TEDITKEY

By: Greg Nuyens

DESCRIPTION

TEditKey is a package which provides a keyboard interface to TEdit. On a Dandelion, the interface takes advantage of the non-Alto keys. On Dorados and Dolphins, a window mimicking the Dlion function keys provides the same abilities.

The abilities provided include allowing the user to alter the *caret looks* (the looks of characters typed in) or the selection looks. These commands are given using the Dlion function keys and/or metacodes (keys typed while a meta key is held down. The default meta key is the tab key, to alter this see User Switches, below). Other metakeys move the cursor within the document (beginning/end of line, back/forward character, up/down a line, etc.)

Thus, many of the non-Alto keys are made to function in TEdit the way they are labeled. The following keys change their behaviour once TEditKey is loaded.

CENTER modifies the justification of the paragraph(s) containing the current selection. If the selection was left justified, then hitting the CENTER key makes it centered. Hitting it again produces right and left justification.

BOLD makes the selection bold. All other properties remain unchanged. Holding the shift key down while hitting BOLD, will make the selection become un-bold.

ITALICS makes the selection italicized. Shift-ITALICS is the opposite.

UNDERLINE makes the selection underlined. Shift-UNDERLINE is the opposite.

SUPERSCRIPT superscripts the selection by a constant amount. Any relative superscripts (or subscripts) are maintained. Thus if "Xi" is selected in "the set X*i* is empty" then pressing the SUPERSCRIPT button produces "the set X*i* is empty". See USER SWITCHES below for how to set the increment. Shift-SUPERSCRIPT is the same as SUBSCRIPT

SUBSCRIPT analogous to SUPERSCRIPT.

SMALLER decreases the font size of the selection. All relative size differences are maintained. e.g. "this is bigger than that" produces "this is bigger than that". shift-SMALLER (labelled LARGER) does the opposite.

DEFAULTS makes the selection have default looks. The default looks can be set to the current caret looks by typing shift-DEFAULTS

The above keys all affect the caret looks if the keyboard key is held down when they are hit. Thus holding down KEYBOARD, and then hitting UNDERLINE, makes the caret looks be underlined.

FONT change the font of the selection or caret looks according to the following table: (to alter this table see USER SWITCHES below:)

1 timesroman
2 helvetica
3 gacha
4 modern
5 classic
6 terminal
7 symbol
8 hippo

Thus, to change the font of the selection to Classic, hold down FONT and hit '5'. To change the caret font to Classic, hold down FONT (to signal the font change) and KEYBOARD (to direct the change to the caret looks) then hit '5'. Note that this table is part of the menu displayed when the HELP button is pressed.

KEYBOARD apply any changes that occur while this key is down to the caret looks instead of the selection.

AGAIN is an ESCape key, which acts as the TEdit REDO syntax class. (see p 20.22 Interlisp Reference Manual)

OPEN Opens a blank line at the current cursor position.

FIND prompts the user for a target string, then searches from the selection forward.

NEXT acts as the TEdit NEXT syntax class.

EXPAND expands TEdit abbreviations. (see p. 20.31 IRM)

HELP displays a menu of the keybindings until a mouse key is clicked.

UNDO acts as the TEdit UNDO syntax class. Note that it still retains its TELERAID function as does STOP.

ESC (the right arrow key on Dandelions) enters \, and | when shifted. (Recall that AGAIN is an escape key)

As well as the previous functions available on the Dandelion's non-Alto keys, the following functions are available on the standard keyboard (thus usable on the Dandelion, Dolphin and Dorado). Each function is shown with the key that invokes it (in conjunction with the control (denoted ↑) or meta (denoted #) key). Thus, for the sixth entry, holding down the metakey and hitting f (or "F") would move the caret one character forward. (To find out how to get a metakey see USER SWITCHES below.)

#/	default the caret looks
# =	query caret looks
#9	smaller caret font
#0	larger caret font
↑b	back char
↑f	forward char
↑p	previous line
↑n	next line
↑a	beginning of line
↑e	end of line
#<	beginning of document
#>	end of document
↑k	kill line (delete from caret to end of line)
↑o	open line
↑d	delete character forward (also on shift backspace)
↑t	transpose characters
#[	indent paralooks. also available on the MARGINS key.
#]	exdent paralooks. also available as shift-MARGINS.
j	justification change (same as CENTER key)

#u uppercasify selection
#l lowercasify selection

#? show keybindings

Note that the positions of any of these functions can be individually changed using TEDIT.SETFUNCTION (see p. 20.30 IRM). For wholesale customization see User Switches below.

User Switches:

TEDITKEY.METAKEY The user must chooses a metakey to make use of TEditKey. The value of the variable **TEDITKEY.METAKEY** is the name of the key which will be your metakey. For instance to make TAB (the default) your metakey, (SETQ TEDITKEY 'TAB) before loading TEditKey. (note that even in the standard system, TAB is available as Control-I).

Alternately, METASHIFT will make the swat key a meta. (see p 18.9 of the IRM)

The operation of TEditKey is controlled by the following (INITVARed) variables:

TEDITKEY.LOCKTOGGLEKEY This is the key which will be turned into a lock-toggle. If it is non-NIL, that key is set to act as a lock-toggle. Thus hitting this switches the case of the typein.

TEDITKEY.FONTS This is an eight element list of the fonts that are invoked by meta-1 through meta-8. The defaults are listed above.

TEDIT.DEFAULT.CHARLOOKS This defines the looks that result when the DEFAULTS key is pressed or when default caret looks are requested. It is an instance of the CHARLOOKS datatype. To set it, for instance, to TIMESROMAN 10 type the following to the lisp top level.

(SETQ TEDIT.DEFAULT.CHARLOOKS (CHARLOOKS.FROM.FONT (FONTCREATE 'TIMESROMAN 10)))

Alternately, select an instance of the desired looks, and type shift-DEFAULTS

TEDITKEY.VERBOSE if T (the default), the functions which modify the caret looks print feedback in the (TEdit) promptwindow.

TEDITKEY.NESTWIDTH the distance (in points) that the indent and exdent functions move the margins. Initially 36 points (0.5 inches).

\TK.SIZEINCREMENT the amount (in points) which the LARGER function increases the selection. (and conversely for SMALLER) Initially 2 points.

TEDITKEY.OFFSETINCREMENT the amount (in points) which the SUBSCRIPT function raises the selection. (and conversely for SUPERSCRIPT) Initially 3 points.

TEDIT.KEYBINDINGS the list which controls the mapping of keys to functions. It consists of triples of function name, list of CHARCODE-style character specifications, and a comment describing what the function does. (the comments are used by the automated menu building tools and their inclusion is encouraged)

TEDIT.DLION.KEYACTIONS the list which specifies the keyactions of the non-Alto keys (to the left and right) on the DLion.

TEDIT.DLION.FNKEYACTIONS the list which specifies the keyactions of the function keys (center, bold, etc.).

TEDIT.DLION.KEYBINDINGS the list specifying the functions to be tied to the characters generated from above. The keynames in the CAR of each element are comments. Note that TEDIT.DLION.KEYACTIONS and TEDIT.DLION.KEYBINDINGS must be coordinated (similarly for TEDIT.DLION.FNKEYACTIONS and TEDIT.DLION.FNKEYBINDINGS).

TEDIT.DLION.FNKEYBINDINGS analogous to TEDIT.DLION.KEYBINDINGS but for the fn keys.

TEDIT.DLION.KEYSYNTAX the list of syntax classes to be applied to the DLION keys.

The previous variables in conjunction with the following functions specify the effect of TEditKey.

(TEDIT.INSTALL.KEYBINDINGS *readtable*)

invokes the keyactions specified by TEDIT.DLION.KEYACTIONS and installs the values of TEDIT.KEYBINDINGS and TEDIT.DLION.KEYBINDINGS on *readtable*. (*readtable* defaults to TEDIT.READTABLE).

(\TK.BUILD.MENU)

function which builds the help menu from the values of the above variables

TINYTIDY

By: Larry Masinter (Masinter.pa@Xerox.ARPA)

TINYTIDY takes the icons on your screen and lines them up along the edge:

(TINYTIDY edge spacing)
edge defaults to RIGHTBOTTOM, but LEFT, LEFTBOTTOM, RIGHT, RIGHTBOTTOM, TOP, TOPRIGHT, BOTTOM, BOTTOMRIGHT are all allowed. spacing (pixels between icons) defaults to 4

TINYTIDY also adds a background menu item.

TMENU

By: Mark Stefik (Stefik.pa @ Xerox.ARPA)

The TMENU package provides the following features:

TMenus. These are interactive menus intended for reducing the amount of typing to an Interlisp-D program. When an item is selected in a menu, an expression is inserted into the TTY input buffer. It can be used to simplify the entry of common LISP commands or long names.

Windowshades. Windowshades are a modification that can be made to any window in order to conserve screen space. They make a window "roll up" when not in use, leaving behind only the title. When the mouse is clicked inside the remaining bar, the window unrolls for interaction, and rolls up again at completion. Windowshades can be used with TMenus.

Msgs to the Prompt window. Two functions are provided for clearing and printing to the Interlisp-D prompt window. These functions take an arbitrary number of arguments and preserve the black background shade of the window.

Interaction with TMenus

TMenus are placed on the display under program control using the functions described below. Once a menu is on the screen, items may be selected using the LEFT mouse button. This causes some text to be inserted into the teletype buffer. In the default case, this text is just the item in the window, but it can alternatively be the result of evaluating an expression. In the default case, the test is followed by a blank space in the buffer, but it can be followed alternatively by an arbitrary string (such as the empty string or a carriage return). Non-default cases are controlled by arguments to the TMenu function.

The MIDDLE Mouse button is used for user-interactions that change the menu. When the yellow button is depressed inside a menu, another pop-up menu (a meta menu!) appears that provides several options for changing the menu, such as adding or deleting items. One of the options is to recompute the set of items by evaluating an expression associated with the menu.

The RIGHT mouse button is used for the usual window commands. These commands work in the standard way except that when a TMenu is reshaped, internal functions are invoked to adjust the configuration of rows and columns in order to create a menu that is visually appealing.

Main Functions

TMenu (itemExpr title displaySpec windowShadeFlg buttonFn default TrailerString)

TMenu is the function for creating a menu in a window on the display. Its arguments are as follows:

itemExpr

An expression for computing the list of items that is saved with the menu. In the usual case, itemExpr is a list, whose items are either atoms or lists of the form:

(displayThis evalThis comment trailerString)

where displayThis is the form displayed in the menu, evalThis is the form evaluated to create the text for the input buffer, comment is displayed in the prompt window if the left button is helt over an item for an extended period, and trailerString is the string inserted after an expression in the input buffer. Most of these fields are optional. The field value for evalThis is the entry in displayThis. The default trailerString can be specified for a TMenu in the defaultTrailerString argument above. Otherwise it is a space if the expression is an atom, and the empty string otherwise. This definition of fields for menu items is compatible with the usual set for the Interlisp-D menu package, with the addition of the trailerString field.

Special cases: If itemExpr is an atom, it must be the name of a global variable to be evaluated to yield the list of the form described above (e.g., MYFNS). If itemExpr is a list and the first element of the list has a functional definition, then that function is evaluated to yield the list. This is intended for cases where the list is tobe computed.

title

The title of the menu. This title is used as the title of the window containing the menu.

displaySpec

This argument has several possible interpretations that control the display of the menu. If displaySpec is a region, then the window for the menu is placed in that region. If displaySpec is a number, that number is used as the number of columns in the menu display and a minimum size window is allocated for displaying the entire menu. The user is prompted with a ghost box to place the menu on the display. If displaySpec is T, then the number of columns is computed by TMenu assuming a maximum of 15 rows per column and the user is prompted for placement as before. If displaySpec is NIL, then the user is prompted to place a bounding box for the menu and TMenu tries to compute an arrangement of rows and columns that is visually pleasing.

windowShadeFlg

If T, the window containing the menu is augmented with a window shade so that the menu "rolls up" if not in use. If windowshadeFlg is NIL, the menu is placed in a window on the screen:

buttonFn

Optional argument that allows the caller to specify his own function for handling the LEFT and MIDDLE buttons.

defaultTrailerString

Optional argument that allows the caller to specify the default string to follow each item printed. Can be overriden for specific items by the trailerString argument in the itemExpr.

Exmaple

The following expression:

```
(TMenu 'MYFNS "Common Fns" T)
```

would create a window titled "CommonFns" and display the list of functions in the window. The window would contain columns of up to 15 functions each, and would be placed on the screen under control. If the user later used the MIDDLE button to add or delete items from the menu, then the list in the variable MYFNS would be updated as the menu is updated.

MakeFileMenus (fileName)

MakeFileMenus is the function for creating a set of menus for the functions and variables in a file. A window is created for each menu. The menus are placed under user control and are all given window shades. The argument fileName is the name of a file.

Example

If the file is MYFILE, then the fns on MYFILENS and the vars on MYFILEVARS would be displayed in menus. MakeFileMenus would look for file commands of the form (FNS * FnsLst) and (VARS * VarsLst) on the command

CloseFileMenus (fileName)

Closes all of the TMenu windows associated with the given file.

Window Shades

MakeWindowShade (windows)

Modifies the given window to provide a window shade. If the window argument is NIL, then the window is selected which is under the cursor. If the window argument is T, it waits for the CTRL key to be depressed, and then selects the window under the cursor.

Prompt Window Functions

PROMPT (arg_1 arg_2 arg_3 ...)

Prints an arbitrary number of arguments to the prompt window after first clearing the window. A call with no arguments simply clears the window.

CPROMPT (arg_1 arg_2 arg_3 ...)

Same as prompt except the arguments are printed centered in the window.

TRACEIN

By: Don Cohen and Dave Dyer
(Don C @ ISIF.ARPA and D Dyer @ ISIB.ARPA)

Tracein is a facility for very low level tracing/stepping, i.e., to show you everything that takes place within a particular piece of code.

IMPORTANT You should load the compiled version of TRACEIN, not the interpreted version. The interpreted version does not do the right thing for RETURNs. The problem is that the lisp interpreter does not care which return goes with which prog. The stepper contains a PROG in which your code is EVAL'd. If that code is a RETURN, it returns to the prog in the stepper. The compiled version does not have this problem.

(TRACEIN fn {T} loc1 loc2 . . .) [NLAMBDA NOSPREAD]

TRACEIN is like BREAKIN except that it arranges to single step the code rather then just breaking when it gets there. The optional T causes an automatic trace, otherwise a prompting trace is started.

The first argument is the name of the function in which the code is to be found. It can also be a list of the form (fn test) where test will be evaluated to decide whether to actually trace. The default condition is T. (Note: the condition NIL is also treated as T - if you want never to break try (NOT T). This is a feature inherited from BREAKIN.)

The other arguments are editor location specifications. (TRACEIN FN) is like (BREAKIN FN) - it allows you to find the desired location in the editor. When you type OK the current expression is the answer. (TRACEIN fn loc1) does a (BREAKIN fn (AROUND loc1) <some trickery>). Since TRACEIN works by calling BREAKIN, it can be undone by calling UNBREAK.

(WATCH form stepaction nohook) [LAMBDA SPREAD]

Watch traces the evaluation of form. If STEPACTION is T an automatic trace is done, if NIL a prompting trace. If nohook is T, the WATCH mode will always be used (rather than EVALHOOK mode as described below).

BREAKMACROS:

These break macros will work at entry breakpoints for interpreted functions, and also at BREAKIN breakpoints where AROUND was the break type.

TRACEALL starts an automatic trace of the current form, so you can use BREAKIN and decide to trace it from the break.

STEP starts a prompting trace.

General Description

TRACEIN uses BREAKIN to set up a breakpoint, and works just like BREAKIN except that BREAK1 is called with a command to start tracing the form. If there is no second arg of T, then you are prompted at each trace step, and given several continuation options. If the second arg is T, then the tracing proceeds automatically, showing the full trace of what goes on within the form being traced.

Note that it only makes sense to tracein expressions that will be evaluated.

You will get into trouble if you attempt to trace a clause of COND, e.g., ((ATOM X) NIL), because it will be interpreted as call to the function (ATOM X).

TRACEIN prompts with "->" if the expression has yet to be evaluated and "<-" if it has already been evaluated. The prompting is done by ASKUSER, so ? will print the full list of options to respond to the prompt. The options are:

S continue stepping (into the form from ->, on to the next one from <-)
E evaluate (or re-evaluate) the form silently, prompt again with the result
F finish the evaluation without further printing
T trace the form, showing every evaluation without prompting
P prettyprint the form
V prettyprint the value (if the prompt is -> this is NOBIND)
R retry - go back to -> mode
X set the result to be returned on exit to a value you supply
B Break - type RETURN to get back to the stepper
<space> escape to eval one expression

There are two slightly different implementations of TRACEIN, the "watch" implementation and the "evalhook" implementation. Currently, the "evalhook" is only available on Interlisp-VAX. The "watch" implementation works in any Interlisp. The "evalhook" mode is standard for the vax.

WATCH mode.

This is in use when the variable WATCH-EVALHOOK is NIL. In this mode, the tracing is done by laboriously generating a translation of the traced expression. There are several constraints on this process; first, the tracer has to guess correctly which forms will actually be evaluated. There is a mechanism (detailed later) to advise the translation process about any nonstandard special forms you are tracing. Also, it is the DWIMIFIED version of your code that is traced, so you never see iteration forms, macros, or changetran expressions. For example, you might see a PROG or a MAPCAR in place of a FOR.

You might prefer this mode to EVALHOOK mode in some circumstances, since only code lexically scoped by the initial BREAKIN is traced: other EXPRS encountered in the process of evaluating the traced form are not automatically traced. Sometimes, this is just what you want.

The WATCHified form (the translation of the form to be traced is the result of embedding each subexpression to be evaluated in a call to WATCH-EVAL) is computed the first time it is needed, and is stored in CLISPARRAY. Thus the WATCHified version will automatically go away (and have to be recomputed) if you ever change the expression. (Detail: before WATCHification, the expression is DWIMified. WATCHification of an expression which already has a CLISP translation stores the WATCHified version of the CLISP translation as the-CLISP translation of the original form. Thus, when the break condition is false, the original CLISP translation is used. When the expression is changed, the CLISP translation goes away, and so the WATCHification (as well as the DWIMification) is redone.)

EVALHOOK mode

This is modeled after the EVALHOOK feature of the MACLSP family. Basically, there is a hook that allows any call to EVAL to be interrupted and some other function called instead. In EVALHOOK mode this feature is used to trap to the stepper. The main difference between this and WATCH mode is that ANY form that is EVAL'd is trapped, not just the lexical code you use BREAKIN on, once the stepper is invoked. This means that you trace all levels of interpreted code below a TRACEIN. Also, because your program does not have to be analyzed in advance, you see the original Iteration, changetran, macros and so on, as well as their translations. Unfortunately, you also occasionally see a random EVAL done in the process of generating the translation.

You disable EVALHOOK mode by setting WATCH-EVALHOOK to NIL.

Advising WATCH mode about nonstandard special forms.

Most users can ignore this. If you are tracing NLAMBDAs of your own construction, this may be important.

TRACEIN traces every subexpression that it EXPECTS to eventually be evaluated. It doesn't expect the arguments of NLAMBDAs to be evaluated, although sometimes they are. Users can tell TRACEIN what will be evaluated by adding an EVL-FIX property to functions that TRACEIN does not understand. (Maybe some masterscope expert can figure out how to use the masterscope templates to get most of the effect of EVL-FIX properties described below.) The system functions are supposed to be already understood, but if that understanding is wrong (or the user wants to change it for any other reason) the same tactic will work. The EVL-FIX property is a pattern to be applied to the tail of a function call. A pattern element of T means that the corresponding argument will be evaluated (and thus ought to be traced). NIL means that it will not. Thus the pattern for SETQ is (NIL T). If a pattern element is itself a list, then that argument is not to be traced, but it is expected to be a list which will in turn be matched with the pattern element to determine whether its elements will be traced. In addition, the special pattern element TAIL allows the pattern element after it to be applied to as many arguments as necessary to exhaust the list. For example, the pattern for COND is (TAIL (TAIL T)). The pattern for SELECTQ is (T TAIL (NIL TAIL T) T).

If the CAR of a pattern is TEST, the next element is an expression which is evaluated. If the result is not NIL, it will be traced. Also, as a special case, if a pattern element is an atom other than T or NIL, it is expected to be a function which will be applied to the corresponding argument, and if the result is not NIL then the argument will be traced. If the CAR of a pattern is EVAL, the next element is evaluated, and what it returns is used as the pattern. In such code, the free variable EXP is the function call that the resulting pattern is supposed to describe. In addition, the variable NOEMBED may be set to T to indicate that the function call itself is not to be traced. This feature is not normally needed, but LAMBDA is an example where it is.

Controlling the Printing and PRINTUPTO

The printing is done by (APPLY* FORMPRINTER <form> and (APPLY* VALUEPRINTER <value>) so you can control the printing. The initial values of these call PRINTUPTO to print as much as will fit on the line after the indentation, leaving space for the resposne to ASKUSER (but never cutting off the printing in less than 20 characters). If anyone can use such a function, it is (PRINTUPTO x limit UsePrin2 Ignore File), which prints up to limit characters of x. If UsePrin2 is not NIL then it prints with PRIN2 (otherwise PRIN1). Ignore is used to skip calls to WATCH-

EVAL. (The WATCHified form is passed to the formprinter.) When the CAR of a list is in Ignore, only the CADR is printed. If you want to replace FORMPRINTER, the function UNWATCH will remove the WATCHes from the form.

Similar applications and EVL-FIX

The WATCHifier might be of use to people who are interested in similar applications. For example, one could keep track of how many times each subexpression is executed by embedding each expression to be evaluated in (Count 0 <expression>) where Count is a function that increments the counter and returns (EVAL <expression>). (EVL-FIX form prefix) returns the result of appending prefix to the front every expression to be evaluated in form. (EVL-FIX ' (ADD1 (COND (FOO BAR))) ' (Count 0)) would be (Count 0 (ADD1 (Count 0 (COND ((Count 0 FOO) (Count 0 BAR)))))).

TRAJECTORY-FOLLOWER

By: D. Austin Henderson, Jr. (A Henderson.pa @ Xerox.ARPA)

TRAJECTORY-FOLLOWER provides a function which causes a "snake" to crawl along a trajectory.

TRAJECTORY.FOLLOW (KNOTS CLOSED N DELAY BITMAP WINDOW) The trajectory is specified by KNOTS (a set of knots) and CLOSED (a flag indicating whether it is an open or closed curve). N is the length of the snake in points along the curve. DELAY is the time (in milliseconds) between each move along the curve; DELAY = 0 or NIL means go as fast as you can. BITMAP is the brush to be used at each point in creating the snake. WINDOW is the window in whose coordinate system the knots are given and in which the snake is to be drawn; if NIL, then the SCREEN bitmap is used. The snake is moved by INVERTing the bitmap at the points along the curve, and then INVERTing the bitmap back out again.

A demonstration function is also provided with the package:

TRAJECTORY.FOLLOWER.TEST ( ) Interacts with the user through prompting in the promptwindow to gather up arguments for TRAJECTORY.FOLLOW and then carries it out. Closed curves are snaked around repeatedly until the left shift key is depress when it reaches the curve's starting point.

The internal functions used by this package are also available for use. They are:

TRAJECTORY.FOLLOWER.SETUP (WINDOW N DELAY BITMAP) Initializes drawing variables.

TRAJECTORY.FOLLOWER.POINT (X Y WINDOW) Defines the next point on the curve. Note that the argument structure of this function is appropriate for use as a BRUSH with the curve drawing functions DRAWCURVE, DRAWCIRCLE, and DRAWELLIPSE. (For an example, see the demonstration function TRAJECTORY.FOLLOWER.TEST.)

TRAJECTORY.FOLLOWER.WRAPUP ( ) Finishes the job after all the points have been defined.

TTY

By: Mark Stefik (Stefik.pa @ Xerox.ARPA)

The TTY package is a set of functions for interacting with the TTY. It is much simpler to use than ASKUSER. The input functions also provide a technique for suppressing prompts that makes it easy to create programs that respond to "commands" or sequenced entry with a varied amount of prompting for input.

INTTY (promptStr goodList helpStr noShiftFlg)

INTTY interacts with a user at the TTY and returns an atom. In its simplest usage, it prompts the user using the promptStr. The user then types a response followed by a carriage return. INTTY returns the user's response which is a member of goodList. If there is a sequence of calls to INTTY, the user can separate the responses by spaces and follow the last one with a carriage treturn.

More formally, the arguments to INTTY are as follows:

promptStr. A prompt typed to the TTY when INTTY is invoked. This prompt is suppressed if the user has typed ahead.

goodList. Optional argument, this is a list of acceptable answers. If the characters typed by the user correspond to the fir characters of any element of goodList, that member is returned as the value of INTTY. If the leading characters are ambiguous then user is warned. If goodList is supplied and the user's reply matches it) entry. INTTY attempt spelling correction. On failure of speedling correction all type ahead is flushed and the user is warned and reprompted. If goodList is .NIL, no checking is performed.

helpStr. A secondary prompt if the suer types a ? to INTTY. Intended to allow short prompts. and longer more explanatory prompts as needed.

noShiftFlg. Normally, INTTY converts the user's response to all upper case. If noShiftFlg is T, then the user's response is passed along as typed.

Example

The following statements:

```
(SETQ name (INTTY "Name: "  NIL  "Please type your first name"))
(SETQ age (INTTY "Age:  "  "Your age in years"))
(SETQ color (INTTY  "Paint color:  " '(RED BLUE YELLOW WHITE)))
```

could yield the following dialog (prompts in boldface):

Name: John <return>
Age: 17 <return>
Paint color: Green <return>
Response not recognized. Type ? for help.
Paint color: ? <return>
Expecting one of (RED BLUE YELLOW WHITE)
Paint color: blue <return>

which illustrates the help facility, or the following dialog

Name: John 17 yell <return>

which illustrates the suppression of type ahead and name completion. Noate that no actual reading of the typing takes place until the <return> is typed. Successive calls to INTTY suppress the prompt, and read through the separating spaces. At the conclusion of execution. the LISP variables name, age, and color have all been set to JOHN. 17. and YELLOW respectively.

INTTYL (promptStr goodList helpStr noShiftFlg)

Similar to INTTY except that it allways returns a list of the items typed. INTTYL reads successive entries until a carriage return is typed.

WRITE (arg1 arg2 ...)

Prints the arguments arg1. arg2. etc., to the TTY (using PRINT1). Prints a carriage return at the end. Output goes to the primary output device.

PrintStatus (arg1 arg2 ...)

Similar to WRITE except that PrintStatus allocates its own window and prints to that. The window is title "Status Window" and is made to scroll as writing proceeds.

TTYIO

By: Reid Smith

TTYIO is a package containing a number of functions built on top of TTYIN that enable you to prompt a user for files, functions, reals, integers, dotted pairs, and atoms with defaults and flexible restriction handling.

When using these functions in sequence, you may type responses ahead by including the responses to successive questions on the same line as the response to the current question.

TTYIO also contains a "text" pointer facility, based on INTERLISP comment pointers, that enables you to store help messages on a file without storing them as strings in memory.

Interaction Functions:

(ASKFLE MODE PROMPT HELP CONFIRMFLG DEFAULT NULFLG FILE)

ASKFLE asks for a filename using TTYINC with PROMPT and HELP. MODE is either INPUT or OUTPUT (defaults to INPUT). CONFIRMFLG is T if user confirmation is required. DEFAULT is returned if <cr> is entered as the response. NULFLG is T if a null response is permitted. If FILE is given, then it is taken as the response without user interaction. However, it is still checked for validity (and user interaction will occur if it is not valid).

(ASKFN PROMPT DEFAULT HELP SPLST FN CONFIRMFLG NULFLG)

ASKFN asks for a function name or LAMBDA expression using TTYINC with PROMPT and HELP. CONFIRMFLG is T if user confirmation is required. DEFAULT is returned if <cr> is entered as the response. NULFLG is T if a null response is permitted. SPLST (if given) is a list of suggested function names. The list is used strictly for <esc> completion. The response does not have to be a member of the list. If FN is given, then it is taken as the response without user interaction. However, it is still checked for validity (and user interaction will occur if FN is not a valid function).

(ASKINT PROMPT DEFAULT HELP LOWERBOUND UPPERBOUND INTEGER CONFIRMFLG NULFLG)

ASKINT asks for an integer using TTYINC with PROMPT and HELP. CONFIRMFLG is T if user confirmation is required. DEFAULT is returned if <cr> is entered as the response. NULFLG is T

if a null response is permitted. LOWERBOUND and UPPERBOUND are the lower and upper bounds on acceptable answers. The response must be greater than or equal to the lower bound and less than or equal to the upper bound. Both bounds may be defaulted to NIL. If INTEGER is given, then it is taken as the response without user interaction. However, it is still checked for validity (and user interaction will occur if it fails to meet the requirements).

(ASKITEM RESTRICTION PROMPT HELP CONFIRMFLG DEFAULT NULFLG ITEM)

ASKITEM asks for an item using TTYINC with PROMPT and HELP. RESTRICTION is either a list of allowable responses or a function that returns the response if it is valid and NIL otherwise. CONFIRMFLG is T if user confirmation is required. DEFAULT is returned if <cr> is entered as the response. NULFLG is T if a null response is permitted. If ITEM is given, then it is taken as the response without user interaction. However, it is still checked for validity (and user interaction will occur if it fails to meet the requirements).

If a restriction list is provided and the first element of the list is "*" then new items are allowed. Otherwise, the response must be a member of the restriction list.

(ASKITEMS RESTRICTION PROMPT HELP CONFIRMFLG ITEMS)

ASKITEMS asks for a list of items using TTYINC with PROMPT and HELP. RESTRICTION is either a list of allowable responses or a function that returns the response if it is valid and NIL otherwise. CONFIRMFLG is T if user confirmation is required. If ITEMS is given, then it is taken as the response without user interaction. However, it is still checked for validity (and user interaction will occur if it fails to meet the requirements).

If a restriction list is provided and the first element of the list is "*" then new items are also allowed. Otherwise, the response must be a member of the restriction list.

Once the list of items has been entered, ASKITEM is called to check the validity of each item.

(ASKPAIR RESTRICTION PROMPT HELP CONFIRMFLG DEFAULT NULFLG ITEM)

ASKPAIR asks for a dotted pair using TTYINC with PROMPT and HELP. RESTRICTION is either a list of allowable responses or a function that returns the response if it is valid and NIL otherwise. CONFIRMFLG is T if user confirmation is required. DEFAULT is returned if <cr> is entered as the response. NULFLG is T if a null response is permitted. If ITEM is given, then it is taken as the response without user interaction. However, it is still checked for validity (and user interaction will occur if it fails to meet the requirements).

If a restriction list is provided and the first element of the list is "*" then new items are also allowed. Otherwise, the response must be a member of the restriction list.

If the value is expected as (A . B), the user should type A B. If a default is supplied, the user can type A alone, and the default B will be supplied.

(ASKRL PROMPT DEFAULT HELP LOWERBOUND UPPERBOUND REAL CONFIRMFLG NULFLG)

ASKRL asks for a floating point number using TTYIN with PROMPT and HELP. CONFIRMFLG is T if user confirmation is required. DEFAULT is returned if <cr> is entered as the response. NULFLG is T if a null response is permitted. LOWERBOUND and UPPERBOUND are the lower and upper bounds on acceptable answers. The response must be greater than or equal to the lower bound and less than or equal to the upper bound. Both bounds may be defaulted to NIL. If REAL is given, then it is taken as the response without user interaction. However, it is still checked for validity (and user interaction will occur if it fails to meet the requirements).

(ASKYN PROMPT DEFAULT HELP RESPONSE)

ASKYN asks for a Yes/No answer using TTYIN with PROMPT and HELP. DEFAULT is returned if <cr> is the response (any non-NIL default is taken as Yes). If RESPONSE is given, then it is taken as the response without user interaction. However, it is still checked for validity (and user interaction will occur if it fails to meet the requirements). ASKYN returns T for Yes and NIL for No.

(TTYINC PROMPT SPLST HELP OPTIONS ECHOTOFILE TABS UNREADBUF RDTBL NOSTOREFLG)

TTYINC is used in conjunction with TTYIN to allow the user to type a number of commands ahead to a prompt. It is typically used with the TTYIN modes COMMAND and STRING. If the global variable TTYIN-COMMAND-LINE is non-NIL upon entry, then it is concatenated with UNREADBUF and a line terminator. TTYIN is then called. CDR of the line returned by TTYIN is stored back in TTYIN-COMMAND-LINE unless NOSTOREFLG is non-NIL.

Text Pointer Functions:

Text pointers are variations on comment pointers. A text pointer is an s-expression of the form (; "text string"). ";" is defined as a read macro. When the global variable TTYIO-TXT-FLG is NIL, only pointers are loaded. When T, the full text is loaded. Upon printing to the primary output file, the leading parenthesis, semi-colon, and trailing parenthesis are stripped off. GET; is a user macro similar to GETCOMMENT. Regardless of the TTYIO-TXT-FLG setting, it loads the actual text.

(DISPLAYHELP KEY)

DISPLAYHELP copies the help message indexed by KEY to the primary output file. KEY can either be a text pointer (with TTYIO-TXT-FLG T or NIL) or a string or an atom (T is ignored). DISPLAYHELP returns NIL if nothing was printed; T if the entry was found, or user typed ↑O. The function SPRINTT is advised to ensure that DISPLAYHELP is called from TTYIN in response to ? or HELP.

Simple Output Functions:

The following functions were written by Bill VanMelle for the MYCIN environment.

(WRITE . N)

WRITE takes an arbitrary number of arguments, each of which is PRIN1ed to the primary output file, followed by EOL. If an argument is a list, it is MAPRINTed; i.e., the outer parentheses will not appear.

(WRITE1 . N)

WRITE without the final EOL.

(WRITEARG X FILE)

WRITEARG PRIN1's X to FILE if it is not a list. Otherwise it uses MAPRINT. If X is not a list, WRITEARG PRIN1's it to FILE, otherwise MAPRINT's it, so that the outer parentheses will not appear.

(TTYOUT . N)

WRITE to the tty display stream.

(TTYOUT1 . N)

TTYOUT without the final EOL.

TwoD

By: Herb Jellinek (Jellinek.pa @ Xerox.ARPA)

Introduction

TwoD provides a subset of the ACM SIGGRAPH 2D Core graphics standard for use by Interlisp-D programs. Specifically, it provides world-to-viewport and viewport-to-screen coordinate transformations, and versions of some Interlisp-D display primitives that use them.

Before describing the functions in the package, we must define a few terms:

The *world* coordinate system is the space in which our picture is defined. It is chosen to suit the application program, and is completely arbitrary: for example, one dimension might be "millions of dollars," and the other "years."

A *viewport* is a region of an Interlisp-D window (as distinct from the SIGGRAPH Core notion of a window).

The *viewing transformation* is the operation we perform on points in the world coordinate system to map them into viewport coordinates.

We can depict the arrangement like this:

The purpose of the TwoD package, then, is to provide versions of a number of Interlisp-D display primitives (BITBLT, DRAWLINE, etc.) that operate in the world-coordinate domain.

Functions

TwoD contains the following functions:

(TwoD.Init *W*)

Initializes the window *W* for use with the TwoD primitives. Must be called before any other TwoD operations are performed on *W*.

(**TwoD.BitBLT** *SOURCEBITMAP SOURCELEFT*
SOURCEBOTTOM destinationWindow
DESTINATIONLEFT DESTINATIONBOTTOM
WIDTH HEIGHT SOURCETYPE OPERATION
TEXTURE CLIPPINGREGION)

BITBLTs *SOURCEBITMAP* onto *destinationWindow*. *SOURCELEFT*, *SOURCEBOTTOM*, *DESTINATIONLEFT*, *DESTINATIONBOTTOM*, *WIDTH*, and *HEIGHT* are considered in the world coordinate system of *destinationWindow*. The rest of the arguments are the same as for BITBLT. Bug: *CLIPPINGREGION* is defined in screen coordinates.

(**TwoD.DrawBetween** *Pt1 Pt2 width operation W*)

Draws a line of width *width* from *Pt1* to *Pt2* on *W*. *Pt1* and *Pt2* are defined in *W*'s world coordinate system. What we refer to as "points" are known to Interlisp-D as "positions."

(**TwoD.DrawLine** *x1 y1 x2 y2 width operation W*)

Draws a line of width *width* from (*x1, y1*) to (*x2, y2*) on *W*. The points are defined in terms of W's world coordinate system.

(**TwoD.DrawTo** *point width operation W*)

Draws a line of width *width* from *W*'s "current position" to *point*. *point* is defined in terms of *W*'s world coordinate system.

(**TwoD.MoveTo** *Pt W*)

Moves to point *Pt* on *W*.

(**TwoD.PlotAt** *Pt glyph W*)

Draws *glyph* (a bitmap) on *W* at point *Pt*.

(**TwoD.RelDrawTo** *deltaX deltaY width operation W color*)

Draws a line of width *width* from *W*'s "current position" to the point (*deltaX*, *deltaY*) away.

(**TwoD.SetViewport** *Xmin Ymin Xmax Ymax window*)

Sets the viewport of *window* to be the region bordered by the lines X = *Xmin*, X = *Xmax*, Y = *Ymin*, and Y = *Ymax*.

(**TwoD.SetWorld** *Xmin Ymin Xmax Ymax window*)

Sets the world coordinate system of *window* to be the region bordered by the lines X = *Xmin*, X = *Xmax*, Y = *Ymin*, and Y = *Ymax*.

(**TwoD.WorldToScreen** *Pt W*)

Returns the result of doing the viewing transformation on *Pt* considered in the window *W*'s world coordinate system.

Example

Here's a simple graphing program that makes use of some of TwoD's capabilities.

```
(DEFINEQ (SIMPLEGRAPH (WINDOW DATA)

    (* a simple program to illustrate the use of TwoD)

    (PROG (OldPt)
        (TwoD.Init WINDOW)
        (TwoD.SetWorld 0 0 12 1.0E12)     (* X runs 0..12, Y runs 0..1E12)
        (TwoD.SetViewport 0 0
    (fracOfWidth 1.0 WINDOW)
    (fracOfHeight 1.0 WINDOW))        (* We'll use the whole window)
        (for Pt in DATA do            (* Draw it!)
    (if OldPt then
    (TwoD.DrawBetween OldPt Pt 1 (QUOTE PAINT) WINDOW)
    (SETQ OldPt Pt)
        else (TwoD.MoveTo Pt WINDOW]
```

UTILISOPRS

By: Tom Lipkis (Lipkis @ USC-ISIF)

This package contains some useful CLISP iterative statement operators contributed by various people over the years. Both upper and lower case versions are recognized.

I.S.TYPES

COLLECTWHEN <form>

Collects all non-NIL instances of <form> into a list and returns it as the value of the i.s. when it terminates.

INTERSECT <form>

Computes the intersection (using the function INTERSECTION) of all instances of <form> and returns it as the value of the i.s. <Form> should evaluate to a list (or NIL) on each iteration.

UNION <form>

Similar to INTERSECT, but computes the union (using UNION) of all instances of <form>.

FINTERSECT <form>

Similar to INTERSECT, but the intersection is computed using an EQ test rather than EQUAL. Uses the function FIntersection (also defined in this package). FUNION <form> Similar to UNION, but uses EQ instead of EQUAL. Uses the function FUnion.

MAXIMIZE <form

MINIMIZE <form>

Computes the numerically largest (smallest) instance of <form> and returns it as the value of the i.s. Differs from LARGEST (SMALLEST) in that the largest (smallest) value of <form> is returned, rather than value of the i.v. that produced the largest (smallest) value of <form>.

UNIQUE <form>

If <form> evaluates to non-NIL exactly once during the iteration then that value is returned, else NIL. The i.v. is left set to the value that caused <form> to be non-NIL, for use either in a FINALLY or outside the loop in the case of (FOR OLD x IN --). For example, (FOR x IN '(1 2 3) UNIQUE (EVENP x) FINALLY (PRINT x)) will print 2 and return T (the value of (EVENP 2)).

Other I.S.OPRS

REPEATEACHTIME <form>

Causes <form> to be evaluated on each iteration after the main body of the loop. For example, (FOR x IN L COLLECT (F x) REPEATEACHTIME (G x)) will collect all the (F x)s, evaluating (G x) each time AFTER (F x) is evaluated).

FIRSTTIME <form>

<form> is evaluated on the first iteration instead of the main body. (FOR x IN '(1 2 3) FIRSTTIME (G x) DO (F x)) will evaluate (G 1), (F 2) and (F 3).

YIELD <form>

The value of <form> is returned as the value of the i.s. after the i.s. terminates normally. (Equivalent to FINALLY RETURN <form>).

THEN <form>

If the value of the i.s. would otherwise have been non-NIL, then the value of <form> is returned, else NIL. Most useful with NEVER, ALWAYS and THEREIS. For example, (FOR x IN L NEVER (EVENP x) THEN (PRINT 'NEVER!)) will print (and return) NEVER! iff all elements of L are odd.

ELSE <form>

Similar to THEN but evaluates and returns <form> iff the i.s. would otherwise have returned NIL.

BUGS/QUESTIONS to LIPKIS@ISIF.ARPA

WAM (Window Action Menu)

By: Jeff Shrager (Shrager @ CMS-CS-A.ARPA)

WAM can be used to attach an iconic menu to any window. This menu (attached to the left or right of the target window) contains various icons, that refer to the window action menu commands: CLOSE (a door closing), REDISPLAY (a paint brush), SHRINK (a shirking window icon), and MOVE (a window on wheels).

(WAM [window] [border = {LEFT/RIGHT}])

If window is nil, (WHICHW) is used.
Border defaults to "RIGHT".

Simply select the icon, to call the appropriate window action function.

By adding "wamify" to one's WindowMenuCommands, which, when selected, calls WAM makes it easy to add a window action menu to any window.

* Notes:
My icons are horrible. Anyone wishing to hack them, may feel free.

The action menu gets reshaped a little bizarrely, by virtue of the way that attache windows work.

You can use TOP and BOTTOM for the "edge" argument, but boy will you be surprised by the result.

* Recommended enhancements:

Fix the icons.

Make the attached window (menu) respond correctly to repaint so that it fixes its width appropriately.

Permit TOP and BOTTOM for attachement.

(WHICHW) is used in the WAM menu items. This can lose if you've moved the mouse in the mean time (ie. when waiting for the handler to be swapped in). You might want to use (WFROMMENU FROMMENU). DEFAULTWHENSELECTEDFN binds FROMMENU.

"Close" selected from the window menu in a WAM window should close the WAM only, thus giving a way of getting rid of it. Probably something like WFN would do it.

YAPFF

By: Larry Masinter (Masinter.pa @ Xerox)

Yet Another Page Full Function, YAPFF is different than PageHold in that it changes the title the held window and flashes it with a grey every once and a while. To get rid of the "Hold", the held process has to have the TTY when you press down "shift".

Pressing both shift-keys lets the process print forever.

Could be better, but it is yet another try.