

## CHAPTER 6

### INPUT/OUTPUT

#### 6.1 FILES

All input/output functions in Interlisp can specify their source/destination *le* with an optional extra argument, which is the name of the *le*, given as a litatom. These functions generally require that the *le* be *open*. Files are opened and manipulated by the functions described below. The name *T* designates terminal input and output, and is always considered open. It is also possible to supply a string as an input “*le*”, without needing to open it; input operations remove successive characters from the string. Note that because of this feature, *le* names must always be specified as litatoms, not strings.

(*OPENFILE* *FILE* *ACCESS* *RECOG* *BYTESIZE* *MACHINE.DEPENDENT.PARAMETERS* ) [Function]  
Opens *FILE* with access rights as specified by *ACCESS*, one of *INPUT*, *OUTPUT*, *BOTH*, or *APPEND*, and returns the full name of the *le*. Causes error *FILE NOT FOUND* if *FILE* is not recognized by the *le* system, or other errors if *FILE* is recognized but cannot be opened, e.g. *FILE WON'T OPEN* if the *le* is already opened by someone else or is protected against the operation, *FILE SYSTEM RESOURCES EXCEEDED* if there is no more room in the *le* system.

For *ACCESS* = *INPUT*, only input operations are permitted on the *le*; for *ACCESS* = *OUTPUT* or *ACCESS* = *APPEND*, only output operations are permitted. Note: in Interlisp-10 and Interlisp-D, *ACCESS* = *OUTPUT* implies that one intends to write a new or different *le*, even if a version number was specified and the corresponding *le* already exists. Thus any previous contents of the *le* are discarded, and the *le* is empty immediately after the *OPENFILE*. If it is desired to write on an already existing *le* while preserving the old contents, the *le* must be opened for access *BOTH* or *APPEND*.

*RECOG* specifies the recognition mode of *FILE*, as described on page 6.4. If *RECOG* = *NIL*, it defaults according to the value of *ACCESS*: for *ACCESS* = *INPUT*, *RECOG* = *OLD* is used; for *ACCESS* = *OUTPUT*, *RECOG* = *NEW* is used; for the other values of *ACCESS*, *RECOG* = *OLD/NEW* is used.

*BYTESIZE*, if supplied, is the byte size in which to open the *le*. If *BYTESIZE* = *NIL*, the bytesize used is the default for the implementation (8 for Interlisp-D, 7 for Interlisp-10).

*MACHINE.DEPENDENT.PARAMETERS* is a list specifying additional opening parameters. In Interlisp-10, this list may contain the following litatoms:

*WAIT*        Wait if *le* is busy.

*DON'T.CHANGE.DATE*

## Files

Don't change the access dates.

THAWED    Open `le` in "thawed" mode.

In Interlisp-D, `MA CHINE.DEPENDENT.P ARAMETERS` should be a list of pairs `(ATTRIB VALUE)`, where `ATTRIB` is any `le` attribute that the `le` system is willing to allow the user to set (see `SETFILEINFO`, page 6.7).

If the `FILE` argument to an input (output) function is not given (has value `NIL`), the `le` specified as "primary" for input (output) is used. Normally these are both `T`, for terminal input and output. However, the primary input or output `le` may be changed with the functions below.

`(INPUT FILE)` [Function]  
Sets `FILE` as the primary input `le`; returns the name of the old primary input `le`. `FILE` must be open for input. `INPUT` can also be given a string as argument, interpreted as described above.

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

`(OUTPUT FILE)` [Function]  
Sets `FILE` as the primary output `le`; returns the name of the old primary output `le`. `FILE` must be open for output. A string cannot be used as an output `le`.

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

`(INFILE FILE)` [Function]  
Opens `FILE` for input, and sets it as the primary input `le`. Equivalent to `(INPUT (OPENFILE FILE 'INPUT 'OLD))`

`(OUTFILE FILE)` [Function]  
Opens `FILE` for output, and sets it as the primary output `le`. Equivalent to `(OUTPUT (OPENFILE FILE 'OUTPUT 'NEW))`.

`(IOFILE FILE)` [Function]  
`(OPENFILE FILE 'BOTH 'OLD)`; opens `FILE` for both input and output. Does not affect the primary input or output `le`.

`(OPENP FILE ACCESS)` [Function]  
If `ACCESS = NIL`, returns the full name of `FILE` if `FILE` is open either for input or for output; otherwise `NIL`.

If `ACCESS` is `INPUT`, `OUTPUT` or `BOTH`, returns the full name of `FILE` if it is open in that access mode; otherwise `NIL`.

Note: If `FILE` is not recognized, `OPENP` returns `NIL` without generating an error.

`(OPENP)` returns a list of all `les` open for input or output, excluding `T` and the current typescript (dribble) `le`, if any (page 6.12).

`(CLOSEF FILE)` [Function]  
Closes `FILE`. Generates an error, `FILE NOT OPEN`, if `FILE` is not open. If `FILE` is `NIL`, it attempts to close the primary input `le` if other than terminal. Failing that, it attempts to close the primary output `le` if other than terminal. Failing both, it

## INPUT/OUTPUT

returns NIL. If it closes any *le*, it returns the name of that *le*. If it closes either of the primary *les*, it resets that primary *le* to terminal.

WHENCLOSE (page 6.11) allows the user to “advise” CLOSEF to perform various operations when a *le* is closed.

(CLOSEF? FILE) [Function]

Closes FILE if it is open, otherwise does nothing. Returns FILE.

(CLOSEALL ALLFL G) [Function]

Closes all open *les*, except T and the current typescript *le*, if any. Returns a list of the *les* closed.

WHENCLOSE (page 6.11) allows certain *les* to be “protected” from CLOSEALL. (CLOSEALL T) overrides this protection.

(DELFILE FILE) [Function]

Deletes FILE if possible. Returns FILE if deleted, else NIL.

(RENAMEFILE OLDFILE NEWFILE) [Function]

Renames OLDFILE to be NEWFILE. Returns NEWFILE if successful, else NIL.

### 6.1.1 File Naming and Recognition

In Interlisp, a *le* name is a literal atom composed of one or more *elds*, separated by suitable punctuation. The precise *elds* and their interpretation is dependent on the implementation; the functions PACKFILENAME and UNPACKFILENAME (page 6.6) are used to construct and take apart *lenames* in an implementation-independent way.

Depending on the *le* system implementation, *le* names given to input/output functions may be incompletely specified, with the *le* system handling the task of obtaining a specific *le* from a partial name, or *recognizing* the *le*. For example, in *le* systems that support version numbers, one can call OPENFILE giving a *le* name without a version number, and the *le* system will supply a default version number based on the context (opening a new *le* for output vs. an old *le* for input). Internally, however, each open *le* has associated with it a completely-specified *lename*, one that uniquely identifies the *le* to the *le* system in any context. It is this “full” *le* name that is returned from OPENFILE and other functions that return names of open *les*. For example, (OPENFILE 'FOO 'OUTPUT) might return <LISP>FOO.;3. Any time that an input/output function is called with a *le* name other than the full *le* name, Interlisp must perform recognition on the partial *le* name in order to determine which open *le* is intended. Thus if repeated operations are to be performed, it is considerably more efficient to use the full *le* name returned from OPENFILE than to repeatedly use the possibly incomplete name that was used to open the *le*.

In Interlisp-10, *lenames* follow the conventions of the operating system (either TENEX or TOPS-20), i.e., FILE can be prefixed by a directory name enclosed in angle brackets, can contain <esc>s or control-F's, and can include suffixes and/or version numbers. When a *le* is opened for input and no version number is given, the highest existing version number is used. Similarly, when a *le* is opened for output and no version number is given, a new *le* is created with a version number one higher than the highest one currently in use with that *le* name. The full *lename* in Interlisp-10 consists of directory, name, extension, and version. In Interlisp-D, it also includes a device or host name in brackets, i.e.,

## File Naming and Recognition

{PHYLUM}<LISP>FOO.;3).

The following functions can be used to perform le recognition without opening a le:

*Warning: In some implementations of Interlisp (such as Interlisp-D), it may not be possible to determine the full name of a new le without trying to open it. In this case, OUTFILEP and FULLNAME may not always return the correct value. These functions should not be used in general, because the idea “what a le would be named if it were opened” is not well defined in some le systems.*

(INFILEP FILE) [Function]  
Returns full le name of FILE if FILE is recognized as specifying the name of an existing le that could potentially be opened for input, NIL otherwise. Recognition is in input context, i.e., in Interlisp-10, if no version number is given, the highest existing version number is returned.

(OUTFILEP FILE) [Function]  
Similar to INFILEP, except recognition is in output context, i.e., in Interlisp-10, if no version number is given, a version number one higher than the highest existing version number is returned. Roughly speaking, OUTFILEP returns the full name of the le that would be created if OUTFILE were called with the same argument.

A more general version of INFILEP and OUTFILEP is provided by the function FULLNAME:

(FULLNAME X RECOG) [Function]  
If x is recognized in the recognition mode specified by RECOG as an abbreviation for some le, returns the le's full name, otherwise NIL. RECOG can be OLD, meaning choose the (newest) existing version of the le; NEW, meaning make the full le name one which does not yet exist (version number one higher than highest existing version); OLDEST, meaning choose the existing le with the lowest version number; or OLD/NEW, meaning to recognize an existing version if possible, otherwise a new version (useful only for writing a le). RECOG = NIL defaults to OLD. For all other values of RECOG, generates an error ILLEGAL ARG. If x is not a literal atom, generates an error, ARG NOT LITATOM.

For example, INFILEP could be defined as (FULLNAME FILE 'OLD) and OUTFILEP as (FULLNAME FILE 'NEW).

The RECOG argument is used only for defaulting unspecified parts of the lename (in Interlisp-10 and Interlisp-D, the version), not to pass judgment on the specified parts. In particular, RECOG = NEW does not require that the le be new. For example, (FULLNAME 'FOO.;2 'NEW) may return <MASINTER>FOO.;2 if that le already exists, even though (FULLNAME 'FOO 'NEW) would default the version to a new number, perhaps returning <MASINTER>FOO.;5.

Note that INFILEP, OUTFILEP and FULLNAME do not open any les, or change the primary les; they are pure predicates. In general they are also only hints, as they do not necessarily imply that the caller has access rights to the le. For example, INFILEP might return non-NIL, but OPENFILE might fail for the same le because the le is read-protected against the user, or the le happens to be open for output by another user at the time. Similarly, OUTFILEP could return non-NIL, but OPENFILE could fail with a FILE SYSTEM RESOURCES EXCEEDED error. Note also that in a multi-user le system, intervening le operations by another user could contradict the information returned by recognition. For example, a le that was INFILEP might be deleted, or between an OUTFILEP and the subsequent OPENFILE,

## INPUT/OUTPUT

another user might create a new version or delete the highest version, causing the names returned by `OUTFILEP` and `OPENFILE` to have different version numbers. Thus, in general, the “truth” about a le can only be obtained by actually opening the le; in particular, creators of les should rely on the name returned from `OPENFILE`, not from `OUTFILEP`.

If the le system does not successfully recognize an incomplete le name, a `FILE NOT FOUND` error is generated (except for `INFILEP`, `OUTFILEP`, `FULLNAME` and `OPENP`, which in this case return `NIL`). As described on page 9.16, before a `FILE NOT FOUND` error occurs, it is intercepted via an entry on `ERRORTYPELIST`, which causes `SPELLFILE` (page 15.20) to be called. `SPELLFILE` will search alternate directories and possibly attempt spelling correction on the le name. Only if `SPELLFILE` is unsuccessful will the error actually occur.

Note that recognition is performed on the user’s entire directory, not just the open les, which can result in certain anomalies. Thus, even if only one le is open, say `FOO. ;1`, the name `F$` (`F<esc>`) will not be recognized if the user’s directory also contains the le `FIE. ;1`. Similarly, it is possible for a le name that was previously recognized to become ambiguous. For example, a program performs `(INFILE 'FOO)`, opening `FOO. ;1`, and reads several expressions from `FOO`. Then the user interrupts the program, creates a `FOO. ;2` and reenters his program. Now a call to `READ` giving it `FOO` as its `FILE` argument will generate a `FILE NOT OPEN` error, because `FOO` will be recognized as `FOO. ;2`.

### 6.1.2 Manipulating File Names

Different operating systems have different conventions for naming les. However, it is desirable for Interlisp to be as implementation independent as possible. Therefore, all programs that need to reference parts of a filename, or construct new le names from existing ones, should use the functions described below. The implementation of these functions obviously is dependent on the operating system they will run under, but as far as the programs that use them are concerned, they permit expressing operations that are implementation independent.<sup>1</sup>

Every le name is composed of a collection of *elds* which have different semantic interpretations. A *eld name* is a literal atom which is the name of a le-name eld. Interlisp assumes that `NAME` and `EXTENSION` are valid eld names; the implementor is free to allow other elds. In Interlisp-10, allowable eld names are: `DEVICE`, `DIRECTORY`, `NAME`, `EXTENSION`, `VERSION`, `PROTECTION`, `ACCOUNT`, and `TEMPORARY`. Interlisp-D allows `HOST`, `DIRECTORY`, `NAME`, `EXTENSION`, and `VERSION`.

`(FILENAMEFIELD FILENAME FIELDNAME )` [Function]  
Returns the contents of the `FIELDNAME` eld of `FILENAME` .

`(UNPACKFILENAME FILENAME _ )` [Function]  
Returns a list of alternating eld names and eld contents.

Examples from Interlisp-D:

```
_ (UNPACKFILENAME 'FOO.BAR)
(NAME FOO EXTENSION BAR)
_ (UNPACKFILENAME '{PHYLUM}<SANNELLA>LISP>IMTRAN.DCOM;21)
```

---

<sup>1</sup>In particular, the Interlisp-10 implementation recognizes le names in both Tenex and TOPS-20 format, and builds new names as appropriate.

## File Attributes

```
(HOST PHYLUM DIRECTORY SANNELLA>LISP NAME IMTRAN  
EXTENSION DCOM VERSION 21)
```

Examples from Interlisp- 10 on Tenex:

```
_ (UNPACKFILENAME ' <LISP>MAC.COM;3)  
(DIRECTORY LISP NAME MAC EXTENSION COM VERSION 3)  
_ (UNPACKFILENAME 'WORK.;T)  
(NAME WORK EXTENSION NIL TEMPORARY T)
```

Note: In Interlisp- 10, (UNPACKFILENAME 'DSK:FOO) returns (DEVICE DSK:  
NAME FOO), i.e. the : is left in. This is so (DEVICE NIL:) may be distinguished  
from (DEVICE NIL).

```
(PACKFILENAME FIELDNAME 1 FIELDCONTENTS 1 FIELDNAME N FIELDCONTENTS N)  
[NoSpread Function]
```

Takes a list of alternating eld names and eld contents (atoms or strings),  
and returns the corresponding le name. For example, (PACKFILENAME  
'DIRECTORY 'LISP 'NAME 'NET) returns <LISP>NET.

If the same eld name is given twice, the *rst* occurrence is used.

If the “eld name” BODY is given, this means that the operand to BODY should  
itself be unpacked and spliced into the argument list at that point. This is useful  
for providing default eld names, or to change just one eld in an existing name.

For example, to take a le name FILE and change the DIRECTORY eld, perform  
(PACKFILENAME 'DIRECTORY NEWDIRECTOR Y 'BODY FILE). Alternatively,  
to provide a default for the EXTENSION eld, perform (PACKFILENAME 'BODY  
FILE 'EXTENSION DEFAULT). This uses DEFAULT as the extension unless one is  
already speci ed in FILE.

Note that a null eld is a eld that *has* been speci ed, e.g., if FILE= FOO;1 in the  
above example, the default extension will be used, but if FILE= FOO.;1, it will  
not, because a null extension has been speci ed.

If the *rst* argument to PACKFILENAME is a list, PACKFILENAME is called on that  
argument. Thus PACKFILENAME and UNPACKFILENAME operate as inverses.

### 6.1.3 File Attributes

Any le has a number of “le attributes”, such read date, protection, and bytesize. The exact attributes  
that a le can have is implementation- dependent. The functions GETFILEINFO and SETFILEINFO  
allow the user to conveniently access le attributes:

```
(GETFILEINFO FILE ATTRIB ) [Function]  
Returns the current setting of the ATTRIB attribute of FILE. In Interlisp- 10, FILE  
may also be a JFN as returned by GTJFN (page 22.22).
```

In Interlisp- 10, GETFILEINFO takes an optional third argument, SCRA TCH , which  
is analogous to the third argument of GDATE (page 14.10): a string pointer to reuse

## INPUT/OUTPUT

for those `ATTRIB` 's which return string values.

(`SETFILEINFO FILE ATTRIB VALUE`) [Function]  
 Sets the attribute `ATTRIB` of `FILE` to be `VALUE`. `SETFILEINFO` returns `T` if it is able to change the attribute `ATTRIB`, and `NIL` if unsuccessful (some attributes cannot be changed, e.g. it doesn't make sense to change the `SIZE` of a `le` without writing something on it).

`GETFILEINFO` and `SETFILEINFO` currently recognize the following values for `ATTRIB`:

`ACCESS`                The current access mode of `FILE` (e.g. `INPUT`, `OUTPUT`, `BOTH`, `APPEND`) or `NIL` if `FILE` is not open.

`BYTESIZE`             The byte size of the `le`.

`LENGTH`               The byte position of the end-of-`le`. Like (`GETEOFPTR FILE`), but `FILE` does not have to be open.

`SIZE`                   The size of `FILE` in pages.

`WRITEDATE`, `READDATE`, `CREATIONDATE`  
 The date (and time) as a string that `FILE` was respectively last written, last read, and originally created.

`IWRITEDATE`, `IREADDATE`, `ICREATIONDATE`  
 The respective date in integer form, as `IDATE` (page 14.10) would return.

`TYPE`                   (Interlisp- D) Either `TEXT` or `BINARY`.

`OPENBYTESIZE`        (Interlisp- 10) It is possible that the byte size for the “opening” of a `le` might differ from the “permanent” bytesize. For example, a 7-bit text `le` can be opened in 36-bit mode. To obtain the “open” bytesize, use attribute `OPENBYTESIZE`.

`PROTECTION`           (Interlisp- 10) The “protection code” of `FILE`, as an integer.

`DELETED`               (Interlisp- 10) `T` if `FILE` is the name of a deleted `le`, `NIL` otherwise.

Additional attributes which are available for Interlisp- 10 on TOPS-20 systems (DEC release 4 or later) are:

`INVISIBLE`            `T` if `FILE` has the invisible attribute, `NIL` otherwise.

`ARCHIVED`             `T` if `FILE` has been archived, `NIL` otherwise.

`OFF-LINE`              `T` if the contents of `FILE` are o - line (i.e. `FILE` has been archived and its contents flushed), `NIL` otherwise.

(`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 (`POSITION FILE`) is *not* the same as (`GETFILEPTR FILE`) which gives the position in the `le`, not on the *line*.

## Randomly Accessible Files

(LINELENGTH N FILE) [Function]  
Sets the length of the print line for the output le FILE to N; returns the former setting of the line length. FILE defaults to the primary output le. (LINELENGTH NIL FILE) returns the current setting for FILE. When a le is rst opened, its linelength is set to the value of the variable FILELINELENGTH.

Whenever printing an atom or string would increase a le's position *beyond* the line length of the le, an end of line is automatically inserted rst. This action can be defeated by using PRIN3 and PRIN4 (page 6.17).

(SETLINELENGTH N) [Function]  
If N is NIL, interrogates the operating system for the line length of the terminal device, and sets the variable TTYLINELENGTH to this value. If N is not NIL, instructs the operating system to set the terminal line length to N, and also sets TTYLINELENGTH to N. Then, in either case, SETLINELENGTH performs (and returns as its value) (LINELENGTH TTYLINELENGTH T).

Both AFTERSYSOUTFORMS and RESETFORMS (page 8.19) contain a (SETLINELENGTH) so that when the user rst runs a SYSOUT, or types control-D, the system obtains the latest information about the terminal.

### 6.1.4 Randomly Accessible Files

For most applications, les 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, les are written sequentially. However, it is also possible to read/write characters at arbitrary positions in a le, essentially treating the le as a large block of auxiliary storage. For example, one application might involve writing an expression at the *beginning* of the le, and then reading an expression from a specied point in its *middle*. This particular example requires the le be open for *both* input and output. However, random le input or output can also be performed on les that have been opened for only input or only output.

Associated with each le is a "le pointer" that points to the location where the next character is to be read from or written to. The le pointer to a le is automatically advanced after each input or output operation. This section describes functions which can be used to *reposition* the le pointer on those les that can be randomly accessed. A le 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, les can be enlarged. For example, if the le pointer is positioned at the end of a le and anything is written, the le "grows." It is also possible to position the le pointer *beyond* the end of le and then to write. (If the program attempts to *read* beyond the end of le, an END OF FILE error occurs.) In this case, the le 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 le; it is not possible to make more room in the middle of a le. 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, which is 200 characters long, part of B will be altered.

The address of a character (byte) is the number of characters (bytes) that precede it in the le, i.e., 0 is the address of the beginning of the le. However, the user should be careful about computing the space needed for an expression, since end-of-line may be represented by a different number of characters in different implementations, even though NCHARS only counts it as one; e.g., end-of-line in Interlisp-10 les is represented as the two characters carriage-return, line-feed. Output functions may also introduce end-of-line's as a result of LINELENGTH considerations.



## INPUT/OUTPUT

- (GETFILEPTR FILE) [Function]  
Returns the current position of the le pointer for FILE, i.e., the byte address at which the next input/output operation will commence.
- (SETFILEPTR FILE ADR) [Function]  
Sets the le pointer for FILE to the position ADR ; returns ADR . The special value ADR = -1 is interpreted to mean the address of the end of le.<sup>2</sup>
- (GETEOFPTR FILE) [Function]  
Returns the byte address of the end of le, i.e., the number of bytes in the le. Equivalent to performing (SETFILEPTR FILE -1) and returning (GETFILEPTR FILE) except that it does not change the current le pointer.
- (EOFP FILE) [Function]  
Returns T if the le pointer to FILE is pointing to the end of le; NIL otherwise. FILE must be open for (at least) input, or an error is generated, FILE NOT OPEN.
- (RANDACCESSP FILE) [Function]  
Returns FILE if FILE is randomly accessible, NIL otherwise. The le T is not randomly accessible, nor are the les LPT:, NIL: in Interlisp- 10, or certain network le connections in Interlisp- D. FILE must be open or an error is generated, FILE NOT OPEN.
- (COPYBYTES SRCFIL DSTFIL START END) [Function]  
Copies bytes (characters) 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 le is reached.
- (FILEPOS PATTERN FILE START END SKIP TAIL CASEARRA Y) [Function]  
Analogous to STRPOS (page 2.31), but searches a le rather than a string. FILEPOS searches FILE for the string PATTERN . Search begins at START (or the current position of the le 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 le. If TAIL is T, and the search is successful, the value is the address of the rst character *after* the sequence of characters corresponding to PATTERN , instead of the starting address of the sequence. In either case, the le is left so that the next i/o operation begins at the address returned as the value of FILEPOS.

---

<sup>2</sup>Note: If a le is opened for output only, the end of le is initially zero, even if an old le by the same name had existed (see OPENFILE, page 6.1). If a le is opened for both input and output, the initial le pointer is the beginning of the le, but (SETFILEPTR FILE -1) will set it to the end of the le. If the le had been opened in append mode by (OPENFILE FILE 'APPEND), the le pointer right after opening would be set to the end of the existing le, in which case a SETFILEPTR to position the le at the end would be unnecessary.

## Randomly Accessible Files

`CASEARRA Y` should be a “casearray” that specifies that certain characters should be transformed to other characters before matching. Casearrays are returned by `CASEARRAY` or `SEPRCASE` below. `CASEARRA Y = NIL` means no transformation will be performed.

A casearray 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 le “through” `CASEARRA Y` in the sense that each character code is transformed into the corresponding character code from `CASEARRA Y` 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 casearrays.

For example, to search without regard to upper and lower case differences, `CASEARRA Y` would be a casearray 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 `CASEARRA Y` in which all of the break and separator characters mapped into the same code as space.

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

(`FFILEPOS` `PATTERN` `FILE` `START` `END` `SKIP` `TAIL` `CASEARRA Y`) [Function]

Like `FILEPOS`, except much faster in most applications.<sup>3</sup> `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 le 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` `OLD` `ARRA Y`) [Function]

Creates and returns a new casearray, with all elements set to themselves, to indicate the identity mapping.

(Interlisp- D) If `OLD` `ARRA Y` is given, it is reused.

(`SETCASEARRAY` `CASEARRA Y` `FROMCODE` `TOCODE`) [Function]

Modifies the casearray `CASEARRA Y` so that character code `FROMCODE` is mapped to character code `TOCODE`.

(`SEPRCASE` `CLFLG`) [Function]

Returns a new casearray 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 will be mapped into this character as well. This is useful for finding a delimited atom in a le. For example, if `PATTERN`

---

<sup>3</sup>In Interlisp- 10, a speedup of 10 to 50 times is typical. In Interlisp- D the speedup is much smaller.

## INPUT/OUTPUT

is " FOO ", and (SEPRCASE T) is used for CASEARRA Y, then FILEPOS will  
nd "(FOO\_".

### 6.1.5 Closing and Reopening Files

The function WHENCLOSE permits the user to associate certain operations with open les that govern how and when the le will be closed, and how the le's status will be restored when a SYSOUT is started up. The user can specify that certain functions will be executed before CLOSEF closes the le and/or after CLOSEF closes the le. The user can make a particular le be invisible to CLOSEALL, so that it will remain open across user invocations of CLOSEALL. Finally, the user can associate a status-saving function with a le which will be called before SYSOUT and which can specify what to do when a SYSOUT is restarted.

(WHENCLOSE FILE PROP<sub>1</sub> VAL<sub>1</sub> PROP<sub>N</sub> VAL<sub>N</sub>) [NoSpread Function]  
FILE must specify the name of an open le other than T (NIL defaults to the primary input le, if other than T, or primary output le if other than T). The remaining arguments specify properties to be associated with the full name of FILE. WHENCLOSE returns the full name of FILE as its value.

WHENCLOSE recognizes the following property names:

BEFORE VAL is a function that CLOSEF will apply to the full name of FILE just before it is closed. This might be used, for example, to copy information about the le from an in-core data structure to the le just before it is closed.

AFTER VAL is a function that CLOSEF will apply to the full name of FILE just after it is closed. This capability permits in-core data structures that know about the le to be cleaned up when the le is closed.

BEFORE and AFTER differ in their behavior with respect to SYSOUT. If a le that was open before SYSOUT does not have a STATUS function associated with it that causes the le to be successfully restored after the SYSOUT is started, then the le is considered to have been "closed" by the SYSOUT, and its AFTER function will be executed after the SYSOUT starts.

STATUS This property provides a way of restoring the status of les when a SYSOUT is resumed. VAL is a function that will be applied to the full name of FILE just before a SYSOUT. VAL is expected to return a list, CAR of which is a function which will be APPLY'd to the CDR when the SYSOUT is started up and which will restore the status of FILE. If the value of the APPLY is NIL, it is assumed the le could not be successfully restored, a warning message is printed, and then any AFTER functions associated with the le are executed.

The function PERMSTATUS (page 23.17) produces an expression for re-opening a le after SYSOUT and restoring as many of its attributes as possible.

CLOSEALL VAL is either YES or NO and determines whether FILE will be closed by CLOSEALL (YES) or whether CLOSEALL will ignore it (NO). CLOSEALL uses CLOSEF, so that any AFTER functions will be executed if the le is in fact closed.

EOF VAL is a function that will be applied to the full name of FILE when an end-of-le

## Dribble Files

error occurs, and the `ERRORTYPELIST` entry for that error, if any, returns `NIL`. The function can examine the context of the error, and can decide whether to close the `le`, `RETFROM` some function, or perform some other computation. If the function supplied returns normally (i.e. does not `RETFROM` some function), the normal error machinery will be invoked (but `FILE` will not be automatically closed if the `EOF` function did not close it).

Note that multiple `AFTER` and `BEFORE` functions may be associated with a `le`; they are executed in sequence with the most recently associated function executed `rst`. However, a second `STATUS` specification will supercede an earlier one. The `CLOSEALL` and `EOF` values will also override earlier values, so only the last value specified will have an effect. Files are initialized with `CLOSEALL - YES`, `EOF - CLOSEF`.

### 6.1.6 Dribble Files

A dribble `le` is a “transcript” of all of the input and output on a terminal. The following function enables dribble `les` for Interlisp:

```
(DRIBBLE FILENAME APPENDFL G THAWEDFL G) [Function]
```

Opens `FILENAME` and begins recording the typescript. Returns the old dribble `le` if any, otherwise `NIL`. If `APPENDFL G = T`, the typescript will be appended to the end of `FILENAME`. If `THAWEDFL G = T`, the `le` will be opened in “thawed” mode, for those implementations that support it. `(DRIBBLE)` closes the dribble `le`. Only one dribble `le` can be active at any one time, so `(DRIBBLE FILE1)` followed by `(DRIBBLE FILE2)` will cause `FILE1` to be closed.

In Interlisp-D, `DRIBBLE` opens a dribble `le` for the current process, recording the input and output for that process. Multiple processes can have separate dribble `les` open at the same time.

```
(DRIBBLEFILE) [Function]
```

Returns the name of the current dribble `le`, if any, otherwise `NIL`.

Terminal input is echoed to the dribble `le` a line bufer at a time. Thus, the typescript produced is somewhat neater than that appearing on the user’s terminal, because it does *not* show characters that were erased via control-A or control-Q. Note that the typescript `le` is *not* included in the list of `les` returned by `(OPENP)`, nor will it be closed by a call to `CLOSEALL` or `CLOSEF`. Only `(DRIBBLE)` closes the typescript `le`.

## 6.2 INPUT FUNCTIONS

Most of the functions described below have an argument `FILE`, which species the name of the `le` on which the operation is to take place. If `FILE` is `NIL`, the primary input `le` will be used. If the `le` argument is a string, input will be taken from that string (and the string pointer reset accordingly).

Most input functions also have a `RDTBL` argument, which species the readtable to be used for input. If `RDTBL` is `NIL`, the primary readtable will be used. Readtables are described on page 6.32.

## INPUT/OUTPUT

Note: in all Interlisp-10 symbolic les, end-of-line is indicated by the characters carriage-return and line-feed in that order. Accordingly, on input from les, Interlisp-10 skips all line-feeds that immediately follow carriage-returns. On input from the terminal, Interlisp echos a line-feed whenever a carriage-return is input.

When reading from the terminal, the input is buered a line at a time (unless buering has been inhibited by (CONTROL T), or the input is being read by READC or PEEKC) and can be backed up over using speci ed editing characters. The user can erase a character at a time, the whole line, or, in Interlisp-D, a word at a time. The keys that perform these editing functions are assignable via the SETSYNTAX function (page 6.34), with the intial settings chosen to be those most natural for the given operating system: characters are deleted one at a time by control-A under Tenex, Delete under Tops20, and BackSpace in Interlisp-D; the whole line is erased by control-Q under Tenex and in Interlisp-D, and control-U under Tops20; words are erased by control-W in Interlisp-D.

The character-deleting action on normal terminals is to echo a \ followed by the erased character; on the Interlisp-D display the character is physically erased from the screen (this action can also be speci ed for display terminals in other Interlisps; see page 6.43). The line-deleting action is normally to print ## and start over on a new line. Neither will back up beyond the previous carriage-return.

When reading from a le, and an end of le is encountered, all input functions close the le and generate an error, END OF FILE (unless WHENCLOSE has been used to alter this behavior; see page 6.11).

(READ FILE RDTBL FLG) [Function]  
Reads one expression from FILE. Atoms are delimited by the break and separator characters as de ned in RDTBL. To include a break or separator character in an atom, the character must be preceded by the input escape character %, e.g., AB%(C is the atom AB(C, %% is the atom %, %control-A is the atom control-A. 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., ^VC for control-C.

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 octal by terminating it with a Q, e.g., 17Q and 15 read in as the same integer. The setting of RADIX (page 6.19) determines in which base integers are printed.

When reading from the terminal, all input is line-buered to enable the action of the backspacing control characters (unless inhibited by (CONTROL T) (page 6.45)). Thus no characters are actually seen by the program until a carriage-return is typed.<sup>4</sup> 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

---

<sup>4</sup>Actually, the line buering is terminated by the character with terminal syntax class EOL (see page 6.33), which in most cases is carriage-return.

## Input Functions

characters are transmitted.<sup>5</sup> To indicate this, Interlisp also prints a carriage-return line-feed on the terminal.

In Interlisp-10, the character control-W is defined as an IMMEDIATE read macro that erases the last expression read, echoing a `\\` and the erased expression, e.g., `(NOW IS THE TIME^W \\ TIME)` returns `(NOW IS THE)`. Control-W can be used repeatedly, and can also back up and erase expressions on previous lines. However, since control-W is implemented as an IMMEDIATE read-macro character, (page 6.36), once it is typed, then individual *characters* typed before it cannot be deleted by control-A or control-Q, since they will already have passed through the line buffer.

In Interlisp-D, control-W is instead defined as an editing character that deletes the last “word” of input, i.e., back to the first non-OTHER character preceding the first non-SEPR character, essentially a repeated BackSpace. The character performing this function is assignable using the WORDDELETE syntax (page 6.34).

`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., the user does 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 an escape character 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. Control-A, 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 last atom read by RATOM, `NIL` otherwise.

---

<sup>5</sup>The line buffer is also transmitted to READ whenever an IMMEDIATE read-macro character is typed (page 6.36).

## INPUT/OUTPUT

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 `%` (as an escape character, e.g., `%[` 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, separator, or escape character. The action of `READC` is subject to line-buering, 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-buering, e.g., determining when input has been terminated. If (`CONTROL T`) has been executed (page 6.45), defeating line-buering, 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 buer).

(`PEEKC FILE RDTBL`) [Function]  
Returns the next character, but does not actually read it and remove it from the buer. If `RDTBL = NIL`, `PEEKC` is not subject to line-buering,<sup>6</sup> i.e., it returns a value as soon as a character has been typed. Otherwise, `PEEKC` waits until the line has been terminated before returning its value. This means that control-A, control-Q, and control-V will be able to perform their usual editing functions.

(`LASTC FILE`) [Function]  
Returns the last character read from `FILE`.

`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 buer of `FILE`, `NIL` otherwise. Note that because of line-buering, `READP` may return `T`, indicating there is input in the buer, but `READ` may still have to wait.

Frequently, the terminal's input buer contains a single EOL character left over from a previous input. For most applications, this situation wants to be treated as though the buer were empty, and so (`READP T`) returns `NIL` in this case. However, if `FLG = T`, `READP` also returns `T` in this case, i.e., (`READP T T`) returns `T` if there is *any* character in the input buer.

(`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`

---

<sup>6</sup>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 buer, where a subsequent `del` (or control-Z on TOPS-20) before the character is read can clear it, and where subsequent line buer backspacing could change it. Thus it is possible for the value returned by `PEEKC` to "disagree" in the first character with a subsequent `READ`.

## Output Functions

`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.

In Interlisp-10, `WAITFORINPUT` operates by dismissing, checking for available input, and then, if there is none, dismissing again, each time for an increasingly larger interval. The initial interval is `DISMISSINIT` milliseconds (initially 500), and the interval grows by 1/16 for each dismissal, up to a maximum of `DISMISSMAX` milliseconds (initially 10,000).

(`SKREAD FILE REREADSTRING` ) [Function]  
“Skip Read”. It moves the `le` pointer for `FILE` ahead 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 user 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 getting its paren-count, double-quote count, etc. set up properly.

`SKREAD` always uses `FILERDTBL` for its readtable. `SKREAD` may have difficulties if unusual read-macros have been added to `FILERDTBL`. `SKREAD` will 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 `le` which does not follow the normal parenthesis and string-quote conventions in `FILERDTBL`.

`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`.

### 6.3 OUTPUT FUNCTIONS

Most of the functions described below have an argument `FILE`, which specifies the name of the `le` on which the operation is to take place. If `FILE` is `NIL`, the primary output `le` is used. Some of the functions have a `RDTBL` argument, which specifies the readtable to be used for output. If `RDTBL` is `NIL`, the primary readtable is used.

Unless otherwise specified by `DEFPRINT` (page 6.23), 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 the user’s terminal, and will *not* read back in correctly; if the form above is read, it will produce the atom “`{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 `le` system being used by the given implementation of Interlisp. For example, in Interlisp-10 end-of-line is indicated by the characters carriage-return and line-feed in that



## INPUT/OUTPUT

order.

(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 lists as well as atoms and strings; PRIN1 is usually used only for explicitly printing formatting characters, e.g., (PRIN1 (QUOTE %[ ])) might be used to print a left square bracket (the % would not be printed by PRIN1). PRIN2 is used for printing S-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 RADIX=8 (page 6.19), PRIN2 prints a Q after integers but PRIN1 does not (but both print the integer in octal).

(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.

(SPACES N FILE) [Function]

Prints N spaces. Returns NIL.

(TERPRI FILE) [Function]

Prints an end-of-line. Returns NIL.

(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 (page 6.25).

(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 (page 8.1) for printing the values of expressions and for printing the history list, by various commands of the break package (page 9.1), e.g. ?= and BT commands, and various other system packages. The idea is that by simply setting or binding SYSPRETTYFLG to T (initially NIL), the user instructs the system when interacting with the user

## Printlevel

to PRETTYPRINT expressions (page 6.47) instead of printing them.

(PRINTBELLS) [Function]  
Used by DWIM (page 15.1) to print a sequence of bells to alert the user to stop typing. Can be advised or redefined for special applications, e.g., to flash the screen on a display terminal.

(DOBE) [Function]  
(Interlisp-10) Dismiss until Output Buffer is Empty, i.e., until all of the characters that have been printed by Interlisp functions have actually been printed on the user's terminal. For example, it is important to perform a DOBE after printing an error message before clearing the input buffers to make sure that the user has actually seen the error message.

In systems that do not handle output to the display asynchronously with user computation, such as Interlisp-D, DOBE is a no-op.

### 6.3.1 Printlevel

When using Interlisp one often has to handle large, complicated lists, which are difficult to understand when printed out. PRINTLEVEL allows the user 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 CAR VAL CDR VAL) [Function]  
Sets the CAR print level to CAR VAL, and the CDR print level to CDR VAL. 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 CAR VAL is a list cell it is equivalent to (PRINTLEVEL (CAR CAR VAL) (CDR CAR VAL)).

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

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 &. For example, suppose  $x = (A (B C (D (E F) G) H) K)$ . If CAR VAL=3, (PRINT x) would print (A (B C (D & G) H) K), if CAR VAL=2, (A (B C & H) K), if CAR VAL=1, (A & K), and if CAR VAL=0, just &.

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 CDR VAL=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. For example, if CDR VAL=2, then (A (B C (D (E F) G) H) K L) will print as (A (B --) --)

## INPUT/OUTPUT

and if `CDR VAL=3`, as `(A (B C --) K --)`.

If the `CDR printlevel` is negative, then it is the same as if the `CDR printlevel` were infinite.

The `printlevel` setting can be changed dynamically, even while Interlisp is printing, by typing control-P followed by a number, i.e., a string of digits, followed by a period or exclamation point. As soon as control-P is typed, Interlisp clears and saves the input buffer, clears the output buffer, rings the bell indicating it has seen the control-P, and then waits for input, which is terminated by any non-number. The input buffer is then restored and the program continues. If the input was terminated by a period or an exclamation point, the `CAR printlevel` is immediately set to this number; otherwise, the input is ignored. Characters cleared from the output buffer will have been lost in either case, and printing continues with the (possibly new) `printlevel`. If the print routine is currently deeper than the new level, all unfinished lists above that level will be terminated by “--”. Thus, if a circular or long list of atoms, is being printed out, typing “control-P0.” will cause the list to be terminated immediately.

If the string of digits following a control-P is terminated by a comma, another number may be typed terminated by a period or exclamation point. The `CAR printlevel` will then be set to the first number, the `CDR printlevel` to the second number.

In either case, if a period is used to terminate the `printlevel` setting, the `printlevel` will be returned to its previous setting after the current printout has finished. If an exclamation point is used, the change is permanent and the `printlevel` is not restored (until it is changed again).

`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.

### 6.3.2 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 `FLTFRMT` 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`. If `N` is negative, integers are interpreted by the print routines as unsigned numbers; i.e., the actual two's complement representation of the integer in the integer size of the particular implementation is interpreted as if it were a positive number on a machine of infinite integer size. Thus, numeric output under a negative radix varies with the implementation, and numbers printed in this way by one implementation will not read correctly in an implementation whose integers are of a different size.

For example, in Interlisp-10, whose integer size is 36 bits, -9 will print as shown with the following radices:

## Printing numbers

(RADIX)	(PRINT -9)
10	-9
8	-11Q
-10	68719476727 (i.e. $2^{36-9}$ )
-8	777777777767Q

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 end in Q, in which case they are interpreted as octal. Thus READ and PRINT are inverses, independent of any radix setting. RADIX also does not affect the behavior of UNPACK, etc., unless the value of PRXFLG (below) is T; e.g., with (RADIX 8), the value of (UNPACK 9) is (9), not (1 1).

(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.

In Interlisp-10, FORMAT may also be a machine-dependent FLOAT format-code as returned by NUMFORMATCODE (page 6.23).

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

## INPUT/OUTPUT

```
(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 (page X.XX), 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 FORM T NUMBER FILE) [Function]  
Prints NUMBER on FILE according to the format FORM T. FORM T is a list structure with one of the forms described below. FORM T can also be a machine dependent format-code as returned by NUMFORMATCODE (page 6.23).

(Interlisp-10) If NUMBER does not fit in the field specified by FORM T, the full print name is printed. Then a TAB is executed so that the line position of the line after PRINTNUM is always the position prior to printing plus the indicated width.

If FORM T 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 of 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 (the vertical bars indicate the field width):

## Printing numbers

FORMA T	NUMBER	PRINTNUM prints
(FIX 2)	3	3
(FIX 2 NIL T)	7	07
(FIX 12 8 T)	14	000000000016
(FIX 5 NIL NIL T)	2	2

If `FORMA T` is a list of the form `(FLOAT WIDTH DECP AR T EXPP AR T PAD0 ROUND )`, this specifies a `FLOAT` format. `NUMBER` is printed as a decimal number in a field `WIDTH` characters wide, with `DECP AR T` digits to the right of the decimal point. If `EXPP AR T` is not 0 (or `NIL`), the number is printed in exponent notation, with the exponent occupying `EXPP AR T` characters in the field. `EXPP AR T` 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.<sup>7</sup>

`FLOAT` format examples:

FORMA T	NUMBER	PRINTNUM prints
(FLOAT 7 2)	27.689	27.69
(FLOAT 7 2 NIL T)	27.689	0027.69
(FLOAT 7 2 2)	27.689	2.77E1
(FLOAT 11 2 4)	27.689	2.77E+01  <sup>8</sup>
(FLOAT 7 2 NIL NIL 1)	27.689	30.00
(FLOAT 7 2 NIL NIL 2)	27.689	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 `FORMA T`. This option facilitates the printing of numbers in aggregates with missing values coded as `NIL`.

---

<sup>7</sup>The interpretation of `WIDTH = NIL` and `DECP AR T = NIL` are not specified, and are currently a function of the implementation. Interlisp-10 prohibits `WIDTH = NIL`, and treats `DECP AR T = NIL` as equivalent to `DECP AR T = 0`; Interlisp-D interprets `WIDTH = NIL` to mean no padding, i.e., to use however much space the number needs, and interprets `DECP AR T = NIL` to mean as many decimal places as needed.

<sup>8</sup>As of this writing, the Interlisp-10 implementation actually does something less intuitive with the `EXPP AR T` field: the placement of the decimal point is affected by `DECP AR T`, and padding never occurs. These two examples in Interlisp-10 would actually print as `|.28E+02|` and `|27.69E+0000|`.

## INPUT/OUTPUT

In some implementations, formatted printing of numbers receives assistance from the operating system, provided that the format is specified in some sort of special code. PRINTNUM works by converting the machine-independent format specifications described above into machine-*dependent* codes the exact form of which may vary from implementation to implementation. This conversion process takes place on each call to PRINTNUM. For efficiency purposes, if the user is going to be performing a particular call to PRINTNUM frequently, he may wish to separate the conversion from the actual printing, performing the conversion process just once and saving the result. The function NUMFORMATCODE is available for this purpose: NUMFORMATCODE takes a format, performs the conversion and returns a machine dependent format-code, which can be given to PRINTNUM in place of a list structure format as described above. In this case, PRINTNUM will not have to perform the conversion, but can simply use the machine-dependent format code directly.

(NUMFORMATCODE *FORMAT* *SMASHCODE* ) [Function]  
Converts the FIX or FLOAT format *FORMAT* to a machine-dependent format-code. If *SMASHCODE* is recognized as a format-code data-structure, then the new format-code is smashed into that structure instead of allocating new storage. (NUMFORMATCODE) returns an uninitialized datum that can later be smashed.

In Interlisp-D, this function is a no-op, as there is no special internal representation for number formats.

### 6.3.3 User Defined Printing

(DEFPRINT *TYPE* *FN* ) [Function]  
*TYPE* is a type name (see page 2.1). Whenever a printing function (PRINT, PRIN1, PRIN2, etc.) encounters an object of the indicated type, *FN* is called with the item to be printed as its argument. If it returns NIL, the datum is printed in the manner the system defaults; for user data types, it is printed as {datatype}#nnnnnn. If *FN* wishes to specify how the datum should be printed, it should return a list of the form (*ITEM1* . *ITEM2* ). *ITEM1* is printed using PRIN1 (unless it is NIL), and then *ITEM2* printed using PRIN2 with no spaces between the two items. (Typically, *ITEM1* is a read macro character.)

In Interlisp-10, *TYPE* may also be a type number (see page 22.2). Note that the user can specify different action for type names ARRAYP, HARRAYP, TERMTABLEP, READTABLEP, and CCODEP, even though they all have the same type *number*.

Note that DEFPRINT also affects internal calls to print from PACK, CONCAT, etc., i.e. any operation that involves obtaining a print name (see page 2.8). A consequence of this fact is that in implementations that do not have reentrant printing code (in particular, Interlisp-10), the user's DEFPRINT function must *not* call any print name manipulating functions itself, or the results of the whole printing operation are undefined.

### 6.3.4 Dumping 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

## READFILE and WRITEFILE

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 page 3.14), or an array or hash array, it prints the data contained therein, surrounded by special characters defined as read-macro characters (see page 6.36). 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 readtable.

(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 will assume 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 disk files, since the algorithm depends on being able to reset the file pointer. If FILE is not a disk 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.

(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 (page 11.25) are two file package commands for dumping and reloading circular and re-entrant data structures. They provide a convenient interface to HPRINT and HREAD.

## 6.4 READFILE AND WRITEFILE

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

(READFILE FILE) [Function]  
Reads successive expressions from file using READ (with FILERDTBL as readtable)



## INPUT/OUTPUT

until the single atom `STOP` is read, or an end of file encountered. Returns a list of these expressions.

`(WRITEFILE X FILE)` [Function]  
Inverse of `READFILE`. Writes a date expression onto `FILE`, followed by successive expressions from `x`, using `FILERDTBL` as a readtable. 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, a `STOP` is printed on `FILE` and it is closed. Returns `FILE`.

`(ENDFILE FILE)` [Function]  
Prints `STOP` on `FILE` and closes it.

### 6.5 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 the user 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 FILE PRINTCOM 1 PRINTCOM 2 PRINTCOM N)`

`FILE` is evaluated to obtain the name of the file to which the output from this specification is directed. The `PRINTOUT` commands are strung together, one after the other without punctuation, after `FILE`. 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 the user wanted 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:

## Horizontal Spacing Commands

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

Here the `rst T` species output to the terminal, the commas cause single spaces to be printed, and the final `T` species 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 file, the user 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-justified 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 named `BOLDFONT` and the values in font `SMALLFONT`. The program in ordinary Lisp 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` (page 6.21) to print `X` in the desired format. The `.F10.2` species a `FLOAT` format for `PRINTNUM`.

### 6.5.1 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.

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

## INPUT/OUTPUT

<code>.TAB POS</code>	Species <code>TAB</code> to position (the value of) <code>POS</code> . 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 <code>PRINTOUT</code> form is more concise and is prettyprinted more compactly. Note that <code>.TAB N</code> and <code>N</code> , where <code>N</code> is an integer, are equivalent.
<code>.TAB0 POS</code>	Like <code>.TAB</code> except that it can result in zero spaces (i.e. the call to <code>TAB</code> species <code>MINSPACES=0</code> ).
<code>-N</code>	Negative integers indicate relative (as opposed to absolute) spacing. Translates as <code>(SPACES  N )</code> .
<code>, , , , ,</code>	Provides a short-hand way of specifying 1, 2 or 3 spaces, i.e., these commands are equivalent to <code>-1</code> , <code>-2</code> , and <code>-3</code> , respectively.
<code>.SP DISTANCE</code>	Translates as <code>(SPACES DISTANCE)</code> . Note that <code>.SP N</code> and <code>-N</code> , where <code>N</code> is an integer, are equivalent.
<code>.RESET</code>	Resets the current line by causing a carriage-return to be printed without a line-feed. Useful for overprinting, or for regaining control of a line on which characters have been printed in a variable pitched font.

### 6.5.2 Vertical Spacing Commands

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

<code>T</code>	Translates as <code>(TERPRI)</code> . This command is functionally equivalent to the integer command <code>0</code> ; they both move to position <code>0</code> (= column <code>1</code> ) of the next line. To print the letter <code>T</code> , use the string <code>"T"</code> .
<code>.SKIP LINES</code>	Equivalent to a sequence of <code>LINES</code> <code>(TERPRI)</code> 's. The <code>.SKIP</code> command allows for skipping large constant distances and for computing the distance to be skipped.
<code>.PAGE</code>	Puts a form-feed (control-L) out on the line. Care is taken to make sure that Interlisp's view of the current line position is correctly updated.

### 6.5.3 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:

<code>.FONT FONTSPEC</code>	Puts out a control sequence that causes a change to font <code>FONTSPEC</code> (the association between <code>FONTSPEC</code> and a specific font must be defined in the user's font profile, as described in page 6.55). <code>FONTSPEC</code> may be a font-name variable or an expression that evaluates to the value of a font-name variable. <code>FONTSPEC</code> may also be a positive integer <code>N</code> , which is taken as an abbreviated reference to the font named <code>FONTN</code> (e.g. <code>1 =&gt; FONT1</code> ).
<code>.SUP</code>	Species superscripting. All subsequent characters are printed above the base of the current line. Note that this is absolute, not relative: a <code>.SUP</code> following a <code>.SUP</code>

## Printing Specifications

is a no-op.

- `.SUB` Species subscripting. Subsequent printing is below the base of the current line. As with superscripting, the effect is absolute.
- `.BASE` 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.

### 6.5.4 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` Causes `THING` to be printed using `PRIN2`; translates as `(PRIN2 THING)`.
- `.PPF THING` Causes `THING` to be prettyprinted at the current line position via `PRINTDEF` (page 6.49). 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` Prettyprints `THING` as a variable; no special interpretation is given to `SELECTQ`, `PROG`, etc.
- `.PPFTL THING` 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` Like `.PPV`, but prettyprints `THING` as a tail.

#### 6.5.4.1 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 LMAR G RMAR G LIST`  
Prints `LIST` in paragraph format, using `PRIN1`. Translates as `(PRINTPARA LMAR G RMAR G LIST)` (see page 6.31). 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 LMAR G RMAR G LIST`

## INPUT/OUTPUT

Print as paragraph using PRIN2 instead of PRIN1. Translates as (PRINTPARALMAR G RMAR G LIST T).

### 6.5.4.2 Right-Flushing

Two commands are provided for printing simple expressions flushed-right against a specified line position, using the function FLUSHRIGHT (page 6.31). 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      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      Flush-right using PRIN2 instead of PRIN1.

### 6.5.4.3 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      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      Centers using PRIN2 instead of PRIN1.

### 6.5.4.4 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 (see page 6.21).

.IFORMAT NUMBER      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) will translate as (PRINTNUM '(FIX 5 -8 T) FOO). It will cause the value of FOO to be printed with radix -8 right-flushed in a field of width 5,

## Escaping to LISP

with 0's used for padding on the left. Internal NIL's may be omitted, e.g. the commands `.I5..T` and `.I5.NIL.T` are equivalent.

`.F` *FORMA T* *NUMBER*

Species oating- number printing. Like the `.I` format command, except translates with a `FLOAT` format- list.

`.N` *FORMA T* *NUMBER*

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

### 6.5.5 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 `FILE` is `T`), or lower-level iterative printing within a higher-level print specification.

`#` *FORM*

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 file 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` (page 9.20) 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.

### 6.5.6 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 the user 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

## INPUT/OUTPUT

whose CDR is the list of commands still remaining to be translated in the normal way.

For example, suppose the user wanted 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:

```
(LAMBDA (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.

### 6.5.7 Special Printing Functions

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

```
(PRINTPARA LMAR G RMAR G LIST P2FLAG PARENFLAG FILE) [Function]
```

Prints LIST on FILE in line- lled paragraph format with its first element beginning at the current line position and ending at or before RMAR G, and with subsequent lines appearing between LMAR G and RMAR G. 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 LMAR G is zero or positive, it is interpreted as an absolute column position. If it is negative, then the left margin will be at |LMAR G| + (POSITION). If LMAR G = NIL, the left margin will be at (POSITION), and the paragraph will appear in block format.

If RMAR G 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) + |RMAR G|. 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-ushing and centering commands translate as calls to the function FLUSHRIGHT:

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

If CENTERFLAG = NIL, prints X right-ushed 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,

## Readtables

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.

### 6.6 READTABLES

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 readtables. Readtables are objects that specify the syntactic properties of characters for input routines. Since the input routines parse character sequences into objects, the readtable in use determines which sequences are recognized as literal atoms, strings, list structures, etc.

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

The Interlisp system uses three readtables: `T` for input/output from terminals, the value of `FILERDTBL` for input/output from files, and the value of `EDITRDTBL` for input from terminals while in the editor. These three tables are initially copies of the `ORIG` readtable, with changes made to some of them to provide read macros (page 6.36) that are specific to terminal input or file input. Using the functions described below, the user may further change, reset, or copy these tables. The user can also create new readtables, and either explicitly pass them to input/output functions as arguments, or install them as the primary readtable, via `SETREADTABLE`, and then not specify a `RDTBL` argument, i.e., use `NIL`.

#### 6.6.1 Readtable Functions

`(READTABLEP RDTBL)` [Function]  
Returns `RDTBL` if `RDTBL` is a real readtable (*not* `T` or `ORIG`), otherwise `NIL`.

`(GETREADTABLE RDTBL)` [Function]  
If `RDTBL = NIL`, returns the primary read table. If `RDTBL = T`, returns the system terminal readtable. If `RDTBL` is a real readtable, returns `RDTBL`. Otherwise, generates an `ILLEGAL READTABLE` error.

`(SETREADTABLE RDTBL FLG)` [Function]  
Sets the primary readtable to `RDTBL`. If `FLG = T`, `SETREADTABLE` sets the system terminal readtable, `T`. Note that the user can reset the other system readtables with `SETQ`, e.g., `(SETQ FILERDTBL (GETREADTABLE))`.

Generates an `ILLEGAL READTABLE` error if `RDTBL` is not `NIL`, `T`, or a real readtable. Returns the previous setting of the primary readtable, so `SETREADTABLE` is suitable for use with `RESETFORM` (page 9.20).



## INPUT/OUTPUT

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

Note that COPYREADTABLE is the only function that *creates* a readtable.

(RESETREADTABLE RDTBL FR OM ) [Function]  
Copies (smashes) FR OM into RDTBL . FR OM and RDTBL can be NIL, T, or a real readtable. In addition, FR OM can be ORIG, meaning use the system's original readtable.

### 6.6.2 Syntax Classes

A readtable 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, pre x 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

## Syntax Classes

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*$` would be read by 6 calls to `RATOM` returning respectively `ABC`, `DEF`, `$`, `GH`, `$`, `$`.

Although normally there is only one character in a readtable 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 readtable provides the *association* between a character and its syntax class, and the input/output routines enforce the abstraction by using readtables to drive the parsing.

The functions below are used to obtain and set the syntax class of a character in a readtable. `CH` can either be a character code (a number), or a character (a single-character atom); those Interlisp objects that happen to be both, viz., one-digit numbers, are interpreted as character codes. For example, in Interlisp-10, 1 indicates control-A, and 49 indicates the *character* 1.

Note: Terminal tables, described in page 6.40, also associate characters with syntax classes, and they can also be manipulated with the functions below. The set of readtable 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 readtable 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 readtable 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 page 6.36).

`NIL`, `T`, `ORIG`, or a real readtable 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 readtable.

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 readtable or terminal table.

## INPUT/OUTPUT

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.

Note: 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 readtable:

(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 characters 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 readtable 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.

## Read-Macros

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.
```

### 6.6.3 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 file and readtable 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

```
[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:

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

and the value of FOO is (A B C), then when the user inputs (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 (page 2.17), 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, + could be defined by:

## INPUT/OUTPUT

```
(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 `OPTION1` `OPTIONN`, 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 escape character), and is a break character, i.e., a member of <code>(GETSYNTAX 'BREAK RDTBL)</code> .   |
| FIRST  | The character is interpreted as a read-macro character <i>only</i> 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 <i>not</i> 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 <code>(QUOTE T)</code> . |
| ALONE  | The read-macro character is <i>not</i> 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. For example, * is an ALONE read-macro character in order to implement the comment pointer feature (see page 6.51).   |

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:

- |                     |  |
|---------------------|--|
| ESCQUOTE or ESC     | The default. When printed with PRIN2, the read-macro character will be preceded by the output escape character (%).  |
| NOESCQUOTE or NOESC | The read-macro character will be printed without an escape, e.g., ' 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:

## Read-Macros

### 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 the user 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 page 9.17).

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.

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. Therefore, in Interlisp-10, reading a single RIGHTPAREN or RIGHTBRACKET via a READ inside of a read-macro function is disallowed. If this occurs, the paren/bracket is put back into the input buffer, and a READ-MACRO CONTEXT ERROR is generated. The following two functions are useful for avoiding this error:

```
(INREADMACROP)
```

[Function]

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

```
(SETREADMACROFLG FLG)
```

[Function]

Resets the “read-macro” flag, i.e., the internal system flag that informs READ that it is under a read macro function, and causes it to generate a READ-MACRO CONTEXT ERROR, if an unmatched ) or ] is encountered. Returns the previous

## INPUT/OUTPUT

value of the `ag`. The main use for this is when debugging read-macro functions: to avoid spurious READ-MACRO CONTEXT error messages when typing into breaks, the user can put `(SETREADMACROFLG)` on `BREAKRESETFORMS` (page 9.13).

The READ-MACRO CONTEXT error does not occur in Interlisp-D; a READ inside of a read-macro when the next input character is a `RIGHTPAREN` or `RIGHTBRACKET` eats 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.

`(READMACROS FLG RDTBL )` [Function]  
If `FLG = NIL`, turns off action of read-macros. If `FLG = T`, turns them on. Returns previous setting.

In Interlisp-D, turns on/off action of read-macros in readtable `RDTBL`.

The following read macros are standardly defined in Interlisp:

- `'` (single-quote) Currently defined only in `T` and `EDITRDTBL`. 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-YFOO 2)` is read as `(1 (A B) 2)`, but note that no structure is copied; the `CADR` 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` (page 17.59).
- ``` (back-quote) Back-quote makes it easier to write programs to construct complex data structures. Back-quote is like quote, except that within the back-quoted expression, forms can be evaluated. The general idea is that the back-quoted expression is a “template” containing some constant parts (as with a quoted form) and some parts to be filled in by evaluating something.
- Within the back-quoted expression, the character `“,”` (comma) introduces a form to be evaluated. A form preceded by `“,@”` is to be spliced in, using `APPEND`, and a form preceded by `“,.”` is to be spliced in, using `NCONC`. Unlike with control-Y, however, the evaluation occurs not at the time the form is read, but at the time the back-quoted expression is evaluated. That is, the back-quote macro returns an expression which, when evaluated, produces the desired structure.
- 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))))`. Back-quote is particularly useful for writing compiler macros. For example,

## Terminal Tables

```
'(COND
  ((FIXP , (CAR X))
   , (CADR X))
  (T ,@(CDDR X)))
```

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 back-quote context.

For users without a back-quote character on their keyboards, back-quote can also be written as |' (vertical-bar, quote). In Interlisp-D, back-quote is typed as shift-linefeed.

?	Defined in T and EDITRDTBL. Implements the ?= command for on-line help regarding the function currently being “called” in the typein (see page 9.5).
*	Defined in FILERDTBL only. Implements the comment pointer feature for saving space by keeping the text of comments outside memory (page 6.51).
control-W	Defined in T and EDITRDTBL, Interlisp-10 only. An IMMEDIATE read macro that deletes the previous expression. In Interlisp-D, control-W is an editing character that deletes the previous “word”.
(vertical bar)	When followed by ' (quote), is a synonym for back-quote; followed by certain other characters, it is used by HPRINT and HREAD to print and read in unusual expressions; otherwise is ignored, i.e., treated as a separator character, enabling the editor's CHANGECHAR feature (page 6.55).

## 6.7 TERMINAL TABLES

A readtable contains input/output information that is *media-independent*. For example, the action of parentheses is the same regardless of the device from which the input is being performed. A terminal table is an object that contains information that pertains to *terminal* input/output operations only, such as the character to type to delete the last character or to delete the last line. In addition, terminal tables contain such information as how line-buffering is to be performed, how control characters are to be echoed/printed, whether lower case input is to be converted to upper case, etc.

Using the functions below, the user may change, reset, or copy terminal tables, or create a new terminal table and install it as the primary terminal table via SETTERMTABLE. However, unlike readtables, terminal tables cannot be passed as arguments to input/output functions.



## INPUT/OUTPUT

### 6.7.1 Terminal Table Functions

(TERMTABLEP TTBL )	[Function]
Returns TTBL , if TTBL is a real terminal table, NIL otherwise.	
(GETTERMTABLE TTBL )	[Function]
If TTBL = NIL, returns the primary (i.e., current) terminal table. If TTBL is a real terminal table, return TTBL . Otherwise, generates an ILLEGAL TERMINAL TABLE error.	
(SETTERMTABLE TTBL )	[Function]
Sets the primary terminal table to be TTBL . Returns the previous TTBL . Generates an ILLEGAL TERMINAL TABLE error if TTBL is not a real terminal table.	
(COPYTERMTABLE TTBL )	[Function]
Returns a copy of TTBL . TTBL can be a real terminal table, NIL, or ORIG, in which case it returns a copy of the <i>original</i> system terminal table. Note that COPYTERMTABLE is the only function that <i>creates</i> a terminal table.	
(RESETTERMTABLE TTBL FROM )	[Function]
Copies (smashes) FROM into TTBL . FROM and TTBL can be NIL or a real terminal table. In addition, FROM can be ORIG, meaning to use the system's original terminal table.	

### 6.7.2 Terminal Syntax Classes

A terminal table associates with each character a single “terminal syntax class”, one of CHARDELETE, LINEDELETE, WORDDELETE (Interlisp-D only), RETYPE, CTRLV, EOL, and NONE. Unlike readtable classes, only one character in a particular terminal table can belong to each of the classes (except for the default class NONE). When a new character is assigned one of these syntax classes by SETSYNTAX, the previous character is disabled (i.e., reassigned the syntax class NONE), and the value of SETSYNTAX is the code for the previous character of that class, if any, otherwise NIL.

The terminal syntax classes are interpreted as follows:

CHARDELETE or DELETECHAR	(Initially control-A under Tenex, del under Tops20, BackSpace in Interlisp-D) Typing this character deletes the previous character typed. Repeated use of this character deletes successive characters back to the beginning of the line.
LINEDELETE or DELETEDLINE	(Initially control-Q in Interlisp-10 under Tenex and in Interlisp-D, control-U under Tops20) Typing this character deletes the whole line; it cannot be used repeatedly.
WORDDELETE	(Interlisp-D only; initially control-W) Typing this character deletes the previous “word”, i.e., sequence of non-separator characters.
RETYPE	(Initially control-R) Causes the line to be retyped as Interlisp sees it (useful when repeated deletions make it difficult to see what remains).

## Terminal Control Functions

CTRLV or CNTRLV	(Initially control- V) When followed by A, B, ..., Z, inputs the corresponding control character control- A, control- B, ..., control- Z. This allows interrupt characters to be input without causing an interrupt.
EOL	On input from a terminal, the EOL character signals to the line buffering routine to pass the input back to the calling function. It also is used to terminate inputs to READLINE (page 8.30). In general, whenever the phrase carriage-return linefeed is used, what is meant is the character with terminal syntax class EOL.
NONE	The terminal syntax class of all other characters.

GETSYNTAX, SETSYNTAX, and SYNTAXP all work on terminal tables as well as readtables (see page 6.34). When given NIL as a TABLE argument, GETSYNTAX and SYNTAXP use the primary readtable or primary terminal table depending on which table contains the indicated CLASS argument. For example, (SETSYNTAX CH 'BREAK) refers to the primary readtable, and (SETSYNTAX CH 'CHARDELETE) refers to the primary terminal table. In the absence of such information, all three functions default to the primary readtable; e.g., (SETSYNTAX '{ '%[) refers to the primary read table. If given incompatible CLASS and table arguments, all three functions generate errors. For example, (SETSYNTAX CH 'BREAK TTBL), where TTBL is a terminal table, generates an ILLEGAL READTABLE error, and (GETSYNTAX 'CHARDELETE RDTBL) generates an ILLEGAL TERMINAL TABLE error.

### 6.7.3 Terminal Control Functions

(ECHOCONTROL CHAR MODE TTBL)	[Function]
	Used to indicate how control characters are to be echoed or printed. CHAR is a character or character code. MODE may be one of the atoms IGNORE, REAL, SIMULATE, or INDICATE, <sup>9</sup> which specify how the control character should be printed:
IGNORE	CHAR is never printed.
REAL	CHAR itself is printed; i.e., the raw control character is sent to the terminal. Some terminals, particularly displays, respond to certain control characters in interesting ways.
SIMULATE	Output of CHAR is simulated. For example, control-I (tab) may be simulated by printing spaces. The simulation is machine-specific and beyond the control of the user.
INDICATE	CHAR is printed as ^ followed by the corresponding alphabetic character.

The value of ECHOCONTROL is the previous output mode for CHAR. If MODE = NIL, ECHOCONTROL returns the current output mode without changing it.

Note that although the name of this function suggests echoing only, it affects *all* output of the control character, both echoing of input and printing of output.

---

<sup>9</sup>UPARROW is an obsolete synonym of INDICATE.

## INPUT/OUTPUT

The two cannot be specified independently, which can lead to some trickiness in DELETECONTROL messages (below).

In Interlisp-10, echoing information can be specified only for control characters (although *all* echoing can be disabled using ECHOMODE, below). Therefore, if CHAR is an alphabetic character (or code), it refers to the corresponding control character, e.g., (ECHOCONTROL 'Z 'INDICATE) makes control-Z echo as ^Z. All other values of CHAR generate ILLEGAL ARG errors. In Interlisp-D and Interlisp-VAX, it is possible to specify echoing information for *all* characters, using the function ECHOCHAR.

(ECHOCHAR CHAR CODE MODE TTBL) [Function]  
(Interlisp-D, Interlisp-VAX only) Like ECHOCONTROL, but CHAR CODE must be a character code, and can specify *any* character no coercions are performed. The INDICATE mode for "meta" characters, i.e., characters whose codes are in the range 200Q through 377Q, causes the character to be printed following a #. For example, meta-A would print as #A, meta-control-B as #^B.

CHAR CODE can also be a list of characters, in which case ECHOCHAR is applied to each of them with arguments MODE and TTBL.

(ECHOMODE FLG TTBL) [Function]  
If FLG = T, turns echoing for terminal table TTBL on. If FLG = NIL, turns echoing off. Returns the previous setting.

(GETECHOMODE TTBL) [Function]  
Returns the current echo mode for TTBL.

(DELETECONTROL TYPE MESSAGE TTBL) [Function]  
Specifies the output protocol when a CHARDELETE or LINEDELETE is typed. In the case of character deletion, Interlisp-10 is initially set up for hardcopy terminals: it echos the characters being deleted, preceding the rst by a \ and following the last by a \, so that it is easy to see exactly what was deleted, viz., the characters between the \s. Interlisp-D and Interlisp-VAX are initially set up to physically erase the deleted characters from the display, backing up over them. The various values of TYPE specify different phases of the deletion, as follows:

1STCHDEL	MESSAGE is the message printed the rst time CHARDELETE is typed. Initially "\ " in Interlisp-10.
NTHCHDEL	MESSAGE is the message printed on subsequent CHARDELETE's (without intervening characters). Initially "" in Interlisp-10.
POSTCHDEL	MESSAGE is the message printed when input is resumed following a sequence of one or more CHARDELETE's. Initially "\ " in Interlisp-10.
EMPTYCHDEL	MESSAGE is the message printed when a CHARDELETE is typed and there are no characters in the buffer. Initially "# CR" in Interlisp-10.
ECHO	The characters deleted by CHARDELETE are echoed. MESSAGE

## Terminal Control Functions

is ignored.

NOECHO                    The characters deleted by CHARDELETE are not echoed  
MESSA GE is ignored.

LINEDELETE      MESSA GE is the message printed when LINEDELETE character is typed. Initially ‘##<sup>cr</sup>’.

Note: In Interlisp-10, the LINEDELETE, 1STCHDEL, NTHCHDEL, POSTCHDEL, and EMPTYCHDEL messages must be 4 characters or fewer in length.

DELETECONTROL returns the previous message as a string. If MESSAGE = NIL, the value returned is the previous message without changing it. For ECHO and NOECHO, the value of DELETECONTROL is the previous echo mode, i.e., ECHO or NOECHO.

(GETDELETECONTROL TYPE TTBL) [Function]  
Returns the current DELETECONTROL mode for TYPE in TTBL.

If the user's terminal is a display, `DELETECONTROL` and `ECHOCONTROL` can be used to make it really delete the last character by performing the following:

(ECHOCONTROL 8 'REAL)  
8 is the code for control-H, which is backspace; we want the terminal to really backspace when we send ^H.

```
(DELETECONTROL 'NOECHO)
    Do not echo the deleted characters.
```

```
(DELETECONTROL '1STCHDEL " ^H ^H")
(DELETECONTROL 'NTHCHDEL " ^H ^H")
```

Erase each character by backspacing over it, printing a space, then backspacing again to put the carriage in the right place.

The following functions manipulate the RAISE mode, which determines whether lower case characters are converted to upper case when input from the terminal. (There currently is no “raise” mode for input from les.)

(RAISE FLG TTBL )	[Function]
<p>Sets the RAISE mode for terminal table TTBL . If FLG= NIL, all characters are passed as typed. If FLG= T, input is echoed as typed, but lowercase letters are converted to upper case. If FLG= 0, input is converted to upper case <i>before</i> it is echoed. Returns the previous setting.<sup>10</sup></p>	

<sup>10</sup>In Interlisp-10, both (RAISE) and (RAISE T) execute Tenex/Tops20 JSYS calls corresponding to the Executive command NORAISE, while (RAISE 0) executes the JSYS calls corresponding to the Executive command RAISE. Thus with (RAISE T), the conversion to uppercase is performed by Interlisp, while with (RAISE 0) the conversion is performed at the operating system level, i.e., before Interlisp-10 even sees the characters. The initial setting of RAISE in Interlisp-10 is determined by the terminal mode at the time the user rst starts up the system. When a SYSOUT is started, the RAISE mode is restored to whatever it was prior to the SYSOUT.

## INPUT/OUTPUT

(GETRAISE TTBL )

[Function]

Returns the current RAISE mode for TTBL .

### 6.7.4 Line-Bu ering

Characters typed at the terminal are stored in two bu ers before they are passed to an input function. All characters typed in are put into the low-level “system bu er”, which allows type-ahead. When an input function is entered, characters are transferred to the “line bu er” until a character with terminal syntax class EOL appears (or, for calls from READ, when the count of unbalanced open parentheses reaches 0).<sup>11</sup>

Until this time, the user can delete characters one at a time from the line bu er by typing the current CHARDELETE character, or delete the entire line bu er back to the last carriage-return by typing the current LINEDELETE.

Note that this line editing is *not* performed by READ or RATOM, but by Interlisp, i.e., it does not matter (nor is it necessarily known) which function will ultimately process the characters, only that they are still in the Interlisp line bu er. However, the function that is requesting input at the time the bu ering starts does determine whether parentheses counting is observed. For example, if a program performs (PROGN (RATOM) (READ)) and the user types in “A (B C D)”, the user must type in the carriage-return following the right parenthesis before any action is taken, because the line bu ering is happening under RATOM. If the program had performed (PROGN (READ) (READ)), the line-bu ering would be under READ, so that the right parenthesis would terminate line bu ering, and no terminating carriage-return would be required.

Once a carriage-return has been typed, the entire line is “available” even if not all of it is processed by the function initiating the request for input. If any characters are “left over”, they are returned immediately on the next request for input. For example, (LIST (RATOM) (READC) (RATOM)) when the input is “A B<sup>cr</sup>” returns the three-element list (A % B) and leaves the carriage-return in the bu er.

If a carriage-return is typed when the input under READ is not “complete” (the parentheses are not balanced or a string is in progress), line bu ering continues, but the lines completed so far are not available for editing with CHARDELETE or LINEDELETE.

The function CONTROL is available to defeat line-bu ering:

(CONTROL MODE TTBL )

[Function]

If MODE = T, eliminates Interlisp’s normal line-bu ering for the terminal table TTBL .  
If MODE = NIL, restores line-bu ering (normal). When operating with a terminal table in which (CONTROL T) has been performed, characters are returned to the calling function without line-bu ering as described below.

CONTROL returns its previous setting.

(GETCONTROL TTBL )

[Function]

Returns the current control mode for TTBL .

The function that initiates the request for input determines how the line is treated when (CONTROL T) is in effect:

---

<sup>11</sup>PEEK is an exception; it returns the character immediately when its second argument is NIL.

## Line-Bu ering

**READ** If the expression being typed is a list, the effect is the same as though done with (CONTROL NIL), i.e., line-bu ering continues until a carriage-return or matching parentheses. If the expression being typed is not a list, it is returned as soon as a break or separator character is encountered, e.g., (READ) when the input is 'ABC<space>' immediately returns ABC. CHARDELETE and LINEDELETE are available on those characters still in the bu er. Thus, if a program is performing several reads under (CONTROL T), and the user types 'NOW IS THE TIME' followed by control-Q, only TIME is deleted, since the rest of the line has already been transmitted to READ and processed.

An exception to the above occurs when the break or separator character is an opening parenthesis, bracket or double-quote, since returning at this point would leave the line bu er in a "funny" state. Thus if the input to (READ) is 'ABC(', the ABC is not read until a carriage-return or matching parentheses is encountered. In this case the user could LINEDELETE the entire line, since all of the characters are still in the bu er.

**RATOM** Characters are returned as soon as a break or separator character is encountered. Until then, LINEDELETE and CHARDELETE may be used as with READ. For example, (RATOM) followed by 'ABC<control-A><space>' returns AB. (RATOM) followed by '(<control-A>' returns ( and types ## indicating that control-A was attempted with nothing in the bu er, since the ( is a break character and would therefore already have been read.

**READC or PEEKC** The character is returned immediately; no line editing is possible. In particular, (READC) is perfectly happy to return the CHARDELETE or LINEDELETE characters, or the ESCAPE character (%).

The system bu er and line bu er can be directly manipulated using the following functions.

(CLEARBUF FILE FLG) [Function]  
Clears the input bu er for FILE. If FILE is T and FLG is T, the contents of Interlisp's system bu er and line bu er are saved (and can be obtained via SYSBUF and LINBUF described below).

When control-D or control-E is typed, or any of the interrupt characters that require terminal interaction is typed (control-H, control-P, or control-S), Interlisp automatically performs (CLEARBUF T T). For control-P, control-S, and, when the break is exited normally, control-H, Interlisp restores the bu er after the interaction.

The action of (CLEARBUF T), i.e., clearing of typeahead, is also available as the RUBOUT interrupt character, initially assigned to the del key in Interlisp-D and in Interlisp-10 under Tenex, control-Z under Tops20. Note that this interrupt clears both bu ers at the time it is *typed*, whereas the action of the CHARDELETE and LINEDELETE character occur at the time they are *read*.

(SYSBUF FLG) [Function]  
If FLG = T, returns the contents of the system bu er (as a string) that was saved at the last (CLEARBUF T T). If FLG = NIL, clears this internal bu er.

## INPUT/OUTPUT

(LINBUF FLG) [Function]  
Same as SYSBUF for the line bu er .

If both the system bu er and Interlisp's line bu er are empty, the internal bu ers associated with LINBUF and SYSBUF are not changed by a (CLEARBUF T T).

(BKSYSBUF X FLG RDTBL) [Function]  
BKSYSBUF sets the system bu er to the PRIN1-name of x. The effect is the same as though the user typed x. Some implementations have a limit on the length of x, in which case characters in x beyond the limit are ignored. Returns x.

If FLG is T, then the PRIN2-name of x is used, computed with respect to the readtable RDTBL .

Note that if the user is typing at the same time as the BKSYSBUF is being performed, the relative order of the type-in and the characters of x is unpredictable.

Compatibility note: Some implementations of BKSYSBUF (Interlisp-10) use a "system" bu er, from which keyboard interrupts are also processed. In this case, BKSYSBUF of an interrupt character actually invokes the interrupt at some (asynchronous) time after the BKSYSBUF is initiated. In other implementations (Interlisp-D), the characters are not processed for interrupts, and it is possible to BKSYSBUF characters which would otherwise be impossible to type.

(BKLINBUF STR) [Function]  
STR is a string. BKLINBUF sets Interlisp's line bu er to STR. Some implementations have a limit on the length of STR, in which case characters in STR beyond the limit are ignored. Returns STR.

BKLINBUF, BKSYSBUF, LINBUF, and SYSBUF provide a way of "undoing" a CLEARBUF. Thus to "peek" at various characters in the bu er, one could perform (CLEARBUF T T), examine the bu ers via LINBUF and SYSBUF, and then put them back.

The more common use of these functions is in saving and restoring typeahead when a program requires some unanticipated (from the user's standpoint) input. The function RESETBUFS provides a convenient way of simply clearing the input bu er, performing an interaction with the user, and then restoring the input bu er.

(RESETBUFS FORM<sub>1</sub> FORM<sub>2</sub> ... FORM<sub>N</sub>) [NLambda NoSpread Function]  
Clears any typeahead (ringing the terminal's bell if there was, indeed, typeahead), evaluates FORM<sub>1</sub>, FORM<sub>2</sub>, ... FORM<sub>N</sub>, then restores the typeahead. Returns the value of FORM<sub>N</sub>. Compiles open.

## 6.8 PRETTYPRINT

The standard way of printing out function definitions (on the terminal or into files) is to use PRETTYPRINT.

(PRETTYPRINT FNS PRETTYDEFL G \_ ) [Function]  
FNS is a list of functions. If FNS is atomic, its value is used). The definitions of

## Prettyprint

the functions are printed in a pretty format on the primary output le using the primary readtable. For example, if FACTORIAL were defined by typing

```
(DEFINEQ (FACTORIAL [LAMBDA (N) (COND ((ZEROP N) 1)
(T (ITIMES N (FACTORIAL (SUB1 N))
```

(PRETTYPRINT '(FACTORIAL)) would print out

```
(FACTORIAL
  [LAMBDA (N)
    (COND
      ((ZEROP N)
        1)
      (T (ITIMES N (FACTORIAL (SUB1 N))
```

Prettyprint is T when called from PRETTYDEF (and hence MAKEFILE). Among other actions taken when this argument is true, PRETTYPRINT indicates its progress in writing the current output le: whenever it starts a new function, it prints on the terminal the name of that function if more than 30 seconds (real time) have elapsed since the last time it printed the name of a function.

Prettyprint operates correctly on functions that are BROKEN, BROKEN-IN, ADVISED, or have been compiled with their definitions saved on their property lists: it prints the original, pristine definition, but does not change the current state of the function. If a function is not defined but is known to be on one of the les noticed by the le package, PRETTYPRINT loads in the definition (using LOADFNS) and prints it (except when called from PRETTYDEF). If PRETTYPRINT is given an atom which is not the name of a function, but has a value, it prettyprints the value. Otherwise, PRETTYPRINT attempts spelling correction. If all fails, PRETTYPRINT returns (FN NOT PRINTABLE).

(PP FN<sub>1</sub> FN<sub>N</sub>) [NLambda NoSpread Function]

For prettyprinting functions to the terminal. PP calls PRETTYPRINT with the primary output le set to T and the primary read table set to T. The primary output le and primary readtable are restored after printing.

(PP FOO) is equivalent to (PRETTYPRINT '(FOO)); (PP FOO FIE) is equivalent to (PRETTYPRINT '(FOO FIE)).

As described above, when PRETTYPRINT, and hence PP, is called with the name of a function that is not defined, but whose definition is on a le known to the le package, the definition is automatically read in and then prettyprinted. However, if the user does not intend on editing or running the definition, but simply wants to *see* the definition, the function PF described below can be used to simply copy the corresponding characters from the le to the terminal. This results in a savings in both space and time, since it is not necessary to allocate storage to actually read in the definition, and it is not necessary to re-prettyprint it (since the function is already in prettyprint format on the le).

(PF FN FROMFILES TOFILE) [NLambda NoSpread Function]

Copies the definition of FN found on each of the les in FROMFILES to TOFILE. If TOFILE = NIL, defaults to T. If FROMFILES = NIL, defaults to (WHEREIS FN NIL T) (see page 11.10). The typical usage of PF is simply to type 'PF FN'.

When printing to the terminal, PF performs several transformations on the characters in the le that comprise the definition for FN: (1) font information (page 6.55) is stripped out (except in Interlisp-D,



## INPUT/OUTPUT

whose display supports multiple fonts); (2) occurrences of the `CHANGECHAR` (page 6.55) are not printed; (3) since functions typically tend to be printed to a file with a larger linelength than when printing to a terminal, the number of leading spaces on each line is cut in half;<sup>12</sup> and (4) comments are elided, if `**COMMENT**FLG` is non-NIL (see page 6.50).

While the function `PRETTYPRINT` prints entire function definitions, the function `PRINTDEF` can be used to print parts of functions, or arbitrary Interlisp structures:

```
(PRINTDEF EXPR LEFT DEF TAILFLG FNSLST FILE) [Function]
Prints the expression EXPR in a pretty format on FILE using the primary readtable.
LEFT is the left hand margin (LINELENGTH determines the right hand margin.)13

DEF = T means EXPR is a function definition, or a piece of one. If DEF = NIL,
no special action is taken for LAMBDA's, PROG's, COND's, comments, CLISP, etc.
DEF is NIL when PRETTYDEF calls PRETTYPRINT to print variables and property
lists, and when PRINTDEF is called from the editor via the command PPV.

TAILFLG = T means EXPR is interpreted as a tail of a list, to be printed without
parentheses.

FNSLST is for use with the Font package (page 6.55). PRINTDEF prints occurrences
of any function in the list FNSLST in a different font, for emphasis. MAKEFILE
passes as FNSLST the list of all functions on the file being made.
```

### 6.8.1 Comment Feature

A facility for annotating Interlisp functions is provided in `PRETTYPRINT`. Any expression beginning with the atom `*` is interpreted as a comment and printed in the right margin. Example:

```
(FACTORIAL
  [LAMBDA (N)                                (* COMPUTES N! )
    (COND
      ((ZEROP N)                             (* 0!=1 )
        1)
      (T                                     (* RECURSIVE DEFINITION:
                                           N!=N*N-1! )
        (ITIMES N (FACTORIAL (SUB1 N))
```

These comments actually form a part of the function definition. Accordingly, `*` is defined as an `nlambda` nospread function that returns its argument, similar to `QUOTE`. When running an interpreted function, `*` is entered the same as any other Interlisp function. Therefore, comments should only be placed where they will not harm the computation, i.e., where a quoted expression could be placed. For example, writing

```
(ITIMES N (FACTORIAL (SUB1 N)) (* RECURSIVE DEFINITION))
```

---

<sup>12</sup>Unless `PFDEFAULT` is T. `PFDEFAULT` is initially NIL.

<sup>13</sup>`PRINTDEF` initially performs `(TAB LEFT T)`, which means to space to position `LEFT`, unless already beyond this position, in which case it does nothing.

## Comment Feature

in the above function would cause an error when `ITIMES` attempted to multiply `N,N-1!`, and `RECURSIVE`.

For compilation purposes, `*` is defined as a macro which compiles into no instructions (unless the comment has been placed where it has been used for value, in which case the compiler prints an appropriate error message and compiles `*` as `QUOTE`). Thus, the compiled form of a function with comments does not use the extra atom and list structure storage required by the comments in the source (interpreted) code. This is the way the comment feature is intended to be used.

A comment of the form `(* E x)` causes `x` to be evaluated at prettyprint time, as well as printed as a comment in the usual way. For example, `(* E (RADIX 8))` as a comment in a function containing octal numbers can be used to change the radix to produce more readable printout.

The comment character `*` is stored in the variable `COMMENTFLG`. The user can set it to some other value, e.g. `“;”`, and use this to indicate comments.

`COMMENTFLG` [Variable]  
If `CAR` of an expression is `EQ` to `COMMENTFLG`, the expression is treated as a comment by `PRETTYPRINT`. `COMMENTFLG` is initialized to `*`. Note that whatever atom is chosen for `COMMENTFLG` should also have an appropriate function definition and compiler macro, for example, by copying those of `*`.

Comments are designed mainly for documenting *listings*. Therefore, when prettyprinting to the terminal, comments are suppressed and printed as the string `**COMMENT**`. The value of `**COMMENT**FLG` determines the action.

`**COMMENT**FLG` [Variable]  
If `**COMMENT**FLG` is `NIL`, comments are printed. Otherwise, the value of `**COMMENT**FLG` is printed. Initially `" **COMMENT** "`.

The functions `PP*` and `PF*` are provided to easily print functions, including their comments.

`(PP* x)` [NLambda NoSpread Function]  
`PP*` operates exactly like `PP` except it `rst` sets `**COMMENT**FLG` to `NIL`, so comments are printed in full.

`(PF* FN FROMFILES TOFILE)` [NLambda NoSpread Function]  
`PF*` operates exactly like `PF` except it `rst` sets `**COMMENT**FLG` to `NIL`, so comments are printed in full.

`(COMMENT1 L _)` [Function]  
Prints the comment `L`. `COMMENT1` is a separate function<sup>14</sup> to permit the user to write prettyprint macros (page 6.54) that use the regular comment printer. For example, to cause comments to be printed at a larger than normal linelength, one could put an entry for `*` on `PRETTYPRINTMACROS`:

```
(* LAMBDA (X) (RESETFORM (LINELENGTH 100) (COMMENT1 X)))
```

---

<sup>14</sup>`COMMENT1` is an entry to the `PRETTYPRINT` block. However, it is called internally by `PRETTYPRINT` so that advising or redefining it will not affect the action of `PRETTYPRINT`. `COMMENT1` should *not* be called when not under a `PRINTDEF`.

## INPUT/OUTPUT

This macro resets the line length, prints the comment, and then restores the line length.

COMMENT1 expects to be called from within the environment established by PRINTDEF, so ordinarily the user should call it *only* from within prettyprint macros.

### 6.8.2 Comment Pointers

For a well-commented collection of programs, the list structure, atom, and pname storage required to represent the comments in core can be significant. If the comments already appear on a line and are not needed for editing, a significant savings in storage can be achieved by simply leaving the text of the comment on the line when the line is loaded, and instead retaining in core only a *pointer* to the comment. This feature has been implemented by defining \* as a read-macro in FILERDTBL which, instead of reading in the entire text of the comment, constructs an expression containing (1) the name of the line in which the text of the comment is contained, (2) the address of the first character of the comment, (3) the number of characters in the comment, and (4) a flag indicating whether the comment appeared at the right hand margin or centered on the page. For output purposes, \* is defined on PRETTYPRINTMACROS (page 6.54) so that it prints the comments represented by such pointers by simply copying the corresponding characters from one line to another, or to the terminal. Normal comments are processed the same as before, and can be intermixed freely with comment pointers.

The comment pointer feature is controlled by the value of NORMALCOMMENTSFLG.

NORMALCOMMENTSFLG [Variable]  
The comment pointer feature is enabled by setting NORMALCOMMENTSFLG to NIL. NORMALCOMMENTSFLG is initially T.

NORMALCOMMENTSFLG can be changed as often as desired. Thus, some lines can be loaded normally, and others with their comments converted to comment pointers.

For convenience of editing selected comments, an edit macro, GET\*, is included, which loads in the text of the corresponding comment. The editor's PP\* command, in contrast, prints the comment *without* reading it by simply copying the corresponding characters to the terminal. GET\* is defined in terms of GETCOMMENT:

(GETCOMMENT X DESTFL \_ ) [Function]  
If x is a comment pointer, replaces x with the actual text of the comment, which it reads from its line. Returns x in all cases. If DESTFL is non-NIL, it is the name of an open line, to which GETCOMMENT copies the comment; in this case, x remains a comment pointer, but it has been changed to point to the new line (unless NORMALCOMMENTSFLG = DONTUPDATE).

(PRINTCOMMENT X ) [Function]  
Defined as the prettyprint macro for \*: copies the comment to the primary output line by using GETCOMMENT.

(READCOMMENT FL RDTBL LST ) [Function]  
Defined as the read macro for \* in FILERDTBL: if NORMALCOMMENTSFLG is NIL,

## Converting Comments to Lower Case

it constructs a comment pointer.<sup>15</sup>

Note that a certain amount of care is required in using the comment pointer feature. Since the text of the comment resides on the `le` pointed to by the comment pointer, that `le` must remain in existence as long as the comment is needed. `GETCOMMENT` helps out by changing the comment pointer to always point at the most recent `le` that the comment lives on. However, if the user has been performing repeated `MAKEFILE`'s (page 11.6) in which differing functions have changed at each invocation of `MAKEFILE`, it is possible for the comment pointers in memory to be pointing at several versions of the same `le`, since a comment pointer is only updated when the function it lives in is prettyprinted, not when the function has been copied verbatim to the new `le`. This can be a problem for `le` systems, such as Tenex and Tops20, that have a built-in limit on the number of versions of a given `le` that will be made before old versions are expunged. In such a case, the user should set the version retention count of any directories involved to be in nite. `GETCOMMENT` prints an error message if the `le` that the comment pointer points at has disappeared.

Similarly, one should be cognizant of comment pointers in `SYSOUT`s, and be sure to retain any `les` thus pointed to.

When using comment pointers, the user should also not set `PRETTYFLG` (page 6.54) to `NIL` or call `MAKEFILE` with option `FAST`, since this will prevent functions from being prettyprinted, and hence not get the text of the comment copied into the new `le`.

If the user changes the value of `COMMENTFLG` but still wishes to use the comment pointer feature, the new `COMMENTFLG` should be given the same read-macro definition in `FILERDTBL` as `*` has, and the same entry be put on `PRETTYPRINTMACROS`. For example, if `COMMENTFLG` is reset to be `';`', then `(SETSYNTAX ' ; '* FILERDTBL)` should be performed, and `( ; . PRINTCOMMENT)` added to `PRETTYPRINTMACROS`.

### 6.8.3 Converting Comments to Lower Case

This section is for users operating on terminals without lower case, e.g. model 33 teletypes, who nevertheless would like their comments to be converted to lower case for more readable line-printer listings. If the second atom in a comment is `%%`, the text of the comment is converted to lower case so that it looks like English instead of LISP. Note that comments are converted *only* when they are actually written to a `le` by `PRETTYPRINT`.

The algorithm for conversion to lower case is the following: If the first character in an atom is `^`, do not change the atom (but remove the `^`). If the first character is `%`, convert the atom to lower case.<sup>16</sup> If the atom (minus any trailing punctuation marks) is an Interlisp word,<sup>17</sup> do not change it. Otherwise, convert the atom to lower case. Conversion only affects the upper case alphabet, i.e., atoms already converted to lower case are not changed if the comment is converted again. When converting, the first character in the comment and the first character following each period are left capitalized. After conversion, the comment is physically modified to be the lower case text minus the `%%` ag, so that conversion is thus

---

<sup>15</sup>Unless it believes the expression beginning with `*` is not actually a comment, e.g., if the next atom is `“.”` or `E`.

<sup>16</sup>User must type `%%` as `%` is the escape character.

<sup>17</sup>i.e., is a bound or free variable for the function containing the comment, or has a top level value, or is a defined function, or has a non-`NIL` property list.

## INPUT/OUTPUT

only performed once (unless the user edits the comment inserting additional upper case text and another %% ag).

LCASELST [Variable]  
Words on LCASELST will always be converted to lower case. LCASELST is initialized to contain words which are Interlisp functions but also appear frequently in comments as English words (AND, EVERY, GET, GO, LAST, LENGTH, LIST, etc.). Therefore, if one wished to type a comment including the lisp function GO, it would be necessary to type ^GO in order that it might be left in upper case.

UCASELST [Variable]  
Words on UCASELST (that do not appear on LCASELST) will be left in upper case. UCASELST is initialized to NIL.

ABBREVLST [Variable]  
ABBREVLST is used to distinguish between abbreviations and words that end in periods. Normally, words that end in periods and occur more than halfway to the right margin cause carriage-returns. Furthermore, during conversion to lowercase, words ending in periods, except for those on ABBREVLST, cause the rst character in the *next* word to be capitalized. ABBREVLST is initialized to the upper and lower case forms of ETC., I.E., and E.G..

### 6.8.4 Special Prettyprint Controls

PRETTYTABFLG [Variable]  
In order to save space on les, tabs are used instead of spaces for the initial spaces on each line, assuming that each tab corresponds to 8 spaces. This results in a reduction of le size by about 30%. Tabs are not used if PRETTYTABFLG is set to NIL (initially T).

#RPARS [Variable]  
Controls the number of right parentheses necessary for square bracketing to occur. If #RPARS= NIL, no brackets are used. #RPARS is initialized to 4.

FIRSTCOL [Variable]  
The starting column for comments. Initial setting is 48. Comments run between FIRSTCOL and LINELENGTH. If a word in a comment ends with a “.” and is not on the list ABBREVLST, and the position is greater than halfway between FIRSTCOL and LINELENGTH, the next word in the comment begins on a new line. Also, if a list is encountered in a comment, and the position is greater than halfway, the list begins on a new line.

PRETTYLCOM [Variable]  
If a comment is bigger (using COUNT) than PRETTYLCOM in size, it is printed starting at column 10, instead of FIRSTCOL. PRETTYLCOM is initialized to 14 (arrived at empirically). Comments are also printed starting at column 10 if their second element is also a \*, i.e., comments of the form (\* \* --).

#CAREFULCOLUMNS [Variable]  
In the interests of efficiency, PRETTYPRINT approximates the number of characters

## Special Prettyprint Controls

in each atom, rather than calling NCHARS, when computing how much will fit on a line. This procedure works satisfactorily in most cases. However, users with unusually long atoms in their programs, e.g., such as produced by CLISPIFY, may occasionally encounter some glitches in the output produced by PRETTYPRINT. The value of #CAREFULCOLUMNS tells PRETTYPRINT how many columns (counting from the right hand margin) in which to actually compute NCHARS instead of approximating. Setting #CAREFULCOLUMNS to 20 or 30 will eliminate the glitches, although it will slow down PRETTYPRINT slightly. #CAREFULCOLUMNS is initially 0.

(WIDEPAPER FLG)

[Function]

(WIDEPAPER T) sets FILELINELENGTH to 120, FIRSTCOL to 80, and PRETTYLCOM to 28. These are useful settings for prettyprinting lines to be listed on wide paper. (WIDEPAPER) restores these parameters to their initial values. The value of WIDEPAPER is its previous setting.

PrettyFLG

[Variable]

If PrettyFLG is NIL, PRINTDEF uses PRIN2 instead of prettyprinting. This is useful for producing a fast symbolic dump (see FAST option of MAKEFILE, page 11.6). Note that the line loads the same as if it were prettyprinted. PrettyFLG is initially set to T. PrettyFLG should not be set to NIL if comment pointers (page 6.51) are being used.

CLISPIFYPRETTYFLG

[Variable]

Used to inform PRETTYPRINT to call CLISPIFY on selected function definitions before printing them (see page 16.20).

PrettyPRINTMACROS

[Variable]

An association-list that enables the user to control the formatting of selected expressions. CAR of each expression being PRETTYPRINTed is looked up on PRETTYPRINTMACROS, and if found, CDR of the corresponding entry is applied to the expression. If the result of this application is NIL, PRETTYPRINT ignores the expression; i.e., it prints nothing, assuming that the prettyprintmacro has done any desired printing. If the result of applying the prettyprint macro is non-NIL, the result is prettyprinted in the normal fashion. This gives the user the option of computing some other expression to be prettyprinted in its place. PRETTYPRINTMACROS is initially NIL.

Note: “prettyprinted in the normal fashion” includes processing prettyprint macros, unless the prettyprint macro returns a structure EQ to the one it was handed, in which case the potential recursion is broken.

PrettyPRINTYPEMACROS

[Variable]

A list of elements of the form (TYPENAME . FN). For types other than lists and atoms, the type name of each datum to be prettyprinted is looked up on PRETTYPRINTYPEMACROS, and if found, the corresponding function is applied to the datum about to be printed, instead of simply printing it with PRIN2. PRETTYPRINTYPEMACROS is initially NIL.

PrettyEQUIVLST

[Variable]

An association-list that tells PRETTYPRINT to treat a CAR-of-form the same as some other CAR-of-form. For example, if (QLAMBDA . LAMBDA) appears

## INPUT/OUTPUT

on PRETTYEQUIVLST, then expressions beginning with QLAMBDA are prettyprinted the same as LAMBDA's. PRETTYEQUIVLST is initially NIL. Currently, PRETTYEQUIVLST only allows (i.e., supports in an interesting way) equivalences to forms that PRETTYPRINT internally handles. Equivalence to forms for which the user has specified a prettyprint macro should be made by adding further entries to PRETTYPRINTMACROS

CHANGECHAR

[Variable]

If non-NIL, and PRETTYPRINT is printing to a file or display terminal, PRETTYPRINT prints CHANGECHAR in the right hand margin while printing those expressions marked by the editor as having been changed (see page 17.22). CHANGECHAR is initially |.

### 6.8.5 Font Package

PrettyPRINT contains a facility for printing elements of various classes (user functions, system functions, clisp words, comments, etc.) in different fonts to emphasize (or deemphasize) their importance, and in general to provide for more pleasing printout when printing to a file. Of course, in order to be useful, this facility requires that the user has access to a printer which supports multiple fonts, such as an XGP.

Prettyprint signals font changes by inserting a user-defined escape sequence, e.g. ^F^C meaning change to font 3, ^F^A change back to font 1, etc. It is convenient if these sequences can consist of control characters, because by making these characters be separator characters in FILERDTBL, a file with font changes in it can also be loaded back in. Otherwise, the user would have to dump two files, one for listing, and one for loading.

Currently, the user can specify fonts for each of the following eight classes, each different, or the same for several classes.

LAMBDAFONT	The font for printing the name of the function being prettyprinted, before the actual definition (usually a large font).
CLISPFONT	If CLISPFLG is on, the font for printing any clisp words, i.e. atoms with property CLISPWORD.
COMMENTFONT	The font for everything inside of a comment.
USERFONT	The font for the name of any function in the file, or any member of the list FONTFNS.
SYSTEMFONT	The font for any other (defined) function.
CHANGEFONT	The font for anything in an expression marked by the editor as having been changed.
PrettyCOMFONT	The font used in printing the operand of a file package command.
DEFAULTFONT	The font for everything else, or any of the above classes for which a font is not specified.

Note: the output primitives PRINT, PRIN1, etc., currently do not know about variable width fonts, so

## Font Package

the user may have to experiment to find a compatible (pleasing) set of fonts. Note also that the user does not set LAMBDAFONT, CLISPFONT, et al, but indicates what font to be used by including an appropriate entry in FONTPROFILE. FONTSET will then set LAMBDAFONT, CLISPFONT, et al, to a data structure that contains the necessary information for performing the font change.

FONTPROFILE [Variable]

A list of elements of the form (FONTCLASS NIL FONT *q*),<sup>18</sup> where FONTCLASS is one of the eight font classes and FONT *q* is the font number for that class. It is assumed that the user has some way of communicating to the printing device the correspondence between font numbers and fonts. For each fontclass, the escape sequence consists of FONTESCAPECHAR followed by the character *code* for the font number, i.e. for font number 1, ^A, for font number 2, ^B, etc.

If FONT *q* is NIL for any fontclass, the DEFAULTFONT is used. Note that the DEFAULTFONT must be specified or an error is generated.

The operation of the font package is affected by a large number of parameters, e.g. FILELINELENGTH, LISTFILESTR, etc. plus the various fontnames themselves. To facilitate switching back and forth between various configurations, the font package allows the user to set the various parameters to their desired values, and then use the function FONTNAME to package up and save this configuration. Subsequently, the user invokes this configuration by performing (FONTSET NAME).

Note that the user may also want to reset FILELINELENGTH (page 23.14), PRETTYLCOM (page 6.53), and FIRSTCOL (page 6.53) as a part of various font configurations.

(FONTNAME NAME) [Function]

Performs some processing on FONTPROFILE, and then collects names and values of variables on FONTDEFSVARS, and saves them on FONTDEFS.

(FONTSET NAME) [Function]

Restores font configuration for NAME. Generates an error if NAME not previously defined.

FONTDEFSVARS [Variable]

The list of variables to be packaged by a FONTNAME. Initially FONTCHANGEFLG, FILELINELENGTH, COMMENTLINELENGTH, FIRSTCOL, PRETTYLCOM, LISTFILESTR, and FONTPROFILE.

FONTESCAPECHAR [Variable]

The character or string used to signal the start of a font escape sequence.

FONTCHANGEFLG [Variable]

If T, enables fonts, if NIL, disables fonts, i.e. no font changes are performed when prettyprinting.

---

<sup>18</sup>The NIL is a place marker. FONTNAME replaces (RPLACA) CADR when the font configuration is defined.



## INPUT/OUTPUT

LISTFILESTR	[Variable]
Passed to the operating system by LISTFILES (page 11.9). Can be used to specify subcommands to the LIST command, e.g. to establish correspondance between font number and font name.	
COMMENTLINELENGTH	[Variable]
Since comments are usually printed in a smaller font, COMMENTLINELENGTH is provided to o set the fact that Interlisp does not know about font widths. When FONTCHANGEFLG= T, CAR of COMMENTLINELENGTH is the linelength used to print short comments, i.e. those printed in the right margin, and CDR is the linelength used when printing full width comments.	
(CHANGEFONT FONTCLASS )	[Function]
Prints the font escape sequence to change to FONTCLASS . Note that FONTCLASS is not a font name, so one should use (CHANGEFONT LAMBDAFONT), not (CHANGEFONT 'LAMBDAFONT). For use in PRETTYPRINTMACROS .	
FONTDEFS	[Variable]
The dictionary of font con gurations. FONTDEFS is a list of elements of form (NAME . PARAMETER- PAIRS ). To save a con guration on a le after performing a FONTNAME to de ne it, the user could either save the entire value of FONTDEFS, or simply use an ALISTS le package command (page 11.23) to dump out just the one con guration.	

## 6.9 ASKUSER

DWIM, the compiler, the editor, and many other system packages all use ASKUSER, an extremely general user interaction package, for their interactions with the user at the terminal. ASKUSER takes as its principal argument KEYLST which is used to drive the interaction. KEYLST species what the user can type at any given point, how ASKUSER should respond to the various inputs, what value should be returned by ASKUSER, and is also used to present the user at any given point with a list of the possible responses. ASKUSER also takes other arguments which permit specifying a wait time, a default value, a message to be printed on entry, a ag indicating whether or not typeahead is to be permitted, a ag indicating whether the transaction is to be stored on the history list (page 8.1), a default set of options, and an (optional) input le/string.

### 6.9.1 Startup Protocol

Interlisp permits and encourages the user to typeahead; in actual practice, the user frequently does this. This presents a problem for ASKUSER. When ASKUSER is entered and there has been typeahead, was the input intended for ASKUSER, or was the interaction unanticipated, and the user simply typing ahead to some other program, e.g. the programmer's assistant? Even where there was no typeahead, i.e., the user starts typing *after* the call to ASKUSER, the question remains of whether the user had time to see the message from ASKUSER and react to it, or simply began typing ahead at an inauspicious moment. Thus, what is needed is an interlock mechanism which warns the user to stop typing, gives him a chance to respond to the warning, and then allows him to begin typing to ASKUSER.

## Startup Protocol

Therefore, when ASKUSER is rst entered, and the interaction is to take place with a terminal, and typeahead to ASKUSER is not permitted, the following protocol is observed:

(1) If there is typeahead, ASKUSER clears and saves the input buers and rings the bell to warn the user to stop typing. The buers will be restored when ASKUSER completes operation and returns.

(2) If MESS, the message to be printed on entry, is not NIL (the typical case), ASKUSER then prints MESS if it is a string, otherwise CAR of MESS, if MESS is a list.

(3) After printing MESS or CAR of MESS, ASKUSER waits until the output has actually been printed on the terminal to make sure that the user has actually had a chance to see the output. This also give the user a chance to react. ASKUSER then checks to see if anything additional has been typed in the intervening period since it rst warned the user in (1). If something has been typed, ASKUSER clears it out and again rings the bell. This latter material, i.e., that typed between the entry to ASKUSER and this point, is discarded and will not be restored since it is not certain whether the user simply reacted quickly to the rst warning (bell) and this input is intended for ASKUSER, or whether the user was in the process of typing ahead when the call to ASKUSER occurred, and did not stop typing at the rst warning, and therefore this input is a continuation of input intended for another program.

Anything typed after (3) is considered to be intended for ASKUSER, i.e., once the user sees MESS or CAR of MESS, he is free to respond. For example, UNDO (page 8.11) calls ASKUSER when the number of undosaves are exceeded for an event with MESS = (LIST NUMBER- UNDOSA VES "undosaves, continue saving"). Thus, the user can type a response as soon as NUMBER- UNDOSA VES is typed.

(4) ASKUSER then types the rest of MESS, if any.

(5) Then ASKUSER goes into a wait loop until something is typed. If WAIT, the wait time, is not NIL, and nothing is typed in WAIT seconds, ASKUSER will type "... " and treat the elements of DEFAULT, the default value, as a list of characters, and begin processing them exactly as though they had been typed. If the user does type anything within WAIT seconds, he can then wait as long as he likes, i.e., once something has been typed, ASKUSER will not use the default value specied in DEFAULT.

If the user wants to consider his response for more than WAIT seconds, and does not want ASKUSER to default, he can type a carriage return or a space, which are ignored if they are not specied as acceptable inputs by KEYLST (see below) and they are the rst thing typed.

If the calling program knows that the user is expecting an interaction with ASKUSER, e.g. another interaction preceded this one, it can specify in the call to ASKUSER that typeahead is permitted. In this case, ASKUSER simply notes whether there is any typeahead,<sup>19</sup> then prints MESS and goes into a wait loop as described above.

(6) Finally, if the interaction is not with the terminal, i.e., the optional input le/string is specied, ASKUSER simply prints MESS and begins reading from the le/string.

---

<sup>19</sup>In this case, if the typeahead turns out to contain unacceptable input, ASKUSER will assume that the typeahead was not intended for ASKUSER, and will restore the typeahead when it completes operation and returns.

## INPUT/OUTPUT

### 6.9.2 Operation

All input operations are executed with the terminal table in the variable `ASKUSERTTBL`., in which (1) `(CONTROL T)` has been executed, so that `ASKUSER` can interact with the user after each character is typed; and (2) `(ECHOMODE NIL)` has been executed, so that `ASKUSER` can decide *after* it reads a character whether or not the character should be echoed, and with what, e.g. unacceptable inputs are never echoed.

As each character is typed, it is matched against `KEYLST`, and appropriate echoing and/or prompting is performed. If the user types an unacceptable character, `ASKUSER` simply rings the bell and allows him to try again.

At any point, the user can type `?` and receive a list of acceptable responses at that point (generated from `KEYLST`), or type a control- A, control- Q, control- X, or `<del>`, which causes `ASKUSER` to reinitialize, and start over.

Note that `?`, Control- A, Control- Q, and Control- X will not work if they are acceptable inputs, i.e., they match one of the keys on `KEYLST`. `<del>` will not work if it is an interrupt character, in which case it is not seen by `ASKUSER`.

When an acceptable sequence is completed, `ASKUSER` returns the indicated value.

### 6.9.3 Format of KEYLST

`KEYLST` is a list of elements of the form `(KEY PROMPTSTRING . OPTIONS)`, where `KEY` is an atom or a string (equivalent), `PROMPTSTRING` is an atom or a string, and `OPTIONS` a list of options in property list format. The following options are recognized and explained below: `KEYLST`, `CONFIRMFLG`, `PROMPTCONFIRMFLG`, `NOCASEFLG`, `RETURN`, `EXPLAINSTRING`, `NOECHOFLG`, `KEYSTRING`, `PROMPTON`, `COMPLETEON`, `AUTOCOMLETEFLG`. If an option is specified in `OPTIONS`, the value of the option is the next element. Otherwise, if the option is specified in `OPTIONSLST` (the seventh argument to `ASKUSER`), its value is the next element on `OPTIONSLST`. Thus, `OPTIONSLST` can be used to provide default options for an entire `KEYLST`, rather than having to include the option at each level. If an option does not appear on either `OPTIONS` or `OPTIONSLST`, its value is `NIL`.

For convenience, an entry on `KEYLST` of the form `(KEY . ATOM/STRING)`, can be used as an abbreviation for `(KEY ATOM/STRING CONFIRMFLG T)`, and an entry of just the form `KEY`, i.e., a non-list, as an abbreviation for `(KEY NIL CONFIRMFLG T)`.

As each character is read, it is matched against the currently active keys. A character matches a key if it is the same character as that in the corresponding position in the key, or, if the character is an alphabetic character, if the characters are the same without regard for upper/lower case differences, i.e. 'A' matches 'a' and vice versa.<sup>20</sup> In other words, if two characters have already been input and matched, the third character is matched with each active key by comparing it with the third character of that key. If the character matches with one or more of the keys, the entries on `KEYLST` corresponding to the remaining keys are discarded. If the character does not match with any of the keys, the character is not echoed, and a bell is rung instead.

---

<sup>20</sup>Unless the `NOCASEFLG` option (page 6.62) is `T`.

## Format of KEYLST

When a key is complete, `PROMPTSTRING` is printed (NIL is equivalent to "", the empty string, i.e., nothing will be printed). Then, if the value of the `CONFIRMFLG` option is T, ASKUSER waits for confirmation of the key by a<sup>cr21</sup> or space. Otherwise, the key does not require confirmation.

Then, if the value of the `KEYLST` option is not NIL, its value becomes the new `KEYLST`, and the process recurses. Otherwise, the key is a "leaf," i.e., it terminates a particular path through the original, top-level `KEYLST`, and ASKUSER returns the result of packing all the keys that have been matched and completed along the way (unless the `RETURN` option is used to specify some other value, as described below).

For example, the following `KEYLST` is the default `KEYLST`, i.e., is used when ASKUSER is called with `KEYLST = NIL`: `((Y "escr") (N "ocr"))`

This `KEYLST` specifies that if (as soon as) the user types Y (or y), ASKUSER echoes with Y, prompts with "es<sup>cr</sup>", and returns Y as its value. Similarly, if the user types N, ASKUSER echoes the N, prompts with "o<sup>cr</sup>", and returns N. If the user types ?, ASKUSER prints:

Yes

No

to indicate his possible responses. All other inputs are unacceptable, and ASKUSER will ring the bell and not echo or print anything.

Here is a more complicated example, the `KEYLST` used for the compiler questions (page 12.1):

```
((ST "ore and redefine " KEYLST (" (F . "orget exprs"))
 (S . "ame as last time")
 (F . "File only")
 (T . "o terminal")
 1
 2
 (Y . "es")
 (N . "o"))
```

When ASKUSER is called with this `KEYLST`, and the user types an S, two keys are matched: ST and S. The user can then type a T, which matches only the ST key, or confirm the S key by typing a<sup>cr</sup> or space. If the user confirms the S key, ASKUSER prompts with "ame as last time", and returns S as its value. (Note that the confirming character is not included in the value.) If the user types a T, ASKUSER prompts with "ore and redefine", and makes `(( " (F . "orget exprs"))` be the new `KEYLST`, and waits for more input. The user can then type an F, or confirm the "" (which essentially starts out with all of its characters matched). If he confirms the "", ASKUSER returns ST as its value the result of packing ST and "". If he types F, ASKUSER prompts with "orget exprs", and waits for confirmation again. If the user then confirms, ASKUSER returns STF, the result of packing ST and F.

As mentioned earlier, at any point the user can type a ? and be prompted with the possible responses. For example, if the user types S and then ?, ASKUSER will type:

```
STore and redefine Forget exprs
STore and redefine
Same as last time
```

---

<sup>21cr</sup> is used throughout the discussion to denote carriage return.

## INPUT/OUTPUT

### 6.9.4 Completing a Key

The decision about when a key is complete is more complicated than simply whether or not all of its characters have been matched. In the example above, all of the characters in the S key are matched as soon as the S has been typed, but until the next character is typed, ASKUSER does not know whether the S completes the S key, or is simply the first character in the ST key. Therefore, a key is considered to be complete when:

- (1) All of its characters have been matched and it is the only key left, i.e., there are no other keys for which this key is a substring; or
- (2) All of its characters have been matched and a confirming character is typed; or
- (3) All of its characters have been matched, and the value of the CONFIRMFLG option is NIL, and the value of the KEYLST option is not NIL, and the next character matches one of the keys on the value of the KEYLST option; or
- (4) There is only one key left and a confirming character is typed. Note that if the value of CONFIRMFLG is T, the key still has to be confirmed, regardless of whether or not it is complete. For example, if the first entry in the above example were instead

```
(ST "ore and redefine " CONFIRMFLG T KEYLST (" (F . "orget exprs"))
```

and the user wanted to specify the STF path, he would have to type ST, *then* confirm before typing F, even though the ST completed the ST key by the rule in case (1). However, he would be prompted with ‘ore and redefine’ as soon as he typed the T, and completed the ST key.

Case (2) says that confirmation can be used to complete a key in the case where it is a substring of another key, even where the value of CONFIRMFLG is NIL. In this case, the confirming character doubles as both an indicator that the key is complete, and also to confirm it, if necessary. This situation corresponds to typing S<sup>cr</sup> in the above example.

Case (3) says that if there were another entry whose key was STX in the above example, so that after the user typed ST, two keys, ST and STX, were still active, then typing F would complete the ST key, because F matches the (F . "orget exprs") entry on the value of the KEYLST option of the ST entry. In this case, ‘ore and redefine’ would be printed *before* the F was echoed.

Finally, case (4) says that the user can use confirmation to specify completion when only one key is left, even when all of its characters have not been matched. For example, if the first key in the above example were STORE, the user could type ST and then confirm, and ORE would be echoed, followed by whatever prompting was specified. In this case, the confirming character also confirms the key if necessary, so that no further action is required, even when the value of CONFIRMFLG is T.

Case (4) permits the user not to have to type every character in a key when the key is the only one left. Even when there are several active keys, the user can type type \$ (the ESC key, or on some terminals, the key labelled ALT) to specify the next N>0 common characters among the currently active keys. The effect is exactly the same as though these characters had been typed. If there are no common characters in the active keys at that point, i.e. N=0, the \$ is treated as an incorrect input, and the bell is rung. For example, if KEYLST is (CLISPFLG CLISPIFYPACKFLG CLISPIFYTRANFLG), and the user types C followed by \$, ASKUSER will supply the L, I, S, and P. The user can then type F followed by<sup>cr</sup> or space to complete and confirm CLISPFLG, as per case (4), or type I, followed by \$, and ASKUSER will supply the F, etc. Note that the characters supplied do not have to correspond to a terminal segment of

## Options

any of the keys. Note also that the \$ does not confirm the key, although it may complete it in the case that there is only one key active.

If the user types a confirming character when several keys are left, the next  $N > 0$  common characters are still supplied, the same as with \$. However, ASKUSER assumes the intent was to complete a key, i.e., case (4) is being invoked. Therefore, after supplying the next  $N$  characters, the bell is rung to indicate that the operation was not completed. In other words, typing a confirming character has the same effect as typing a \$ in that the next  $N$  common characters are supplied. Then, if there is only one key left, the key is complete (case 4) and confirmation is not required. If the key is not the only key left, the bell is rung.

### 6.9.5 Options

KEYLST	When a key is complete, if the value of the KEYLST option is not NIL, this value becomes the new KEYLST and the process recurses. Otherwise, the key terminates a path through the original, top-level KEYLST, and ASKUSER returns the indicated value.
CONFIRMFLG	If T, the key must be confirmed with either a <sup>cr</sup> or a space. If the value of CONFIRMFLG is a <i>list</i> , the confirming character may be any member of the list.
PROMPTCONFIRMFLG	If T, whenever confirmation is required, the user is prompted with the string "[confirm] ".
NOCASEFLG	If T, says do <i>not</i> perform case independent matching on alphabetic characters. If NIL, do perform case independent matching, i.e. 'A' matches with 'a' and vice versa.
RETURN	If non-NIL, EVAL of the value of the RETURN option is returned as the value of ASKUSER. Note that different RETURN options can be specified for different keys. The variable ANSWER is bound in ASKUSER to the list of keys that have been matched. In other words, RETURN (PACK ANSWER) would be equivalent to what ASKUSER normally does.
EXPLAINSTRING	If the value of the EXPLAINSTRING option is non-NIL, its value is printed when the user types a ?, rather than KEY + PROMPTSTRING. EXPLAINSTRING enables more elaborate explanations in response to a ? than what the user sees when he is prompted as a result of simply completing keys. See example below.
NOECHOFLG	If non-NIL, characters that are matched (or automatically supplied as a result of typing \$ or confirming) are not echoed, nor is the confirming character, if any. The value of NOECHOFLG is automatically NIL when ASKUSER is reading from a file or string. The decision about whether or not to echo a character that matches several keys is determined by the value of the NOECHOFLG option for the first key.

Example: one of the entries on the KEYLST used by ADDTOFILES? (page 11.8) is:

```
([] "Nowherecr" NOECHOFLG T
  EXPLAINSTRING "]" - nowhere, item is marked as a dummycr)
```

## INPUT/OUTPUT

When the user types ], ASKUSER just prints 'Nowhere<sup>cr</sup>', i.e., the ] is not echoed. If the user types ?, the explanation corresponding to this entry will be:

] - nowhere, item is marked as a dummy

KEYSTRING	If non-NIL, characters that are matched are echoed as though the value of KEYSTRING were used in place of the key. KEYSTRING is also used for computing the value returned. The main reason for this feature is to enable echoing in lowercase.
PROMPTON	If non-NIL, PROMPTSTRING is printed <i>only</i> when the key is confirmed with a member of the value of PROMPTON. See example below.
COMPLETEON	When a confirming character is typed, the N characters that are automatically supplied, as specified in case (4), are echoed <i>only</i> when the key is confirmed with a member of the value of PROMPTON.

The PROMPTON and COMPLETEON options enable the user to construct a KEYLIST which will cause ASKUSER to emulate the action of the TENEX exec. The protocol followed by the TENEX exec is that the user can type as many characters as he likes in specifying a command. The command can be completed with a<sup>cr</sup> or space, in which case no further output is forthcoming, or with a \$, in which case the rest of the characters in the command are echoed, followed by some prompting information. The following KEYLIST would handle the TENEX COPY and CONNECT commands:

```
((COPY " (FILE LIST) "
    PROMPTON ($)
    COMPLETEON ($)
    CONFIRMFLG ($))
(CONNECT " (TO DIRECTORY) "
    PROMPTON ($)
    COMPLETEON ($)
    CONFIRMFLG ($)))
```

### AUTOCOMLETEFLG

If the value of the AUTOCOMLETEFLG option is not NIL, ASKUSER will automatically supply unambiguous characters whenever it can, i.e., ASKUSER acts as though \$ were typed after each character (except that it does not ring the bell if there are no unambiguous characters).

### MACROCHARS

value is a list of dotted pairs of form (CHARACTER . FORM). When CHARACTER is typed, and it does not match any of the current keys, FORM is evaluated and nothing else happens, i.e. the matching process stays where it is. For example, ? could have been implemented using this option. Essentially MACROCHARS provides a read macro facility while inside of ASKUSER (since ASKUSER does READC's, read macros defined via the readtable are never invoked).

### EXPLAINDELIMITER

value is what is printed to delimit explanation in response to ?. Initially '<sup>cr</sup>' but can be reset, e.g. to ', ', for more linear output.

## Special Keys

### 6.9.6 Special Keys

& can be used as a key to match with any single character, provided the character does not match with some other key at that level. For the purposes of echoing and returning a value, the effect is the same as though the character that were matched actually appeared as the key.

\$ (esc) can be used as a key to match with the result of a single call to READ. For example, if the first entry in the TENEX KEYLST above were:

```
(COPY " (FILE LIST) "  
    PROMPTON ($)   
    COMPLETEON ($)   
    CONFIRMFLG ($)   
    KEYLST (($ NIL RETURN ANSWER)))
```

then if the user typed `COP FOO`<sup>22</sup>, `(COPY FOO)` would be returned as the value of `ASKUSER`. One advantage of using \$, rather than having the calling program perform the READ, is that the call to READ from inside `ASKUSER` is `ERRORSET` protected, so that the user can back out of this path and reinitialize `ASKUSER`, e.g. to change from a `COPY` command to a `CONNECT` command, simply by typing control-E.

\$\$ can be used as a key to match with the result of a single call to `READLINE`.

A list can be used as a key, in which case the list/form is evaluated and its value “matches” the key. This feature is provided primarily as an escape hatch for including arbitrary input operations as part of an `ASKUSER` sequence. For example, the effect of \$\$ could be achieved simply by using `(READLINE T)` as a key.<sup>22</sup>

“” can be used as a key. Since it has no characters, all of its characters are automatically matched. “” essentially functions as a place marker. For example, one of the entries on the `KEYLST` used by `ADDTFILES?` is:

```
(" "File/list: "  
    EXPLAINSTRING "a file name or name of a function list"  
    KEYLST ($))
```

Thus, if the user types a character that does not match any of the other keys on the `KEYLST`, then the character completes the “” key, by virtue of case (4), since the character *will* match with the \$ in the inner `KEYLST`. `ASKUSER` then prints ‘File/list: ’ *before* echoing the character, then calls `READ`. The character will be read as part of the `READ`. The value returned by `ASKUSER` will be the value of the `READ`.

```
(ASKUSER WAIT DEFAULT MESS KEYLST TYPEAHEAD LISPXPRTFL G OPTIONSLST FILE)
```

[Function]

`WAIT` is either `NIL` or a number (of seconds). `DEFAULT` is a single character or a sequence (list) of characters to be used as the default inputs for the case when `WAIT` is not `NIL` and more than `WAIT` seconds elapse without any input. In this

---

<sup>22</sup>For \$, \$\$, or a list, if the last character read by the input operation is a separator, the character is treated as a confirming character for the key. However, if the last character is a break character, it will be matched against the next key.



## INPUT/OUTPUT

case, the character(s) from `DEFAULT` are processed exactly as though they had been typed, except that `ASKUSER` rst types “...”.

`MESS` is the initial message to be printed by `ASKUSER`, if any, and can be a string, or a list. In the latter case, each element of the list is printed, separated by spaces, and terminated with a “ ? ”. `KEYLST` and `OPTIONSLST` were described earlier. `TYPEAHEAD` is T if the user is permitted to typeahead a response to `ASKUSER`. `NIL` means any typeahead should be cleared and saved. `LISPXPRTFLG` determines whether or not the interaction is to be recorded on the history list. `FILE` can be either `NIL` (in which case it is set to T), the name of a file, or a string.<sup>23</sup> All input operations take place from `FILE` until an unacceptable input is encountered, i.e., one that does not conform to the protocol defined by `KEYLST`. At that point, `FILE` is set to T, `DEFAULT` is set to `NIL`, the input buffer is cleared, and a bell is rung. Unacceptable inputs are not echoed.

The value of `ASKUSER` is the result of packing all the keys that were matched, unless the `RETURN` option is specified (page 6.62).

(`MAKEKEYLST` `LST` `DEFAULTKEY` `LCASEFLG` `_`) [Function]  
`LST` is a list of atoms or strings. `MAKEKEYLST` returns an `ASKUSER` `KEYLST` which will permit the user to specify one of the elements on `LST` by either typing enough characters to make the choice unambiguous, or else typing a number between 1 and `N`, where `N` is the length of `LST`.

For example, if `ASKUSER` is called with `KEYLST = (MAKEKEYLST '(CONNECT SUPPORT COMPILE))`, then the user can type C-O-N, S, C-O-M, 1, 2, or 3 to indicate one of the three choices.

If `LCASEFLG = T`, then echoing of upper case elements will be in lower case (but the value returned will still be one of the elements of `LST`). If `DEFAULTKEY` is non-`NIL`, it will be the last key on the `KEYLST`. Otherwise, a key which permits the user to indicate “No - none of the above” choices, in which case the value returned by `ASKUSER` will be `NIL`.

---

<sup>23</sup>If `FILE` is a string, and all of its characters are read before `ASKUSER` finishes, `FILE` will be reset to T, and the interaction will continue with `ASKUSER` reading from the terminal.

## Special Keys