations for
```
(:date :never-p t)
(:date :past-p t)
(:date :never-p t :past-p t)
```

:character — Directs prompt-and-read to read a single character and return a character object representing it.

:string — Directs prompt-and-read to read a line and return its contents as a string, using readline.

:string-or-nil — Directs prompt-and-read to read a line and return its contents as a string, using readline-trim. In addition, if the result would be empty, nil is returned instead of the empty string.

:string-list — Like :string-trim but regards the line as a sequence of input strings separated by commas. Each substring between commas is trimmed, and a list of the strings is returned.

:keyword-list — Like :string-list but converts each string to a keyword by interning it in the keyword package. The value is therefore a list of keywords.

:font-list — Like :string-list but converts each string to a font name by interning it in the fonts package. The symbols must already exist in that package or the user is required to retype the input.

:delimited-string

Directs prompt-and-read to read a string terminated by specified delimiters. With
```
(:delimited-string :delimiter delimiter-list
                   :buffer-size size)
```
you can specify a list of delimiter characters and an initial size for the buffer. The list defaults to (#\end) and the size to 100.

The work is done by read-delimited-string (page 450). The delimiters and size are passed to that function.

:delimited-string-or-nil
: Like :delimited-string except that nil is returned instead of the empty string if the first character read is a delimiter.

:host
: Directs prompt-and-read to read a line and interpret the contents as a network host name. The value returned is the host, looked up using si:parse-host (page 576). An option is defined:

```
        (:host :default default-name :chaos-only chaos-only)
```

If the line read is empty, the host named *default-name* is used. If *chaos-only* is non-nil, only hosts on the Chaosnet are permitted input.

:host-list
: Like :host but regards the line as a sequence of host names separated by commas. Each host name is looked up as in :host and a list of the resulting hosts is returned.

:pathname-host
: Like :host but uses fs:get-pathname-host to look up the host object from its name (page 577). Thus, you find hosts that can appear in pathnames rather than hosts that are on the network.

:pathname
: Directs prompt-and-read to read a line and parse it as a pathname, merging it with the defaults. If the line is empty, the default pathname is used. These options are defined:

```
        (:pathname :defaults defaults-alist-or-pathname
                   :version default-version)
```

uses *defaults-alist-or-pathname* as the *defaults* argument to fs:merge-pathname-defaults, and *default-version* as the *version* argument to it.

:pathname-or-nil
: Is like :pathname, but if the user types just End it is interpreted as meaning "no pathname" rather than "use the default". Then nil is returned.

:pathname-list
: Like :pathname but regards the line as a sequence of filenames separated by commas. Each filename is parsed and defaulted and a list of the resulting pathnames is returned.

:fquery
: Directs prompt-and-read to query the user for a fixed set of alternatives, using fquery. *type-of-parsing* should always be a list, whose car is :fquery and whose cdr is a list to be passed as the list of options (fquery's first argument).

Example:

```
        (prompt-and-read '(:fquery
                            . ,format:y-or-p-options)
                         "Eat it? ")
```

is equivalent to

```
        (y-or-n-p "Eat it? ")
```

This keyword is most useful as a way to get to **fquery** when going through an interface defined to call **prompt-and-read**.

22.2 Output Functions

These functions all take an optional argument called *stream*, which is where to send the output. If unsupplied *stream* defaults to the value of ***standard-output***. If *stream* is **nil**, the value of ***standard-output*** (i.e. the default) is used. If it is **t**, the value of ***terminal-io*** is used (i.e. the interactive terminal). This is all more-or-less compatible with Maclisp, except that instead of the variable ***standard-output*** Maclisp has several variables and complicated rules. For detailed documentation of streams, refer to section 22.3, page 459.

For **print** and the other expression output functions, see section 23.4, page 527.

write-char *char* &optional *stream*
tyo *char* &optional *stream*

Outputs *char* to *stream* (using **:tyo**). *char* may be an integer or a character object; in the latter case, it is converted to an integer before the **:tyo**.

write-byte *number* &optional *stream*

Outputs number to stream using :tyo. In strict Common Lisp, output to binary streams can be done only with **write-byte** and output to character streams requires **write-char**. In fact, the two functions are identical on the Lisp Machine.

write-string *string* &optional *stream* &key (*start* 0) *end*

Outputs *string* (or the specified portion of it) to *stream*.

write-line *string* &optional *stream* &key (*start* 0) *end*

Outputs *string* (or the specified portion) to *stream*, followed by a **Return** character.

fresh-line &optional *stream*

Outputs a **Return** character to **stream** unless either

(1) nothing has been output to *stream* yet, or

(2) the last thing output was a **Return** character, or

(3) *stream* does not remember what previous output there has been.

This uses the **:fresh-line** stream operation. The value is **t** if a **Return** is output, **nil** if nothing is output.

force-output &optional *stream*

Causes *stream*'s buffered output, if any, to be transmitted immediately. This uses the **:force-output** stream operation.

finish-output &optional *stream*

Causes *stream*'s buffered output, if any, to be transmitted immediately, and waits until that is finished. This uses the :finish stream operation.

clear-output &optional *stream*

Discards any output buffered in *stream*. This uses the :clear-output stream operation.

terpri &optional *stream*

Outputs a Return character to *stream*. It returns t for Maclisp compatibility. It is wise not to depend on the value terpri returns.

cli:terpri &optional *stream*

Outputs a Return character to *stream*. Returns nil to meet Common Lisp specifications. It is wise not to depend on the value cli:terpri returns.

The format function (see page 483) is very useful for producing nicely formatted text. It can do anything any of the above functions can do, and it makes it easy to produce good looking messages and such. format can generate a string or output to a stream.

stream-copy-until-eof *from-stream to-stream* &optional *leader-size*

stream-copy-until-eof inputs characters from *from-stream* and outputs them to *to-stream*, until it reaches the end of file on the *from-stream*. For example, if x is bound to a stream for a file opened for input, then (stream-copy-until-eof x *terminal-io*) prints the file on the console.

If *from-stream* supports the :line-in operation and *to-stream* supports the :line-out operation, then stream-copy-until-eof uses those operations instead of :tyi and :tyo, for greater efficiency. *leader-size* is passed as the argument to the :line-in operation.

beep &optional *beep-type* (*stream* *terminal-io*)

This function is intended to attract the user's attention by causing an audible beep, or flashing the screen, or something similar. If the stream supports the :beep operation, then this function sends it a :beep message, passing *beep-type* along as an argument. Otherwise it just causes an audible beep on the terminal.

beep-type is a keyword which explains the significance of this beep. Users can redefine beep to make different noises depending on the beep type. The defined beep types are:

zwei:converse-problem	Used for the beep that is done when Converse is unable to send a message.
zwei:converse-message-received	Used for the beeps done when a Converse message is received.
zwei:no-completion	Used when you ask for completion in the editor and the string does not complete.
tv:notify	Used for the beep done when you get a notification that cannot be printed on the selected window.

fquery	Used for the beep done by yes-or-no-p or by fquery with the :beep option specified.
supdup:terminal-bell	Used for the beep requested by the remote host being used through a Supdup window.
nil	Used whenever no other beep type applies.

The :beep operation is described on page 467.

cursorpos &rest *args*

This function exists primarily for Maclisp compatibility. Usually it is preferable to send the appropriate messages (see the Window System manual).

cursorpos normally operates on the *standard-output* stream; however, if the last argument is a stream or t (meaning *terminal-io*) then cursorpos uses that stream and ignores it when doing the operations described below. cursorpos only works on streams that are capable of these operations, such as windows. A stream is taken to be any argument that is not a number and not a symbol, or that is a symbol other than nil with a name more than one character long.

(cursorpos) => (*line* . *column*), the current cursor position.

(cursorpos *line column*) moves the cursor to that position. It returns t if it succeeds and nil if it doesn't.

(cursorpos *op*) performs a special operation coded by *op*, and returns t if it succeeds and nil if it doesn't. *op* is tested by string comparison, it is not a keyword symbol and may be in any package.

f	Moves one space to the right.
b	Moves one space to the left.
d	Moves one line down.
u	Moves one line up.
t	Homes up (moves to the top left corner). Note that t as the last argument to cursorpos is interpreted as a stream, so a stream *must* be specified if the t operation is used.
z	Home down (moves to the bottom left corner).
a	Advances to a fresh line. See the :fresh-line stream operation.
c	Clears the window.
e	Clear from the cursor to the end of the window.
l	Clear from the cursor to the end of the line.
k	Clear the character position at the cursor.
x	b then k.

22.3 I/O Streams

An *I/O stream*, or just *stream*, is a source and/or sink of characters or bytes. A set of *operations* is available with every stream; operations include things like "output a character" and "input a character". The way to perform an operation on a stream is the same for all streams, although what happens inside the stream is very different depending on what kind of a stream it is. So all a program has to know is how to deal with streams using the standard, generic operations. A programmer creating a new kind of stream only needs to implement the appropriate standard operations.

A stream is a message-receiving object. This means that it is something that you can apply to arguments. The first argument is a keyword symbol which is the name of the operation you wish to perform. The rest of the arguments depend on what operation you are doing. Message-passing and generic operations are explained in the flavor chapter (chapter 21, page 401).

Some streams can only do input, some can only do output, and some can do both. Some operations are only supported by some streams. Also, there are some operations that the stream may not support by itself, but which work anyway, albeit slowly, because the *stream default handler* can handle them. All streams support the operation :which-operations, which returns a list of the names of all of the operations that are supported "natively" by the stream. (:which-operations itself is not in the list.)

All input streams support all the standard input operations, and all output streams support all the standard output operations. All bidirectional streams support both.

streamp *object*

According to Common Lisp, this returns t if *object* is a stream. In the Lisp machine, a stream is any object which can be called as a function with certain calling conventions. It is theoretically impossible to test for this. However, streamp does return t for any of the usual types of streams, and nil for any Common Lisp datum which is not a stream.

22.3.1 Standard Streams

There are several variables whose values are streams used by many functions in the Lisp system. These variables and their uses are listed here. By convention, variables that are expected to hold a stream capable of input have names ending with -input, and similarly for output. Those expected to hold a bidirectional stream have names ending with -io. The names with asterisks are synonyms introduced for the sake of Common Lisp.

standard-input *Variable*
standard-input *Variable*

In the normal Lisp top-level loop, input is read from *standard-input* (that is, whatever stream is the value of *standard-input*). Many input functions, including tyi and read, take a stream argument that defaults to *standard-input*.

standard-output *Variable*
standard-output *Variable*

In the normal Lisp top-level loop, output is sent to *standard-output* (that is, whatever stream is the value of *standard-output*). Many output functions, including tyo and print, take a stream argument that defaults to *standard-output*.

error-output *Variable*
error-output *Variable*

The value of *error-output* is a stream on which noninteractive error or warning messages should be printed. Normally this is the same as *standard-output*, but *standard-output* might be bound to a file and *error-output* left going to the terminal.

debug-io *Variable*
debug-io *Variable*

The value of *debug-io* is used for all input and output by the error handler. Normally this is a synonym for *terminal-io*. The value may be nil, which is regarded as equivalent to a synonym for *terminal-io*. This feature is provided because users often set *debug-io* by hand, and it is much easier to set it back to nil afterward than to figure out the proper synonym stream pointing to *terminal-io*.

query-io *Variable*
query-io *Variable*

The value of *query-io* is a stream that should be used when asking questions of the user. The question should be output to this stream, and the answer read from it. The reason for this is that when the normal input to a program may be coming from a file, questions such as "Do you really want to delete all of the files in your directory??" should be sent directly to the user, and the answer should come from the user, not from the data file. *query-io* is used by fquery and related functions; see page 769.

terminal-io *Variable*
terminal-io *Variable*

The value of *terminal-io* is the stream that the program should use to talk to the user's console. In an interactive program, it is the window from which the program is being run; I/O on this stream reads from the keyboard and displays on the screen. However, in a background process that has no window, *terminal-io* defaults to a stream that does not ever expect to be used. If it is used, perhaps by an error printout, it turns into a background window and requests the user's attention.

trace-output *Variable*
trace-output *Variable*

The value of *trace-output* is the stream on which the trace function prints its output.

standard-input, *standard-output*, *error-output*, *debug-io*, *trace-output*, and *query-io* are initially bound to synonym streams that pass all operations on to the stream that is the value of *terminal-io*. Thus any operations performed on those streams go to the keyboard and screen.

Most user programs should not change the value of *terminal-io*. A program which wants (for example) to divert output to a file should do so by binding the value of *standard-output*; that way queries on *query-io*, debugging on *debug-io* and error messages sent to *error-output* can still get to the user by going through *terminal-io*, which is usually what is desired.

22.3.2 Standard Input Stream Operations

:tyi &optional *eof* *Operation on streams*

The stream inputs one character and returns it. For example, if the next character to be read in by the stream is a 'C', then the form

```
(send s :tyi)
```

returns the value of #/C (that is, 103 octal). Note that the :tyi operation does not echo the character in any fashion; it just does the input. The **tyi** function (see page 452) does echoing when reading from the terminal.

The optional *eof* argument to the :tyi operation tells the stream what to do if it gets to the end of the file. If the argument is not provided or is **nil**, the stream returns **nil** at the end of file. Otherwise it signals a **sys:end-of-file** error. Note that this is *not* the same as the eof-option argument to **read**, **tyi**, and related functions.

The :tyi operation on a binary input stream returns a non-negative number, not necessarily to be interpreted as a character.

For some streams (such as windows), not all the input data are numbers. Some are lists, called *blips*. The :tyi operation returns only numbers. If the next available input is not a number, it is discarded, and so on until a number is reached (or end of file is reached).

:any-tyi &optional *eof* *Operation on streams*

Like :tyi but returns any kind of datum. Non-numbers are not discarded as they would be by :tyi. This distinction only makes a difference on streams which can provide input which is not composed of numbers; currently, only windows can do that.

:tyipeek &optional *eof* *Operation on streams*

Peeks at the next character or byte from the stream without discarding it. The next :tyi or :tyipeek operation will get the same character.

eof is the same as in the :tyi operation: if **nil**, end of file returns **nil**; otherwise, it signals a **sys:end-of-file** error.

:untyi *char* *Operation on streams*

Unreads the character or byte *char*; that is to say, puts it back into the input stream so that the next :tyi operation will read it again. For example,

```
(send s :untyi 120)
(send s :tyi) ==> 120
```

This operation is used by **read**, and any stream that supports :tyi must support :untyi as well.

You are only allowed to :untyi one character before doing a :tyi, and the character you :untyi must be the last character read from the stream. That is, :untyi can only be used to back up one character, not to stuff arbitrary data into the stream. You also can't :untyi after you have peeked ahead with :tyipeek since that does one :untyi itself. Some streams implement :untyi by saving the character, while others implement it by backing up the pointer to a buffer.

:string-in *eof-option string* &optional (*start* 0) *end* *Operation on streams*

Reads characters from the stream and stores them into the array *string*. Many streams can implement this far more efficiently that repeated :tyi's. *start* and *end*, if supplied, delimit the portion of *string* to be stored into. If *eof-option* is non-nil then a sys:end-of-file error is signaled if end of file is reached on the stream before the string has been filled. If *eof-option* is nil, any number of characters before end of file is acceptable, even no characters.

If *string* has an array-leader, the fill pointer is adjusted to *start* plus the number of characters stored into *string*.

Two values are returned: the index of the next position in *string* to be filled, and a flag that is non-nil if end of file was reached before *string* was filled. Most callers do not need to look at either of these values.

string may be any kind of array, not necessarily a string; this is useful when reading from a binary input stream.

:line-in &optional *leader* *Operation on streams*

The stream should input one line from the input source, and return it as a string with the carriage return character stripped off. Contrary to what you might assume from its name, this operation is not much like the readline function.

Many streams have a string that is used as a buffer for lines. If this string itself were returned, there would be problems caused if the caller of the stream attempted to save the string away somewhere, because the contents of the string would change when the next line was read in. In order to solve this problem, the string must be copied. On the other hand, some streams don't reuse the string, and it would be wasteful to copy it on every :line-in operation. This problem is solved by using the *leader* argument to :line-in. If *leader* is nil (the default), the stream does not bother to copy the string and the caller should not rely on the contents of that string after the next operation on the stream. If *leader* is t, the stream does make a copy. If *leader* is a fixnum then the stream makes a copy with an array leader *leader* elements long. (This is used by the editor, which represents lines of buffers as strings with additional information in their array-leaders, to eliminate an extra copy operation.)

If the stream reaches end of file while reading in characters, it returns the characters it has read in as a string and returns a second value of t. The caller of the stream should therefore arrange to receive the second value, and check it to see whether the string returned was a whole line or just the trailing characters after the last carriage return in the input source.

This operation should be implemented by all input streams whose data are characters.

:string-line-in *eof-option string* &optional (*start* 0) *end* *Operation on streams*

Reads characters, storing them in *string*, until *string* is full or a Return character is read. If input stops due to a Return, the Return itself is not put in the buffer.

Thus, this operation is nearly the same as :string-in, except that :string-in always keeps going until the buffer is full or until end of file.

start and *end*, if supplied, delimit the portion of *string* to be stored into. If *eof-option* is non-nil then a sys:end-of-file error is signaled if end of file is reached on the stream before the string has been filled. If *eof-option* is nil, any number of characters before end of file is acceptable, even no characters.

If *string* has an array-leader, the fill pointer is adjusted to *start* plus the number of characters stored into *string*.

string may be any kind of array, not necessarily a string; this is useful when reading from a binary input stream.

Three values are returned:

(1) The index in *string* at which input stopped. This is the first index not stored in.

(2) t if input stopped due to end of file.

(3) t if the line is incomplete; that is, if a Return character did not terminate it.

:read-until-eof *Operation on streams*

Discards all data from the stream until it is at end of file, or does anything else with the same result.

:close &optional *ignore* *Operation on streams*

Releases resources associated with the stream, when it is not going to be used any more. On some kinds of streams, this may do nothing. On Chaosnet streams, it closes the Chaosnet connection, and on file streams, it closes the input file on the file server.

The argument is accepted for compatibility with :close on output streams.

22.3.3 Standard Output Stream Operations

:tyo *char* *Operation on streams*

The stream outputs the character *char*. For example, if s is bound to a stream, then the form

```
        (send s :tyo #/B)
```

outputs a B to the stream. For binary output streams, the argument is a non-negative number rather than specifically a character.

:fresh-line *Operation on streams*

Tells the stream that it should position itself at the beginning of a new line. If the stream is already at the beginning of a fresh line it should do nothing; otherwise it should output a carriage return. If the stream cannot tell whether it is at the beginning of a line, it should always output a carriage return.

:string-out (*string* 0) &optional *start end* *Operation on streams*

Outputs the characters of *string* successively to *stream*. This operation is provided for two reasons: first, it saves the writing of a loop which is used very often, and second, many streams can perform this operation much more efficiently than the equivalent sequence of :tyo operations.

If *start* and *end* are not supplied, the whole string is output. Otherwise a substring is output; *start* is the index of the first character to be output (defaulting to 0), and *end* is one greater than the index of the last character to be output (defaulting to the length of the string). Callers need not pass these arguments, but all streams that handle :string-out must check for them and interpret them appropriately.

:line-out *string* &optional (*start* 0) *end* *Operation on streams*

Outputs the characters of *string* successively to *stream*, then outputs a Return character. *start* and *end* optionally specify a substring, as with :string-out. If the stream doesn't support :line-out itself, the default handler implements it by means of :tyo.

This operation should be implemented by all output streams whose data are characters.

:close &optional *mode* *Operation on streams*

Closes the stream to make the output final if this is necessary. The stream becomes *closed* and no further output operations should be performed on it. However, it is all right to :close a closed stream. On many file server hosts, a file being written is not accessible to be read until the output stream is closed.

This operation does nothing on streams for which it is not meaningful.

The *mode* argument is normally not supplied. If it is :abort, we are abnormally exiting from the use of this stream. If the stream is outputting to a file, and has not been closed already, the stream's newly-created file is deleted; it will be as if it was never opened in the first place. Any previously existing file with the same name remains undisturbed.

:eof *Operation on streams*

Indicates the end of data on an output stream. This is different from :close because some devices allow multiple data files to be transmitted without closing. :close implies :eof when the stream is an output stream and the close mode is not :abort.

This operation does nothing on streams for which it is not meaningful.

22.3.4 Asking Streams What They Can Do

All streams are supposed to support certain operations which enable a program using the stream to ask which operations are available.

:which-operations *Operation on streams*

Returns a list of operations handled natively by the stream. Certain operations not in the list may work anyway, but slowly, so it is just as well if any programs that work with or without them choose not to use them.

:which-operations itself need not be in the list.

:operation-handled-p *operation* *Operation on streams*

Returns t if *operation* is handled natively by the stream: if *operation* is a member of the :which-operations list, or is :which-operations.

:send-if-handles *operation* &rest *arguments* *Operation on streams*

Performs the operation *operation*, with the specified *arguments*, only if the stream can handle it. If *operation* is handled, this is the same as sending an *operation* message directly, but if *operation* is not handled, using :send-if-handles avoids any error.

If *operation* is handled, :send-if-handles returns whatever values the execution of the *operation* returns. If *operation* is not handled, :send-if-handles returns nil.

:direction *Operation on streams*

Returns :input, :output, or :bidirectional for a bidirectional stream.

There are a few kinds of streams, which cannot do either input or output, for which the :direction operation returns nil. For example, open with the :direction keyword specified as nil returns a stream-like object which cannot do input or output but can handle certain file inquiry operations such as :truename and :creation-date.

:characters *Operation on streams*

Returns t if the data input or output on the stream represent characters, or nil if they are just numbers (as for a stream reading a non-text file).

:element-type *Operation on streams*

Returns a type specified describing in principle the data input or output on the stream. Refer to the function stream-element-type, below, which works using this operation.

These functions for inquiring about streams are defined by Common Lisp.

input-stream-p *stream*
: t if *stream* handles input operations (at least, if it handles :tyi).

output-stream-p *stream*
: t if *stream* handles output operations (at least, if it handles :tyo).

stream-element-type *stream*
: Returns a type specifier which describes, conceptually, the kind of data input from or output to *stream*. The value is always a subtype of integer (for a binary stream) or a subtype of character (for a character stream). If it is a subtype of integer, a Common Lisp program should use read-byte (page 450) or write-byte (page 456) for I/O. If it is a subtype of character, read-char (page 450) or write-char (page 456) should be used.

The value returned is not intended to be rigidly accurate. It describes the typical or characteristic sort of data transferred by the stream, but the stream may on occasion deal with data that do not fit the type; also, not all objects of the type may be possible as input or even make sense as output. For example, windows describe their element type as character even though they may offer blips, which are lists, as input on occasion. In addition, streams which say they provide characters really return integers if the :tyi operation is used rather than the standard Common Lisp function read-char.

22.3.5 Operations for Interactive Streams

The operations :listen, :tyi-no-hang, :rubout-handler and :beep are intended for interactive streams, which communicate with the user. :listen and :tyi-no-hang are supported in a trivial fashion by other streams, for compatibility.

:listen *Operation on streams*
: On an interactive device, the :listen operation returns non-nil if there are any input characters immediately available, or nil if there is no immediately available input. On a non-interactive device, the operation always returns non-nil except at end of file.

The main purpose of :listen is to test whether the user has hit a key, perhaps trying to stop a program in progress.

:tyi-no-hang &optional *eof* *Operation on streams*
: Just like :tyi except that it returns nil rather than waiting if it would be necessary to wait in order to get the character. This lets the caller check efficiently for input being available and get the input if there is any.

:tyi-no-hang is different from :listen because it reads a character.

Streams for which the question of whether input is available is not meaningful treat this operation just like :tyi. So do Chaosnet file streams. Although in fact reading a character from a file stream may involve a delay, these delays are *supposed* to be insignificant, so we pretend they do not exist.

:any-tyi-no-hang &optional *eof* *Operation on streams*

Like :tyi-no-hang but does not filter and discard input which is not numbers. It is therefore possible to see blips in the input stream. The distinction matters only for input from windows.

:rubout-handler *options function* &rest *args* *Operation on streams*

This is supported by interactive bidirectional streams, such as windows on the terminal, and is described in its own section below (see section 22.5, page 500).

:beep &optional *type* *Operation on streams*

This is supported by interactive streams. It attracts the attention of the user by making an audible beep and/or flashing the screen. *beep-type* is a keyword selecting among several different beeping noises; see **beep** (page 457) for a list of them.

22.3.6 Cursor Positioning Stream Operations

:read-cursorpos &optional (*units* :pixel) *Operation on streams*

This operation is supported by all windows and some other streams.

It returns two values, the current *x* and *y* coordinates of the cursor. It takes one optional argument, which is a symbol indicating in what units *x* and *y* should be; the symbols :pixel and :character are understood. :pixel means that the coordinates are measured in display pixels (bits), while :character means that the coordinates are measured in characters horizontally and lines vertically.

This operation and :increment-cursorpos are used by the **format** ~T request (see page 487), which is why ~T doesn't work on all streams. Any stream that supports this operation should support :increment-cursorpos as well.

Some streams return a meaningful value for the horizontal position but always return zero for the vertical position. This is sufficient for ~T to work.

:increment-cursorpos *Operation on streams*
x-increment y-increment &optional (*units* :pixel)

Moves the stream's cursor left or down according to the specified increments, as if by outputting an appropriate number of space or return characters. *x* and *y* are like the values of :read-cursorpos and *units* is the same as the *units* argument to :read-cursorpos.

Any stream which supports this operation should support :read-cursorpos as well, but it need not support :set-cursorpos.

Moving the cursor with :increment-cursorpos differs from moving it to the same place with :set-cursorpos in that this operation is thought of as doing output and :set-cursorpos is not. For example, moving a window's cursor down with :increment-cursorpos when it is near the bottom to begin with will wrap around, possibly doing a **MORE**. :set-cursorpos, by comparison, cannot move the cursor "down" if it is at the bottom of the window; it can move the cursor explicitly to the top of the window, but then no **MORE** will happen.

Some streams, such as those created by with-output-to-string, cannot implement arbitrary cursor motion, but do implement this operation.

:set-cursorpos *x y* &optional (*units* :pixel) *Operation on streams*
This operation is supported by the same streams that support :read-cursorpos. It sets the position of the cursor. *x* and *y* are like the values of :read-cursorpos and *units* is the same as the *units* argument to :read-cursorpos.

:clear-screen *Operation on streams*
Erases the screen area on which this stream displays. Non-window streams don't support this operation.

There are many other special-purpose stream operations for graphics. They are not documented here, but in the window-system documentation. No claim that the above operations are the most useful subset should be implied.

22.3.7 Operations for Efficient Pretty-Printing

grindef runs much more efficiently on streams that implement the :untyo-mark and :untyo operations.

:untyo-mark *Operation on streams*
This is used by the grinder (see page 528) if the output stream supports it. It takes no arguments. The stream should return some object that indicates how far output has gotten up to in the stream.

:untyo *mark* *Operation on streams*
This is used by the grinder (see page 528) in conjunction with :untyo-mark. It takes one argument, which is something returned by the :untyo-mark operation of the stream. The stream should back up output to the point at which the object was returned.

22.3.8 Random Access File Operations

The following operations are implemented only by streams to random-access devices, principally files.

:read-pointer *Operation on streams*
Returns the current position within the file, in characters (bytes in fixnum mode). For text files on ASCII file servers, this is the number of Lisp Machine characters, not ASCII characters. The numbers are different because of character-set translation.

:set-pointer *new-pointer* *Operation on streams*
Sets the reading position within the file to *new-pointer* (bytes in fixnum mode). For text files on ASCII file servers, this does not do anything reasonable unless *new-pointer* is 0, because of character-set translation. Some file systems support this operation for input streams only.

:rewind *Operation on streams*

This operation is obsolete. It is the same as **:set-pointer** with argument zero.

22.3.9 Buffered Stream Operations

:clear-input *Operation on streams*

Discards any buffered input the stream may have. It does nothing on streams for which it is not meaningful.

:clear-output *Operation on streams*

Discards any buffered output the stream may have. It does nothing on streams for which it is not meaningful.

:force-output *Operation on streams*

This is for output streams to buffered asynchronous devices, such as the Chaosnet. **:force-output** causes any buffered output to be sent to the device. It does not wait for it to complete; use **:finish** for that. If a stream supports **:force-output**, then **:tyo**, **:string-out**, and **:line-out** may have no visible effect until a **:force-output** is done.

This operation does nothing on streams for which it is not meaningful.

:finish *Operation on streams*

This is for output streams to buffered asynchronous devices, such as the Chaosnet. **:finish** does a **:force-output**, then waits until the currently pending I/O operation has been completed.

This operation does nothing on streams for which it is not meaningful.

The following operations are implemented only by buffered input streams. They allow increased efficiency by making the stream's internal buffer available to the user.

:read-input-buffer &optional *eof* *Operation on streams*

Returns three values: a buffer array, the index in that array of the next input byte, and the index in that array just past the last available input byte. These values are similar to the *string*, *start*, *end* arguments taken by many functions and stream operations. If the end of the file has been reached and no input bytes are available, this operation returns **nil** or signals an error, based on the *eof* argument, just like the **:tyi** operation. After reading as many bytes from the array as you care to, you must use the **:advance-input-buffer** operation.

:get-input-buffer &optional *eof* *Operation on streams*

This is an obsolete operation similar to **:read-input-buffer**. The only difference is that the third value is the number of significant elements in the buffer-array, rather than a final index. If found in programs, it should be replaced with **:read-input-buffer**.

:advance-input-buffer &optional *new-pointer* *Operation on streams*

If *new-pointer* is non-nil, it is the index in the buffer array of the next byte to be read. If *new-pointer* is nil, the entire buffer has been used up.

22.3.10 Obtaining Streams to Use

Windows are one important class of streams. Each window can be used as a stream. Output is displayed on the window and input comes from the keyboard. A window is created using make-instance on a window flavor. Simple programs use windows implicitly through *terminal-io* and the other standard stream variables.

Also important are *file streams*, which are produced by the function open (see page 582). These read or write the contents of a file.

Chaosnet streams are made from Chaosnet connections. Data output to the stream goes out over the network; data coming in over the network is available as input from the stream. File streams that deal with Chaosnet file servers are very similar to Chaosnet streams, but Chaosnet streams can be used for many purposes other than file access.

String streams read or write the contents of a string. They are made by with-output-to-string or with-input-from-string (see page 473), or by make-string-input-stream or make-string-output-stream, below.

Editor buffer streams read or write the contents of an editor buffer.

The *null stream* may be passed to a program that asks for a stream as an argument. It returns immediate end of file if used for input and throws away any output. The null stream is the symbol si:null-stream. This is to say, you do not call that function to get a stream or use the symbol's value as the stream; *the symbol itself* is the object that is the stream.

The *cold-load stream* is able to do I/O to the keyboard and screen without using the window system. It is what is used by the error handler, if you type Terminal Call, to handle a background error that the window system cannot deal with. It is called the cold-load stream because it is what is used during system bootstrapping, before the window system has been loaded.

si:null-stream *operation* &rest *arguments*

This function is the null stream. Like any stream, it supports various operations. Output operations are ignored and input operations report end of file immediately, with no data. Usage example:

```
(let ((*standard-output* 'si:null-stream))
  (function-whose-output-I-dont-want))
```

si:cold-load-stream *Constant*

The one and only cold-load stream. Usage example:

```
(let ((*query-io* si:cold-load-stream))
  (yes-or-no-p "Clear all window system locks? "))
```

with-open-stream (*variable expression*) *body...* *Macro*

body is executed with *variable* bound to the value of *expression*, which ought to be a stream. On exit, whether normal or by throwing, a :close message with argument :abort is sent to the stream.

This is a generalization of with-open-file, which is equivalent to using with-open-stream with a call to open as the *expression*.

with-open-stream-case (*variable expression*) *clauses...* *Macro*

Like with-open-stream as far as opening and closing the stream are concerned, but instead of a simple body, it has clauses like those of a condition-case that say what to do if *expression* does or does not get an error. See with-open-file-case, page 580.

make-synonym-stream *symbol-or-locative*
make-syn-stream *symbol-or-locative*

Creates and returns a *synonym* stream ('syn' for short). Any operations sent to this stream are redirected to the stream that is the value of the argument (if it is a symbol) or the contents of it (if it is a locative).

A synonym stream is actually an uninterned symbol whose function defnition is forwarded to the function cell of the argument or to the contents of the argument as appropriate. If the argument is a symbol, the synonym stream's print-name is *symbol*-syn-stream; otherwise the name is just syn-stream. Once a synonym stream is made for a symbol, it is recorded, and the same one is handed out again if there is another request for it.

The two names for this function are synonyms too.

make-concatenated-stream &rest *streams*

Returns an input stream which will read its input from the first of *streams* until that reaches its eof, then read input from the second of *streams*, and so on until the last of *streams* has reached end of file.

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

Returns a bidirectional stream which passes input operations to *input-stream* and passes output operations to *output-stream*. This works by attempting to recognize all standard input operations; anything not recognized is passed to *output-stream*.

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

Like make-two-way-stream except that each input character read via *input-stream* is output to *output-stream* before it is returned to the caller.

make-broadcast-stream &rest *streams*

Returns a stream that only works in the output direction. Any output sent to this stream is forwarded to all of the streams given. The :which-operations is the intersection of the :which-operations of all of the streams. The value(s) returned by a stream operation are the values returned by the last stream in *streams*.

zwei:interval-stream *interval-or-from-bp* &optional *to-bp in-order-p hack-fonts*

Returns a bidirectional stream that reads or writes all or part of an editor buffer. Note that editor buffer streams can also be obtained from **open** by using a pathname whose host is ED, ED-BUFFER or ED-FILE (see section 24.7.6, page 575).

The first three arguments specify the buffer or portion to be read or written. Either the first argument is an *interval* (a buffer is one kind of interval), and all the text of that interval is read or written, or the first two arguments are two buffer pointers delimiting the range to be read or written. The third argument is used only in the latter case; if non-nil, it tells the function to assume that the second buffer pointer comes later in the buffer than the first and not to take the time to verify the assumption.

The stream has only one pointer inside it, used for both input and output. As you do input, the pointer advances through the text. When you do output, it is inserted in the buffer at the place where the pointer has reached. The pointer starts at the beginning of the specified range.

hack-fonts tells what to do about fonts. Its possible values are

t	The character ε is recognized as special when you output to the stream; sequences such as ε2 are interpreted as font-changes. They do not get inserted into the buffer; instead, they change the font in which following output will be inserted. On input, font change sequences are included to indicate faithfully what was in the buffer.
:tyo	You are expected to read and write 16-bit characters containing font numbers.
nil	All output is inserted in font zero and font information is discarded in the input you receive. This is the best mode to use if you are reading or otherwise parsing the contents of an editor buffer.

sys:with-help-stream (*stream options...*) *body...* *Macro*

Executes the *body* with the variable *stream* bound to a suitable stream for printing a large help message. If *standard-output* is a window, then *stream* is also a window; a temporary window which fills the screen. Otherwise, *stream* is just the same as *standard-output*.

The purpose of this is to spare the user the need to read a large help printout in a small window, or have his data overwritten by it permanently. This is the mechanism used if you type the Control-Help key while in the rubout handler.

options is a list of alternating keywords and values.

:label	The value (which is evaluated) is used as the label of the temporary window, if one is used.
:width	The value, which is not evaluated, is a symbol. While *body* is executed, this symbol is bound to the width, in characters, available for the message.

:height — The value is a symbol, like the value after :width, and it is bound to the height in lines of the area available for the help message.

:superior — The value, which is evaluated, specifies the original stream to use in deciding where to print the help message. The default is *standard-output*.

22.3.11 String I/O Streams

The functions and special forms in this section allow you to create I/O streams that input from or output to the contents of a string.

make-string-input-stream *string* &optional (*start* 0) *end*

Returns a stream which can be used to read the contents of *string* (or the portion of it from index *start* to index *end*) as input. End of file occurs on reading past position *end* or the end of string.

make-string-output-stream &optional *string*

Returns an output stream which will accumulate all output in a string. If *string* is non-nil, output is added to it with string-nconc (page 216). Otherwise, a new string is created and used to hold the output.

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

Returns the string of output accumulated so far by a stream which was made by make-string-output-stream. The accumulated output is cleared out, so it will not be obtained again if get-output-stream-string is called another time on the same stream.

with-input-from-string (*var string* &key *start end index*) *body...* *Macro*

The form

```
(with-input-from-string (var string)
    body)
```

evaluates the forms in *body* with the variable *var* bound to a stream which reads characters from the string which is the value of the form *string*. The value of the construct is the value of the last form in its body.

If the *start* and *end* arguments are specified, they should be forms. They are evaluated at run time to produce the indices starting and ending the portion of *string* to be read.

If the *index* argument is specified, it should be something setf can store in. When *body* is finished, the index in the string at which reading stopped is stored there. This is the index of the first character not read. If the entire string was read, it is the length of the string. The value of *index* is not updated until with-input-from-string is exited, so you can't use its value within the body to see how far the reading has gotten. Example:

```
(with-input-from-string
    (foo "This is a test." :start (+ 2 2) :end 8 :index bar)
  (readline))
```

returns " is " and sets bar to eight.

An older calling sequence which used positional rather than keyword arguments is still accepted:

```
(with-input-from-string (var string index end)
  body)
```

The functions read-from-string and cli:read-from-string are convenient special cases of what with-input-from-string can do. See page 533.

with-output-to-string (*var* [*string* [*index*]]) *body*... *Macro*

This special form provides a variety of ways to send output to a string through an I/O stream.

```
(with-output-to-string (var)
  body)
```

evaluates the forms in *body* with *var* bound to a stream which saves the characters output to it in a string. The value of the special form is the string.

```
(with-output-to-string (var string)
  body)
```

appends its output to the string which is the value of the form *string*. (This is like the string-nconc function; see page 216.) The value returned is the value of the last form in the body, rather than the string. Multiple values are not returned. *string* must have a fill pointer. If *string* is too small to contain all the output, adjust-array-size is used to make it bigger.

```
(with-output-to-string (var string index)
  body)
```

is similar to the above except that *index* is a variable or setf-able reference which contains the index of the next character to be stored into. It must be initialized before the with-output-to-string and it is updated upon normal exit. The value of *index* is not updated until with-output-to-string returns, so you can't use its value within the body to see how far the writing has gotten. The presence of *index* means that *string* is not required to have a fill-pointer; if there is one, it is updated on exit.

Another way of doing output to a string is to use the format facility (see page 483).

22.3.12 Implementing Streams

There are two ways to implement a stream: using defun or using flavors.

Using flavors is best when you can take advantage of the predefined stream mixins, including those which perform buffering, or when you wish to define several similar kinds of streams that can inherit methods from each other.

defun (or defselect, which is a minor variation of the technique) may have an advantage if you are dividing operations into broad groups and handling them by passing them off to one or more other streams. In this case, the automatic operation decoding provided by flavors may get in the way. A number of streams in the system are implemented using defun or defselect for historical reasons. It isn't yet clear whether there is any reason not to convert most of them to use flavors.

If you use **defun**, you can use the *stream default handler* to implement some of the standard operations for you in a default manner. If you use flavors, there are predefined mixins to do this for you.

A few streams are individual objects, one of a kind. For example, there is only one null stream, and no need for more, since two null streams would behave identically. But most streams are elements of a general class. For example, there can be many file streams for different files, even though all behave the same way. There can also be multiple streams reading from different points in the same file.

If you implement a class of streams with **defun**, then the actual streams must be closures of the function you define, made with **closure**.

If you use flavors to implement the streams, having a class of similar streams comes naturally: each instance of the flavor is a stream, and the instance variables distinguish one stream of the class from another.

22.3.13 Implementing Streams with Flavors

To define a stream using flavors, define a flavor which incorporates the appropriate predefined stream flavor, and then redefine those operations which are peculiar to your own type of stream.

Flavors for defining unbuffered streams:

si:stream *Flavor*

This flavor provides default definitions for a few standard operations such as **:direction** and **:characters**. Usually you do not have to mention this explicitly; instead you use the higher level flavors below, which are built on this one.

si:input-stream *Flavor*

This flavor provides default definitions of all the mandatory input operations except **:tyi** and **:untyi**, in terms of those two. You can make a simple non-character input stream by defining a flavor incorporating this one and giving it methods for **:tyi** and **:untyi**.

si:output-stream *Flavor*

This flavor provides default definitions of all the mandatory output operations except **:tyo**, in terms of **:tyo**. All you need to do to define a simple unbuffered non-character output stream is to define a flavor incorporating this one and give it a method for the **:tyo** operation.

si:bidirectional-stream *Flavor*

This is a combination of **si:input-stream** and **si:output-stream**. It defines **:direction** to return **:bidirectional**. To define a simple unbuffered non-character bidirectional stream, build on this flavor and define **:tyi**, **:untyi** and **:tyo**.

The unbuffered streams implement operations such as :string-out and :string-in by repeated use of :tyo or :tyi.

For greater efficiency, if the stream's data is available in blocks, it is better to define a buffered stream. You start with the predefined buffered stream flavors, which define :tyi or :tyo themselves and manage the buffers for you. You must provide other operations that the system uses to obtain the next input buffer or to write or discard an output buffer.

Flavors for defining buffered streams:

si:buffered-input-stream *Flavor*

This flavor is the basis for a non-character buffered input stream. It defines :tyi as well as all the other standard input operations, but you must define the two operations :next-input-buffer and :discard-input-buffer, which the buffer management routines use.

:next-input-buffer *Operation on* si:buffered-input-stream

In a buffered input stream, this operation is used as a subroutine of the standard input operations, such as :tyi, to get the next bufferful of input data. It should return three values: an array containing the data, a starting index in the array, and an ending index. For example, in a Chaosnet stream, this operation would get the next packet of input data and return pointers delimiting the actual data in the packet.

:discard-input-buffer *buffer-array* *Operation on* si:buffered-input-stream

In a buffered input stream, this operation is used as a subroutine of the standard input operations such as :tyi. It says that the buffer management routines have used or thrown away all the input in a buffer, and the buffer is no longer needed.

In a Chaosnet stream, this operation would return the packet buffer to the pool of free packets.

si:buffered-output-stream *Flavor*

This flavor is the basis for a non-character buffered output stream. It defines :tyo as well as all the other standard output operations, but you must define the operations :new-output-buffer, :send-output-buffer and :discard-output-buffer, which the buffer management routines use.

:new-output-buffer *Operation on* si:buffered-output-stream

In a buffered output stream, this operation is used as a subroutine of the standard output operations, such as :tyo, to get an empty buffer for storing more output data. How the buffer is obtained depends on the kind of stream, but in any case this operation should return an array (the buffer), a starting index, and an ending index. The two indices delimit the part of the array that is to be used as a buffer.

For example, a Chaosnet stream would get a packet from the free pool and return indices delimiting the part of the packet array which can hold data bytes.

:send-output-buffer *buffer-array ending-index* — *Operation on* **si:buffered-output-stream**

In a buffered output stream, this operation is used as a subroutine of the standard output operations, such as **:tyo**, to send the data in a buffer that has been completely or partially filled.

ending-index is the first index in the buffer that has not actually been stored. This may not be the same as the ending index that was returned by the **:new-output-buffer** operation that was used to obtain this buffer; if a **:force-output** is being handled, *ending-index* indicates how much of the buffer is currently full.

The method for this operation should process the buffer's data and, if necessary, return the buffer to a free pool.

:discard-output-buffer *buffer-array* — *Operation on* **si:buffered-output-stream**

In a buffered output stream, this operation is used as a subroutine of the standard output operations, such as **:clear-output**, to free an output buffer and say that the data in it should be ignored.

It should simply return *buffer-array* to a free pool, if appropriate.

Some buffered output streams simply have one buffer array which they use over and over. For such streams, **:new-output-buffer** can simply return that particular array each time; **:send-output-buffer** and **:discard-output-buffer** do not have to do anything about returning the buffer to a free pool. In fact, **:discard-output-buffer** can probably do nothing.

si:buffered-stream — *Flavor*

This is a combination of **si:buffered-input-stream** and **si:buffered-output-stream**, used to make a buffered bidirectional stream. The input and output buffering are completely independent of each other. You must define all five of the low level operations: **:new-output-buffer**, **:send-output-buffer** and **:discard-output-buffer** for output, and **:next-input-buffer** and **:discard-input-buffer** for input.

The data in most streams are characters. Character streams should support either **:line-in** or **:line-out** in addition to the other standard operations.

si:unbuffered-line-input-stream — *Flavor*

This flavor is the basis for unbuffered character input streams. You need only define **:tyi** and **:untyi**.

si:line-output-stream-mixin — *Flavor*

To make an unbuffered character output stream, mix this flavor into the one you define, together with **si:output-stream**. In addition, you must define **:tyo**, as for unbuffered non-character streams.

si:buffered-input-character-stream *Flavor*

This is used just like si:buffered-input-stream, but it also provides the :line-in operation and makes :characters return t.

si:buffered-output-character-stream *Flavor*

This is used just like si:buffered-output-stream, but it also provides the :line-out operation and makes :characters return t.

si:buffered-character-stream *Flavor*

This is used just like si:buffered-stream, but it also provides the :line-in and :line-out operations and makes :characters return t.

To make an unbuffered random-access stream, you need only define the :read-pointer and :set-pointer operations as appropriate. Since you provide the :tyi or :tyo handler yourself, the system cannot help you.

In a buffered random-access stream, the random access operations must interact with the buffer management. The system provides for this.

si:input-pointer-remembering-mixin *Flavor*

Incorporate this into a buffered input stream to support random access. This flavor defines the :read-pointer and :set-pointer operations. If you wish :set-pointer to work, you must provide a definition for the :set-buffer-pointer operation. You need not do so if you wish to support only :read-pointer.

:set-buffer-pointer *new-pointer* *Operation on* si:input-pointer-remembering-mixin

You must define this operation if you use si:input-pointer-remembering-mixin and want the :set-pointer operation to work.

This operation should arrange for the next :next-input-buffer operation to provide a bufferful of data that includes the specified character or byte position somewhere inside it.

The value returned should be the file pointer corresponding to the first character or byte of that next bufferful.

si:output-pointer-remembering-mixin *Flavor*

Incorporate this into a buffered output stream to support random access. This mixin defines the :read-pointer and :set-pointer operations. If you wish :set-pointer to work, you must provide definitions for the :set-buffer-pointer and :get-old-data operations. You need not do so if you wish to support only :read-pointer.

:set-buffer-pointer *Operation on* si:output-pointer-remembering-mixin
new-pointer

This is the same as in si:input-pointer-remembering-mixin.

:get-old-data *Operation on* si:output-pointer-remembering-mixin
buffer-array lower-output-limit

The buffer management routines perform this operation when you do a :set-pointer that is outside the range of pointers that fit in the current output buffer. They first send the old buffer, then do :set-buffer-pointer as described above to say where in the file the next output buffer should come, then do :new-output-buffer to get the new buffer. Then the :get-old-data operation is performed.

It should fill current buffer (*buffer-array*) with the *old* contents of the file at the corresponding addresses, so that when the buffer is eventually written, any bytes skipped over by random access will retain their old values.

The instance variable si:stream-output-lower-limit is the starting index in the buffer of the part that is supposed to be used for output. si:stream-output-limit is the ending index. The instance variable si:output-pointer-base is the file pointer corresponding to the starting index in the buffer.

si:file-stream-mixin *Flavor*

Incorporate this mixin together with si:stream to make a *file probe stream*, which cannot do input or output but records the answers to an enquiry about a file. You should specify the init option :pathname when you instantiate the flavor.

You must provide definitions for the :plist and :truename operations; in terms of them, this mixin defines the operations :get, :creation-date, and :info.

si:input-file-stream-mixin *Flavor*

Incorporate this mixin into input streams that are used to read files. You should specify the file's pathname with the :pathname init option when you instantiate the flavor.

In addition to the services and requirements of si:file-stream-mixin, this mixin takes care of mentioning the file in the who-line. It also includes si:input-pointer-remembering-mixin so that the :read-pointer operation, at least, will be available.

si:output-file-stream-mixin *Flavor*

This is the analogue of si:input-file-stream-mixin for output streams.

22.3.14 Implementing Streams Without Flavors

You do not need to use flavors to implement a stream. Any object that can be used as a function, and decodes its first argument appropriately as an operation name, can serve as a stream. Although in practice using flavors is as easy as any other way, it is educational to see how to define streams "from scratch".

We could begin to define a simple output stream, which accepts characters and conses them onto a list, as follows:

```
(defvar the-list nil)

(defun list-output-stream (op &optional arg1 &rest rest)
  (ecase op
    (:tyo
     (setq the-list (cons arg1 the-list)))
    (:which-operations '(:tyo))))
```

This is an output stream, and so it supports the :tyo operation. All streams must support :which-operations.

The lambda-list for a stream defined with a defun must always have one required parameter (*op*), one optional parameter (*arg1*), and a rest parameter (*rest*).

This definition is not satisfactory, however. It handles :tyo properly, but it does not handle :string-out, :direction, :send-if-handles, and other standard operations.

The function stream-default-handler exists to spare us the trouble of defining all those operations from scratch in simple streams like this. By adding one additional clause, we let the default handler take care of all other operations, if it can.

```
(defun list-output-stream (op &optional arg1 &rest rest)
    (selectq op
        (:tyo
         (setq the-list (cons arg1 the-list)))
        (:which-operations '(:tyo))
        (otherwise
          (stream-default-handler #'list-output-stream
                                  op arg1 rest))))
```

If the operation is not one that the stream understands (e.g. :string-out), it calls stream-default-handler. Note how the rest argument is passed to it. This is why the argument list must look the way it does. stream-default-handler can be thought of as a restricted analogue of flavor inheritance.

If we want to have only one stream of this sort, the symbol list-output-stream can be used as the stream. The data output to it will appear in the global value of the-list. One more step is required, though:

```
(defprop list-output-stream t si:io-stream-p)
```

This tells certain functions including read to treat the symbol list-output-stream as a stream rather than as an end of file option.

If we wish to be able to create any number of list output streams, each accumulating its own list, we must use closures:

```
(defvar the-stream nil
  "Inside a list output stream, holds the stream itself.")
(defvar the-list nil
  "Inside a list output stream,
holds the list of characters being accumulated.")

(defun list-output-stream (op &optional arg1 &rest rest)
    (selectq op
        (:tyo
         (push arg1 the-list)))
        (:withdrawal (prog1 the-list (setq the-list nil)))
        (:which-operations '(:tyo :withdrawal))
        (otherwise
          (stream-default-handler the-stream
                                  op arg1 rest))))

(defun make-list-output-stream ()
  (let ((the-stream the-list))
    (setq the-stream
          (closure '(the-stream the-list)
                   'list-output-stream))))
```

We have added a new operation :withdrawal that can be used to find out what data has been accumulated by a stream. This is necessary because we can no longer simply look at or set the global value of the-list; that is not the same as the value closed into the stream.

In addition, we have a new variable the-stream which allows the function list-output-stream to know which stream it is serving at any time. This variable is passed to stream-default-handler so that when it simulates :string-out by means of :tyo, it can do the :tyo's to the same stream that the :string-out was done to.

The same stream could be defined with defselect instead of defun. It actually makes only a small difference. The defun for list-output-stream could be replaced with this code:

```
(defselect (list-output-stream list-output-d-h)
  (:tyo (arg1)
    (push arg1 the-list))
  (:withdrawal ()
    (prog1 the-list (setq the-list nil))))

(defun list-output-d-h (op &optional arg1 &rest rest)
  (stream-default-handler the-stream op arg1 rest))
```

defselect takes care of decoding the operations, provides a definition for :which-operations, and allows you to write a separate lambda list for each operation.

By comparison, the same stream defined using flavors looks like this:

```
(defflavor list-output-stream ((the-list nil))
           (si:line-output-stream-mixin si:output-stream))

(defmethod (list-output-stream :tyo) (character)
  (push character the-list))

(defmethod (list-outut-stream :withdrawal) ()
  (prog1 the-list (setq the-list nil)))

(defun make-list-output-stream ()
  (make-instance 'list-output-stream))
```

Here is a simple input stream, which generates successive characters of a list.

```
(defvar the-list)       ;Put your input list here
(defvar the-stream)
(defvar untyied-char nil)

(defun list-input-stream (op &optional arg1 &rest rest)
  (selectq op
    (:tyi
     (cond ((not (null untyied-char))
              (prog1 untyied-char (setq untyied-char nil)))
             ((null the-list)
              (and arg1 (error arg1)))
             (t (pop the-list))))
    (:untyi
     (setq untyied-char arg1))
    (:which-operations '(:tyi :untyi))
    (otherwise
      (stream-default-handler the-stream
                              op arg1 rest))))

(defun make-list-input-stream (the-list)
  (let (the-stream untyied-char)
    (setq the-stream
          (closure '(the-list the-stream untyied-char)
                   'list-input-stream))))
```

The important things to note are that :untyi must be supported, and that the stream must check for having reached the end of the information and do the right thing with the argument to the :tyi operation.

stream-default-handler *stream op arg1 rest*
Tries to handle the *op* operation on *stream*, given arguments of *arg1* and the elements of *rest*. The exact action taken for each of the defined operations is explained with the documentation on that operation, above.

22.4 Formatted Output

There are two ways of doing general formatted output. One is the function **format**. The other is the **output** subsystem. **format** uses a control string written in a special format specifier language to control the output format. **format:output** provides Lisp functions to do output in particular formats.

For simple tasks in which only the most basic format specifiers are needed, **format** is easy to use and has the advantage of brevity. For more complicated tasks, the format specifier language becomes obscure and hard to read. Then **format:output** becomes advantageous because it works with ordinary Lisp control constructs.

22.4.1 The Format Function

format *destination control-string* &rest *args*

Produces formatted output. **format** outputs the characters of *control-string*, except that a tilde ('~') introduces a directive. The character after the tilde, possibly preceded by prefix parameters and modifiers, specifies what kind of formatting is desired. Most directives use one or more elements of *args* to create their output; the typical directive puts the next element of *args* into the output, formatted in some special way.

The output is sent to *destination*. If *destination* is **nil**, a string is created which contains the output; this string is returned as the value of the call to **format**. In all other cases **format** returns no interesting value (generally it returns **nil**). If *destination* is a stream, the output is sent to it. If *destination* is **t**, the output is sent to ***standard-output***. If *destination* is a string with an array-leader, such as would be acceptable to **string-nconc** (see page 216), the output is added to the end of that string.

A directive consists of a tilde, optional prefix parameters separated by commas, optional colon (':') and atsign ('@') modifiers, and a single character indicating what kind of directive this is. The alphabetic case of the character is ignored. The prefix parameters are generally decimal numbers. Examples of control strings:

```
"~S"          ; This is an S directive with no parameters.
"~3,4:@s"     ; This is an S directive with two parameters, 3 and 4,
              ;   and both the colon and atsign flags.
"~,4S"        ; The first prefix parameter is omitted and takes
              ;   on its default value, while the second is 4.
```

format includes some extremely complicated and specialized features. It is not necessary to understand all or even most of its features to use **format** efficiently. The beginner should skip over anything in the following documentation that is not immediately useful or clear. The more sophisticated features are there for the convenience of programs with complicated formatting requirements.

Sometimes a prefix parameter is used to specify a character, for instance the padding character in a right- or left-justifying operation. In this case a single quote (" ' ") followed by the desired character may be used as a prefix parameter, so that you don't have to know the decimal numeric values of characters in the character set. For example, you can use "~5,'0d" instead of "~5,48d" to print a decimal number in five columns with leading zeros.

In place of a prefix parameter to a directive, you can put the letter V, which takes an argument from *args* as a parameter to the directive. Normally this should be a number but it doesn't really have to be. This feature allows variable column-widths and the like. Also, you can use the character # in place of a parameter; it represents the number of arguments remaining to be processed.

Here are some relatively simple examples to give you the general flavor of how **format** is used.

```
(format nil "foo") => "foo"
(setq x 5)
(format nil "The answer is ~D." x) => "The answer is 5."
(format nil "The answer is ~3D." x) => "The answer is   5."
(setq y "elephant")
(format nil "Look at the ~A!" y) => "Look at the elephant!"
(format nil "The character ~:@C is strange." #\meta-beta)
      => "The character Meta-β (Greek-b) is strange."
(setq n 3)
(format nil "~D item~:P found." n) => "3 items found."
(format nil "~R dog~:[s are~; is~] here." n (= n 1))
      => "three dogs are here."
(format nil "~R dog~:*~[~1; is~:;s are~] here." n)
      => "three dogs are here."
(format nil "Here ~[~1;is~:;are~] ~:*~R pupp~:@P." n)
      => "Here are three puppies."
```

The directives will now be described. *arg* will be used to refer to the next argument from *args*.

~A *arg*, any Lisp object, is printed without escaping (as by **princ**). ~:A prints ( ) if *arg* is nil; this is useful when printing something that is always supposed to be a list. ~*n*A inserts spaces on the right, if necessary, to make the column width at least *n*. The @ modifier causes the spaces to be inserted on the left rather than the right. ~*mincol,colinc,minpad,padchar*A is the full form of ~A, which allows elaborate control of the padding. The string is padded on the right with at least *minpad* copies of *padchar*; padding characters are then inserted *colinc* characters at a time until the total width is at least *mincol*. The defaults are 0 for *mincol* and *minpad*, 1 for *colinc*, and space for *padchar*.

~S This is just like ~A, but *arg* is printed *with* escaping (as by **prin1** rather than **princ**).

~D *arg*, a number, is printed in base ten. Unlike **print**, ~D never puts a decimal point after the number. ~*n*D uses a column width of *n*; spaces are inserted on the left if the number requires less than *n* columns for its digits and sign. If the number doesn't fit in *n* columns, additional columns are used as needed. ~*n*,*m*D uses *m* as the pad character instead of space. If *arg* is not a number, it is printed in ~A format and decimal base. The @ modifier causes the number's sign to be printed always; the default is only to print it if the number is negative. The : modifier causes commas to be printed between groups of three digits; the third prefix parameter may be used to change the character used as the comma. Thus the most general form of ~D is ~*mincol*,*padchar*,*commachar*D.

~O This is just like ~D but prints in octal instead of decimal.

~X This is just like ~D but prints in hex instead of decimal. Note that ~X used to have a different meaning: print one or more spaces. Uses of ~X intended to have this meaning should be replaced with ~@T.

~B This is just like ~D but prints in binary instead of decimal.

~*w*,*d*,*k*,*ovfl*,*pad* F
arg is printed in nonexponential floating point format, as in '10.5'. (If the magnitude of *arg* is very large or very small, it is printed in exponential notation.) The parameters control the details of the formatting.

w is the total field width desired. If omitted, this is not constrained.

d is the number of digits to print after the decimal point. If *d* is omitted, it is chosen to do a good job based on *w* (if specified) and the value of *arg*.

k is a scale factor. *arg* is multiplied by (exp 10. *k*) before it is printed.

ovfl is a character to use for overflow. If *arg* is too big to print and fit the constraints of field width, etc., and *ovfl* is specified then the whole field is filled with *ovfl*. If *ovfl* is not specified, *arg* is printed using extra width as needed.

pad is a character to use for padding on the left, when the field width is specified and not that many characters are really needed.

If the @ modifier is used, a sign is printed even if *arg* is positive.

Rational numbers are converted to floats and then printed. Anything else is printed with ~*w*D format.

~*w*,*d*,*e*,*k*,*ovfl*,*pad*,*expt* E
arg is printed in exponential notation, as in '.105e+2'. The parameters control the details of the formatting.

w is the total field width desired. If omitted, this is not constrained.

d and *k*
control the number of mantissa digits and their arrangement around the decimal point. $d+1$ digits are printed. If k is positive, all of them are significant digits, and the decimal point is printed after the first k of them. If k is zero or negative, the first $|k|+1$ of the $d+1$ digits are leading zeros, and the decimal point follows the first zero. (This zero can be omitted if necessary to fit the number in *w* characters.) So the number of significant figures is less than *d* if *k* is negative.

The exponent printed always compensates for any powers of ten introduced according to *k*, so 10.5 might be printed as 0.105e+2 or as 1050.0e-2.

If *d* is omitted, the system chooses enough significant figures to represent the float accurately. If *k* is omitted, the default is one.

e is the number of digits to use for the exponent. If it is not specified, however many digits are needed are used.

ovfl is the overflow character. If the exponent doesn't fit in *e* digits or the entire number does not fit in *w* characters, then if *ovfl* is specified, the field of *w* characters is filled with *ovfl*. Otherwise more characters are used as needed.

pad is a character to use for padding on the left, when the field width is specified and not that many characters are really needed.

expt is a character to use to separate the mantissa from the exponent. The default is e or s or f, whichever would be used in printing the number normally.

If the @ modifier is used, a sign is printed even if *arg* is positive.

~*w,d,e,k,ovfl,pad,expt* G
Prints a floating point number *arg* in either ~F or ~E format. Fixed format is used if the absolute value of *arg* is less than (expt 10. *d*), and exponential format otherwise. (If *d* is not specified, it defaults based on the value of *arg*.) If fixed format is used, *e*+2 blanks are printed at the end (where the exponent and its separator and sign would go, in exponential format). These count against the width *w* if that is specified. Four blanks are used if *e* is omitted. The diminished width available, *d*, *ovfl* and *pad* are used as specified. The scale factor used in fixed format is always zero, not *k*.

If exponential format needs to be used, all the parameters are passed to the ~E directive to print the number.

Rational numbers are converted to floats and then printed. Anything else is printed with ~*w*D format.

~$ ~*rdig,ldig,field,padchar*$ prints *arg*, a float, with exactly *rdig* digits after the decimal point. The default for *rdig* is 2, which is convenient for printing amounts of money. At least *ldig* digits are printed preceding the decimal point; leading zeros are printed if there would be fewer than *ldig*. The default for *ldig* is 1. The number is right justified in a field *field* columns long, padded out with *padchar*. The colon modifier means that the sign character is to be at the beginning of the field, before the padding, rather than just to the left of the number. The atsign modifier says that the sign character should always be output.

If *arg* is not a number, or is unreasonably large, it is printed in ~*field,,,padchar*@A format; i.e. it is princ'ed right-justified in the specified field width.

~C (character *arg*) is put in the output. *arg* is treated as a keyboard character (see page 206), thus it may contain extra control-bits. These are printed first by representing them with abbreviated prefixes: 'C-' for Control, 'M-' for Meta, 'H-' for Hyper, and 'S-' for Super.

With the colon flag (~:C), the names of the control bits are spelled out (e.g. 'Control-Meta-F') and non-printing characters are represented by their names (e.g. 'Return') rather than being output as themselves. The printing characters Space and Altmode are also represented as their names, but all others are printed directly.

With both colon and atsign (~:@C), the colon-only format is printed, and then if the character requires the Top or Greek (Front) shift key(s) to type it, this fact is mentioned (e.g. '∀ (Top-U)'). This is the format used for telling the user about a key he is expected to type, for instance in prompt messages.

For all three of these formats, if the character is a mouse character, it is printed as Mouse-, the name of the button, '-', and the number of clicks.

With just an atsign (~@C), the character is printed in such a way that the Lisp reader can understand it, using '#\' or '#/', depending on the escaping character of *readtable* (see page 516).

~% Outputs a carriage return. ~*n*% outputs *n* carriage returns. No argument is used. Simply putting a carriage return in the control string would work, but ~% is usually used because it makes the control string look nicer in the Lisp source program.

~& The :fresh-line operation is performed on the output stream. Unless the stream knows that it is already at the front of a line, this outputs a carriage return. ~*n*& does a :fresh-line operation and then outputs *n*-1 carriage returns.

~| Outputs a page separator character (#\page). ~*n*| does this *n* times. With a : modifier, if the output stream supports the :clear-screen operation this directive clears the screen, otherwise it outputs page separator character(s) as if no : modifier were present. | is vertical bar, not capital I.

~~ Outputs a tilde. ~*n*~ outputs *n* tildes.

~<CR> Tilde immediately followed by a carriage return ignores the carriage return and any whitespace at the beginning of the next line. With a :, the whitespace is left in place. With an @, the carriage return is left in place. This directive is typically used when a format control string is too long to fit nicely into one line of the program.

~* *arg* is ignored. ~*n** ignores the next *n* arguments. ~:* "ignores backwards"; that is, it backs up in the list of arguments so that the argument last processed will be processed again. ~*n*:* backs up *n* arguments. ~*n*@* is absolute; it moves to argument *n* (*n* = 0 specifies the first argument).

When within a ~{ construct (see below), the ignoring (in either direction) is relative to the list of arguments being processed by the iteration.

~P If *arg* is not 1, a lower-case 's' is printed. ('P' is for 'plural'.) ~:P does the same thing, after doing a ~:*; that is, it prints a lower-case s if the *last* argument was not 1. ~@P prints 'y' if the argument is 1, or 'ies' if it is not. ~:@P does the same thing, but backs up first.

~T Spaces over to a given column. ~*n*,*m*T outputs sufficient spaces to move the cursor to column *n*. If the cursor is already past column *n*, it outputs spaces to move it to column *n*+*mk*, for the smallest integer value *k* possible. *n* and *m* default to 1. Without the colon flag, *n* and *m* are in units of characters; with it, they are in units of pixels.

Note: this operation works properly *only* on streams that support the :read-cursorpos and :increment-cursorpos stream operations (see page 467). On other streams, any ~T operation simply outputs two spaces. When **format** is creating a string, ~T works by assuming that the first character in the string is at the left margin.

~@T simply outputs a space. ~*rel* T simply outputs *rel* spaces. ~*rel*, *period* T outputs *rel* spaces and then additional spaces until it reaches a column which is a multiple of *period*. If the output stream does not support :read-cursorpos then it simply outputs *rel* spaces.

~R ~R prints *arg* as a cardinal English number, e.g. four. ~:R prints *arg* as an ordinal number, e.g. fourth. ~@R prints *arg* as a Roman numeral, e.g. IV. ~:@R prints *arg* as an old Roman numeral, e.g. IIII.

~*n*R prints *arg* in radix *n*. The flags and any remaining parameters are used as for the ~D directive. Indeed, ~D is the same as ~10R. The full form here is therefore ~*radix*,*mincol*,*padchar*,*commachar*R.

~? Uses up two arguments, and processes the first one as a format control string using the second one's elements as arguments. Thus,

```
        (format nil "~? ~D" "~O ~O" '(4 20.) 9)
```

returns "4 24 9".

~@? processes the following argument as a format control string, using all the remaining arguments. Any arguments it does not use are left to be processed by the format directives following the ~@? in the original control string.

```
        (format nil "~@? ~D" "~O ~O" 4 20. 9)
```

likewise returns "4 24 9".

~→*str*~← Performs the formatting specified by *str*, with indentation on any new lines. Each time a Return is printed during the processing of *str*, it is followed by indentation sufficient to line up underneath the place where the cursor was at the beginning of *str*. For example,

```
        (format t "Foo: ~8T~→~A~←" string)
```

prints *string* with each line starting at column 8. If *string* is (string-append "This is" #\return "the string") then the output is

```
        Foo:    This is
                the string
```

~(*str*~) Performs output with case conversion. The formatting specified by *str* is done, with all the letters in the resulting output being converted to upper or lower case according to the modifiers given to the ~(command:

~(without modifiers
Converts all the letters to lower case.

~:(Converts the first letter of each word to upper case and the rest to lower case.

~@(Converts the first letter of the first word to upper case, and all other letters to lower case.

~:@(Converts all the letters to upper case.

~1(Converts the first letter of the first word to upper case and does not change anything else. If you arrange to generate all output in lower case except for letters that should be upper case regardless of context, you can use this directive when the output appears at the beginning of a sentence.

Example:
```
"~(FoO BaR~) ~:(FoO BaR~) ~@(FoO BaR~) ~:@(FoO BaR~)
~1(at the White Hart~)"
```
produces
```
foo bar Foo Bar Foo bar FOO BAR
At the White Hart
```

~[*str0*~;*str1*~;...~;*strn*~]

This is a set of alternative control strings. The alternatives (called *clauses*) are separated by ~; and the construct is terminated by ~]. For example,
```
"~[Siamese ~;Manx ~;Persian ~;Tortoise-Shell ~
   ~;Tiger ~;Yu-Shiang ~]kitty"
```
The *arg*th alternative is selected; 0 selects the first. If a prefix parameter is given (i.e. ~*n*[), then the parameter is used instead of an argument (this is useful only if the parameter is '#'). If *arg* is out of range no alternative is selected. After the selected alternative has been processed, the control string continues after the ~].

~[*str0*~;*str1*~;...~;*strn*~:;*default*~] has a default case. If the *last* ~; used to separate clauses is instead ~:;, then the last clause is an "else" clause, which is performed if no other clause is selected. For example,
```
"~[Siamese ~;Manx ~;Persian ~;Tiger ~
   ~;Yu-Shiang ~:;Bad ~] kitty"
```

~[~*tag00*,*tag01*,...;*str0*~*tag10*,*tag11*,...;*str1*...~] allows the clauses to have explicit tags. The parameters to each ~; are numeric tags for the clause which follows it. That clause is processed which has a tag matching the argument. If ~*a1*,*a2*,*b1*,*b2*,...:; (note the colon) is used, then the following clause is tagged not by single values but by ranges of values *a1* through *a2* (inclusive), *b1* through *b2*, etc. ~:; with no parameters may be used at the end to denote a default clause. For example,
```
"~[~'+,'-,'*,'//;operator ~'A,'Z,'a,'z:;letter ~
   ~'0,'9:;digit ~:;other ~]"
```

~:[*false*~;*true*~] selects the *false* control string if *arg* is nil, and selects the *true* control string otherwise.

~@[*true*~] tests the argument. If it is not nil, then the argument is not used up, but is the next one to be processed, and the one clause is processed. If it is nil, then the argument is used up, and the clause is not processed. For example,
```
(setq *print-level* nil *print-length* 5)
(format nil
        "~@[ *PRINT-LEVEL*=~D~]~@[ *PRINT-LENGTH*=~D~]"
        prinlevel prinlength)
   =>  " *PRINT-LENGTH*=5"
```

The combination of ~[and # is useful, for example, for dealing with English conventions for printing lists:

```
(setq foo "Items:~#[ none~; ~S~; ~S and ~
          ~S~:;~@{~#[~1; and~] ~S~^,~}~].")
(format nil foo)
          =>  "Items: none."
(format nil foo 'foo)
          =>  "Items: FOO."
(format nil foo 'foo 'bar)
          =>  "Items: FOO and BAR."
(format nil foo 'foo 'bar 'baz)
          =>  "Items: FOO, BAR, and BAZ."
(format nil foo 'foo 'bar 'baz 'quux)
          =>  "Items: FOO, BAR, BAZ, and QUUX."
```

~; Separates clauses in ~[and ~< constructions. It is undefined elsewhere.

~] Terminates a ~[. It is undefined elsewhere.

~{*str*~} This is an iteration construct. The argument should be a list, which is used as a set of arguments as if for a recursive call to **format**. The string *str* is used repeatedly as the control string. Each iteration can absorb as many elements of the list as it likes; if *str* uses up two arguments by itself, then two elements of the list get used up each time around the loop. If before any iteration step the list is empty, then the iteration is terminated. Also, if a prefix parameter *n* is given, then there can be at most *n* repetitions of processing of *str*. Here are some simple examples:

```
(format nil "Here it is:~{ ~S~}." '(a b c))
      => "Here it is: A B C."
(format nil "Pairs of things:~{ <~S,~S>~}." '(a 1 b 2 c 3))
      => "Pairs of things: <A,1> <B,2> <C,3>."
```

Using ~^ as well, to terminate *str* if no arguments remain, we can print a list with commas between the elements:

```
(format nil "Elements: ~{~S~^, ~}." '(a b c))
      => "Elements: A, B, C."
```

~:{*str*~} is similar, but the argument should be a list of sublists. At each repetition step one sublist is used as the set of arguments for processing *str*; on the next repetition a new sublist is used, whether or not all of the last sublist had been processed. Example:

```
(format nil "Pairs of things:~:{ <~S,~S>~}."
            '((a 1) (b 2) (c 3)))
      => "Pairs of things: <A,1> <B,2> <C,3>."
```

~@{*str*~} is similar to ~{*str*~}, but instead of using one argument which is a list, all the remaining arguments are used as the list of arguments for the iteration. Example:

```
(format nil "Pairs of things:~@{ <~S,~S>~}."
            'a 1 'b 2 'c 3)
     => "Pairs of things: <A,1> <B,2> <C,3>."
```

~:@{*str*~} combines the features of ~:{*str*~} and ~@{*str*~}. All the remaining arguments are used, and each one must be a list. On each iteration the next argument is used as a list of arguments to *str*. Example:

```
(format nil "Pairs of things:~:@{ <~S,~S>~}."
            '(a 1) '(b 2) '(c 3))
     => "Pairs of things: <A,1> <B,2> <C,3>."
```

Terminating the repetition construct with ~:} instead of ~} forces *str* to be processed at least once even if the initial list of arguments is null (however, it does not override an explicit prefix parameter of zero).

If *str* is empty, then an argument is used as *str*. It must be a string, and precedes any arguments processed by the iteration. As an example, the following are equivalent:

```
(apply #'format stream string args)
(format stream "~1{~:}" string args)
```

This uses string as a formatting string. The ~1{ says it must be processed at most once, and the ~:} says it must be processed at least once. Therefore it is processed exactly once, using args as the arguments.

As another example, the format function itself uses format-error (a routine internal to the format package) to signal error messages, which in turn uses ferror, which uses format recursively. Now format-error takes a string and arguments, just like format, but also prints some additional information: if the control string in ctl-string actually is a string (it might be a list—see below), then it prints the string and a little arrow showing where in the processing of the control string the error occurred. The variable ctl-index points one character after the place of the error.

```
(defun format-error (string &rest args)
  (if (stringp ctl-string)
      (ferror nil "~1{~:}~%~VT↓~%~3@T/"~A/"~%"
                  string args (+ ctl-index 3) ctl-string)
    (ferror nil "~1{~:}" string args)))
```

This first processes the given string and arguments using ~1{~:}, then tabs a variable amount for printing the down-arrow, then prints the control string between double-quotes. The effect is something like this:

```
(format t "The item is a ~[Foo~;Bar~;Loser~]." 'quux)
>>ERROR: The argument to the FORMAT "~[" command
              must be a number
                          ↓
   "The item is a ~[Foo~;Bar~;Loser~]."
...
```

~} Terminates a ~{. It is undefined elsewhere.

~< ~*mincol,colinc,minpad,padchar*<*text*~> justifies *text* within a field at least *mincol* wide. *text* may be divided up into segments with ~;—the spacing is evenly divided between the text segments. With no modifiers, the leftmost text segment is left justified in the field, and the rightmost text segment right justified; if there is only one, as a special case, it is right justified. The : modifier causes spacing to be introduced before the first text segment; the @ modifier causes spacing to be added after the last. *Minpad*, default 0, is the minimum number of *padchar* (default space) padding characters to be output between each segment. If the total width needed to satisfy these constraints is greater than *mincol*, then *mincol* is adjusted upwards in *colinc* increments. *colinc* defaults to 1. *mincol* defaults to 0. For example,

```
(format nil "~10<foo~;bar~>")            =>  "foo    bar"
(format nil "~10:<foo~;bar~>")           =>  "  foo  bar"
(format nil "~10:@<foo~;bar~>")          =>  "  foo bar "
(format nil "~10<foobar~>")              =>  "    foobar"
(format nil "~10:<foobar~>")             =>  "    foobar"
(format nil "~10@<foobar~>")             =>  "foobar    "
(format nil "~10:@<foobar~>")            =>  "  foobar  "
(format nil "$~10,,.'*<~3f~>" 2.5902)    =>  "$******2.59"
```

Note that *text* may include format directives. The last example illustrates how the ~< directive can be combined with the ~f directive to provide more advanced control over the formatting of numbers.

Here are some examples of the use of ~^ within a ~< construct. ~^ is explained in detail below, however the general idea is that it eliminates the segment in which it appears and all following segments if there are no more arguments.

```
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo)
        =>  "            FOO"
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar)
        =>  "FOO         BAR"
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar 'baz)
        =>  "FOO   BAR   BAZ"
```

The idea is that if a segment contains a ~^, and **format** runs out of arguments, it just stops there instead of getting an error, and it as well as the rest of the segments are ignored.

If the first clause of a ~< is terminated with ~:; instead of ~;, then it is used in a special way. All of the clauses are processed (subject to ~^, of course), but the first one is omitted in performing the spacing and padding. When the padded result has been determined, then if it will fit on the current line of output, it is output, and the text for the first clause is discarded. If, however, the padded text will not fit on the current line, then the text segment for the first clause is output before the padded text. The first clause ought to contain a carriage return (~%). The first clause is always processed, and so any arguments it refers to will be used; the decision is whether to use the resulting segment of text, not whether to process the first clause. If the ~:; has a prefix parameter *n*, then the padded text must fit on the current line with *n* character positions to spare to avoid outputting the first clause's text. For example, the control string

```
"~%;; ~{~<~%;; ~1:; ~S~>~^,~}.~%"
```

can be used to print a list of items separated by commas, without breaking items over line boundaries, and beginning each line with ';; '. The prefix parameter 1 in ~1:; accounts for the width of the comma which will follow the justified item if it is not the last element in the list, or the period if it is. If ~:; has a second prefix parameter, then it is used as the width of the line, thus overriding the natural line width of the output stream. To make the preceding example use a line width of 50, one would write

```
        "~%;; ~{~<~%;; ~1,50:; ~S~>~^,~}.~%"
```

If the second argument is not specified, then **format** sees whether the stream handles the **:size-in-characters** message. If it does, then **format** sends that message and uses the first returned value as the line length in characters. If it doesn't, **format** uses 72. as the line length.

Rather than using this complicated syntax, one can often call the function **format:print-list** (see page 495).

~> Terminates a ~<. It is undefined elsewhere.

~^ This is an escape construct. If there are no more arguments remaining to be processed, then the immediately enclosing ~{ or ~< construct is terminated. If there is no such enclosing construct, then the entire formatting operation is terminated. In the ~< case, the formatting *is* performed, but no more segments are processed before doing the justification. The ~^ should appear only at the *beginning* of a ~< clause, because it aborts the entire clause. ~^ may appear anywhere in a ~{ construct.

If a prefix parameter is given, then termination occurs if the parameter is zero. (Hence ~^ is the same as ~#^.) If two parameters are given, termination occurs if they are equal. If three are given, termination occurs if the second is between the other two in ascending order. Of course, this is useless if all the prefix parameters are constants; at least one of them should be a # or a V parameter.

If ~^ is used within a ~:{ construct, then it merely terminates the current iteration step (because in the standard case it tests for remaining arguments of the current step only); the next iteration step commences immediately. To terminate the entire iteration process, use ~:^.

~Q An escape to arbitrary user-supplied code. *arg* is called as a function; its arguments are the prefix parameters to ~Q, if any. *args* can be passed to the function by using the V prefix parameter. The function may output to ***standard-output*** and may look at the variables **format:colon-flag** and **format:atsign-flag**, which are **t** or **nil** to reflect the : and @ modifiers on the ~Q. For example,

```
        (format t "~VQ" foo bar)
```

is a fancy way to say

```
        (funcall bar foo)
```

and discard the value. Note the reversal of order; the V is processed before the Q.

~\ This begins a directive whose name is longer than one character. The name is terminated by another \ character. The following directives have names longer than one character and make use of the ~\ mechanism as part of their operation.

~\lozenged-string\
: This is like ~A except when output is to a window, in which case the argument is printed in a small font inside a lozenge.

~\lozenged-character\
: This is like ~C except when output is to a window, in which case the argument is printed in a small font inside a lozenge if it has a character name, even if it is a formatting character or graphic character.

~\date\ This expects an argument that is a universal time (see page 776), and prints it as a date and time using time:print-universal-date.
Example:

```
        (format t "It is now ~\date\" (get-universal-time))
```

prints

```
        It is now Saturday the fourth of December, 1982; 4:00:32 am
```

~\time\ This expects an argument that is a universal time (see page 776), and prints it in a brief format using time:print-universal-time.
Example:

```
        (format t "It is now ~\time\" (get-universal-time))
```

prints

```
        It is now 12/04/82 04:01:38
```

~\datime\
: This prints the current time and date. It does not use an argument. It is equivalent to using the ~\time\ directive with (time:get-universal-time) as argument.

~\time-interval\
: This prints a time interval measured in seconds using the function time:print-interval-or-never.
Example:

```
        (format t "It took ~\time-interval\." 3601.)
```

prints

```
        It took 1 hour 1 second.
```

You can define your own directives. How to do this is not documented here; read the code. Names of user-defined directives longer than one character may be used if they are enclosed in backslashes (e.g. ~4,3\GRAPH\).

(Note: format also allows *control-string* to be a list. If the list is a list of one element, which is a string, the string is simply printed. This is for the use of the format:outfmt function below. The old feature wherein a more complex interpretation of this list was possible is now considered obsolete; use format:output if you like using lists.)

A condition instance can also be used as the *control-string*. Then the :report operation is used to print the condition instance; any other arguments are ignored. This way, you can pass a condition instance directly to any function that normally expects a format string and arguments.

format:print-list *destination element-format list* &optional *separator start-line tilde-brace-options*

This function provides a simpler interface for the specific purpose of printing comma-separated lists with no list element split across two lines; see the description of the ~:; directive (page 492) to see the more complex way to do this within **format**. *destination* tells where to send the output; it can be **t**, **nil**, a **string-nconc**'able string, or a stream, as with **format**. *element-format* is a **format** control-string that tells how to print each element of *list*; it is used as the body of a ~{...~} construct. *separator*, which defaults to ", " (comma, space) is a string which goes after each element except the last. **format** control commands are not recommended in *separator*. *start-line*, which defaults to three spaces, is a **format** control-string that is used as a prefix at the beginning of each line of output, except the first. **format** control commands are allowed in *separator*, but they should not swallow arguments from *list*. *tilde-brace-options* is a string inserted before the opening '{'; it defaults to the null string, but allows you to insert colon and/or atsign. The line-width of the stream is computed the same way that the ~:; command computes it; it is not possible to override the natural line-width of the stream.

22.4.2 The Output Subsystem

The formatting functions associated with the **format:output** subsystem allow you to do formatted output using Lisp-style control structure. Instead of a directive in a **format** control string, there is one formatting function for each kind of formatted output.

The calling conventions of most of the formatting functions are similar. The first argument is usually the datum to be output. The second argument is usually the minimum number of columns to use. The remaining arguments are keyword arguments.

Most of the functions accept the keyword arguments *padchar*, *minpad* and *tab-period*. *padchar* is a character to use for padding. *minpad* is a minimum number of padding characters to output after the data. *tab-period* is the distance between allowable places to stop padding. To make the meaning of *tab-period* clearer, if the value of *tab-period* is 5, if the minimum size of the field is 10, and if the value of *minpad* is 2, then a datum that takes 9 characters is padded out to 15 characters. The requirement to use at least two characters of padding means it can't fit into 10 characters, and the *tab-period* of 5 means the next allowable stopping place is at 10+5 characters. The default values for *minpad* and *tab-period*, if they are not specified, are zero and one. The default value for *padchar* is space.

The formatting functions always output to ***standard-output*** and do not require an argument to specify the stream. The macro **format:output** allows you to specify the stream or a string, just as **format** does, and also makes it convenient to concatenate constant and variable output.

format:output *stream string-or-form...* *Macro*

Makes it convenient to intersperse arbitrary output operations with printing of constant strings. ***standard-output*** is bound to *stream*, and each *string-or-form* is processed in succession from left to right. If it is a string, it is printed; otherwise it is a form, which is evaluated for effect. Presumably the forms will send output to ***standard-output***.

If *stream* is written as **nil**, then the output is put into a string which is returned by **format:output**. If *stream* is written as **t**, then the output goes to the prevailing value of ***standard-output***. Otherwise *stream* is a form, which must evaluate to a stream.

Here is an example:

```
(format:output t "FOO is " (prin1 foo) " now." (terpri))
```

Because **format:output** is a macro, what matters about *stream* is not whether it *evaluates* to **t** or **nil**, but whether it is actually written as **t** or **nil**.

format:outfmt *string-or-form...* *Macro*

Some system functions ask for a **format** control string and arguments, to be printed later. If you wish to generate the output using the formatted output functions, you can use **format:outfmt**, which produces a control argument that will eventually make **format** print the desired output (this is a list whose one element is a string containing the output). A call to **format:outfmt** can be used as the second argument to **ferror**, for example:

```
(ferror nil (format:outfmt "Foo is " (format:onum foo)
                           " which is too large"))
```

format:onum *number* &optional *radix minwidth* &key *padchar minpad tab-period signed commas*

Outputs *number* in base *radix*, padding to at least *minwidth* columns and obeying the other padding options specified as described above.

radix can be a number, or it can be :roman, :english, or :ordinal. The default *radix* is 10. (decimal).

If *signed* is non-nil, a + sign is printed if the number is positive. If *commas* is non-nil, a comma is printed every third digit in the customary way. These arguments are meaningful only with numeric radices.

format:ofloat *number* &optional *n-digits force-exponential-notation minwidth* &key *padchar minpad tab-period*

Outputs *number* as a floating point number using *n-digits* digits. If *force-exponential-notation* is non-nil, then an exponent is always used. *minwidth* and the padding options are interpreted as usual.

format:ostring *string* &optional *minwidth* &key *padchar minpad tab-period right-justify*

Outputs *string*, padding to at least *minwidth* columns if *minwidth* is not nil, and obeying the other padding options specified as described above.

Normally the data are left justified; any padding follows the data. If *right-justify* is non-nil, the padding comes before the data. The amount of padding is not affected.

The argument need not really be a string. Any Lisp object is allowed, and it is output with princ.

format:oprint *object* &optional *minwidth* &key *padchar minpad tab-period right-justify*

Prints *object*, any Lisp object, padding to at least *minwidth* columns if *minwidth* is not nil, and obeying the padding options specified as described above.

Normally the data are left justified; any padding follows the data. If *right-justify* is non-nil, the padding comes before the data. The amount of padding is not affected.

The printing of the object is done with prin1.

format:ochar *character* &optional *style top-explain minwidth* &key *padchar minpad tab-period*

Outputs *character* in one of three styles, selected by the *style* argument. *minwidth* and the padding options control padding as usual.

:read or nil	The character is printed using #\ or #/ so that it could be read back in.
:editor	Output is in the style of 'Meta-Rubout'. Non-printing characters, and the two printing characters Space and Altmode, are represented by their names. Other printing characters are printed directly.

:brief   Brief prefixes such as 'C-' and 'M-' are used, rather than 'Control-' or 'Meta-'. Also, character names are used only if there are meta bits present.

:lozenged   The output is the same as that of the :editor style, but If the character is not a graphic character or if it has meta bits, and the stream supports the :display-lozenged-string operation, that operation is used instead of :string-out to print the text. On windows this operation puts the character name inside a lozenge.

:sail   'α', '', etc. are used to represent Control and Meta, and shorter names for characters are also used when possible. See section 10.1.1, page 205.

top-explain is useful with the :editor, :brief and :sail styles. It says that any character that has to be typed using the Top or Greek keys should be followed by an explanation of how to type it. For example: '→ (Top-K)' or 'α (Greek-a)'.

format:tab *mincol* &key *padchar minpad tab-period terpri unit*

Outputs padding at least until column *mincol*. It is the only formatting function that bases its actions on the actual cursor position rather than the width of what is being output. The padding options *padchar*, *minpad*, and *tab-period* are obeyed. Thus, at least the *minpad* number of padding characters are output even if that goes past *mincol*, and once past *mincol*, padding can only stop at a multiple of *tab-period* characters past *mincol*.

In addition, if the *terpri* option is t, then if column *mincol* is passed, format:tab starts a new line and indents it to *mincol*.

The *unit* option specifies the units of horizontal position. The default is to count in units of characters. If *unit* is specified as :pixel, then the computation (and the argument *mincol* and the *minpad* and *tab-period* options) are in units of pixels.

format:pad (*minwidth* &key *padchar minpad tab-period...*) *body...*   *Macro*

format:pad is used for printing several items in a fixed amount of horizontal space, padding between them to use up any excess space. Each of the *body* forms prints one item. The padding goes between items. The entire format:pad always uses at least *minwidth* columns; any columns that the items don't need are distributed as padding between the items. If that isn't enough space, then more space is allocated in units controlled by the *tab-period* option until there is enough space. If it's more than enough, the excess is used as padding.

If the *minpad* option is specified, then at least that many pad characters must go between each pair of items.

Padding goes only between items. If you want to treat several actual pieces of output as one item, put a progn around them. If you want padding before the first item or after the last, as well as between the items, include a dummy item nil at the beginning or the end.

If there is only one item, it is right justified. One item followed by nil is left-justified. One item preceded and followed by nil is centered. Therefore, format:pad can be used to provide the usual padding options for a function that does not provide them itself.

format:plural *number singular* &optional *plural*

Outputs either the singular or the plural form of a word depending on the value of *number*. The singular is used if and only if *number* is 1. *singular* specifies the singular form of the word. string-pluralize is used to compute the plural, unless *plural* is explicitly specified.

It is often useful for *number* to be a value returned by format:onum, which returns its argument. For example:

```
(format:plural (format:onum n-frobs) " frob")
```

prints "1 frob" or "2 frobs".

format:breakline *linel print-if-terpri print-always...* *Macro*

Goes to the next line if there is not enough room for something to be output on the current line. The *print-always* forms print the text which is supposed to fit on the line. *linel* is the column before which the text must end. If it doesn't end before that column, then format:breakline moves to the next line and executes the *print-if-terpri* form before doing the *print-always* forms.

Constant strings are allowed as well as forms for *print-if-terpri* and *print-always*. A constant string is just printed.

To go to a new line unconditionally, simply call terpri.

Here is an example that prints the elements of a list, separated by commas, breaking lines between elements when necessary.

```
(defun pcl (list linel)
  (do ((l list (cdr l))) ((null l))
    (format:breakline linel "  "
      (princ (car l))
      (and (cdr l) (princ ", ")))))
```

22.5 Rubout Handling

The rubout handler is a feature of all interactive streams, that is, streams that connect to terminals. Its purpose is to allow the user to edit minor mistakes made during type-in. At the same time, it is not supposed to get in the way; input is to be seen by Lisp as soon as a syntactically complete form has been typed. The definition of 'syntactically complete form' depends on the function that is reading from the stream; for **read**, it is a Lisp expression.

Some interactive streams ('editing Lisp listeners') have a rubout handler that allows input to be edited with the full power of the ZWEI editor. (ZWEI is the general editor implementation on which Zmacs and ZMail are based.) Most windows have a rubout handler that apes ZWEI, implementing about twenty common ZWEI commands. The cold load stream has a simple rubout handler that allows just rubbing out of single characters, and a few simple commands like clearing the screen and erasing the entire input typed so far. All three kinds of rubout handler use the same protocol, which is described in this section. We also say a little about the most common of the three rubout handlers.
[Eventually some version of ZWEI will be used for all streams except the cold load stream]

The tricky thing about the rubout handler is the need for it to figure out when you are all done. The idea of a rubout handler is that you can type in characters, and they are saved up in a buffer so that if you change your mind, you can rub them out and type different characters. However, at some point, the rubout handler has to decide that the time has come to stop putting characters into the buffer and to let the function parsing the input, such as **read**, return. This is called *activation*. The right time to activate depends on the function calling the rubout handler, and may be very complicated (if the function is **read**, figuring out when one Lisp expression has been typed requires knowledge of all the various printed representations, what all currently-defined reader macros do, and so on). Rubout handlers should not have to know how to parse the characters in the buffer to figure out what the caller is reading and when to activate; only the caller should have to know this. The rubout handler interface is organized so that the calling function can do all the parsing, while the rubout handler does all the handling of editing commands, and the two are kept completely separate.

The basic way that the rubout handler works is as follows. When an input function that reads characters from a stream, such as **read** or **readline** (but not **tyi**), is invoked with a stream which has **:rubout-handler** in its **:which-operations** list, that function "enters" the rubout handler. It then goes ahead :tyi'ing characters from the stream. Because control is inside the rubout handler, the stream echoes these characters so the user can see what he is typing. (Normally echoing is considered to be a higher-level function outside of the province of streams, but when the higher-level function tells the stream to enter the rubout handler it is also handing it the responsibility for echoing.) The rubout handler is also saving all these characters in a buffer, for reasons disclosed in the following paragraph. When the parsing function decides it has enough input, it returns and control "leaves" the rubout handler. This is the easy case.

If the user types a rubout, a **throw** is done out of all recursive levels of **read**, reader macros, and so forth, back to the point where the rubout handler was entered. Also the rubout is echoed by erasing from the screen the character which was rubbed out. Now the **read** is tried over again, re-reading all the characters that have not been rubbed out, not echoing them this time. When the saved characters have been exhausted, additional input is read from the user in the usual fashion.

The effect of this is a complete separation of the functions of rubout handling and parsing, while at the same time mingling the execution of these two functions in such a way that input is always activated at just the right time. It does mean that the parsing function (in the usual case, **read** and all macro-character definitions) must be prepared to be thrown through at any time and should not have non-trivial side-effects, since it may be called multiple times.

If an error occurs while inside the rubout handler, the error message is printed and then additional characters are read. When the user types a rubout, it rubs out the error message as well as the character that caused the error. The user can then proceed to type the corrected expression; the input will be reparsed from the beginning in the usual fashion.

The rubout handler based on the ZWEI editor interprets control characters in the usual ZWEI way: as editing commands, allowing you to edit your buffered input.

The common rubout handler also recognizes a subset of the editor commands, including **Rubout**, **Control-F** and **Meta-F** and others. Typing **Help** while in the rubout handler displays a list of the commands. The kill and yank commands in the rubout handler use the same kill ring as the editor, so you can kill an expression in the editor and yank it back into a rubout handler with **Control-Y**, or kill an expression in the rubout handler with **Control-K** or **Clear-input** and yank it back in the editor. The rubout processor also keeps a ring buffer of most recent input strings (a separate ring for each stream), and the commands **Control-C** and **Meta-C** retrieve from this ring just as **Control-Y** and **Meta-Y** do from the kill ring.

When not inside the rubout handler, and when typing at a program that uses control characters for its own purposes, control characters are treated the same as ordinary characters.

Some programs such as the debugger allow the user to type either a control character or an expression. In such programs, you are really not inside the rubout handler unless you have typed the beginning of an expression. When the input buffer is empty, a control character is treated as a command for the program (such as, **Control-C** to continue in the debugger); when there is text in the rubout handler buffer, the same character is treated as a rubout handler command. Another consequence of this is that the message you get by typing **Help** varies, being either the rubout handler's documentation or the debugger's documentation.

To write a parsing function that reads with rubout handling, use **with-input-editing**.

with-input-editing (*stream options*) *body...* *Macro*

Invokes the rubout handler on *stream*, if *stream* supports it, and then executes *body*. *body* is executed in any case, within the rubout handler if possible. **rubout-handler** is non-**nil** while in *body* if rubout handling is in use.

options are used as the rubout handler options. If already within an invocation of the rubout handler, *options* are appended to the front of the options already in effect. This happens if a function which reads input using **with-input-editing**, such as **read** or **readline**, is called from the body of another **with-input-editing**. The **:norecursive** option can be used to cause the outer set of options to be completely ignored even when not overridden by new ones.

body's values are returned by with-input-editing. *body* ought to read input from *stream* and return a Lisp object that represents the input. It should have no nontrivial side effects aside from reading input from *stream* structure, as it may be aborted at any time it reads input and may be executed over and over.

If the :full-rubout option is specified, and the user types some input and rubs it all out, the with-input-editing form returns immediately. See :full-rubout, below.

If a preemptive command is input by the user, with-input-editing returns immediately with the values being as specified below under the :command and :preemptable options. *body* is aborted from its call to the :tyi operation, and the input read so far remains in the rubout handler editing buffer to be read later.

rubout-handler *Variable*
If control is inside the rubout handler in this process, the value is the stream on which rubout handling is being done. Otherwise, the value is nil.

:rubout-handler *options function* &rest *args* *Operation on streams*
Invokes the rubout handler on the stream, with *options* as the options, and parses by applying *function* to *args*. with-input-editing uses this operation.

:read-bp *Operation on streams*
This operation may be used only from within the code for parsing input from this stream inside the rubout handler. It returns the index within the rubout handler buffer which parsing has reached.

:force-rescan *Operation on streams*
This operation may be used only from within the code for parsing input from this stream inside the rubout handler. It causes parsing to start again immediately from the beginning of the buffer.

:rescanning-p *Operation on streams*
This operation may be used only from within the code for parsing input from this stream inside the rubout handler. It returns t if parsing is now being done on input already in the buffer, nil if parsing has used up all the buffered input and the next character parsed will come from the keyboard.

Each option in the list of rubout handler options consists of a list whose first element is a keyword and whose remaining elements are the arguments of that keyword. Note that this is not the same format as the arguments to a typical function that takes keyword arguments; rather this is an alist of options. The standard options are:

(:activation *fn args...*)
Activate if certain characters are typed in. When the user types an activation character, the rubout handler moves the editing pointer immediately to the end of the buffer and inserts the activation character. This immediately causes the parsing function to begin rescanning the input.

fn is used to test characters for being activators. It is called with an input character as the first arg (possibly a fixnum, possibly a character object) and *args* as additional args. If *fn* returns non-nil, the character is an activation. *fn* is not called for blips.

After the parsing function has read the entire contents of the buffer, it sees the activation character as a blip (:activation *char numeric-arg*) where *char* is the character that activated and *numeric-arg* is the numeric arg that was pending for the next rubout handler command. Normally the parsing function will return at this point. Then the activation character does not echo. But if the parsing function continues to read input, the activation character echoes and is inserted in the buffer.

(:do-not-echo *chars...*)
Poor man's activation characters. Like :activation except that the characters that should activate are listed explicitly, and the character itself is returned to the parsing function rather than a blip.

(:full-rubout *val*)
If the user rubs out all the characters he typed, then control is returned from the rubout handler immediately. Two values are returned; the first is nil and the second is *val*. (If the user doesn't rub out all the characters, then the rubout handler propagates multiple values back from the function that it calls, as usual.) In the absence of this option, the rubout handler would simply wait for more characters to be typed in and would ignore any additional rubouts.

This is how the debugger knows to remove Eval: from the screen if you type the beginning of a form and rub it all out.

(:pass-through *char1 char2...*)
The characters *char1*, *char2*, etc. are not to be treated as special by the rubout handler. They are read as input by the parsing function. If the parsing function does not return, they can be rubbed out. This works only for characters with no modifier bits.

(:preemptable *value*)
Makes all blips read as input by the rubout handler act as preemptive commands. If this option is specified, the rubout handler returns immediately when it reads a blip. It returns two values: the blip that was read, and *value*. The parsing function is not allowed to finish parsing up to a delimiter; instead, any buffered input remains in the buffer for the next time input is done. In the mean time, the preemptive command character can be processed by the command loop.

While this applies to all blips, the blips which it is probably intended for are mouse blips.

(:command *fn args...*)
Makes certain characters preemptive commands. A preemptive command returns instantly to the caller of the :rubout-handler operation, regardless

of the input in the buffer. It returns two values: a list (:command *char numeric-arg*) and the keyword :command. The parsing function is not allowed to finish parsing up to a delimiter; instead, any buffered input remains in the buffer for the next time input is done. In the mean time, the preemptive command character can be processed by the command loop.

The test for whether a character should be a preemptive command is done using *fn* and *args* just as in :activation.

(:editing-command (*char doc*)...)

Defines editing commands to be executed by the parsing function itself. This is how qsend implements the Control-Meta-Y command. Each *char* is such a command, and *doc* says what it does. (*doc* is printed out by the rubout handler's Help command.) If any of these characters is read by the rubout handler, it is returned immediately to the parsing function regardless of where the editing pointer is in the buffer. (Normal inserted text is not returned immediately when read unless the editing pointer is at the end of the buffer.)

The parsing function should not regard these characters as part of the input. There are two reasonable things that the parsing function can do when it receives one of the editing command characters: print some output, or force some input.

If it prints output, it should invoke the :refresh-rubout-handler operation afterward before the next :tyi. This causes the rubout handler to redisplay so that the input being edited appears after the output that was done.

If the parsing function forces input, the input is read by the rubout handler. This can be used to modify the buffered input. qsend's Control-Meta-Y command works by forcing the yanked text as input. There is no way to act directly on the buffered input because different implementations of the rubout handler store it in different ways.

(:prompt *function*)
(:reprompt *function*)

When it is time for the user to be prompted, *function* is called with two arguments. The first is a stream it may print on; the second is the character which caused the need for prompting, e.g. #\clear-input or #\clear-screen, or nil if the rubout handler was just entered.

The difference between :prompt and :reprompt is that the latter does not call the prompt function when the rubout handler is first entered, but only when the input is redisplayed (e.g. after a screen clear). If both options are specified then :reprompt overrides :prompt except when the rubout handler is first entered.

function may also be a string. Then it is simply printed.

If the rubout handler is exited with an empty buffer due to the **:full-rubout** option, whatever prompt was printed is erased.

(:initial-input *string*)
: Pretends that the user typed *string*. When the rubout handler is entered, *string* is typed out. The user can input more characters or rub out characters from it.

(:initial-input-index *index*)
: Positions the editing pointer initially *index* characters into the initial input string. Used only in company with with **:initial-input**.

(:no-input-save t)
: Don't save this batch of input in the input history when it is done. For example, **yes-or-no-p** specifies this option.

(:norecursive t)
: If this invocation of the rubout handler is within another one, the options specified in the previous call should be completely ignored during this one. Normally, individual options specified this time override the previous settings for the same options, but any of the previous options not individually overridden are still in effect.

Rubout handlers handle the condition **sys:parse-error** if it is signaled by the parsing function. The handling consists of printing the error message, waiting for the user to rub out, erasing the error message, and parsing the input again. All errors signaled by a parsing function that signify that the user's input was syntactically invalid should have this condition name. For example, the errors **read** signals have condition name **sys:parse-error** since it is is a consequence of **sys:read-error**.

sys:parse-error (error) *Condition*

The condition name for syntax errors in input being parsed.

The compiler handles **sys:parse-error** by proceeding with proceed-type **:no-action**. All signalers of **sys:parse-error** should offer this proceed type, and respond to its use by continuing to parse, ignoring the invalid input.

sys:parse-ferror *format-string* &rest *args*

Signals a **sys:parse-error** error, using *format-string* and *args* to print the error message. The proceed-type **:no-action** is provided, and if a handler uses it, this function returns **nil**.