25. USER/ INPUT/OUTPUT PACKAGES

Interlisp-D can perform input/output operations on a large variety of physical devices.

This chapter presents a number of packages that have been developed for displaying and allowing the user to enter information. These packages are used to implement the user interface of many system facilities.

INSPECT (see the INSPECT section below) provides a window-based facility for displaying and changing the fields of a data object.

PROMPTFORWORD (see the PROMPTFORWORD section below) is a function used for entering a simple string of characters. Basic editing and prompting facilities are provided.

ASKUSER (see the ASKUSER section below) provides a more complicated prompting and answering facility, allowing a series of questions to be printed. Prompts and argument completion are supported.

TTYIN (see the TTYIN Display Typein Editor section below) is a display typein editor, that provides complex text editing facilities when entering an input line.

PRETTYPRINT (see the Prettyprint section below) is used for printing function definitions and other list structures, using multiple fonts and indenting lines to show the structure of the list.

Inspector

The Inspector provides a display-oriented facility for looking at and changing arbitrary Interlisp-D data structures. The inspector can be used to inspect all user datatypes and many system datatypes (although some objects such as numbers have no inspectable structure). The inspector displays the field names and values of an arbitrary object in a window that allows setting of the properties and further inspection of the values. This latter feature makes it possible to "walk" around all of the data structures in the system at the touch of a button. In addition, the inspector is integrated with the break package to allow inspection of any object on the stack and with the display and teletype structural editors to allow the editors to be used to "inspect" list structures and the inspector to "edit" datatypes.

The underlying mechanisms of the data inspector have been designed to allow their use as specialized editors in user applications. This functionality is described at the end of this section.

Note: Currently, the inspector does not have UNDOing. Also, variables whose values are changed will not be marked as such.

Calling the Inspector

There are several ways to open an inspect window onto an object. In addition to calling INSPECT

directly (below), the inspector can also be called by buttoning an Inspect command inside an existing inspector window. Finally, if a non-list is edited with EDITDEF (see Chapter 17), the inspector is called. This also causes the inspector to be called by the Dedit command from the display editor or the EV command from the teletype editor if the selected piece of structure is a non-list.

(**INSPECT** *OBJECT ASTYPE WHERE*) [Function]

Creates an inspect window onto *OBJECT*. If *ASTYPE* is given, it will be taken as the record type of *OBJECT*. This allows records to be inspected with their property names. If *ASTYPE* is NIL, the data type of *OBJECT* will be used to determine its property names in the inspect window.

WHERE specifies the location of the inspect window. If *WHERE* is NIL, the user will be prompted for a location. If *WHERE* is a window, it will be used as the inspect window. If *WHERE* is a region, the inspect window will be created in that region of the screen. If *WHERE* is a position, the inspect window will have its lower left corner at that position on the screen.

INSPECT returns the inspect window onto *OBJECT*, or NIL if no inspection took place.

(**INSPECTCODE** *FN WHERE* — — — —) [Function]

Opens a window and displays the compiled code of the function *FN* using PRINTCODE. The window is scrollable.

WHERE determines where the window should appear. It can be a position, a region, or a window. If NIL, the user is prompted to specify the position of the window.

Note: If the Tedit library package is loaded, INSPECTCODE uses it to create the code inspector window. Also, if INSPECTCODE is called to inspect the frame name in a break window (see Chapter 14), the location in the code that the frame's PC indicates it was executing at the time is highlighted.

Multiple Ways of Inspecting

For some datatypes there is more than one aspect that is of interest or more than one method of inspecting the object. In these cases, the inspector will bring up a menu of the possibilities and wait for the user to select one.

If the object is a litatom, the commands are the types for which the litatom has definitions as determined by HASDEF. Some typical commands are:

FNS Edit the definition of the selected litatom.

VARS Inspect the value.

PROPS Inspect the property list.

If the object is a list, there will be choice of how to inspect the list:

Inspect Opens an inspect window in which the properties are numbers and the values are the elements of the list.

TtyEdit	Calls the teletype list structure editor on the list (see Chapter 16).
DisplayEdit	Calls the DEdit display editor on the list (see Chapter 16).
As a PLIST	Inspects the list as a property list, if the list is in property list form: ((PROP$_1$ VAL$_1$) ... (PROP$_N$ VAL$_N$)).
As an ALIST	Inspects the list as an association-list, if the list is in ASSOC list form: (PROP$_1$ VAL$_1$ ... PROP$_N$ VAL$_N$).
As a record	Brings up a submenu with all of the RECORDs in the system and inspect the list with the one chosen.
As a "record type"	Inspects the list as the record of the type named in its CAR, if the CAR of the list is the name of a TYPERECORD (see Chapter 8).

If the object is a bitmap, the choice is between inspecting the bitmap's contents with the bitmap editor (EDITBM) or inspecting the bitmap's fields.

Other datatypes may include multiple methods for inspecting objects of that type.

Inspect Windows

An inspect window displays two columns of values. The lefthand column lists the property names of the structure being inspected. The righthand column contains the values of the properties named on the left. For variable length data such as lists and arrays, the "property names" are numbers from 1 to the length of the inspected item and the values are the corresponding elements. For arrays, the property names are the array element numbers and the values are the corresponding elements of the array.

For large lists or arrays, or datatypes with many fields, the initial window may be too small to contain all of them. In these cases, the unseen elements can be scrolled into view (from the bottom) or the window can be reshaped to increase its size.

In an inspect window, the LEFT button is used to select things, the MIDDLE button to invoke commands that apply to the selected item. Any property or value can be selected by pointing the cursor directly at the text representing it, and clicking the LEFT button. There is one selected item per window and it is marked by having its surrounding box inverted.

The options offered by the MIDDLE button depend on whether the selection is a property or a value. If the selected item is a value, the options provide different ways of inspecting the selected structure. The exact commands that are given depend on the type of the value. An example of the menu you may see is:

If the selected item is a property name, the command SET will appear. If selected, the user will be asked to type in an expression, and the selected property will be set to the result of evaluating the read form. The evaluation of the read form and the replacement of the selected item property will appear as their own history events and are individually undoable. Properties of system datatypes cannot be set. (There are often consistency requirements which can be inadvertently violated in ways that crash the system. This may be true of some user datatypes as well, however the system doesn't know which ones. Users are advised to exercise caution.)

It is possible to copy-select property names or values out of an inspect window. Litatoms, numbers and strings are copied as they are displayed. Unprintable objects (such as bitmaps, etc.) come out as an appropriate system expression, such that if is evaluated, the object is re-created.

Inspect Window Commands

By pressing the MIDDLE button in the title of the inspect window, a menu of commands that apply to the inspect window is brought up:

ReFetch [Inspect Window Command]

An inspect window is not automatically updated when the structure it is inspecting is changed. The ReFetch command will refetch and redisplay all of the fields of the object being inspected in the inspect window.

IT_datum [Inspect Window Command]

Sets the variable IT to object being inspected in the inspect window.

IT_selection [Inspect Window Command]

Sets the variable IT to the property name or value currently selected in the inspect window.

Interaction With Break Windows

The break window facility (see Chapter 14) knows about the inspector in the sense that the backtrace frame window is an inspect window onto the frame selected from the back trace menu during a break. Thus you can call the inspector on an object that is bound on the stack by selecting its frame in the back trace menu, selecting its value with the LEFT button in the back trace frame window, and selecting the inspect command with the MIDDLE button in the back trace frame window. The values of variables in frames can be set by selecting the variable name with the LEFT button and then the "Set" command with the MIDDLE button.

Note: The inspector will only allow the setting of named variables. Even with this restriction it is still possible to crash the system by setting variables inside system frames. Exercise caution in setting variables in other than your own code.

Controlling the Amount Displayed During Inspection

The amount of information displayed during inspection can be controlled using the following variables:

MAXINSPECTCDRLEVEL [Variable]

The inspector prints only the first MAXINSPECTCDRLEVEL elements of a long list, and will make the tail containing the unprinted elements the last item. The last item can be inspected to see further elements. Initially 50.

MAXINSPECTARRAYLEVEL [Variable]

The inspector prints only the first MAXINSPECTARRAYLEVEL elements of an array. The remaining elements can be inspected by calling the function (INSPECT/ARRAY ARRAY BEGINOFFSET) which inspects the BEGINOFFSET through the BEGINOFFSET + MAXINSPECTARRAYLEVEL elements of ARRAY. Initially 300.

INSPECTPRINTLEVEL [Variable]

When printing the values, the inspector resets PRINTLEVEL (see Chapter 25) to the value of INSPECTPRINTLEVEL. Initially (2 . 5).

INSPECTALLFIELDSFLG [Variable]

If INSPECTALLFIELDSFLG is T, the inspector will show computed fields (ACCESSFNS, Chapter 8) as well as regular fields for structures that have a record definition. Initially T.

Inspect Macros

The Inspector can be extended to inspect new structures and datatypes by adding entries to the list INSPECTMACROS. An entry should be of the form (OBJECTTYPE . INSPECTINFO). OBJECTTYPE is used to determine the types of objects that are inspected with this macro. If OBJECTTYPE is a litatom, the INSPECTINFO will be used to inspect items whose type name is OBJECTTYPE. If OBJECTTYPE is a list of the form (FUNCTION DATUM-PREDICATE), DATUM-PREDICATE will be APPLYed to the item and if it returns non-NIL, the INSPECTINFO will be used to inspect the item.

INSPECTINFO can be one of two forms. If INSPECTINFO is a litatom, it should be a function that will be applied to three arguments (the item being inspected, OBJECTTYPE, and the value of WHERE passed to INSPECT) that should do the inspection. If INSPECTINFO is not a litatom, it should be a list of (PROPERTIES FETCHFN STOREFN PROPCOMMANDFN VALUECOMMANDFN TITLECOMMANDFN TITLE SELECTIONFN WHERE PROPPRINTFN) where the elements of this list are the arguments for INSPECTW.CREATE, described below. From this list, the WHERE argument will be evaluated; the others will not. If WHERE is NIL, the value of WHERE that was passed to INSPECT will be used.

Examples:

The entry ((FUNCTION MYATOMP) PROPNAMES GETPROP PUTPROP) on INSPECTMACROS would

cause all objects satisfying the predicate MYATOMP to have their properties inspected with GETPROP and PUTPROP. In this example, MYATOMP should make sure the object is a litatom.

The entry (MYDATATYPE . MYINSPECTFN) on INSPECTMACROS would cause all datatypes of type MYDATATYPE to be passed to the function MYINSPECTFN.

INSPECTWs

The inspector is built on the abstraction of an INSPECTW. An INSPECTW is a window with certain window properties that display an object and respond to selections of the object's parts. It is characterized by an object and its list of properties. An INSPECTW displays the object in two columns with the property names on the left and the values of those properties on the right. An INSPECTW supports the protocol that the LEFT mouse button can be used to select any property name or property value and the MIDDLE button calls a user provided function on the selected value or property. For the Inspector application, this function puts up a menu of the alternative ways of inspecting values or of the ways of setting properties. INSPECTWs are created with the following function:

(**INSPECTW.CREATE** *DATUM PROPERTIES FETCHFN STOREFN PROPCOMMANDFN VALUECOMMANDFN TITLECOMMANDFN TITLE SELECTIONFN WHERE PROPPRINTFN*) [Function]

Creates an INSPECTW that views the object *DATUM*. If *PROPERTIES* is a list, it is taken as the list of properties of *DATUM* to display. If *PROPERTIES* is a litatom, it is APPLYed to *DATUM* and the result is used as the list of properties to display.

FETCHFN is a function of two arguments (OBJECT PROPERTY) that should return the value of the PROPERTY property of OBJECT. The result of this function will be printed (with PRIN2) in the INSPECTW as the value.

STOREFN is a function of three arguments (OBJECT PROPERTY NEWVALUE) that changes the PROPERTY property of OBJECT to NEWVALUE. It is used by the default *PROPCOMMANDFN* and *VALUECOMMANDFN* to change the value of a property and also by the function INSPECTW.REPLACE (described below). This can be NIL if the user provides command functions which do not call INSPECTW.REPLACE. Each replace action will be a separate event on the history list. Users are encouraged to provide UNDOable STOREFNs.

PROPCOMMANDFN is a function of three arguments (PROPERTY OBJECT INSPECTW) which gets called when the user presses the MIDDLE button and the selected item in the INSPECTW is a property name. PROPERTY will be the name of the selected property, OBJECT will be the datum being viewed, and INSPECTW will be the window. If *PROPCOMMANDFN* is a string, it will get printed in the PROMPTWINDOW when the MIDDLE button is pressed. This provides a convenient way to notify the user about disabled commands on the properties. DEFAULT.INSPECTW.PROPCOMMANDFN, the default *PROPCOMMANDFN*, will present a menu with the single command Set on it. If selected, the Set command will read a value from the user and set the selected property to the result of EVALuating this read value.

VALUECOMMANDFN is a function of four arguments (VALUE PROPERTY OBJECT

INSPECTW) that gets called when the user presses the MIDDLE button and the selected item in the INSPECTW is a property value. VALUE will be the selected value (as returned by *FETCHFN*), PROPERTY will be the name of the property VALUE is the value of, OBJECT will be the datum being viewed, and INSPECTW will be the INSPECTW window.

DEFAULT.INSPECTW.VALUECOMMANDFN, the default *VALUECOMMANDFN*, will present a menu of possible ways of inspecting the value and create a new Inspect window if one of the menu items is selected.

TITLECOMMANDFN is a function of two arguments (INSPECTW OBJECT) which gets called when the user presses the MIDDLE button and the cursor is in the title or border of the inspect window INSPECTW. This command function is provided so that users can implement commands that apply to the entire object. The default *TITLECOMMANDFN* (DEFAULT.INSPECTW.TITLECOMMANDFN) presents a menu with the commands ReFetch, IT_datum, and IT_election .

TITLE specifies the title of the window. If *TITLE* is NIL, the title of the window will be the printed form of *DATUM* followed by the string " Inspector". If *TITLE* is the litatom DON'T, the inspect window will not have a title. If *TITLE* is any other litatom, it will be applyed to the *DATUM* and the potential inspect window (if it is known). If this result is the litatom DON'T, the inspect window will not have a title; otherwise the result will be used as a title. If *TITLE* is not a litatom, it will be used as the title.

SELECTIONFN is a function of three arguments (PROPERTY VALUEFLG INSPECTW) which gets called when the user releases the left button and the cursor is on one of the items. The *SELECTIONFN* allows a program to take action on the user's selection of an item in the inspect window. At the time this function is called, the selected item has been "selected". The function INSPECTW.SELECTITEM (described below) can be used to turn off this selection. PROPERTY will be the name of the property of the selected item. *VALUEFLG* will be NIL if the selected item is the property name; T if the selected item is the property value.

WHERE indicates where the inspect window should go. Its interpretation is described in INSPECT (see above).

PROPPRINTFN is a function of two arguments (PROPERTY DATUM) which gets called to determine what to print in the property place for the property PROPERTY. If *PROPPRINTFN* returns NIL, no property name will be printed and the value will be printed to the left of the other values.

An inspect window uses the following window property names to hold information: *DATUM*, *FETCHFN*, *STOREFN*, *PROPCOMMANDFN*, *VALUECOMMANDFN*, *SELECTIONFN*, *PROPPRINTFN*, *INSPECTWTITLE*, *PROPERTIES*, *CURRENTITEM* and *SELECTABLEITEMS*.

(**INSPECTW.REDISPLAY** *INSPECTW PROPS* —) [Function]

Updates the display of the objects being inspected in INSPECTW. If *PROPS* is a property name or a list of property names, only those properties are updated. If *PROPS* is NIL, all properties are redisplayed. This function is provided because inspect windows do not automatically update their display when the object they are showing changes.

This function is called by the ReFetch command in the title command menu of an

INSPECTW (see above).

(**INSPECTW.REPLACE** *INSPECTW PROPERTY NEWVALUE*) [Function]

Calls the STOREFN of the inspect window *INSPECTW* to change the property named *PROPERTY* to the value *NEWVALUE* and updates the display of *PROPERTY*'s value in the display. This provides a functional interface for user PROPCOMMANDFNs.

(**INSPECTW.SELECTITEM** *INSPECTW PROPERTY VALUEFLG*) [Function]

Sets the selected item in an inspect window. The item is inverted on the display and put on the window property CURRENTITEM of *INSPECTW*. If *INSPECTW* has a CURRENTITEM, it is deselected. *PROPERTY* is the name of the property of the selected item. *VALUEFLG* is NIL if the selected item is the property name; T if the selected item is the property value. If *PROPERTY* is NIL, no item will be selected. This provides a way of deselecting all items.

PROMPTFORWORD

PROMPTFORWORD is a function that reads in a sequence of characters, generally from the keyboard, without involving READ-like syntax. A user can supply a prompting string, as well as a "candidate" string, which is printed and used if the user types only a word terminator character (or doesn't type anything before a given time limit). As soon as any characters are typed the "candidate" string is erased and the new input takes its place.

PROMPTFORWORD accepts user type-in until one of the "word terminator" characters is typed. Normally, the word terminator characters are EOL, ESCAPE, LF, SPACE, or TAB. This list can be changed using the TERMINCHAR.LST argument to PROMPTFORWORD, for example if it is desirable to allow the user to input lines including spaces.

PROMPTFORWORD also recognizes the following special characters:

Control-A
BACKSPACE
DELETE Any of these characters deletes the last character typed and appropriately erases it from the echo stream if it is a display stream.

Control-Q Erases all the type-in so far.

Control-R Reprints the accumulated string.

Control-V "Quotes" the next character: after typing Control-V, the next character typed is added to the accumulated string, regardless of any special meaning it has. Allows the user to include editing characters and word terminator characters in the accumulated string.

Control-W Erases the last word.

? Calls up a "help" facility. The action taken is defined by the GENERATE?LIST.FN argument to PROMPTFORWORD (see below). Normally,

this prints a list of possible candidates.

(**PROMPTFORWORD** *PROMPT.STR CANDIDATE.STR GENERATE?LIST.FN ECHO.CHANNEL DONTECHOTYPEIN.FLG URGENCY.OPTION TERMINCHARS.LST KEYBD.CHANNEL*) [Function]

PROMPTFORWORD has a multiplicity of features, which are specified through a rather large number of input arguments, but the default settings for them (i.e., when they aren't given, or are given as NIL) is such to minimize the number needed in the average case, and an attempt has been made to order the more frequently non-defaulted arguments at the beginning of the argument list. The default input and echo are both to the terminal; the terminal table in effect during input allows most control characters to be INDICATE'd.

PROMPTFORWORD returns NIL if a null string is typed; this would occur when no candidate is given and only a terminator is typed, or when the candidate is erased and a terminator is typed with no other input still un-erased. In all other cases, PROMPTFORWORD returns a string.

PROMPTFORWORD is controlled through the following arguments:

PROMPT.STR If non-NIL, this is coerced to a string and used for prompting; an additional space is output after this string.

CANDIDATE.STR If non-NIL, this is coerced to a string and offered as initial contents of the input buffer.

GENERATE?LIST.FN If non-NIL, this is either a string to be printed out for help, or a function to be applied to *PROMPT.STR* and *CANDIDATE.STR* (after both have been coerced to strings), and which should return a list of potential candidates. The help string or list of potential candidates will then be printed on a separate line, the prompt will be restarted, and any type-in will be re-echoed.
Note: If *GENERATE?LIST.FN* is a function, its value list will be cached so that it will be run at most once per call to PROMPTFORWORD.

ECHO.CHANNEL Coerced to an output stream; NIL defaults to T, the "terminal output stream", normally (TTYDISPLAYSTREAM). To achieve echoing to the "current output stream", use (GETSTREAM NIL 'OUTPUT). If echo is to a display stream, it will have a flashing caret showing where the next input is to be echoed.

DONTECHOTYPEIN.FLG If T, there is no echoing of the input characters. If the value of *DONTECHOTYPEIN.FLG* is a single-character atom or string, that character is echoed instead of the actual input. For example, LOGIN prompts for a password with *DONTECHOTYPEIN.FLG* being "*".

URGENCY.OPTION If NIL, PROMPTFORWORD quietly wait for input, as READ does; if a number, this is the number of seconds to wait for the user to respond (if timeout is reached, then *CANDIDATE.WORD* is returned, regardless of any other type-in activity); if T, this means to wait forever, but periodically flash the window to alert the user; if TTY, then PROMPTFORWORD grabs the TTY immediately. When URGENCY.OPTION = TTY, the cursor is temporarily changed to a different shape to indicate the urgent nature of the request.

TERMINCHARS.LST This is list of "word terminator" character codes; it defaults to (CHARCODE (EOL ESCAPE LF SPACE TAB)). This may also be a single character code.

KEYBD.CHANNEL If non-NIL, this is coerced to a stream, and the input bytes are taken from that stream. NIL defaults to the keyboard input stream. Note that this is not the same as the terminal input stream T, which is a buffered keyboard input

stream, not suitable for use with PROMPTFORWORD.

Examples:

```
(PROMPTFORWORD
   "What is your FOO word?" 'Mumble
    (FUNCTION (LAMBDA () '(Grumble Bletch)))
   PROMPTWINDOW NIL 30)
```

This first prompts the user for input by printing the first argument as a prompt into PROMPTWINDOW; then the proffered default answer, Mumble, is printed out and the caret starts flashing just after it to indicate that the upcoming input will be echoed there. If the user fails to complete a word within 30 seconds, then the result will be the string Mumble.

```
(FRESHLINE T)
(LIST
   (PROMPTFORWORD
      (CONCAT "{"  HOST  "} Login:")
      (USERNAME NIL NIL T))
   (PROMPTFORWORD
      " (password)" NIL NIL NIL '*))
```

This first prompts in whatever window is currently (TTYDISPLAYSTREAM), and then takes in a username; the second call prompts with (password) and takes in another word (the password) without proffering a candidate, echoing the typed-in characters as "*".

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 specifies 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 flag indicating whether or not typeahead is to be permitted, a flag indicating whether the transaction is to be stored on the history list (see Chapter 13), a default set of options, and an (optional) input file/string.

(**ASKUSER** *WAIT DEFAULT MESS KEYLST TYPEAHEAD LISPXPRNTFLG 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 case, the character(s) from *DEFAULT* are processed exactly as though they had been typed, except that ASKUSER first 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* are described. *TYPEAHEAD* is T if the user is permitted to typeahead a response to ASKUSER. NIL means any typeahead should be cleared and saved. *LISPXPRNTFLG* determines whether or not the interaction is to be recorded on the history list. *FILE* can be either NIL (in which case it defaults to the terminal input stream, T) or a stream.

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 (see the Options section below).

(**MAKEKEYLST** *LST DEFAULTKEY LCASEFLG AUTOCOMPLETEFLG*) [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.

AUTOCOMPLETEFLG is used as the value of the *AUTOCOMPLETEFLG* option of the resulting key list.

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 options are explained below. If an option is specified in OPTIONS, the value of the option is the next element. Otherwise, if the option is specified in the OPTIONSLST 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 (unless the NOCASEFLG option is T, see the Options section below). 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.

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 carriage return 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, when ASKUSER is called with KEYLST = NIL, the following KEYLST is used as the default:

```
((Y "escr") (N "ocr"))
```

This KEYLST specifies that if (as soon as) the user types Y (or y), ASKUSER echoes with Y, prompts with escr, and returns Y as its value. Similarly, if the user types N, ASKUSER echoes the N, prompts with ocr, 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.

For a more complicated example, the following is the KEYLST used for the compiler questions:

```
((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 **cr** 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.

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
```

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 carriage return or a space. If the value of CONFIRMFLG is a list, 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 not 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.
NOECHOFLG	If non-NIL, characters that are matched (or automatically supplied as a result of typing $ (escape) 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.
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. For example: One of the entries on the KEYLST used by ADDTOFILES? is: (] "Nowherecr" NOECHOFLG T EXPLAINSTRING "] - nowhere, item is marked as a dummycr") When the user types], ASKUSER just prints Nowherecr, 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 only when the key is confirmed with a member of the value of PROMPTON.

COMPLETEON — When a confirming character is typed, the N characters that are automatically supplied, as specified in case (4), are echoed only when the key is confirmed with a member of the value of PROMPTON.

The PROMPTON and COMPLETEON options enable the user to construct a KEYLST 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 carriage return or space, in which case no further output is forthcoming, or with a $ (escape), in which case the rest of the characters in the command are echoed, followed by some prompting information. The following KEYLST would handle the TENEX COPY and CONNECT comands:

```
((COPY " (FILE LIST) "
       PROMPTON ($)
       COMPLETEON ($)
       CONFIRMFLG ($))

 (CONNECT " (TO DIRECTORY) "
       PROMPTON ($)
       COMPLETEON ($)
       CONFIRMFLG ($)))
```

AUTOCOMPLETEFLG — If the value of the AUTOCOMPLETEFLG option is not NIL, ASKUSER will automatically supply unambiguous characters whenever it can, i.e., ASKUSER acts as though $ (escape) 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 a carriage return, but can be reset, e.g. to a comma, for more linear output.

Operation

All input operations are executed with the terminal table in the variable ASKUSERTTBL, in which the

following is true:

- (CONTROL T) has been executed (see the Line-Buffering section of Chapter 30), so that ASKUSER can interact with the user after each character is typed

  (ECHOMODE NIL) has been executed (see the Terminal Control Functions section of Chapter 30), 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 delete, 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. Delete 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.

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 compiler questions 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.
2. All of its characters have been matched and a confirming character is typed.
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.
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 Scr 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 <escape> 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 CLISPIFTRANFLG), and the user types C followed by $, ASKUSER will supply the L, I, S, and P. The user can then type F followed by a carriage return 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 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 an $ 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.

Special Keys

`&` This 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.

<escape> This can be used as a key to match with the result of a single call to READ. For example, if the KEYLST were:

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

then if the user typed COP FOOcr, (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.

Escape Escape This can be used as a key to match with the result of a single call to READLINE.

A list 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 $$ (escape, escape) could be achieved simply by using (READLINE T) as a key.

`""` The empty string 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 ADDTOFILES? 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.

Note: For Escape, Escape Escape, 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.

Startup Protocol and Typeahead

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.

Therefore, when ASKUSER is first 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 buffers and rings the bell to warn the user to stop typing. The buffers 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 first 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 first 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 first 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 (see Chapter 13) calls ASKUSER when the number of undosaves are exceeded for an event with `MESS = (LIST NUMBER-UNDOSAVES "undosaves, continue saving")`. Thus, the user can type a response as soon as NUMBER-UNDOSAVES 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 specified 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 specified as acceptable inputs by KEYLST (see below) and they are the first 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, then prints MESS and goes into a wait loop as described above.

   If there is typeahead that contains unacceptable input, ASKUSER will assume that the typeahead was not intended for ASKUSER, and will restore the typeahead when it completes operation and returns.

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

TTYIN is an Interlisp function for reading input from the terminal. It features altmode completion, spelling correction, help facility, and fancy editing, and can also serve as a glorified free text input function. This document is divided into two major sections: how to use TTYIN from the user's point of view, and from the programmer's.

TTYIN exists in implementations for Interlisp-10 and Interlisp-D. The two are substantially compatible, but the capabilities of the two systems differ (Interlisp-D has a more powerful display and allows greater access to the system primitives needed to control it effectively; it also has a mouse, greatly reducing the need for keyboard-oriented editing commands). Descriptions of both are included in this document for completeness, but Interlisp-D users may find large sections irrelevant.

Entering Input With TTYIN

There are two major ways of using TTYIN: set LISPXREADFN to TTYIN, so the LISPX executive uses it to obtain input; and call TTYIN from within a program to gather text input. Mostly the same rules apply to both; places where it makes a difference are mentioned below.

The following characters may be used to edit your input, independent of what kind of terminal you are on. The more TTYIN knows about your terminal, of course, the nicer some of these will behave. Some functions are performed by one of several characters; any character that you happen to have assigned as an interrupt character will, of couse, not be read by TTYIN. There is a (somewhat inelegant) way of changing which characters perform which functions, described under TTYINREADMACROS later on.

Control-A
BACKSPACE
DELETE Deletes a character. At the start of the second or subsequent lines of your input, deletes the last character of the previous line.

Control-W Deletes a "word". Generally this means back to the last space or parenthesis.

Control-Q Deletes the current line, or if the current line is blank, deletes the previous line.

Control-R Refreshes the current line. Two in a row refreshes the whole buffer (when doing multi-line input).

ESCAPE Tries to complete the current word from the spelling list provided to TTYIN, if any. In the case of ambiguity, completes as far as is uniquely determined, or rings the bell. For LISPX input, the spelling list may be USERWORDS (see discussion of TTYINCOMPLETEFLG.

Interlisp-10 only: If no spelling list was provided, but the word begins with a "<", tries directory name completion (or filename completion if there is already a matching ">" in the current word).

? If typed in the middle of a word will supply alternative completions from the SPLST argument to TTYIN (if any). ?ACTIVATEFLG (see the Assorted Flags section below) must be true to enable this feature.

Control-Y Escapes to a Lisp user exec, from which you may return by the

command OK. However, when in READ mode and the buffer is non-empty, Control-Y is treated as Lisp's unquote macro instead, so you have to use meta-Control-Y (below) to invoke the user exec.

LF in Interlisp-10 — Retrieves characters from the previous non-empty buffer when it is able to; e.g., when typed at the beginning of the line this command restores the previous line you typed at TTYIN; when typed in the middle of a line fills in the remaining text from the old line; when typed following ^Q or ^W restores what those commands erased.

; — If typed as the first character of the line means the line is a comment; it is ignored, and TTYIN loops back for more input.

Note: The exact behaviour of this character is determined by the value of TTYINCOMMENTCHAR (see the Assorted Flags section below).

Control-X — Goes to the end of your input (or end of expression if there is an excess right parenthesis) and returns if parentheses are balanced, beeps if not.

During most kinds of input, TTYIN is in "autofill" mode: if a space is typed near the right margin, a carriage return is simulated to start a new line. In fact, on cursor-addressable displays, lines are always broken, if possible, so that no word straddles the end of the line. The "pseudo-carriage return" ending the line is still read as a space, however; i.e., the program keeps track of whether a line ends in a carriage return or is merely broken at some convenient point. You won't get carriage returns in your strings unless you explicitly type them.

Mouse Commands

The mouse buttons are interpreted as follows during TTYIN input:

LEFT — Moves the caret to where the cursor is pointing. As you hold down LEFT, the caret moves around with the cursor; after you let up, any typein will be inserted at the new position.

MIDDLE — Like LEFT, but moves only to word boundaries.

RIGHT — Deletes text from the caret to the cursor, either forward or backward. While you hold down RIGHT, the text to be deleted is complemented; when you let up, the text actually goes away. If you let up outside the scope of the text, nothing is killed (this is how to "cancel" the command). This is roughly the same as CTRL-RIGHT with no initial selection (below).

If you hold down CTRL and/or SHIFT while pressing the mouse buttons, you instead get secondary selection, move selection or delete selection. You make a selection by bugging LEFT (to select a character) or MIDDLE (to select a word), and optionally extend the selection either left or right using RIGHT. While you are doing this, the caret does not move, but your selected text is highlighted in a manner indicating what is about to happen. When you have made your selection (all mouse buttons up now), lift up on CTRL and/or SHIFT and the action you have selected will occur, which is:

SHIFT — The selected text as typein at the caret. The text is highlighted with a broken underline during selection.

CTRL — Delete the selected text. The text is complemented during selection.

CTRL-SHIFT — Combines the above: delete the selected text and insert it at the caret.

This is how you move text about.

You can cancel a selection in progress by pressing LEFT or MIDDLE as if to select, and moving outside the range of the text.

The most recent text deleted by mouse command can be inserted at the caret by typing Middle-blank key (on the Xerox 1132) or the Open key (on the Xerox 1108). This is the same key that retrieves the previous buffer when issued at the end of a line.

Display Editing Commands

On terminals with a meta key: In Interlisp-10, TTYIN reads from the terminal in binary mode, allowing many more editing commands via the meta key, in the style of TVEDIT commands. Note that due to Tenex's unfortunate way of handling typeahead, it is not possible to type ahead edit commands before TTYIN has started (i.e., before its prompt appears), because the meta bit will be thrown away. Also, since Escape has numerous other meanings in Lisp and even in TTYIN (for completion), this is not used as a substitute for the meta key.

In Interlisp-D: Users will probably have little use for most of these commands, as cursor positioning can often be done more conveniently, and certainly more obviously, with the mouse. Nevertheless, some commands, such as the case changing commands, can be useful. The <bottom-blank> key can be used as an meta key if you perform (METASHIFT T) (see Chapter 30). Alternatively, you can use the variable EDITPREFIXCHAR as described in the next paragraph.

On display terminals without a meta key: If you want to type any of these commands, you need to prefix them with the "edit prefix" character. Set the variable EDITPREFIXCHAR to the character code of the desired prefix char. Type the edit prefix twice to give an "meta-escape" command. Some users of the TENEX TVEDIT program like to make escape (33Q) be the edit prefix, but this makes it somewhat awkward to ever use escape completion. EDITPREFIXCHAR is initially NIL.

On hardcopy terminals without a meta key: You probably want to ignore this section, since you won't be able to see what's going on when you issure edit commands; there is no attempt made to echo anything reasonable.

In the descriptions below, "current word" means the word the cursor is under, or if under a space, the previous word. Currently parentheses are treated as spaces, which is usually what you want, but can occasionally cause confusion in the word deletion commands. Most commands can be preceded by numbers or escape (means infinity), only the first of which requires the meta key (or the edit prefix). Some commands also accept negative arguments, but some only look at the magnitude of the arg. Most of these commands are taken from the display editors TVEDIT and/or E, and are confined to work within one line of text unless otherwise noted.

Cursor Movement Commands:

■ **Meta-DELETE**
Meta-BS

Meta-< Back up one (or n) characters.

Meta-SPACE
Meta-> Moves forward one (or n) characters.

Meta-↑ Moves up one (or n) lines.

Meta-lf Moves down one (or n) lines.

Meta-(Moves back one (or n) words.

Meta-) Moves ahead one (or n) words.

Meta-TAB Moves to end of line; with an argument moves to nth end of line; **Meta-ESC-TAB** goes to end of buffer.

Control-Meta-L Moves to start of line (or nth previous, or start of buffer).

Meta-{
Meta-} Go to start and end of buffer, respectively.

Meta-[Moves to beginning of the current list, where cursor is currently under an element of that list or its closing paren. (See also the auto-parenthesis-matching feature below under "Flags".)

Meta-] Moves to end of current list.

Meta-Sx Skips ahead to next (or nth) occurrence of character x, or rings the bell.

Meta-Bx Backward search.

Buffer Modification Commands:

Meta-Zx Zaps characters from cursor to next (or nth) occurrence of x. There is no unzap command yet.

Meta-A
Meta-R Repeat the last S, B or Z command, regardless of any intervening input (note this differs from TEdit's A command).

Meta-K Kills the character under the cursor, or n chars starting at the cursor.

Meta-CR When the buffer is empty is the same as **LF**, i.e. restores buffer's previous contents. Otherwise is just like a **CR** (except that it also terminates an insert). Thus, **Meta-CR CR** will repeat the previous input (as will **LF CR** without the meta key).

Meta-O Does "Open line", inserting a crlf after the cursor, i.e., it breaks the line but leaves the cursor where it is.

Meta-T Transposes the characters before and after the cursor. When typed at the end of a line, transposes the previous two characters. Refuses to handle funny cases, such as tabs.

Meta-G Grabs the contents of the previous line from the cursor position onward. Meta-nG grabs the nth previous line.

Meta-L Lowercases current word, or n words on line. **Meta-ESC-L** lowercases the rest of the line, or if given at the end of line lowercases the entire line.

Meta-U Uppercases analogously.

Meta-C Capitalize. If you give it an argument, only the first word is capitalized; the rest are just lowercased.

Control-Meta-Q Deletes the current line. **Control-Meta-ESC-Q** deletes from the current cursor position to the end of the buffer. No other arguments are handled.

Control-Meta-W Deletes the current word, or the previous word if sitting on a space.

Meta-J "Justify" this line. This will break it if it is too long, or move words up from the next line if too short. Will not join to an empty line, or one starting with a tab (both of which are interpreted as paragraph breaks). Any new line breaks it introduces are considered spaces, not carriage returns. **Meta-nJ** justifies n lines.

The linelength is defined as TTYJUSTLENGTH, ignoring any prompt characters at the margin. If TTYJUSTLENGTH is negative, it is interpreted as relative to the right margin. TTYJUSTLENGTH is initially -8 in Interlisp-D, 72 in Interlisp-10.

Meta-ESC-F "Finishes" the input, regardless of where the cursor is. Specifically, it goes to the end of the input and enters a **CR**, **control-Z** or "", depending on whether normal, REPEAT or READ input is happening. Note that a "" won't necessarily end a READ, but it seems likely to in most cases where you would be inclined to use this command, and makes for more predictable behavior.

Miscellaneous Commands:

Meta-P Interlisp-D: Prettyprint buffer. Clears the buffer and reprints it using prettyprint. If there are not enough right parentheses, it will supply more; if there are too many, any excess remains unprettyprinted at the end of the buffer. May refuse to do anything if there is an unclosed string or other error trying to read the buffer.

Meta-N Refresh line. Same as **Control-R**. Meta-ESC-N refreshes the whole buffer; Meta-nN refreshes n lines. Cursor movement in TTYIN depends on TTYIN being the only source of output to the screen; if you do a **Control-T**, or a system message appears, or line noise occurs, you may need to refresh the line for best results. In Interlisp-10, if for some reason your terminal falls out of binary mode (e.g. can happen when returning to a Lisp running in a lower fork), **Meta-<**anything**>** is unreadable, so you'd have to type **Control-R** instead.

Control-Meta-Y Gets user exec. Thus, this is like regular **Control-Y**, except when doing a READ (when control-Y is a read macro and hence does not invoke this function).

Control-Meta-ESC-Y Gets a user exec, but first unreads the contents of the buffer from the cursor onward. Thus if you typed at TTYIN something destined for the Lisp executive, you can do **Control-Meta-L-ESC-Control-Y** and give it to Lisp.

Meta-_ Adds the current word to the spelling list USERWORDS. With zero arg, removes word. See TTYINCOMPLETEFLG (see the Assorted Flags section below).

Note to Datamedia, Heath users: In addition to simple cursor movement commands and insert/delete, TTYIN uses the display's cursor-addressing capability to optimize cursor movements

longer than a few characters, e.g. **Meta-TAB** to go to the end of the line. In order to be able to address the cursor, TTYIN has to know where it is to begin with. Lisp keeps track of the current print position within the line, but does not keep track of the line on the screen (in fact, it knows precious little about displays, much like Tenex). Thus, TTYIN establishes where it is by forcing the cursor to appear on the last line of the screen. Ordinarily this is the case anyway (except possibly on startup), but if the cursor happens to be only halfway down the screen at the time, there is a possibly unsettling leap of the cursor when TTYIN starts.

Using TTYIN for Lisp Input

When TTYIN is loaded, or a sysout containing TTYIN is started up, the function SETREADFN is called. If the terminal is a display, it sets LISPXREADFN (see Chapter 13) to be TTYINREAD. If the terminal is not a display terminal, SETREADFN will set the variable to READ. (SETREADFN 'READ) will also set it to READ.

There are two principal differences between TTYINREAD and READ: (1) parenthesis balancing. The input does not activate on an exactly balancing right paren/bracket unless the input started with a paren/bracket, e.g., USE (FOO) FOR (FIE) will all be on one line, terminated by **CR**; and (2) read macros.

In Interlisp-10, TTYIN does not use a read table (TTYIN behaves as though using the default initial Lisp terminal input readtable), so read macros and redefinition of syntax characters are not supported; however, "' " (QUOTE) and "Control-Y" (EVAL) are built in, and a simple implementation of ? and ?= is supplied. Also, the TTYINREADMACROS facility described below can supply some of the functionality of immediate read macros in the editor.

In Interlisp-D, read macros are (mostly) supported. Immediate read macros take effect only if typed at the end of the input (it's not clear what their semantics should be elsewhere).

Useful Macros

There are two useful edit macros that allow you to use TTYIN as a character editor: (1) ED loads the current expression into the ttyin buffer to be edited (this is good for editing comments and strings). Input is terminated in the usual way (by typing a balancing right programmer's assistant command FIX will load the buffer with the event's input, rather than calling the editor. If you really wanted the Interlisp editor for your fix, you can say FIX EVENT - TTY: once you got TTYIN's version to force you into the editor.

Programming With TTYIN

(**TTYIN** PROMPT SPLST HELP OPTIONS ECHOTOFILE TABS UNREADBUF RDTBL) [Function]

TTYIN prints PROMPT, then waits for input. The value returned in the normal case is a list of all atoms on the line, with comma and parens returned as individual atoms; OPTIONS may be used to get a different kind of value back.

PROMPT is an atom or string (anything else is converted to a string). If NIL, the value of DEFAULTPROMPT, initially "** ", will be used. If PROMPT is T, no prompt will be given. PROMPT may also be a dotted pair (PROMPT1 . PROMPT2), giving the prompt for the first and subsequent (or overflow) lines, each prompt being a string/atom or NIL to denote absence of prompt. The default prompt for overflow lines is "...". Note that rebinding DEFAULTPROMPT gives a convenient way to affect all the "ordinary" prompts in some program module.

SPLST is a spelling list, i.e., a list of atoms or dotted pairs (SYNONYM . ROOT). If supplied, it is used to check and correct user responses, and to provide completion if the user types escape. If SPLST is one of the Lisp system spelling lists (e.g., USERWORDS or SPELLINGS3), words that are escape-completed get moved to the front, just as if a FIXSPELL had found them. Autocompletion is also performed when user types a break character (cr, space, paren, etc), unless one of the "nofixspell" options below is selected; i.e., if the word just typed would uniquely complete by escape, TTYIN behaves as though escape had been typed.

HELP, if non-NIL, determines what happens when the user types ? or HELP. If HELP = T, program prints back SPLST in suitable form. If HELP is any other litatom, or a string containing no spaces, it performs (DISPLAYHELP HELP). Anything else is printed as is. If HELP is NIL, ? and HELP are treated as any other atoms the user types. [DISPLAYHELP is a user-supplied function, initially a noop; systems with a suitable HASH package, for example, have defined it to display a piece of text from a hashfile associated with the key HELP.]

OPTIONS is an atom or list of atoms chosen from among the following:

NOFIXSPELL — Uses SPLST for HELP and Escape completion, but does not attempt any FIXSPELLing. Mainly useful if SPLST is incomplete and the caller wants to handle corrections in a more flexible way than a straight FIXSPELL.

MUSTAPPROVE — Does spelling correction, but requires confirmation.

CRCOMPLETE — Requires confirmation on spelling correction, but also does autocompletion on <cr> (i.e. if what user has typed so far uniquely identifies a member of SPLST, completes it). This allows you to have the benefits of autocompletion and still allow new words to be typed.

DIRECTORY — (only if SPLST = NIL) Interprets Escape to mean directory name completion [Interlisp-10 only].

USER — Like DIRECTORY, but does username completion. This is identical to DIRECTORY under Tenex [Interlisp-10 only].

FILE — (only if SPLST = NIL) Interprets Escape to mean filename completion [Sumex and Tops20 only].

FIX — If response is not on, or does not correct to, SPLST, interacts with user until an acceptable response is entered. A blank line (returning NIL) is always accepted. Note that if you are willing to accept responses that are not on SPLST, you probably should specify one of the options NOXFISPELL, MUSTAPPROVE or CRCOMPLETE, lest the user's new response get FIXSPELLed

away without their approval.

STRING Line is read as a string, rather than list of atoms. Good for free text.

NORAISE Does not convert lower case letters to upper case.

NOVALUE For use principally with the ECHOTOFILE arg (below). Does not compute a value, but returns T if user typed anything, NIL if just a blank line.

REPEAT For multi-line input. Repeatedly prompts until user types Control-Z (as in Tenex sndmsg). Returns one long list; with STRING option returns a single string of everything typed, with carriage returns (EOL) included in the string.

TEXT Implies REPEAT, NORAISE, and NOVALUE. Additionally, input may be terminated with Control-V, in which case the global flag CTRLVFLG will be set true (it is set to NIL on any other termination). This flag may be utilized in any way the caller desires.

COMMAND Only the first word on the line is treated as belonging to SPLST, the remainder of the line being arbitrary text; i.e., "command format". If other options are supplied, COMMAND still applies to the first word typed. Basically, it always returns (CMD . REST-OF-INPUT), where REST-OF-INPUT is whatever the other options dictate for the remainder. E.g. COMMAND NOVALUE returns (CMD) or (CMD . T), depending on whether there was further input; COMMAND STRING returns (CMD . "REST-OF-INPUT"). When used with REPEAT, COMMAND is only in effect for the first line typed; furthermore, if the first line consists solely of a command, the REPEAT is ignored, i.e., the entire input is taken to be just the command.

READ Parens, brackets, and quotes are treated a la READ, rather than being returned as individual atoms. Control characters may be input via the Control-Vx notation. Input is terminated roughly along the lines of READ conventions: a balancing or over-balancing right paren/bracket will activate the input, or <cr> when no parenthesis remains unbalanced. READ overrides all other options (except NORAISE).

LISPXREAD Like READ, but implies that TTYIN should behave even more like READ, i.e., do NORAISE, not be errorset-protected, etc.

NOPROMPT Interlisp-D only: The prompt argument is treated as usual, except that TTYIN assumes that the prompt for the first line has already been printed by the caller; the prompt for the first line is thus used only when redisplaying the line.

ECHOTOFILE if specified, user's input is copied to this file, i.e., TTYIN can be used as a simple text-to-file routine if NOVALUE is used. If ECHOTOFILE is a list, copies to all files in the list. PROMPT is not included on the file.

TABS is a special addition for tabular input. It is a list of tabstops (numbers). When user types a tab, TTYIN automatically spaces over to the next tabstop (thus the first tabstop is actually the second "column" of input). Also treats specially the characters * and "; they echo normally, and then

automatically tab over.

UNREADBUF allows the caller to "preload" the TTYIN buffer with a line of input. UNREADBUF is a list, the elements of which are unread into the buffer (i.e., "the outer parentheses are stripped off") to be edited further as desired; a simple carriage return (or Control-Z for REPEAT input) will thus cause the buffer's contents to be returned unchanged. If doing READ input, the "PRIN2 names" of the input list are used, i.e., quotes and %'s will appear as needed; otherwise the buffer will look as though UNREADBUF had been PRIN1'ed. UNREADBUF is treated somewhat like READBUF, so that if it contains a pseudo-carriage return (the value of HISTSTR0), the input line terminates there.

Input can also be unread from a file, using the HISTSTR1 format: UNREADBUF = (<value of HISTSTR1> (FILE START . END)), where START and END are file byte pointers. This makes TTYIN a miniature text file editor.

RDTBL [Interlisp-D only] is the read table to use for READing the input when one of the READ options is given. A lot of character interpretations are hardwired into TTYIN, so currently the only effect this has is in the actual READ, and in deciding whether a character typed at the end of the input is an immediate read macro, for purposes of termination.

If the global variable TYPEAHEADFLG is T, or option LISPXREAD is given, TTYIN permits type-ahead; otherwise it clears the buffer before prompting the user.

Using TTYIN as a General Editor

The following may be useful as a way of outsiders to call TTYIN as an editor. These functions are currently only in Interlisp-D.

(**TTYINEDIT** *EXPRS WINDOW PRINTFN PROMPT*) [Function]

This is the body of the edit macro EE. Switches the tty to *WINDOW*, clears it, prettyprints *EXPRS*, a list of expressions, into it, and leaves you in TTYIN to edit it as Lisp input. Returns a new list of expressions.

If *PRINTFN* is non-NIL, it is a function of two arguments, EXPRS and FILE, which is called instead of PRETTYPRINT to print the expressions to the window (actually to a scratch file). Note that EXPRS is a list, so normally the outer parentheses should not be printed. *PRINTFN* = T is shorthand for "unpretty"; use PRIN2 instead of PRETTYPRINT.

PROMPT determines what prompt is printed, if any. If T, no prompt is printed. If NIL, it defaults to the value of TTYINEDITPROMPT.

TTYINAUTOCLOSEFLG [Variable]

If TTYINAUTOCLOSEFLG is true, TTYINEDIT closes the window on exit.

TTYINEDITWINDOW [Variable]

If the WINDOW arg to TTYINEDIT is NIL, it uses the value of TTYINEDITWINDOW, creating

it if it does not yet exist.

TTYINPRINTFN [Variable]

The default value for PRINTFN in EE's call to TTYINEDIT.

(**SET.TTYINEDIT.WINDOW** *WINDOW*) [Function]

Called under a RESETLST. Switches the tty to *WINDOW* (defaulted as in TTYINEDIT) and clears it. The window's position is left so that TTYIN will be happy with it if you now call TTYIN yourself. Specifically, this means positioning an integral number of lines from the bottom of the window, the way the top-level tty window normally is.

(**TTYIN.SCRATCHFILE**) [Function]

Returns, possibly creating, the scratchfile that TTYIN uses for prettyprinting its input. The file pointer is set to zero. Since TTYIN does use this file, beware of multiple simultaneous use of the file.

?= Handler

In Interlisp, the ?= read macro displays the arguments to the function currently "in progress" in the typein. Since TTYIN wants you to be able to continue editing the buffer after a ?=, it processes this macro specially on its own, printing the arguments below your typein and then putting the cursor back where it was when ?= was typed. For users who want special treatment of ?=, the following hook exists:

TTYIN?=FN [Variable]

The value of this variable, if non-NIL, is a user function of one argument that is called when ?= is typed. The argument is the function that ?= thinks it is inside of. The user function should return one of the following:

NIL Normal ?= processing is performed.

T Nothing is done. Presumably the user function has done something privately, perhaps diddled some other window, or called TTYIN.PRINTARGS (below).

a list (ARGS . STUFF) Treats STUFF as the argument list of the function in question, and performs the normal ?= processing using it.

anything else The value is printed in lieu of what ?= normally prints.

At the time that ?= is typed, nothing has been "read" yet, so you don't have the normal context you might expect inside a conventional readmacro. If the user function wants to examine the typed-in arguments being passed to the fn, however, it can call the function TTYIN.READ?=ARGS:

(**TTYIN.READ?=ARGS**) [Function]

When called inside TTYIN?=FN user function, returns everything between the function and the typing of ?= as a list (like an arglist). Returns NIL if ?= was typed immediately

after the function name.

(**TTYIN.PRINTARGS** *FN ARGS ACTUALS ARGTYPE*) [Function]

Does the function/argument printing for ?=. *ARGS* is an argument list, *ACTUALS* is a list of actual parameters (from the typein) to match up with args. *ARGTYPE* is a value of the function *ARGTYPE*; it defaults to (*ARGTYPE FN*).

Read Macros

When doing READ input in Interlisp-10, no Lisp-style read macros are available (but the ' and control-Y macros are built in). Principally because of the usefulness of the editor read macros (set by SETTERMCHARS), and the desire for a way of changing the meanings of the display editing commands, the following exists as a hack:

TTYINREADMACROS [Variable]

Value is a set of shorthand inputs useable during READ input. It is an alist of entries (CHARCODE . SYNONYM). If the user types the indicated character (the meta bit is denoted by the 200Q bit in the char code), TTYIN behaves as though the synonym character had been typed.

Special cases: 0 - the character is ignored; 200Q - pure meta bit; means to read another char and turn on its meta bit; 400Q - macro quote: read another char and use its original meaning. For example, if you have macros ((33Q . 200Q) (30Q . 33Q)), then Escape (33Q) will behave as an edit prefix, and control-X (30Q) will behave like Escape. Note: currently, synonyms for meta commands are not well-supported, working only when the command is typed with no argument.

Slightly more powerful macros also can be supplied; they are recognized when a character is typed on an empty line, i.e., as the first thing after the prompt. In this case, the TTYINREADMACROS entry is of the form (CHARCODE T . RESPONSE) or (CHARCODE CONDITION . RESPONSE), where CONDITION is a list that evaluates true. If RESPONSE is a list, it is EVALed; otherwise it is left unevaluated. The result of this evaluation (or RESPONSE itself) is treated as follows:

NIL — The macro is ignored and the character reads normally, i.e., as though TTYINREADMACROS had never existed.

An integer — A character code, treated as above. Special case: -1 is treated like 0, but says that the display may have been altered in the evaluation of the macro, so TTYIN should reset itself appropriately.

Anything else — This TTYIN input is terminated (with a crlf) and returns the value of "response" (turned into a list if necessary). This is the principal use of this facility. The macro character thus stands for the (possibly computed) reponse, terminated if necessary with a crlf. The original character is not echoed.

Interrupt characters, of course, cannot be read macros, as TTYIN never sees them, but any other characters, even non-control chars, are allowed. The ability to return NIL allows you to have conditional macros that only apply in specified situations (e.g., the macro

might check the prompt (LISPXID) or other contextual variables). To use this specifically to do immediate editor read macros, do the following for each edit command and character you want to invoke it with:

```
(ADDTOVAR TTYINREADMACROS (CHARCODE 'CHARMACRO?
EDITCOM))
```

For example, (ADDTOVAR TTYINREADMACROS (12Q CHARMACRO? !NX)) will make linefeed do the !NX command. Note that this will only activate linefeed at the beginning of a line, not anywhere in the line. There will probably be a user function to do this in the next release.

Note that putting (12Q T . !NX) on TTYINREADMACROS would also have the effect of returning !NX from the READ call so that the editor would do an !NX. However, TTYIN would also return !NX outside the editor (probably resulting in a u.b.a. error, or convincing DWIM to enter the editor), and also the clearing of the output buffer (performed by CHARMACRO?) would not happen.

Assorted Flags

These flags control aspects of TTYIN's behavior. Some have already been mentioned. In Interlisp-D, the flags are all initially set to T.

TYPEAHEADFLG [Variable]

If true, TTYIN always permits typeahead; otherwise it clears the buffer for any but LISPXREAD input.

?**ACTIVATEFLG** [Variable]

If true, enables the feature whereby ? lists alternative completions from the current spelling list.

SHOWPARENFLG [Variable]

If true, then whenever you are typing Lisp input and type a right parenthesis/bracket, TTYIN will briefly move the cursor to the matching parenthesis/bracket, assuming it is still on the screen. The cursor stays there for about 1 second, or until you type another character (i.e., if you type fast you'll never notice it). This feature was inspired by a similar EMACS feature, and turned out to be pretty easy to implement.

TTYINBSFLG [Variable]

Causes TTYIN to always physically backspace, even if you're running on a non-display (not a DM or Heath), rather than print \deletedtext\ (this assumes your hardcopy terminal or glass tty is capable of backspacing). If TTYINBSFLG is LF, then in addition to backspacing, TTYIN x's out the deleted characters as it backs up, and when you stop deleting, it outputs a linefeed to drop to a new, clean line before resuming. To save paper, this linefeed operation is not done when only a single character is deleted, on the grounds that you can probably figure out what you typed anyway.

TTYINRESPONSES [Variable]

An association list of special responses that will be handled by routines designated by the programmer. See "Special Responses", below.

TTYINERRORSETFLG [Variable]

[Interlisp-D only] If true, non-LISPXREAD inputs are errorset-protected (Control-E traps back to the prompt), otherwise errors propagate upwards. Initially NIL.

TTYINCOMMENTCHAR [Variable]

This variable affects the treatment of lines beginning with the comment character (usually ";"). If TTYINCOMMENTCHAR is a character code, and the first character on a line of typein is equal to TTYINCOMMENTCHAR, then the line is erased from the screen and no input function will see it. If TTYINCOMMENTCHAR is NIL, this feature is disabled. TTYINCOMMENTCHAR is initially NIL.

TTYINCOMPLETEFLG [Variable]

If true, enables Escape completion from USERWORDS during READ inputs. Details below.

USERWORDS (see Chapter 20) contains words you mentioned recently: functions you have defined or edited, variables you have set or evaluated at the executive level, etc. This happens to be a very convenient list for context-free escape completion; if you have recently edited a function, chances are good you may want to edit it again (typing "EF xx$") or type a call to it. If there is no completion for the current word from USERWORDS, the escape echoes as "$", i.e. nothing special happens; if there is more than one possible completion, you get beeped. If typed when not inside a word, Escape completes to the value of LASTWORD, i.e., the last thing you typed that the p.a. "noticed" (setting TTYINCOMPLETEFLG to 0 disables this latter feature), except that Escape at the beginning of the line is left alone (it is a p.a. command).

If you really wanted to enter an escape, you can, of course, just quote it with a control-V, like you can other control chars.

You may explicitly add words to USERWORDS yourself that wouldn't get there otherwise. To make this convenient online the edit command [_] means "add the current atom to USERWORDS" (you might think of the command as "pointing out this atom"). For example, you might be entering a function definition and want to "point to" one or more of its arguments or prog variables. Giving an argument of zero to this command will instead remove the indicated atom from USERWORDS.

Note that this feature loses some of its value if the spelling list is too long, for then the completion takes too long computationally and, more important, there are too many alternative completions for you to get by with typing a few characters followed by escape. Lisp's maintenance of the spelling list USERWORDS keeps the "temporary" section (which is where everything goes initially unless you say otherwise) limited to #USERWORDS atoms, initially 100. Words fall off the end if they haven't been used (they are "used" if FIXSPELL corrects to one, or you use <escape> to complete one).

Special Responses

There is a facility for handling "special responses" during any non-READ TTYIN input. This action is independent of the particular call to TTYIN, and exists to allow you to effectively "advise" TTYIN to intercept certain commands. After the command is processed, control returns to the original TTYIN call. The facility is implemented via the list TTYINRESPONSES.

TTYINRESPONSES [Variable]

TTYINRESPONSES is a list of elements, each of the form:

```
(COMMANDS RESPONSE-FORM OPTION)
```

COMMANDS is a single atom or list of commands to be recognized; RESPONSE-FORM is EVALed (if a list), or APPLYed (if an atom) to the command and the rest of the line. Within this form one can reference the free variables COMMAND (the command the user typed) and LINE (the rest of the line). If OPTION is the atom LINE, this means to pass the rest of line as a list; if it is STRING, this means to pass it as a string; otherwise, the command is only valid if there is nothing else on the line. If RESPONSE-FORM returns the atom IGNORE, it is not treated as a special response (i.e. the input is returned normally as the result of TTYIN).

Suggested use: global commands or options can be added to the toplevel value of TTYINRESPONSES. For more specialized commands, rebind TTYINRESPONSES to (APPEND NEWENTRIES TTYINRESPONSES) inside any module where you want to do this sort of special processing.

Special responses are not checked for during READ-style input.

Display Types

[This is not relevant in Interlisp-D]

TTYIN determines the type of display by calling DISPLAYTERMP, which is initially defined to test the value of the GTTYP jsys. It returns either NIL (for printing terminals) or a small number giving TTYIN's internal code for the terminal type. The types TTYIN currently knows about:

0 = glass tty (capable of deleting chars by backspacing, but little else)

1 = Datamedia

2 = Heath

Only the Datamedia has full editing power. DISPLAYTERMP has built into it the correct terminal types for Sumex and Stanford campus 20's: Datamedia = 11 on tenex, 5 on tops20; Heath = 18 on Tenex, 25 on tops20. You can override those values by setting the variable DISPLAYTYPES to be an association list associating the GTTYP value with one of these internal codes. For example, Sumex displays correspond to DISPLAYTYPES = ((11 . 1) (18 . 2)) [although this is actually compiled into DISPLAYTERMP for speed]. Any display terminal other than Datamedia and Heath can

probably safely be assigned to "0" for glass tty.

To add new terminal types, you have to choose a number for it, add new code to TTYIN for it and recompile. The TTYIN code specifies what the capabilities of the terminal are, and how to do the primitive operations: up, down, left, right, address cursor, erase screen, erase to end of line, insert character, etc.

For terminals lacking a meta key (currently only Datamedias have it), set the variable EDITPREFIXCHAR to the ascii code of an edit "prefix" (i.e., anything typed preceded by the prefix is considered to have the meta bit on). If your EDITPREFIXCHAR is 33Q (Escape), you can type a real Escape by typing 3 of them (2 won't do, since that means "Meta-Escape", a legitimate argument to another command). You could also define an Escape synonym with TTYINREADMACROS if you wanted (but currently it doesn't work in filename completion). Setting EDITPREFIXCHAR for a terminal that is not equipped to handle the full range of editing functions (only the Heath and Datamedia are currently so equipped) is not guaranteed to work, i.e. the display will not always be up to date; but if you can keep track of what you're doing, together with an occasional control-R to help out, go right ahead.

Prettyprint

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

(**PRETTYPRINT** *FNS PRETTYDEFLG —*) [Function]

FNS is a list of functions. If *FNS* is atomic, its value is used). The definitions of the functions are printed in a pretty format on the primary output file 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])
```

PRETTYDEFLG 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 file: 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 files noticed by the file 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). Note that PRETTYPRINT will return (FN NOT PRINTABLE) if FN does not have an accessable expr definition, or if it doesn't have any definition at all.

(**PP** *FN$_1$ ... FN$_N$*) [NLambda NoSpread Function]

For prettyprinting functions to the terminal. PP calls PRETTYPRINT with the primary output file set to T and the primary read table set to T. The primary output file 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 file known to the file 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 file 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 file).

(**PF** *FN FROMFILES TOFILE*) [NLambda NoSpread Function]

Copies the definition of *FN* found on each of the files in *FROMFILES* to *TOFILE*. If *TOFILE* = NIL, defaults to T. If *FROMFILES* = NIL, defaults to (WHEREIS FN NIL T) (see Chapter 17). The typical usage of PF is simply to type "PF FN".

PF prints a message if it can't find a file on *FROMFILES*, or it can't find the function FN on a file.

When printing to the terminal, PF performs several transformations on the characters in the file that comprise the definition for FN:

1. Font information is stripped out (except in Interlisp-D, whose display supports multiple fonts)
2. Occurrences of the CHANGECHAR (see the Special Prettyprint Controls section below) 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 (unless PFDEFAULT is T; initially NIL)
4. Comments are elided, if **COMMENT**FLG is non-NIL (see the Comment Feature section below).

(**SEE** *FROMFILE TOFILE*) [NLambda NoSpread Function]

Copies all of the text from *FROMFILE* to *TOFILE* (defaults to T), processing all text as PF does. Used to display the contents of files on the terminal.

(**PP*** *X*) [NLambda NoSpread Function]
(**PF*** *FN FROMFILES TOFILE*) [NLambda NoSpread Function]
(**SEE*** *FROMFILE TOFILE*) [NLambda NoSpread Function]

These functions operate exactly like PP, PF, and SEE, except that they bind **COMMENT**FLG to NIL, so comments are printed in full.

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). PRINTDEF initially performs (TAB LEFT T), which means to space to position *LEFT*, unless already beyond this position, in which case it does nothing.

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 for printing with multiple fonts (see Chapter 27). 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.

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))
```

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

COMMENTFLG is printed. Initially " **COMMENT** ".

(**COMMENT1** *L* —) [Function]

Prints the comment *L*. COMMENT1 is a separate function to permit the user to write prettyprint macros 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)      (COMMEN
T1 X)))
```

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.

Comment Pointers

For a well-commented collection of programs, the list structure, atom, and print name storage required to represent the comments in core can be significant. If the comments already appear on a file and are not needed for editing, a significant savings in storage can be achieved by simply leaving the text of the comment on the file when the file is loaded, and instead retaining in core only a pointer to the comment. When this feature is enabled, * is defined as a read macro (see Chapter 25) in FILERDTBL which, instead of reading in the entire text of the comment, constructs an expression containing

- The name of the file in which the text of the comment is contained
- The address of the first character of the comment
- The number of characters in the comment
- A flag indicating whether the comment appeared at the right hand margin or centered on the page

For output purposes, * is defined on PRETTYPRINTMACROS (see the Prettyprint Control Functions section below) so that it prints the comments represented by such pointers by simply copying the corresponding characters from one file 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 function NORMALCOMMENTS.

(**NORMALCOMMENTS** *FLG*) [Function]

If *FLG* is NIL, the comment pointer feature is enabled. If *FLG* is T, the comment pointer feature is disabled (the default).

NORMALCOMMENTS can be changed as often as desired. Thus, some files 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 file. Returns *X* in all cases. If *DESTFL* is non-NIL, it is the name of an open file, to which GETCOMMENT copies the comment; in this case, *X* remains a comment pointer, but it has been changed to point to the new file (unless NORMALCOMMENTS has been set to DONTUPDATE).

(**PRINTCOMMENT** *X*) [Function]

Defined as the prettyprint macro for *: copies the comment to the primary output file by using GETCOMMENT.

(**READCOMMENT** *FL RDTBL LST*) [Function]

Defined as the read macro for * in FILERDTBL: if NORMALCOMMENTSFLG is NIL, it constructs a comment pointer, unless it believes the expression beginning with * is not actually a comment, e.g., if the next atom is "." or E.

Note that a certain amount of care is required in using the comment pointer feature. Since the text of the comment resides on the file pointed to by the comment pointer, that file 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 file that the comment lives on. However, if the user has been performing repeated MAKEFILE's (see Chapter 17) 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 file, 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 file. This can be a problem for file systems that have a built-in limit on the number of versions of a given file 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 infinite. GETCOMMENT prints an error message if the file that the comment pointer points at has disappeared.

Similarly, one should be cognizant of comment pointers in sysouts, and be sure to retain any files thus pointed to.

When using comment pointers, the user should also not set PRETTYFLG 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 file.

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.

Converting Comments to Lower Case

This section is for users using terminals without lower case, who nevertheless would like their comments to be converted to lower case for more readable 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 file 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. Note that the user must type %% as % is the escape character. If the atom (minus any trailing punctuation marks) is an Interlisp word (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), 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 %% flag, so that conversion is thus only performed once (unless the user edits the comment inserting additional upper case text and another %% flag).

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 fuction 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 first 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..

Special Prettyprint Controls

PRETTYTABFLG [Variable]

In order to save space on files, tabs are used instead of spaces for the inital spaces on each line, assuming that each tab corresponds to 8 spaces. This results in a reduction of file 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. Comments run between FIRSTCOL and the line length set by LINELENGTH (see Chapter 25). 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 has more than PRETTYLCOM elements (using COUNT), it is printed starting at column 10, instead of FIRSTCOL. 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 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 occasionlly 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 (see Chapter 25), FIRSTCOL, and PRETTYLCOM to large values appropriate for pretty printing files to be listed on wide paper. (WIDEPAPER) restores these parameters to their initial values. WIDEPAPER returns the previous setting of *FLG*.

PRETTYFLG [Variable]

If PRETTYFLG is NIL, PRINTDEF uses PRIN2 instead of prettyprinting. This is useful for producing a fast symbolic dump (see the FAST option of MAKEFILE in Chapter 17). Note that the file loads the same as if it were prettyprinted. PRETTYFLG is initially set to T. PRETTYFLG should not be set to NIL if comment pointers are being used.

CLISPIFYPRETTYFLG [Variable]

Used to inform PRETTYPRINT to call CLISPIFY on selected function definitions before printing them (see Chapter 21).

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.

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.

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 on PRETTYEQUIVLST, then expressions beginning with QLAMBDA are prettyprinted the same as LAMBDAs. 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 Chapter 16). CHANGECHAR is initially |.