24. INPUT/OUTPUT FUNCTIONS

This chapter describes the standard I/O functions used for reading and printing characters and Interlisp expressions on files and other streams. First, the primitive input functions are presented, then the output functions, then functions for random-access operations (such as searching a file for a given stream, or changing the "next-character" pointer to a position in a file). Next, the PRINTOUT statement is documented (see below), which provides an easy way to write complex output operations. Finally, read tables, used to parse characters as Interlisp expressions, are documented.

Specifying Streams for Input/Output Functions

Most of the input/output functions in Interlisp-D have an argument named *STREAM* or *FILE*, specifying on which open stream the function's action should occur (the name *FILE* is used in older functions that predate the concept of stream; the two should, however, be treated synonymously). The value of this argument should be one of the following:

a stream — An object of type STREAM, as returned by OPENSTREAM (Chapter 23) or other stream-producing functions, is always the most precise and efficient way to designate a stream argument.

T — The litatom T designates the terminal input or output stream of the currently running process, controlling input from the keyboard and output to the display screen. For functions where the direction (input or output) is ambiguous, T is taken to designate the terminal output stream. The T streams are always open; they cannot be closed.

The terminal output stream can be set to a given window or display stream by using TTYDISPLAYSTREAM (Chapter 28). The terminal input stream cannot be changed. For more information on terminal I/O, see Chapter 30.

NIL — The litatom NIL designates the "primary" input or output stream. These streams are initially the same as the terminal input/output streams, but they can be changed by using the functions INPUT and OUTPUT.

For functions where the direction (input or output) is ambiguous, e.g., GETFILEPTR, the argument NIL is taken to mean the primary input stream, if that stream is not identical to the terminal input stream, else the primary output stream.

a window — Uses the display stream of the window . Valid for output only.

a file name — As of this writing, the name of an open file (as a litatom) can be used as a stream argument. However, there are inefficiencies and possible

future incompatibilities associated with doing so. See Chapter 24 for details.

(**GETSTREAM** *FILE ACCESS*) [Function]

Coerces the argument *FILE* to a stream by the above rules. If *ACCESS* is INPUT, OUTPUT, or BOTH, produces the stream designated by *FILE* that is open for *ACCESS*. If *ACCESS*=NIL, returns a stream for *FILE* open for any kind of input/output (see the list above for the ambiguous cases). If *FILE* does not designate a stream open in the specified mode, causes an error, FILE NOT OPEN.

(**STREAMP** *X*) [Function]

Returns *X* if *X* is a STREAM, otherwise NIL.

Input Functions

While the functions described below can take input from any stream, some special actions occur when the input is from the terminal (the T input stream, see above). When reading from the terminal, the input is buffered a line at a time, unless buffering has been inhibited by CONTROL (Chapter 30) or the input is being read by READC or PEEKC. Using specified editing characters, you can erase a character at a time, a word at a time, or the whole line. The keys that perform these editing functions are assignable via SETSYNTAX, with the initial settings chosen to be those most natural for the given operating system. In Interlisp-D, the initial settings are as follows: characters are deleted one at a time by Backspace; words are erased by control-W; the whole line is erased by Control-Q.

On the Interlisp-D display, deleting a character or a line causes the characters to be physically erased from the screen. In Interlisp-10, the deleting action can be modified for various types of display terminals by using DELETECONTROL (Chapter 30).

Unless otherwise indicated, when the end of file is encountered while reading from a file, all input functions generate an error, END OF FILE. Note that this does not close the input file. The ENDOFSTREAMOP stream attribute (Chapter 24) is useful for changing the behavior at end of file.

Most input functions have a *RDTBL* argument, which specifies the read table to be used for input. Unless otherwise specified, if *RDTBL* is NIL, the primary read table is used.

If the *FILE* or *STREAM* argument to an input function is NIL, the primary input stream is used.

(**INPUT** *FILE*) [Function]

Sets *FILE* as the primary input stream; returns the old primary input stream. *FILE* must be open for input.

(INPUT) returns the current primary input stream, which is not changed.

Note: If the primary input stream is set to a file, the file's full name, rather than the stream itself, is returned. See discussion in Chapter 24.

(**READ** *FILE RDTBL FLG*) [Function]

Reads one expression from *FILE*. Atoms are delimited by the break and separator characters as defined in *RDTBL*. To include a break or separator character in an atom, the character must be preceded by the character %, e.g., AB%(C is the atom AB(C, %% is the atom %, %*control-K* is the atom Control-K. For input from the terminal, an atom containing an interrupt character can be input by typing instead the corresponding alphabetic character preceded by Control-V, e.g., ^VD for Control-D.

Strings are delimited by double quotes. To input a string containing a double quote or a %, precede it by %, e.g., "AB%"C" is the string AB"C. Note that % can always be typed even if next character is not "special", e.g., %A%B%C is read as ABC.

If an atom is interpretable as a number, READ creates a number, e.g., 1E3 reads as a floating point number, 1D3 as a literal atom, 1.0 as a number, 1,0 as a literal atom, etc. An integer can be input in a non-decimal radix by using syntax such as 123Q, |b10101, |5r1234 (see Chapter 7). The function RADIX, sets the radix used to print integers.

When reading from the terminal, all input is line-buffered to enable the action of the backspacing control characters, unless inhibited by CONTROL (Chapter 30). Thus no characters are actually seen by the program until a carriage-return (actually the character with terminal syntax class EOL, see Chapter 30), is typed. However, for reading by READ, when a matching right parenthesis is encountered, the effect is the same as though a carriage-return were typed, i.e., the characters are transmitted. To indicate this, Interlisp also prints a carriage-return line-feed on the terminal. The line buffer is also transmitted to READ whenever an IMMEDIATE read macro character is typed (see below).

FLG=T suppresses the carriage-return normally typed by READ following a matching right parenthesis. (However, the characters are still given to READ; i.e., you do not have to type the carriage-return.)

(**RATOM** *FILE RDTBL*) [Function]

Reads in one atom from *FILE*. Separation of atoms is defined by *RDTBL*. % is also defined for RATOM, and the remarks concerning line-buffering and editing control characters also apply.

If the characters comprising the atom would normally be interpreted as a number by READ, that number is returned by RATOM. Note however that RATOM takes no special action for " whether or not it is a break character, i.e., RATOM never makes a string.

(**RSTRING** *FILE RDTBL*) [Function]

Reads characters from *FILE* up to, but not including, the next break or separator character, and returns them as a string. Backspace, Control-W, Control-Q, Control-V, and % have the same effect as with READ.

Note that the break or separator character that terminates a call to RATOM or RSTRING is *not* read by that call, but remains in the buffer to become the first character seen by the next reading function that is called. If that function is RSTRING, it will return the null string. This is a common source of program bugs.

(**RATOMS** *A FILE RDTBL*) [Function]

Calls RATOM repeatedly until the atom *A* is read. Returns a list of the atoms read, not including *A*.

(**RATEST** *FLG*) [Function]

If *FLG* = T, RATEST returns T if a separator was encountered immediately prior to the atom returned by the last RATOM or READ, NIL otherwise.

If *FLG* = NIL, RATEST returns T if last atom read by RATOM or READ was a break character, NIL otherwise.

If *FLG* = 1, RATEST returns T if last atom read (by READ or RATOM) contained a % used to quote the next character (as in %[or %A%B%C), NIL otherwise.

(**READC** *FILE RDTBL*) [Function]

Reads and returns the next character, including %, ", etc, i.e., is not affected by break or separator characters. The action of READC is subject to line-buffering, i.e., READC does not return a value until the line has been terminated even if a character has been typed. Thus, the editing control characters have their usual effect. *RDTBL* does not directly affect the value returned, but is used as usual in line-buffering, e.g., determining when input has been terminated. If (CONTROL T) has been executed (Chapter 30), defeating line-buffering, the *RDTBL* argument is irrelevant, and READC returns a value as soon as a character is typed (even if the character typed is one of the editing characters, which ordinarily would never be seen in the input buffer).

(**PEEKC** *FILE*) [Function]

Returns the next character, but does not actually read it and remove it from the buffer. If reading from the terminal, the character is echoed as soon as PEEKC reads it, even though it is then "put back" into the system buffer, where Backspace, Control-W, etc. could change it. Thus it is possible for the value returned by PEEKC to "disagree" in the first character with a subsequent READ.

(**LASTC** *FILE*) [Function]

Returns the last character read from *FILE*. LASTC can return an incorrect result when called immediatley following a PEEKC on a file that contains run-coded NS characters.

(**READCCODE** *FILE RDTBL*) [Function]

Returns the next character *code* from *STREAM*; thus, this operation is equivalent to, but more efficient than, (CHCON1 (READC *FILE RDTBL*)).

(**PEEKCCODE** *FILE*) [Function]

Returns, without consuming, the next character *code* from *STREAM*; thus, this operation is equivalent to, but more efficient than, (CHCON1 (PEEKC *FILE*)).

(**BIN** *STREAM*) [Function]

Returns the next byte from *STREAM*. This operation is useful for reading streams of binary, rather than character, data.

Note: BIN is similar to READCCODE, except that BIN always reads a single byte, whereas READCCODE reads a "character" that can consist of more than one byte, depending on the character and its encoding.

READ, RATOM, RATOMS, PEEKC, READC all wait for input if there is none. The only way to test whether or not there is input is to use READP:

(**READP** *FILE FLG*) [Function]

Returns T if there is anything in the input buffer of *FILE*, NIL otherwise. This operation is only interesting for streams whose source of data is dynamic, e.g., the terminal or a byte stream over a network; for other streams, such as to files, (READP *FILE*) is equivalent to (NOT (EOFP *FILE*)).

Note that because of line-buffering, READP may return T, indicating there is input in the buffer, but READ may still have to wait.

Frequently, the terminal's input buffer contains a single EOL character left over from a previous input. For most applications, this situation wants to be treated as though the buffer were empty, and so READP returns NIL in this case. However, if *FLG*=T, READP returns T if there is *any* character in the input buffer, including a single EOL. *FLG* is ignored for streams other than the terminal.

(**EOFP** *FILE*) [Function]

Returns true if *FILE* is at "end of file", i.e., the next call to an input function would cause an END OF FILE error; NIL otherwise. For randomly accessible files, this can also be thought of as the file pointer pointing beyond the last byte of the file. *FILE* must be open for (at least) input, or an error is generated, FILE NOT OPEN.

Note that EOFP can return NIL and yet the next call to READ might still cause an END OF FILE error, because the only characters remaining in the input were separators or otherwise constituted an incomplete expression. The function SKIPSEPRS is sometimes more useful as a way of detecting end of file when it is known that all the expressions in the file are well formed.

(**WAITFORINPUT** *FILE*) [Function]

Waits until input is available from *FILE* or from the terminal, i.e. from T. WAITFORINPUT is functionally equivalent to (until (OR (READP T) (READP FILE)) do NIL),

except that it does not use up machine cycles while waiting. Returns the device for which input is now available, i.e. *FILE* or T.

FILE can also be an integer, in which case WAITFORINPUT waits until there is input available from the terminal, or until *FILE* milliseconds have elapsed. Value is T if input is now available, NIL in the case that WAITFORINPUT timed out.

(**SKREAD** *FILE REREADSTRING RDTBL*) [Function]

"Skip Read". SKREAD consumes characters from *FILE* as if one call to READ had been performed, without paying the storage and compute cost to really read in the structure. *REREADSTRING* is for the case where the caller has already performed some READC's and RATOM's before deciding to skip this expression. In this case, *REREADSTRING* should be the material already read (as a string), and SKREAD operates as though it had seen that material first, thus setting up its parenthesis count, double-quote count, etc.

The read table *RDTBL* is used for reading from *FILE*. If *RDTBL* is NIL, it defaults to the value of FILERDTBL. SKREAD may have difficulties if unusual read macros are defined in *RDTBL*. SKREAD does not recognize read macro characters in *REREADSTRING*, nor SPLICE or INFIX read macros. This is only a problem if the read macros are defined to parse subsequent input in the stream that does not follow the normal parenthesis and string-quote conventions.

SKREAD returns %) if the read terminated on an unbalanced closing parenthesis; %] if the read terminated on an unbalanced %], i.e., one which also would have closed any extant open left parentheses; otherwise NIL.

(**SKIPSEPRS** *FILE RDTBL*) [Function]

Consumes characters from *FILE* until it encounters a non-separator character (as defined by *RDTBL*). SKIPSEPRS returns, but does not consume, the terminating character, so that the next call to READC would return the same character. If no non-separator character is found before the end of file is reached, SKIPSEPRS returns NIL and leaves the stream at end of file. This function is useful for skipping over "white space" when scanning a stream character by character, or for detecting end of file when reading expressions from a stream with no pre-arranged terminating expression.

Output Functions

Unless otherwise specified by DEFPRINT, pointers other than lists, strings, atoms, or numbers, are printed in the form {*DATATYPE*} followed by the octal representation of the address of the pointer (regardless of radix). For example, an array pointer might print as {ARRAYP}#43,2760. This printed representation is for compactness of display on your terminal, and will *not* read back in correctly; if the form above is read, it will produce the litatom {ARRAYP}#43,2760.

Note: The term "end-of-line" appearing in the description of an output function means the character or characters used to terminate a line in the file system being used

by the given implementation of Interlisp. For example, in Interlisp-D end-of-line is indicated by the character carriage-return.

Some of the functions described below have a *RDTBL* argument, which specifies the read table to be used for output. If *RDTBL* is NIL, the primary read table is used.

Most of the functions described below have an argument *FILE*, which specifies the stream on which the operation is to take place. If *FILE* is NIL, the primary output stream is used .

(**OUTPUT** *FILE*) [Function]

Sets *FILE* as the primary output stream; returns the old primary output stream. *FILE* must be open for output.

(OUTPUT) returns the current primary output stream, which is not changed.

Note: If the primary output stream is set to a file, the file's full name, rather than the stream itself, is returned. See the discussion in Chapter 24.

(**PRIN1** *X FILE*) [Function]

Prints *X* on *FILE*.

(**PRIN2** *X FILE RDTBL*) [Function]

Prints *X* on *FILE* with %'s and "'s inserted where required for it to read back in properly by READ, using *RDTBL*.

Both PRIN1 and PRIN2 print any kind of Lisp expression, including lists, atoms, numbers, and strings. PRIN1 is generally used for printing expressions where human readability, rather than machine readability, is important, e.g., when printing text rather than program fragments. PRIN1 does not print double quotes around strings, or % in front of special characters. PRIN2 is used for printing Interlisp expressions which can then be read back into Interlisp with READ; i.e., break and separator characters in atoms will be preceded by %'s. For example, the atom "()" is printed as %(%) by PRIN2. If the integer output radix (as set by RADIX) is not 10, PRIN2 prints the integer using the input syntax for non-decimal integers (see Chapter 7) but PRIN1 does not (but both print the integer in the output radix).

(**PRIN3** *X FILE*) [Function]
(**PRIN4** *X FILE RDTBL*) [Function]

PRIN3 and PRIN4 are the same as PRIN1 and PRIN2 respectively, except that they do not increment the horizontal position counter nor perform any linelength checks. They are useful primarily for printing control characters.

(**PRINT** *X FILE RDTBL*) [Function]

Prints the expression *X* using PRIN2 followed by an end-of-line. Returns *X*.

(**PRINTCCODE** *CHARCODE FILE*) [Function]

Outputs a single character whose code is *CHARCODE* to *FILE*. This is similar to (PRIN1 (CHARACTER *CHARCODE*)), except that numeric characters are guaranteed to print "correctly"; e.g., (PRINTCCODE (CHARCODE 9)) always prints "9", independent of the setting of RADIX.

PRINTCCODE may actually print more than one byte on *FILE*, due to character encoding and end of line conventions; thus, no assumptions should be made about the relative motion of the file pointer (see GETFILEPTR) during this operation.

(**BOUT** *STREAM BYTE*) [Function]

Outputs a single 8-bit byte to *STREAM*. This is similar to PRINTCCODE, but for binary streams the character position in *STREAM* is not updated (as with PRIN3), and end of line conventions are ignored.

Note: BOUT is similar to PRINTCCODE, except that BOUT always writes a single byte, whereas PRINTCCODE writes a "character" that can consist of more than one byte, depending on the character and its encoding.

(**SPACES** *N FILE*) [Function]

Prints *N* spaces. Returns NIL.

(**TERPRI** *FILE*) [Function]

Prints an end-of-line character. Returns NIL.

(**FRESHLINE** *STREAM*) [Function]

Equivalent to TERPRI, except it does nothing if it is already at the beginning of the line. Returns T if it prints an end-of-line, NIL otherwise.

(**TAB** *POS MINSPACES FILE*) [Function]

Prints the appropriate number of spaces to move to position *POS*. *MINSPACES* indicates how many spaces must be printed (if NIL, 1 is used). If the current position plus *MINSPACES* is greater than *POS*, TAB does a TERPRI and then (SPACES *POS*). If *MINSPACES* is T, and the current position is greater than *POS*, then TAB does nothing.

Note: A sequence of PRINT, PRIN2, SPACES, and TERPRI expressions can often be more conveniently coded with a single PRINTOUT statement.

(**SHOWPRIN2** *X FILE RDTBL*) [Function]

Like PRIN2 except if SYSPRETTYFLG=T, prettyprints *X* instead. Returns *X*.

(**SHOWPRINT** *X FILE RDTBL*) [Function]

Like PRINT except if SYSPRETTYFLG=T, prettyprints *X* instead, followed by an end-of-line. Returns *X*.

SHOWPRINT and SHOWPRIN2 are used by the programmer's assistant (Chapter 13) for printing the values of expressions and for printing the history list, by various commands of the beak package (Chapter 14), e.g. ?= and BT commands, and various other system packages. The idea is that by simply settting or binding SYSPRETTYFLG to T (initially NIL), you instruct the system when interacting with you to PRETTYPRINT expressions (Chapter 26)instead of printing them.

(**PRINTBELLS**) [Function]

Used by DWIM (Chapter 19) to print a sequence of bells to alert you to stop typing. Can be advised or redefined for special applications, e.g., to flash the screen on a display terminal.

(**FORCEOUTPUT** *STREAM WAITFORFINISH*) [Function]

Forces any buffered output data in *STREAM* to be transmitted.

If *WAITFORFINISH* is non-NIL, this doesn't return until the data has been forced out.

(**POSITION** *FILE N*) [Function]

Returns the column number at which the next character will be read or printed. After a end of line, the column number is 0. If *N* is non-NIL, *resets* the column number to be *N*.

Note that resetting POSITION only changes Lisp's belief about the current column number; it does not cause any horizontal motion. Also note that (POSITION *FILE*) is *not* the same as (GETFILEPTR *FILE*) which gives the position in the *file*, not on the *line*.

(**LINELENGTH** *N FILE*) [Function]

Sets the length of the print line for the output file *FILE* to *N*; returns the former setting of the line length. *FILE* defaults to the primary output stream. (LINELENGTH NIL *FILE*) returns the current setting for *FILE*. When a file is first opened, its line length is set to the value of the variable FILELINELENGTH.

Whenever printing an atom or string would increase a file's position *beyond* the line length of the file, an end of line is automatically inserted first. This action can be defeated by using PRIN3 and PRIN4.

(**SETLINELENGTH** *N*) [Function]

Sets the line length for the terminal by doing (LINELENGTH *N* T). If *N* is NIL, it determines *N* by consulting the operating system's belief about the terminal's characteristics. In Interlisp-D, this is a no-op.

PRINTLEVEL

When using Interlisp one often has to handle large, complicated lists, which are difficult to understand when printed out. PRINTLEVEL allows you to specify in how much detail lists should be printed. The print functions PRINT, PRIN1, and PRIN2 are all affected by level parameters set by:

(**PRINTLEVEL** *CARVAL CDRVAL*) [Function]

Sets the CAR print level to *CARVAL*, and the CDR print level to *CDRVAL*. Returns a list cell whose CAR and CDR are the old settings. PRINTLEVEL is initialized with the value (1000 . -1).

In order that PRINTLEVEL can be used with RESETFORM or RESETSAVE, if *CARVAL* is a list cell it is equivalent to (PRINTLEVEL (CAR *CARVAL*) (CDR *CARVAL*)).

(PRINTLEVEL *N* NIL) changes the CAR printlevel without affecting the CDR printlevel. (PRINTLEVEL NIL *N*) changes the CDR printlevel with affecting the CAR printlevel. (PRINTLEVEL) gives the current setting without changing either.

Note: Control-P (Chapter 30) can be used to change the PRINTLEVEL setting dynamically, even while Interlisp is printing.

The CAR printlevel specifies how "deep" to print a list. Specifically, it is the number of unpaired left parentheses which will be printed. Below that level, all lists will be printed as &. If the CAR printlevel is *negative*, the action is similar except that an end-of-line is inserted after each right parentheses that would be immediately followed by a left parenthesis.

The CDR printlevel specifies how "long" to print a list. It is the number of top level list elements that will be printed before the printing is terminated with --. For example, if *CDRVAL*=2, (A B C D E) will print as (A B --). For sublists, the number of list elements printed is also affected by the depth of printing in the CAR direction: Whenever the *sum* of the depth of the sublist (i.e. the number of unmatched left parentheses) and the number of elements is greater than the CDR printlevel, -- is printed. This gives a "triangular" effect in that less is printed the farther one goes in either CAR or CDR direction. If the CDR printlevel is negative, then it is the same as if the CDR printlevel were infinite.

Examples:

After:	(A (B C (D (E F) G) H) K L) prints as:
(PRINTLEVEL 3 -1)	(A (B C (D & G) H) K L)
(PRINTLEVEL 2 -1)	(A (B C & H) K L)
(PRINTLEVEL 1 -1)	(A & K L)
(PRINTLEVEL 0 -1)	&
(PRINTLEVEL 1000 2)	(A (B --) --)
(PRINTLEVEL 1000 3)	(A (B C --) K --)

```
(PRINTLEVEL 1 3)    (A & K --)
```

PLVLFILEFLG [Variable]

Normally, PRINTLEVEL only affects terminal output. Output to all other files acts as though the print level is infinite. However, if PLVLFILEFLG is T (initially NIL), then PRINTLEVEL affects output to files as well.

The following three functions are useful for printing isolated expressions at a specified print level without going to the overhead of resetting the global print level.

(**LVLPRINT** *X FILE CARLVL CDRLVL TAIL*) [Function]

Performs PRINT of *X* to *FILE*, using as CAR and CDR print levels the values *CARLVL* and *CDRLVL*, respectively. Uses the T read table. If *TAIL* is specified, and *X* is a tail of it, then begins its printing with " . . . ", rather than on open parenthesis.

(**LVLPRIN2** *X FILE CARLVL CDRLVL TAIL*) [Function]

Similar to LVLPRIN2, but performs a PRIN2.

(**LVLPRIN1** *X FILE CARLVL CDRLVL TAIL*) [Function]

Similar to LVLPRIN1, but performs a PRIN1.

Printing Numbers

How the ordinary printing functions (PRIN1, PRIN2, etc.) print numbers can be affected in several ways. RADIX influences the printing of integers, and FLTFMT influences the printing of floating point numbers. The setting of the variable PRXFLG determines how the symbol-manipulation functions handle numbers. The PRINTNUM package permits greater controls on the printed appearance of numbers, allowing such things as left-justification, suppression of trailing decimals, etc.

(**RADIX** *N*) [Function]

Resets the output radix for integers to the absolute value of *N*. The value of RADIX is its previous setting. (RADIX) gives the current setting without changing it. The initial setting is 10.

Note that RADIX affects output *only*. There is no input radix; on input, numbers are interpreted as decimal unless they are entered in a non-decimal radix with syntax such as 123Q, |b10101, |5r1234 (see Chapter 7). RADIX does not affect the behavior of UNPACK, etc., unless the value of PRXFLG (below) is T. For example, if PRXFLG is NIL and the radix is set to 8 with (RADIX 8), the value of (UNPACK 9) is (9), not (1 1).

Using PRINTNUM (below) or the PRINTOUT command .I (below) is often a more convenient and appropriate way to print a single number in a specified radix than to globally change RADIX.

(**FLTFMT** *FORMAT*) [Function]

Resets the output format for floating point numbers to the FLOAT format *FORMAT* (see PRINTNUM below for a description of FLOAT formats). *FORMAT*=T specifies the default "free" formatting: some number of significant digits (a function of the implementation) are printed, with trailing zeros suppressed; numbers with sufficiently large or small exponents are instead printed in exponent notation.

FLTFMT returns its current setting. (FLTFMT) returns the current setting without changing it. The initial setting is T.

Note: In Interlisp-D, FLTFMT ignores the *WIDTH* and *PAD* fields of the format (they are implemented only by PRINTNUM).

Whether print name manipulation functions (UNPACK, NCHARS, etc.) use the values of RADIX and FLTFMT is determined by the variable PRXFLG:

PRXFLG [Variable]

If PRXFLG=NIL (the initial setting), then the "PRIN1" name used by PACK, UNPACK, MKSTRING, etc., is computed using base 10 for integers and the system default floating format for floating point numbers, independent of the current setting of RADIX or FLTFMT. If PRXFLG=T, then RADIX and FLTFMT do dictate the "PRIN1" name of numbers. Note that in this case, PACK and UNPACK are *not* inverses.

Examples with (RADIX 8), (FLTFMT '(FLOAT 4 2)):

With PRXFLG=NIL,

```
(UNPACK 13)  =>  (1 3)
(PACK '(A 9))  =>  A9
(UNPACK 1.2345)  =>  (1 %. 2 3 4 5)
```

With PRXFLG=T,

```
(UNPACK 13)  =>  (1 5)
(PACK '(A 9))  =>  A11
(UNPACK 1.2345)  =>  (1 %. 2 3)
```

Note that PRXFLG does not effect the radix of "PRIN2" names, so with (RADIX 8), (NCHARS 9 T), which uses PRIN2 names, would return 3, (since 9 would print as 11Q) for either setting of PRXFLG.

Warning: Some system functions will not work correctly if PRXFLG is not NIL. Therefore, resetting the global value of PRXFLG is not recommended. It is much better to rebind PRXFLG as a SPECVAR for that part of a program where it needs to be non-NIL.

The basic function for printing numbers under format control is PRINTNUM. Its utility is considerably enhanced when used in conjunction with the PRINTOUT package, which implements a compact language for specifying complicated sequences of elementary printing operations, and makes fancy output formats easy to design and simple to program.

(**PRINTNUM** FORMAT NUMBER FILE) [Function]

Prints *NUMBER* on *FILE* according to the format *FORMAT*. *FORMAT* is a list structure with one of the forms described below.

If *FORMAT* is a list of the form (FIX *WIDTH RADIX PAD0 LEFTFLUSH*), this specifies a FIX format. *NUMBER* is rounded to the nearest integer, and then printed in a field *WIDTH* characters long with radix set to *RADIX* (or 10 if *RADIX*=NIL; note that the setting from the function RADIX is *not* used as the default). If *PAD0* and *LEFTFLUSH* are both NIL, the number is right-justified in the field, and the padding characters to the left of the leading digit are spaces. If *PAD0* is T, the character "0" is used for padding. If *LEFTFLUSH* is T, then the number is left-justified in the field, with trailing spaces to fill out *WIDTH* characters.

The following examples illustrate the effects of the FIX format options on the number 9 (the vertical bars indicate the field width):

FORMAT:	(PRINTNUM *FORMAT* 9) prints:
(FIX 2)	| 9|
(FIX 2 NIL T)	|09|
(FIX 12 8 T)	|000000000011|
(FIX 5 NIL NIL T)	|9 |

If *FORMAT* is a list of the form (FLOAT *WIDTH DECPART EXPPART PAD0 ROUND*), this specifies a FLOAT format. *NUMBER* is printed as a decimal number in a field *WIDTH* characters wide, with *DECPART* digits to the right of the decimal point. If *EXPPART* is not 0 (or NIL), the number is printed in exponent notation, with the exponent occupying *EXPPART* characters in the field. *EXPPART* should allow for the character E and an optional sign to be printed before the exponent digits. As with FIX format, padding on the left is with spaces, unless *PAD0* is T. If *ROUND* is given, it indicates the digit position at which rounding is to take place, counting from the leading digit of the number.

Interlisp-D interprets *WIDTH*=NIL to mean no padding, i.e., to use however much space the number needs, and interprets *DECPART*=NIL to mean as many decimal places as needed.

The following examples illustrate the effects of the FLOAT format options on the number 27.689 (the vertical bars indicate the field width):

FORMAT:	(PRINTNUM *FORMAT* 27.689) prints:
(FLOAT 7 2)	| 27.69|
(FLOAT 7 2 NIL 0)	|0027.69|
(FLOAT 7 2 2)	| 2.77E1|
(FLOAT 11 2 4)	| 2.77E+01|
(FLOAT 7 2 NIL NIL 1)	| 30.00|
(FLOAT 7 2 NIL NIL 2)	| 28.00|

NILNUMPRINTFLG [Variable]

If PRINTNUM's *NUMBER* argument is not a number and not NIL, a NON-NUMERIC ARG error is generated. If *NUMBER* is NIL, the effect depends on the setting of the variable NILNUMPRINTFLG. If NILNUMPRINTFLG is NIL, then the error occurs as usual. If it is non-NIL, then no error occurs, and the value of NILNUMPRINTFLG is printed right-justified in the field described by *FORMAT*. This option facilitates the printing of numbers in aggregates with missing values coded as NIL.

User Defined Printing

Initially, Interlisp only knows how to print in an interesting way objects of type litatom, number, string, list and stackp. All other types of objects are printed in the form {datatype} followed by the octal representation of the address of the pointer, a format that cannot be read back in to produce an equivalent object. When defining user data types (using the DATATYPE record type, Chapter 8), it is often desirable to specify as well how objects of that type should be printed, so as to make their contents readable, or at least more informative to the viewer. The function DEFPRINT is used to specify the printing format of a data type.

(**DEFPRINT** *TYPE FN*) [Function]

TYPE is a type name. Whenever a printing function (PRINT, PRIN1, PRIN2, etc.) or a function requiring a print name (CHCON, NCHARS, etc.) encounters an object of the indicated type, *FN* is called with two arguments: the item to be printed and the name of the stream, if any, to which the object is to be printed. The second argument is NIL on calls that request the print name of an object without actually printing it.

If *FN* returns a list of the form (*ITEM1* . *ITEM2*), *ITEM1* is printed using PRIN1 (unless it is NIL), and then *ITEM2* is printed using PRIN2 (unless it is NIL). No spaces are printed between the two items. Typically, *ITEM1* is a read macro character.

If *FN* returns NIL, the datum is printed in the system default manner.

If *FN* returns T, nothing further is printed; *FN* is assumed to have printed the object to the stream itself. Note that this case if permitted only when the second argument passed to *FN* is non-NIL; otherwise, there is no destination for *FN* to do its printing, so it must return as in one of the other two cases.

Printing Unusual Data Structures

HPRINT (for "Horrible Print") and HREAD provide a mechanism for printing and reading back in general data structures that cannot normally be dumped and loaded easily, such as (possibly re-entrant or circular) structures containing user datatypes, arrays, hash tables, as well as list structures. HPRINT will correctly print and read back in any structure containing any or all of the above, chasing all pointers down to the level of literal atoms, numbers or strings. HPRINT currently cannot handle compiled code arrays, stack positions, or arbitrary unboxed numbers.

HPRINT operates by simulating the Interlisp PRINT routine for normal list structures. When it encounters a user datatype (see Chapter 8), or an array or hash array, it prints the data contained therein, surrounded by special characters defined as read macro characters. While chasing the pointers of a structure, it also keeps a hash table of those items it encounters, and if any item is encountered a second time, another read macro character is inserted before the first occurrence (by resetting the file pointer with SETFILEPTR) and all subsequent occurrences are printed as a back reference using an appropriate macro character. Thus the inverse function, HREAD merely calls the Interlisp READ routine with the appropriate read table.

(**HPRINT** *EXPR FILE UNCIRCULAR DATATYPESEEN*) [Function]

Prints *EXPR* on *FILE*. If *UNCIRCULAR* is non-NIL, HPRINT does no checking for any circularities in *EXPR* (but is still useful for dumping arbitrary structures of arrays, hash arrays, lists, user data types, etc., that do not contain circularities). Specifying *UNCIRCULAR* as non-NIL results in a large speed and internal-storage advantage.

Normally, when HPRINT encounters a user data type for the first time, it outputs a summary of the data type's declaration. When this is read in, the data type is redeclared. If *DATATYPESEEN* is non-NIL, HPRINT assumes that the same data type declarations will be in force at read time as were at HPRINT time, and not output declarations.

HPRINT is intended primarily for output to random access files, since the algorithm depends on being able to reset the file pointer. If *FILE* is not a random access file (and *UNCIRCULAR* = NIL), a temporary file, HPRINT.SCRATCH, is opened, *EXPR* is HPRINTed on it, and then that file is copied to the final output file and the temporary file is deleted.

You can not use HPRINT to save things that contains pointers to raw storage. Fontdescriptors contain pointers to raw storage and windows contain pointers to fontdescriptors. Netiher can therefor be saved with HPRINT.

(**HREAD** *FILE*) [Function]

Reads and returns an HPRINT-ed expression from *FILE*.

(**HCOPYALL** *X*) [Function]

Copies data structure *X*. *X* may contain circular pointers as well as arbitrary structures.

Note: HORRIBLEVARS and UGLYVARS (Chapter 17) are two file package commands for dumping and reloading circular and re-entrant data structures. They provide a convenient interface to HPRINT and HREAD.

When HPRINT is dumping a data structure that contains an instance of an Interlisp datatype, the datatype declaration is also printed onto the file. Reading such a data structure with HREAD can cause problems if it redefines a system datatype. Redefining a system datatype will almost definitely cause serious errors. The Interlisp system datatypes do not change very often, but there is always a possibility when loading in old files created under an old Interlisp release.

To prevent accidental system crashes, HREAD will *not* redefine datatypes. Instead, it will cause an error "attempt to read DATATYPE with different field

specification than currently defined". Continuing from this error will redefine the datatype.

Random Access File Operations

For most applications, files are read starting at their beginning and proceeding sequentially, i.e., the next character read is the one immediately following the last character read. Similarly, files are written sequentially. However, for files on some devices, it is also possible to read/write characters at arbitrary positions in a file, essentially treating the file as a large block of auxiliary storage. For example, one application might involve writing an expression at the *beginning* of the file, and then reading an expression from a specified point in its *middle*. This particular example requires the file be open for *both* input and output. However, random file input or output can also be performed on files that have been opened for only input or only output.

Associated with each file is a "file pointer" that points to the location where the next character is to be read from or written to. The file position of a byte is the number of bytes that precede it in the file, i.e., 0 is the position of the beginning of the file. The file pointer to a file is automatically advanced after each input or output operation. This section describes functions which can be used to *reposition* the file pointer on those files that can be randomly accessed. A file used in this fashion is much like an array in that it has a certain number of addressable locations that characters can be put into or taken from. However, unlike arrays, files can be enlarged. For example, if the file pointer is positioned at the end of a file and anything is written, the file "grows." It is also possible to position the file pointer *beyond* the end of file and then to write. (If the program attempts to *read* beyond the end of file, an `END OF FILE` error occurs.) In this case, the file is enlarged, and a "hole" is created, which can later be written into. Note that this enlargement only takes place at the *end* of a file; it is not possible to make more room in the middle of a file. In other words, if expression `A` begins at position 1000, and expression `B` at 1100, and the program attempts to overwrite `A` with expression `C`, whose printed representation is 200 bytes long, part of `B` will be altered.

Warning: File positions are always in terms of bytes, not characters. You should thus be very careful about computing the space needed for an expression. In particular, NS characters may take multiple bytes (see below). Also, the end-of-line character (see Chapter 24) may be represented by a different number of characters in different implementations. Output functions may also introduce end-of-line's as a result of `LINELENGTH` considerations. Therefore `NCHARS` (see Chapter 2) does *not* specify how many bytes an expression takes to print, even ignoring line length considerations.

(**GETFILEPTR** *FILE*) [Function]

Returns the current position of the file pointer for *FILE*, i.e., the byte address at which the next input/output operation will commence.

(**SETFILEPTR** *FILE ADR*) [Function]

Sets the file pointer for *FILE* to the position *ADR*; returns *ADR*. The special value *ADR*=`-1` is interpreted to mean the address of the end of file.

Note: If a file is opened for output only, the end of file is initially zero, even if an old file by the same name had existed (see OPENSTREAM, Chapter 24). If a file is opened for both input and output, the initial file pointer is the beginning of the file, but (SETFILEPTR *FILE* -1) sets it to the end of the file. If the file had been opened in append mode by (OPENSTREAM *FILE* 'APPEND), the file pointer right after opening would be set to the end of the existing file, in which case a SETFILEPTR to position the file at the end would be unnecessary.

(**GETEOFPTR** *FILE*) [Function]

Returns the byte address of the end of file, i.e., the number of bytes in the file. Equivalent to performing (SETFILEPTR *FILE* -1) and returning (GETFILEPTR *FILE*) except that it does not change the current file pointer.

(**RANDACCESSP** *FILE*) [Function]

Returns *FILE* if *FILE* is randomly accessible, NIL otherwise. The file T is not randomly accessible, nor are certain network file connections in Interlisp-D. *FILE* must be open or an error is generated, FILE NOT OPEN.

(**COPYBYTES** *SRCFIL DSTFIL START END*) [Function]

Copies bytes from *SRCFIL* to *DSTFIL*, starting from position *START* and up to but not including position *END*. Both *SRCFIL* and *DSTFIL* must be open. Returns T.

If *END*=NIL, *START* is interpreted as the number of bytes to copy (starting at the current position). If *START* is also NIL, bytes are copied until the end of the file is reached.

Warning: COPYBYTES does not take any account of multi-byte NS characters (see Chapter 2). COPYCHARS (below) should be used whenever copying information that might include NS characters.

(**COPYCHARS** *SRCFIL DSTFIL START END*) [Function]

Like COPYBYTES except that it copies NS characters (see Chapter 2), and performs the proper conversion if the end-of-line conventions of *SRCFIL* and *DSTFIL* are not the same (see Chapter 24). *START* and *END* are interpreted the same as with COPYBYTES, i.e., as byte (not character) specifications in *SRCFIL*. The number of bytes actually output to *DSTFIL* might be more or less than the number of bytes specified by *START* and *END*, depending on what the end-of-line conventions are. In the case where the end-of-line conventions happen to be the same, COPYCHARS simply calls COPYBYTES.

(**FILEPOS** *STR FILE START END SKIP TAIL CASEARRAY*) [Function]

Analogous to STRPOS (see Chapter 4), but searches a file rather than a string. FILEPOS searches *FILE* for the string *STR*. Search begins at *START* (or the current position of the file pointer, if *START*=NIL), and goes to *END* (or the end of *FILE*, if *END*=NIL). Returns the address of the start of the match, or NIL if not found.

SKIP can be used to specify a character which matches any character in the file. If *TAIL* is T, and the search is successful, the value is the address of the first character *after* the sequence of characters corresponding to *STR*, instead of the starting address of the sequence. In either case, the file is left so that the next i/o operation begins at the address returned as the value of FILEPOS.

CASEARRAY should be a "case array" that specifies that certain characters should be transformed to other characters before matching. Case arrays are returned by CASEARRAY or SEPRCASE below. *CASEARRAY*=NIL means no transformation will be performed.

A case array is an implementation-dependent object that is logically an array of character codes with one entry for each possible character. FILEPOS maps each character in the file "through" *CASEARRAY* in the sense that each character code is transformed into the corresponding character code from *CASEARRAY* before matching. Thus if two characters map into the same value, they are treated as equivalent by FILEPOS. CASEARRAY and SETCASEARRAY provide an implementation-independent interface to case arrays.

For example, to search without regard to upper and lower case differences, *CASEARRAY* would be a case array where all characters map to themselves, except for lower case characters, whose corresponding elements would be the upper case characters. To search for a delimited atom, one could use " *ATOM* " as the pattern, and specify a case array in which all of the break and separator characters mapped into the same code as space.

For applications calling for extensive file searches, the function FFILEPOS is often faster than FILEPOS.

(**FFILEPOS** *PATTERN FILE START END SKIP TAIL CASEARRAY*) [Function]

Like FILEPOS, except much faster in most applications. FFILEPOS is an implementation of the Boyer-Moore fast string searching algorithm. This algorithm preprocesses the string being searched for and then scans through the file in steps usually equal to the length of the string. Thus, FFILEPOS speeds up roughly in proportion to the length of the string, e.g., a string of length 10 will be found twice as fast as a string of length 5 in the same position.

Because of certain fixed overheads, it is generally better to use FILEPOS for short searches or short strings.

(**CASEARRAY** *OLDARRAY*) [Function]

Creates and returns a new case array, with all elements set to themselves, to indicate the identity mapping. If *OLDARRAY* is given, it is reused.

(**SETCASEARRAY** *CASEARRAY FROMCODE TOCODE*) [Function]

Modifies the case array *CASEARRAY* so that character code *FROMCODE* is mapped to character code *TOCODE*.

(**GETCASEARRAY** *CASEARRAY FROMCODE*) [Function]

Returns the character code that *FROMCODE* is mapped to in *CASEARRAY*.

(**SEPRCASE** *CLFLG*) [Function]

Returns a new case array suitable for use by FILEPOS or FFILEPOS in which all of the break/separators of FILERDTBL are mapped into character code zero. If *CLFLG* is non-NIL, then all CLISP characters are mapped into this character as well. This is useful for finding a delimited atom in a file. For example, if *PATTERN* is " FOO ", and (SEPRCASE T) is used for *CASEARRAY*, then FILEPOS will find "(FOO_".

UPPERCASEARRAY [Variable]

Value is a case array in which every lowercase character is mapped into the corresponding uppercase character. Useful for searching text files.

Input/Output Operations with Characters and Bytes

Interlisp-D supports the 16-bit NS character set (see Chapter 2). All of the standard string and print name functions accept litatoms and strings containing NS characters. In almost all cases, a program does not have to distinguish between NS characters or 8-bit characters. The exception to this rule is the handling of input/output operations.

Interlisp-D uses two ways of writing 16-bit NS characters on files. One way is to write the full 16-bits (two bytes) every time a character is output. The other way is to use "run-encoding." Each 16 NS character can be decoded into a character set (an integer from 0 to 254 inclusive) and a character number (also an integer from 0 to 254 inclusive). In run-encoding, the byte 255 (illegal as either a character set number or a character number) is used to signal a change to a given character set, and the following bytes are all assumed to come from the same character set (until the next change-character set sequence). Run-encoding can reduce the number of bytes required to encode a string of NS characters, as long as there are long sequences of characters from the same character set (usually the case).

Note that characters are not the same as bytes. A single character can take anywhere from one to four bytes bytes, depending on whether it is in the same character set as the preceeding character, and whether run-encoding is enabled. Programs which assume that characters are equal to bytes must be changed to work with NS characters.

The functions BIN and BOUT (see above) should only be used to read and write single eight-bit bytes. The functions READCCODE and PRINTCCODE (see above) should be used to read and write single character codes, interpreting run-encoded NS characters. COPYBYTES should only be used to copy blocks of 8-bit data; COPYCHARS should be used to copy characters. Most I/O functions (READC, PRIN1, etc.) read or write 16-bit NS characters.

The use of NS characters has serious consequences for any program that uses file pointers to access a file in a random access manner. At any point when a file is being read or written, it has a "current character set." If the file pointer is changed with SETFILEPTR to a part of the file with a different character set, any characters read or written may have the wrong character set. The current character set can be accessed with the following function:

(**CHARSET** *STREAM CHARACTERSET*) [Function]

Returns the current character set of the stream *STREAM*. If *CHARACTERSET* is non-NIL, the current character set for *STREAM* is set. Note that for output streams this may cause bytes to be written to the stream.

If *CHARACTERSET* is T, run encoding for *STREAM* is disabled: both the character set and the character number (two bytes total) will be written to the stream for each character printed.

PRINTOUT

Interlisp provides many facilities for controlling the format of printed output. By executing various sequences of PRIN1, PRIN2, TAB, TERPRI, SPACES, PRINTNUM, and PRINTDEF, almost any effect can be achieved. PRINTOUT implements a compact language for specifying complicated sequences of these elementary printing functions. It makes fancy output formats easy to design and simple to program.

PRINTOUT is a CLISP word (like FOR and IF) for interpreting a special printing language in which you can describe the kinds of printing desired. The description is translated by DWIMIFY to the appropriate sequence of PRIN1, TAB, etc., before it is evaluated or compiled. PRINTOUT printing descriptions have the following general form:

(PRINTOUT *STREAM* $PRINTCOM_1$ . . . $PRINTCOM_N$)

STREAM is evaluated to obtain the stream to which the output from this specification is directed. The PRINTOUT commands are strung together, one after the other without punctuation, after *STREAM*. Some commands occupy a single position in this list, but many commands expect to find arguments following the command name in the list. The commands fall into several logical groups: one set deals with horizontal and vertical spacing, another group provides controls for certain formatting capabilities (font changes and subscripting), while a third set is concerned with various ways of actually printing items. Finally, there is a command that permits escaping to a simple Lisp evaluation in the middle of a PRINTOUT form. The various commands are described below. The following examples give a general flavor of how PRINTOUT is used:

Example 1: Suppose you want to print out on the terminal the values of three variables, X, Y, and Z, separated by spaces and followed by a carriage return. This could be done by:

```
(PRIN1 X T)
       (SPACES 1 T)
       (PRIN1 Y T)
       (SPACES 1 T)
       (PRIN1 Z T)
       (TERPRI T)
```

or by the more concise PRINTOUT form:

```
(PRINTOUT T X , Y , Z T)
```

Here the first T specifies output to the terminal, the commas cause single spaces to be printed, and the final T specifies a TERPRI. The variable names are not recognized as special PRINTOUT commands, so they are printed using PRIN1 by default.

Example 2: Suppose the values of X and Y are to be pretty-printed lined up at position 10, preceded by identifying strings. If the output is to go to the primary output stream, you could write either:

```
(PRIN1 "X =")
       (PRINTDEF X 10 T)
       (TERPRI )
       (PRIN1 "Y =")
       (PRINTDEF Y 10 T)
       (TERPRI)
```

or the equivalent:

```
(PRINTOUT NIL "X =" 10 .PPV X T
          "Y =" 10 .PPV Y T)
```

Since strings are not recognized as special commands, "X =" is also printed with PRIN1 by default. The positive integer means TAB to position 10, where the .PPV command causes the value of X to be prettyprinted as a variable. By convention, special atoms used as PRINTOUT commands are prefixed with a period. The T causes a carriage return, so the Y information is printed on the next line.

Example 3. As a final example, suppose that the value of X is an integer and the value of Y is a floating-point number. X is to be printed right-flushed in a field of width 5 beginning at position 15, and Y is to be printed in a field of width 10 also starting at position 15 with 2 places to the right of the decimal point. Furthermore, suppose that the variable names are to appear in the font class named BOLDFONT and the values in font class SMALLFONT. The program in ordinary Interlisp that would accomplish these effects is too complicated to include here. With PRINTOUT, one could write:

```
(PRINTOUT NIL
          .FONT BOLDFONT "X =" 15
```

```
.FONT SMALLFONT .I5 X T
.FONT BOLDFONT "Y =" 15
.FONT SMALLFONT .F10.2 Y T
.FONT BOLDFONT)
```

The .FONT commands do whatever is necessary to change the font on a multi-font output device. The .I5 command sets up a FIX format for a call to the function PRINTNUM (see above) to print X in the desired format. The .F10.2 specifies a FLOAT format for PRINTNUM.

Horizontal Spacing Commands

The horizontal spacing commands provide convenient ways of calling TAB and SPACES. In the following descriptions, *N* stands for a literal positive integer (*not* for a variable or expression whose value is an integer).

N (*N* a number) [PRINTOUT Command]

Used for absolute spacing. It results in a TAB to position *N* (literally, a (TAB *N*)). If the line is currently at position *N* or beyond, the file will be positioned at position *N* on the next line.

.TAB *POS* [PRINTOUT Command]

Specifies TAB to position (the value of) *POS*. This is one of several commands whose effect could be achieved by simply escaping to Lisp, and executing the corresponding form. It is provided as a separate command so that the PRINTOUT form is more concise and is prettyprinted more compactly. Note that .TAB *N* and *N*, where *N* is an integer, are equivalent.

.TAB0 *POS* [PRINTOUT Command]

Like .TAB except that it can result in zero spaces (i.e. the call to TAB specifies *MINSPACES*=0).

-N (*N* a number) [PRINTOUT Command]

Negative integers indicate relative (as opposed to absolute) spacing. Translates as (SPACES |*N*|).

, [PRINTOUT Command]
,, [PRINTOUT Command]
,,, [PRINTOUT Command]

(1, 2 or 3 commas) Provides a short-hand way of specifying 1, 2 or 3 spaces, i.e., these commands are equivalent to -1, -2, and -3, respectively.

.SP *DISTANCE* [PRINTOUT Command]

Translates as (SPACES *DISTANCE*). Note that .SP *N* and -*N*, where *N* is an integer, are equivalent.

Vertical Spacing Commands

Vertical spacing is obtained by calling TERPRI or printing form-feeds. The relevant commands are:

T [PRINTOUT Command]

Translates as (TERPRI), i.e., move to position 0 (the first column) of the next line. To print the letter T, use the string "T".

.SKIP *LINES* [PRINTOUT Command]

Equivalent to a sequence of *LINES* (TERPRI)'s. The .SKIP command allows for skipping large constant distances and for computing the distance to be skipped.

.PAGE [PRINTOUT Command]

Puts a form-feed (Control-L) out on the file. Care is taken to make sure that Interlisp's view of the current line position is correctly updated.

Special Formatting Controls

There are a small number of commands for invoking some of the formatting capabilities of multi-font output devices. The available commands are:

.FONT *FONTSPEC* [PRINTOUT Command]

Changes printing to the font *FONTSPEC*, which can be a font descriptor, a "font list" such as '(MODERN 10), an image stream (coerced to its current font), or a windows (coerced to the current font of its display stream). The DSPFONT is changed permanently. See fonts (Chapter 27) for more information.

FONTSPEC may also be a positive integer *N*, which is taken as an abbreviated reference to the font class named FONT*N* (e.g. 1 => FONT1).

.SUP [PRINTOUT Command]

Specifies superscripting. All subsequent characters are printed above the base of the current line. Note that this is absolute, not relative: a .SUP following a .SUP is a no-op.

.SUB [PRINTOUT Command]

Specifies subscripting. Subsequent printing is below the base of the current line. As with superscripting, the effect is absolute.

.BASE [PRINTOUT Command]

Moves printing back to the base of the current line. Un-does a previous .SUP or .SUB; a no-op, if printing is currently at the base.

Printing Specifications

The value of any expression in a PRINTOUT form that is not recognized as a command itself or as a command argument is printed using PRIN1 by default. For example, title strings can be printed by simply including the string as a separate PRINTOUT command, and the values of variables and forms can be printed in much the same way. Note that a literal integer, say 51, cannot be printed by including it as a command, since it would be interpreted as a TAB; the desired effect can be obtained by using instead the string specification "51", or the form (QUOTE 51).

For those instances when PRIN1 is not appropriate, e.g., PRIN2 is required, or a list structures must be prettyprinted, the following commands are available:

.P2 *THING* [PRINTOUT Command]

Causes *THING* to be printed using PRIN2; translates as (PRIN2 *THING*).

.PPF *THING* [PRINTOUT Command]

Causes *THING* to be prettyprinted at the current line position via PRINTDEF (see Chapter 26). The call to PRINTDEF specifies that *THING* is to be printed as if it were part of a function definition. That is, SELECTQ, PROG, etc., receive special treatment.

.PPV *THING* [PRINTOUT Command]

Prettyprints *THING* as a variable; no special interpretation is given to SELECTQ, PROG, etc.

.PPFTL *THING* [PRINTOUT Command]

Like .PPF, but prettyprints *THING* as a *tail*, that is, without the initial and final parentheses if it is a list. Useful for prettyprinting sub-lists of a list whose other elements are formatted with other commands.

.PPVTL *THING* [PRINTOUT Command]

Like .PPV, but prettyprints *THING* as a tail.

Paragraph Format

Interlisp's prettyprint routines are designed to display the structure of expressions, but they are not really suitable for formatting unstructured text. If a list is to be printed as a textual paragraph, its internal structure is less important than controlling its left and right margins, and the indentation of its first line. The .PARA and .PARA2 commands allow these parameters to be conveniently specified.

.PARA *LMARG RMARG LIST* [PRINTOUT Command]

Prints *LIST* in paragraph format, using PRIN1. Translates as (PRINTPARA *LMARG RMARG LIST*) (see below).

Example: (PRINTOUT T 10 .PARA 5 -5 LST) will print the elements of LST as a paragraph with left margin at 5, right margin at (LINELENGTH)-5, and the first line indented to 10.

.PARA2 *LMARG RMARG LIST* [PRINTOUT Command]

Print as paragraph using PRIN2 instead of PRIN1. Translates as (PRINTPARA *LMARG RMARG LIST* T).

Right-Flushing

Two commands are provided for printing simple expressions flushed-right against a specified line position, using the function FLUSHRIGHT (see below). They take into account the current position, the number of characters in the print-name of the expression, and the position the expression is to be flush against, and then print the appropriate number of spaces to achieve the desired effect. Note that this might entail going to a new line before printing. Note also that right-flushing of expressions longer than a line (e.g. a large list) makes little sense, and the appearance of the output is not guaranteed.

.FR *POS EXPR* [PRINTOUT Command]

Flush-right using PRIN1. The value of *POS* determines the position that the right end of *EXPR* will line up at. As with the horizontal spacing commands, a negative position number means |*POS*| columns from the current position, a positive number specifies the position absolutely. *POS*=0 specifies the right-margin, i.e. is interpreted as (LINELENGTH).

.FR2 *POS EXPR* [PRINTOUT Command]

Flush-right using PRIN2 instead of PRIN1.

Centering

Commands for centering simple expressions between the current line position and another specified position are also available. As with right flushing, centering of large expressions is not guaranteed.

.CENTER *POS EXPR* [PRINTOUT Command]

Centers *EXPR* between the current line position and the position specified by the value of *POS*. A positive *POS* is an absolute position number, a negative *POS* specifies a position relative to the current position, and 0 indicates the right-margin. Uses PRIN1 for printing.

.CENTER2 *POS EXPR* [PRINTOUT Command]

Centers using PRIN2 instead of PRIN1.

Numbering

The following commands provide FORTRAN-like formatting capabilities for integer and floating-point numbers. Each command specifies a printing format and a number to be printed. The format specification translates into a format-list for the function PRINTNUM.

.I*FORMAT NUMBER* [PRINTOUT Command]

Specifies integer printing. Translates as a call to the function PRINTNUM with a FIX format-list constructed from *FORMAT*. The atomic format is broken apart at internal periods to form the format-list. For example, .I5.8.T yields the format-list (FIX 5 8 T), and the command sequence (PRINTOUT T .I5.8.T FOO) translates as (PRINTNUM '(FIX 5 8 T) FOO). This expression causes the value of FOO to be printed in radix 8 right-flushed in a field of width 5, with 0's used for padding on the left. Internal NIL's in the format specification may be omitted, e.g., the commands .I5..T and .I5.NIL.T are equivalent.

The format specification .I1 is often useful for forcing a number to be printed in radix 10 (but not otherwise specially formatted), independent of the current setting of RADIX.

.F *FORMAT NUMBER* [PRINTOUT Command]

Specifies floating-number printing. Like the .I format command, except translates with a FLOAT format-list.

.N *FORMAT NUMBER* [PRINTOUT Command]

The .I and .F commands specify calls to PRINTNUM with quoted format specifications. The .N command translates as (PRINTNUM *FORMAT NUMBER*), i.e., it permits the format to be the value of some expression. Note that, unlike the .I and .F commands, *FORMAT* is a separate element in the command list, not part of an atom beginning with .N.

Escaping to Lisp

There are many reasons for taking control away from PRINTOUT in the middle of a long printing expression. Common situations involve temporary changes to system printing parameters (e.g. LINELENGTH), conditional printing (e.g. print FOO only if FIE is T), or lower-level iterative printing within a higher-level print specification.

Unknown IMAGEOBJ type #GETFN: IM.INDEX.GETFN *FORM* [PRINTOUT Command]

The escape command. *FORM* is an arbitrary Lisp expression that is evaluated within the context established by the PRINTOUT form, i.e., *FORM* can assume that the primary output stream has been set to be the *FILE* argument to PRINTOUT. Note that nothing is done with the *value* of *FORM*; any printing desired is accomplished by *FORM* itself, and the value is discarded.

Note: Although PRINTOUT logically encloses its translation in a RESETFORM (Chapter 14) to change the primary output file to the *FILE* argument (if non-NIL), in most cases it can actually pass *FILE* (or a locally bound variable if *FILE* is a non-trivial expression) to each printing function. Thus, the RESETFORM is only generated when the # command is used, or user-defined commands (below) are used. If many such occur in repeated PRINTOUT forms, it may be more efficient to embed them all in a single RESETFORM which changes the primary output file, and then specify *FILE*=NIL in the PRINTOUT expressions themselves.

User-Defined Commands

The collection of commands and options outlined above is aimed at fulfilling all common printing needs. However, certain applications might have other, more specialized printing idioms, so a facility is provided whereby you can define new commands. This is done by adding entries to the global list PRINTOUTMACROS to define how the new commands are to be translated.

PRINTOUTMACROS [Variable]

PRINTOUTMACROS is an association-list whose elements are of the form **(***COMM FN*). Whenever *COMM* appears in command position in the sequence of PRINTOUT commands (as opposed to an argument position of another command), *FN* is applied to the tail of the command-list (including the command).

After inspecting as much of the tail as necessary, the function must return a list whose CAR is the translation of the user-defined command and its arguments, and whose CDR is the list of commands still remaining to be translated in the normal way.

For example, suppose you want to define a command "?", which will cause its single argument to be printed with PRIN1 only if it is not NIL. This can be done by entering (? ?TRAN) on PRINTOUTMACROS, and defining the function ?TRAN as follows:

```
(DEFINEQ (?TRAN (COMS)
   (CONS
      (SUBST (CADR COMS) 'ARG
         '(PROG ((TEMP ARG))
               (COND (TEMP (PRIN1 TEMP)))))
      (CDDR COMS))]
```

Note that ?TRAN does not do any printing itself; it returns a form which, when evaluated in the proper context, will perform the desired action. This form should direct all printing to the primary output file.

Special Printing Functions

The paragraph printing commands are translated into calls on the function PRINTPARA, which may also be called directly:

(**PRINTPARA** *LMARG RMARG LIST P2FLAG PARENFLAG FILE*) [Function]

Prints *LIST* on *FILE* in line-filled paragraph format with its first element beginning at the current line position and ending at or before *RMARG*, and with subsequent lines appearing between *LMARG* and *RMARG*. If *P2FLAG* is non-NIL, prints elements using PRIN2, otherwise PRIN1. If *PARENFLAG* is non-NIL, then parentheses will be printed around the elements of *LIST*.

If *LMARG* is zero or positive, it is interpreted as an absolute column position. If it is negative, then the left margin will be at |*LMARG*|+(POSITION). If *LMARG*=NIL, the left margin will be at (POSITION), and the paragraph will appear in block format.

If *RMARG* is positive, it also is an absolute column position (which may be greater than the current (LINELENGTH)). Otherwise, it is interpreted as relative to (LINELENGTH), i.e., the right margin will be at (LINELENGTH)+|*RMARG*|. Example: (TAB 10) (PRINTPARA 5 -5 LST T) will PRIN2 the elements of LST in a paragraph with the first line beginning at column 10, subsequent lines beginning at column 5, and all lines ending at or before (LINELENGTH)-5.

The current (LINELENGTH) is unaffected by PRINTPARA, and upon completion, *FILE* will be positioned immediately after the last character of the last item of *LIST*. PRINTPARA is a no-op if *LIST* is not a list.

The right-flushing and centering commands translate as calls to the function FLUSHRIGHT:

(**FLUSHRIGHT** *POS X MIN P2FLAG CENTERFLAG FILE*) [Function]

If *CENTERFLAG*=NIL, prints *X* right-flushed against position *POS* on *FILE*; otherwise, centers *X* between the current line position and *POS*. Makes sure that it spaces over at least *MIN* spaces before printing by doing a TERPRI if necessary; *MIN*=NIL is equivalent to *MIN*=1. A positive *POS* indicates an absolute position, while a negative *POS* signifies the position which is |*POS*| to the right of the current line position. *POS*=0 is interpreted as (LINELENGTH), the right margin.

READFILE and WRITEFILE

For those applications where you simply want to simply read all of the expressions on a file, and not evaluate them, the function READFILE is available:

(**READFILE** *FILE RDTBL ENDTOKEN*) [NoSpread Function]

Reads successive expressions from file using READ (with read table *RDTBL*) until the single litatom *ENDTOKEN* is read, or an end of file encountered. Returns a list of these expressions.

If *RDTBL* is not specified, it defaults to FILERDTBL. If *ENDTOKEN* is not specified, it defaults to the litatom STOP.

(**WRITEFILE** *X FILE*) [Function]

Writes a date expression onto *FILE*, followed by successive expressions from *X*, using FILERDTBL as a read table. If *X* is atomic, its value is used. If *FILE* is not open, it is opened. If *FILE* is a list, (CAR *FILE*) is used and the file is left opened. Otherwise, when *X* is finished, the litatom STOP is printed on *FILE* and it is closed. Returns *FILE*.

(**ENDFILE** *FILE*) [Function]

Prints STOP on *FILE* and closes it.

Read Tables

Many Interlisp input functions treat certain characters in special ways. For example, READ recognizes that the right and left parenthesis characters are used to specify list structures, and that the quote character is used to delimit text strings. The Interlisp input and (to a certain extent) output routines are table driven by read tables. Read tables are objects that specify the syntactic properties of characters for input routines. Since the input routines parse character sequences into objects, the read table in use determines which sequences are recognized as literal atoms, strings, list structures, etc.

Most Interlisp input functions take an optional read table argument, which specifies the read table to use when reading an expression. If NIL is given as the read table, the "primary read table" is used. If T is specified, the system terminal read table is used. Some functions will also accept the atom ORIG (*not* the *value* of ORIG) as indicating the "original" system read table. Some output functions also take a read table argument. For example, PRIN2 prints an expression so that it would be read in correctly using a given read table.

The Interlisp-D system uses the following read tables: T for input/output from terminals, the value of FILERDTBL for input/output from files, the value of EDITRDTBL for input from terminals while in the tty-based editor, the value of DEDITRDTBL for input from terminals while in the display-based editor, and the value of CODERDTBL for input/output from compiled files. These five read tables are initially copies of the ORIG read table, with changes made to some of them to provide read macros that are specific to terminal input or file input. Using the functions described below, you may further change, reset, or copy these tables. However, in the case of FILERDTBL and CODERDTBL, you are cautioned that changing these tables may prevent the system from being able to read files made with the original tables, or prevent users possessing only the standard tables from reading files made using the modified tables.

You can also create new read tables, and either explicitly pass them to input/output functions as arguments, or install them as the primary read table, via SETREADTABLE, and then not specify a *RDTBL* argument, i.e., use NIL.

Read Table Functions

(**READTABLEP** *RDTBL*) [Function]

Returns *RDTBL* if *RDTBL* is a real read table (*not* T or ORIG), otherwise NIL.

(**GETREADTABLE** *RDTBL*) [Function]

If *RDTBL*=NIL, returns the primary read table. If *RDTBL*=T, returns the system terminal read table. If *RDTBL* is a real read table, returns *RDTBL*. Otherwise, generates an ILLEGAL READTABLE error.

(**SETREADTABLE** *RDTBL FLG*) [Function]

Sets the primary read table to *RDTBL*. If *FLG*=T, SETREADTABLE sets the system terminal read table, T. Note that you can reset the other system read tables with SETQ, e.g., (SETQ FILERDTBL (GETREADTABLE)).

Generates an ILLEGAL READTABLE error if *RDTBL* is not NIL, T, or a real read table. Returns the previous setting of the primary read table, so SETREADTABLE is suitable for use with RESETFORM (Chapter 14).

(**COPYREADTABLE** *RDTBL*) [Function]

Returns a copy of *RDTBL*. *RDTBL* can be a real read table, NIL, T, or ORIG (in which case COPYREADTABLE returns a copy of the *original* system read table), otherwise COPYREADTABLE generates an ILLEGAL READTABLE error.

Note that COPYREADTABLE is the only function that *creates* a read table.

(**RESETREADTABLE** *RDTBL FROM*) [Function]

Copies (smashes) *FROM* into *RDTBL*. *FROM* and *RDTBL* can be NIL, T, or a real read table. In addition, *FROM* can be ORIG, meaning use the system's original read table.

Syntax Classes

A read table is an object that contains information about the "syntax class" of each character. There are nine basic syntax classes: LEFTPAREN, RIGHTPAREN, LEFTBRACKET, RIGHTBRACKET, STRINGDELIM, ESCAPE, BREAKCHAR, SEPRCHAR, and OTHER, each associated with a primitive syntactic property. In addition, there is an unlimited assortment of user-defined syntax classes, known as "read macros". The basic syntax classes are interpreted as follows:

LEFTPAREN (normally left parenthesis) Begins list structure.

RIGHTPAREN (normally right parenthesis) Ends list structure.

LEFTBRACKET (normally left bracket) Begins list structure. Also matches RIGHTBRACKET characters.

RIGHTBRACKET (normally left bracket) Ends list structure. Can close an arbitrary numbers of LEFTPAREN lists, back to the last LEFTBRACKET.

STRINGDELIM (normally double quote) Begins and ends text strings. Within the string, all characters except for the one(s) with class ESCAPE are treated as ordinary, i.e., interpreted as if they were of syntax class OTHER. To include the string delimiter inside a string, prefix it with the ESCAPE character.

ESCAPE (normally percent sign) Inhibits any special interpretation of the next character, i.e., the next character is interpreted to be of class OTHER, independent of its normal syntax class.

BREAKCHAR (None initially) Is a break character, i.e., delimits atoms, but is otherwise an ordinary character.

SEPRCHAR (space, carriage return, etc.) Delimits atoms, and is otherwise ignored.

OTHER Characters that are not otherwise special belong to the class OTHER.

Characters of syntax class LEFTPAREN, RIGHTPAREN, LEFTBRACKET, RIGHTBRACKET, and STRINGDELIM are all *break* characters. That is, in addition to their interpretation as delimiting list or string structures, they also terminate the reading of an atom. Characters of class BREAKCHAR serve *only* to terminate atoms, with no other special meaning. In addition, if a break character is the first non-separator encountered by RATOM, it is read as a one-character atom. In order for a break character to be included in an atom, it must be preceded by the ESCAPE character.

Characters of class SEPRCHAR also terminate atoms, but are otherwise completely ignored; they can be thought of as logically spaces. As with break characters, they must be preceded by the ESCAPE character in order to appear in an atom.

For example, if $ were a break character and * a separator character, the input stream ABC**DEF$GH*$$ would be read by six calls to RATOM returning respectively ABC, DEF, $, GH, $, $.

Although normally there is only one character in a read table having each of the list- and string-delimiting syntax classes (such as LEFTPAREN), it is perfectly acceptable for any character to have any syntax class, and for more than one to have the same class.

Note that a "syntax class" is an abstraction: there is no object referencing a collection of characters called a *syntax class*. Instead, a read table provides the *association* between a character and its syntax class, and the input/output routines enforce the abstraction by using read tables to drive the parsing.

The functions below are used to obtain and set the syntax class of a character in a read table. *CH* can either be a character code (a integer), or a character (a single-character atom). Single-digit integers are interpreted as character codes, rather than as characters. For example, 1 indicates Control-A, and 49 indicates the *character* 1. Note that *CH* can be a full sixteen-bit NS character (see Chapter 2).

Note: Terminal tables, described in Chapter 30, also associate characters with syntax classes, and they can also be manipulated with the functions below. The set of read table and terminal table syntax classes are disjoint, so there is never any ambiguity about which type of table is being referred to.

(**GETSYNTAX** *CH TABLE*) [Function]

Returns the syntax class of *CH*, a character or a character code, with respect to *TABLE*. *TABLE* can be NIL, T, ORIG, or a real read table or terminal table.

CH can also be a syntax class, in which case GETSYNTAX returns a list of the character codes in *TABLE* that have that syntax class.

(**SETSYNTAX** *CHAR CLASS TABLE*) [Function]

Sets the syntax class of *CHAR*, a character or character code, in *TABLE*. *TABLE* can be either NIL, T, or a real read table or terminal table. SETSYNTAX returns the previous syntax class of *CHAR*. *CLASS* can be any one of the following:

- The name of one of the basic syntax classes.
- A list, which is interpreted as a read macro (see below).
- NIL, T, ORIG, or a real read table or terminal table, which means to give *CHAR* the syntax class it has in the table indicated by *CLASS*. For example, (SETSYNTAX '%('ORIG *TABLE*) gives the left parenthesis character in *TABLE* the same syntax class that it has in the original system read table.
- A character code or character, which means to give *CHAR* the same syntax class as the character *CHAR* in *TABLE*. For example, (SETSYNTAX '{ '%[*TABLE*) gives the left brace character the same syntax class as the left bracket.

(**SYNTAXP** *CODE CLASS TABLE*) [Function]

CODE is a character code; *TABLE* is NIL, T, or a real read table or terminal table. Returns T if *CODE* has the syntax class *CLASS* in *TABLE*; NIL otherwise.

CLASS can also be a read macro type (MACRO, SPLICE, INFIX), or a read macro option (FIRST, IMMEDIATE, etc.), in which case SYNTAXP returns T if the syntax class is a read macro with the specified property.

SYNTAXP will *not* accept a character as an argument, only a character *code*.

For convenience in use with SYNTAXP, the atom BREAK may be used to refer to *all* break characters, i.e., it is the union of LEFTPAREN, RIGHTPAREN, LEFTBRACKET, RIGHTBRACKET, STRINGDELIM, and BREAKCHAR. For purely symmetrical reasons, the atom SEPR corresponds to all separator characters. However, since the only separator characters are those that also appear in SEPRCHAR, SEPR and SEPRCHAR are equivalent.

Note that GETSYNTAX never returns BREAK or SEPR as a value although SETSYNTAX and SYNTAXP accept them as arguments. Instead, GETSYNTAX returns one of the disjoint basic syntax classes that comprise BREAK. BREAK as an argument to SETSYNTAX is interpreted to mean BREAKCHAR if the character is not already of one of the BREAK classes. Thus, if %(is of class LEFTPAREN, then (SETSYNTAX '%('BREAK) doesn't do anything, since %(is already a break character, but (SETSYNTAX '%('BREAKCHAR) means make %(be *just* a break character, and therefore disables the LEFTPAREN function of %(. Similarly, if one of the format characters is disabled completely, e.g., by (SETSYNTAX '%('OTHER),

then (SETSYNTAX '%('BREAK) would make %(be *only* a break character; it would *not* restore %(as LEFTPAREN.

The following functions provide a way of collectively accessing and setting the separator and break characters in a read table:

(**GETSEPR** *RDTBL*) [Function]

Returns a list of separator character codes in *RDTBL*. Equivalent to (GETSYNTAX 'SEPR *RDTBL*).

(**GETBRK** *RDTBL*) [Function]

Returns a list of break character codes in *RDTBL*. Equivalent to (GETSYNTAX 'BREAK *RDTBL*).

(**SETSEPR** *LST FLG RDTBL*) [Function]

Sets or removes the separator characters for *RDTBL*. *LST* is a list of charactors or character codes. *FLG* determines the action of SETSEPR as follows: If *FLG*=NIL, makes *RDTBL* have exactly the elements of *LST* as separators, discarding from *RDTBL* any old separator characters not in *LST*. If *FLG*=0, removes from *RDTBL* as separator characters all elements of *LST*. This provides an "UNSETSEPR". If *FLG*=1, makes each of the characters in *LST* be a separator in *RDTBL*.

If *LST*=T, the separator characters are reset to be those in the system's read table for terminals, regardless of the value of *FLG*, i.e., (SETSEPR T) is equivalent to (SETSEPR (GETSEPR T)). If *RDTBL* is T, then the characters are reset to those in the original system table.

Returns NIL.

(**SETBRK** *LST FLG RDTBL*) [Function]

Sets the break characters for *RDTBL*. Similar to SETSEPR.

As with SETSYNTAX to the BREAK class, if any of the list- or string-delimiting break characters are disabled by an appropriate SETBRK (or by making it be a separator character), its special action for READ will *not* be restored by simply making it be a break character again with SETBRK. However, making these characters be break characters when they already *are* will have no effect.

The action of the ESCAPE character (normally %) is not affected by SETSEPR or SETBRK. It can be disabled by setting its syntax to the class OTHER, and other characters can be used for escape on input by assigning them the class ESCAPE. As of this writing, however, there is no way to change the output escape character; it is "hardwired" as %. That is, on output, characters of special syntax that need to be preceded by the ESCAPE character will always be preceded by %, independent of the syntax of % or which, if any characters, have syntax ESCAPE.

The following function can be used for defeating the action of the ESCAPE character or characters:

(**ESCAPE** *FLG RDTBL*) [Function]

If *FLG*=NIL, makes characters of class ESCAPE behave like characters of class OTHER on input. Normal setting is (ESCAPE T). ESCAPE returns the previous setting.

Read Macros

This is a description of the OLD-INTERLISP-T read macros. Read macros are user-defined syntax classes that can cause complex operations when certain characters are read. Read macro characters are defined by specifying as a syntax class an expression of the form:

(*TYPE* $OPTION_1$ ... $OPTION_N$ *FN*)

where *TYPE* is one of MACRO, SPLICE, or INFIX, and *FN* is the name of a function or a lambda expression. Whenever READ encounters a read macro character, it calls the associated function, giving it as arguments the input stream and read table being used for that call to READ. The interpretation of the value returned depends on the type of read macro:

MACRO This is the simplest type of read macro. The result returned from the macro is treated as the expression to be read, instead of the read macro character. Often the macro reads more input itself. For example, in order to cause ~EXPR to be read as (NOT EXPR), one could define ~ as the read macro:

```
[MACRO (LAMBDA (FL RDTBL)
         (LIST 'NOT (READ FL RDTBL]
```

SPLICE The result (which should be a list or NIL) is spliced into the input using NCONC. For example, if $ is defined by the read macro:

```
(SPLICE (LAMBDA NIL (APPEND FOO)))
```

and the value of FOO is (A B C), then when you input (X $ Y), the result will be (X A B C Y).

INFIX The associated function is called with a third argument, which is a list, in TCONC format (Chapter 3), of what has been read at the current level of list nesting. The function's value is taken as a new TCONC list which replaces the old one. For example, the infix operator + could be defined by the read macro:

```
(INFIX (LAMBDA (FL RDTBL Z)
         (RPLACA (CDR Z)
            (LIST (QUOTE IPLUS)
                  (CADR Z)
                  (READ FL RDTBL)))
         Z))
```

If an INFIX read macro character is encountered *not* in a list, the third argument to its associated function is NIL. If the function returns NIL, the read macro character is essentially ignored and reading continues. Otherwise, if the function returns a TCONC list of one element, that element is the value of

the READ. If it returns a TCONC list of more than one element, the list is the value of the READ.

The specification for a read macro character can be augmented to specify various options $OPTION_1$ ... $OPTION_N$, e.g., (MACRO FIRST IMMEDIATE *FN*). The following three disjoint options specify when the read macro character is to be effective:

ALWAYS — The default. The read macro character is always effective (except when preceded by the % character), and is a break character, i.e., a member of (GETSYNTAX 'BREAK *RDTBL*).

FIRST — The character is interpreted as a read macro character *only* when it is the first character seen after a break or separator character; in all other situations, the character is treated as having class OTHER. The read macro character is *not* a break character. For example, the quote character is a FIRST read macro character, so that DON'T is read as the single atom DON'T, rather than as DON followed by (QUOTE T).

ALONE — The read macro character is *not* a break character, and is interpreted as a read macro character only when the character would have been read as a separate atom if it were not a read macro character, i.e., when its immediate neighbors are both break or separator characters.

Making a FIRST or ALONE read macro character be a break character (with SETBRK) disables the read macro interpretation, i.e., converts it to syntax class BREAKCHAR. Making an ALWAYS read macro character be a break character is a no-op.

The following two disjoint options control whether the read macro character is to be protected by the ESCAPE character on output when a litatom containing the character is printed:

ESCQUOTE or ESC — The default. When printed with PRIN2, the read macro character will be preceded by the output escape character (%) as needed to permit the atom containing it to be read correctly. Note that for FIRST macros, this means that the character need be quoted only when it is the first character of the atom.

NOESCQUOTE or NOESC — The read macro character will always be printed without an escape. For example, the **?** read macro in the T read table is a NOESCQUOTE character. Unless you are very careful what you are doing, read macro characters in FILERDTBL should never be NOESCQUOTE, since symbols that happen to contain the read macro character will not read back in correctly.

The following two disjoint options control when the macro's function is actually executed:

IMMEDIATE or IMMED — The read macro character is immediately activated, i.e., the current line is terminated, as if an EOL had been typed, a carriage-return line-feed is printed,

and the entire line (including the macro character) is passed to the input function.

IMMEDIATE read macro characters enable you to specify a character that will take effect immediately, as soon as it is encountered in the input, rather than waiting for the line to be terminated. Note that this is not necessarily as soon as the character is *typed*. Characters that cause action as soon as they are typed are interrupt characters (see Chapter 30).

Note that since an IMMEDIATE macro causes any input before it to be sent to the reader, characters typed before an IMMEDIATE read macro character cannot be erased by Control-A or Control-Q once the IMMEDIATE character has been typed, since they have already passed through the line buffer. However, an INFIX read macro can still alter some of what has been typed earlier, via its third argument.

NONIMMEDIATE or NONIMMED The default. The read macro character is a normal character with respect to the line buffering, and so will not be activated until a carriage-return or matching right parenthesis or bracket is seen.

Making a read macro character be both ALONE and IMMEDIATE is a contradiction, since ALONE requires that the next character be input in order to see if it is a break or separator character. Thus, ALONE read macros are always NONIMMEDIATE, regardless of whether or not IMMEDIATE is specified.

Read macro characters can be "nested". For example, if = is defined by

```
(MACRO (LAMBDA (FL RDTBL)
    (EVAL (READ FL RDTBL))))
```

and ! is defined by

```
(SPLICE (LAMBDA (FL RDTBL)
    (READ FL RDTBL)))
```

then if the value of FOO is (A B C), and (X =FOO Y) is input, (X (A B C) Y) will be returned. If (X !=FOO Y) is input, (X A B C Y) will be returned.

Note: If a read macro's function calls READ, and the READ returns NIL, the function cannot distinguish the case where a RIGHTPAREN or RIGHTBRACKET followed the read macro character, (e.g. "(A B)"), from the case where the atom NIL (or "()") actually appeared. In Interlisp-D, a READ inside of a read macro when the next input character is a RIGHTPAREN or RIGHTBRACKET reads the character and returns NIL, just as if the READ had not occurred inside a read macro.

If a call to READ from within a read macro encounters an unmatched RIGHTBRACKET *within* a list, the bracket is simply put back into the buffer to be read (again) at the higher level. Thus, inputting an expression such as (A B '(C D] works correctly.

(**INREADMACROP**) [Function]

Returns NIL if currently *not* under a read macro function, otherwise the number of unmatched left parentheses or brackets.

(**READMACROS** *FLG RDTBL*) [Function]

If *FLG*=NIL, turns off action of read macros in read table *RDTBL*. If *FLG*=T, turns them on. Returns previous setting.

The following read macros are standardly defined in Interlisp in the T and EDITRDTBL read tables:

' (single-quote) Returns the next expression, wrapped in a call to QUOTE; e.g., 'FOO reads as (QUOTE FOO). The macro is defined as a FIRST read macro, so that the quote character has no effect in the middle of a symbol. The macro is also ignored if the quote character is immediately followed by a separator character.

Control-Y Defined in T and EDITRDTBL. Returns the result of evaluating the next expression. For example, if the value of FOO is (A B), then (LIST 1 *control-Y*FOO 2) is read as (LIST 1 (A B) 2). Note that no structure is copied; the third element of that input expression is still EQ to the value of FOO. Control-Y can thus be used to read structures that ordinarily have no read syntax. For example, the value returned from reading (KEY1 *Control-Y*(ARRAY 10)) has an array as its second element. Control-Y can be thought of as an "un-quote" character. The choice of character to perform this function is changeable with SETTERMCHARS (see Chapter 16).

` (backquote) Backquote makes it easier to write programs to construct complex data structures. Backquote is like quote, except that within the backquoted expression, forms can be evaluated. The general idea is that the backquoted expression is a "template" containing some constant parts (as with a quoted form) and some parts to be filled in by evaluating something. Unlike with control-Y, however, the evaluation occurs not at the time the form is read, but at the time the backquoted expression is evaluated. That is, the backquote macro returns an expression which, when evaluated, produces the desired structure.

Within the backquoted expression, the character "," (comma) introduces a form to be evaluated. The value of a form preceded by ",@" is to be spliced in, using APPEND. If it is permissible to destroy the list being spliced in (i.e., NCONC may be used in the translation), then ",." can be used instead of ",@".

For example, if the value of FOO is (1 2 3 4), then the form

```
`(A ,(CAR FOO) ,@(CDDR FOO) D E)
```

evaluates to `(A 1 3 4 D E)`; it is logically equivalent to writing

```
(CONS 'A
   (CONS (CAR FOO)
      (APPEND (CDDR FOO) '(D E))))
.
```

Backquote is particularly useful for writing macros. For example, the body of a macro that refers to `X` as the macro's argument list might be

```
`(COND
   ((FIXP ,(CAR X))
      ,(CADR X))
   (T .,(CDDR X)))
```

which is equivalent to writing

```
(LIST 'COND
      (LIST (LIST 'FIXP (CAR X))
            (CADR X))
      (CONS 'T (CDDR X)))
```

Note that comma does *not* have any special meaning outside of a backquote context.

For users without a backquote character on their keyboards, backquote can also be written as `|'` (vertical-bar, quote).

`?` Implements the `?=` command for on-line help regarding the function currently being "called" in the typein (see Chapter 26).

`|` (vertical bar) When followed by an end of line, tab or space, `|` is ignored, i.e., treated as a separator character, enabling the editor's `CHANGECHAR` feature (see Chapter 26). Otherwise it is a "dispatching" read macro whose meaning depends on the character(s) following it. The following are currently defined:

`'` (quote) -- A synonym for backquote.

`.` (period) -- Returns the evaluation of the next expression, i.e., this is a synonym for Control-Y.

, (comma) -- Returns the evaluation of the next expression *at load time*, i.e., the following expression is quoted in such a manner that the compiler treats it as a literal whose value is not determined until the compiled expression is loaded.

`O` or `o` (the letter O) -- Treats the next number as octal, i.e., reads it in radix 8. For example, `|o12` = `10` (decimal).

`B` or `b` -- Treats the next number as binary, i.e., reads it in radix 2. For example, `|b101` = `5` (decimal).

`X` or `x` -- Treats the next number as hexadecimal, i.e., reads it in radix 16. The uppercase letters `A` though `F` are used as the digits after `9`. For example, `|x1A` = `26` (decimal).

`R` or `r` -- Reads the next number in the radix specified by the (decimal) number that appears between the `|` and the `R`. When inputting a number in a radix above ten, the upper-case letters `A` through `Z` can be used as the digits after `9` (but there is no digit above `Z`, so it is not possible to type all base-99 digits). For example, `|3r120` reads 120 in radix 3, returning 15.

`(`, `{`, `^` -- Used internally by `HPRINT` and `HREAD` (see above) to print and read unusual expressions.

The dispatching characters that are letters can appear in either upper- or lowercase.