MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper 208A Revised June, 1981

Lisp Machine Choice Facilities

David A. Moon

This document is a draft copy of a portion of the Lisp Machine window system manual. It is being published in this form now to make it available, since the complete window system manual is unlikely to be finished in the near future. The information in this document is accurate as of system 70, but is not guaranteed to remain 100% accurate. To understand some portions of this document may depend on background information which is not contained in any published documentation.

The window system contains several facilities to allow the user to make choices. These all work by displaying some arrangement of choices in a window; by pointing to one with the mouse the user can select it. This document explains what the various facilities are, how to use them, and how to customize them for your own purposes.

Table of Contents

1. Choice Facilities

The window system contains several facilities to allow the user to make choices. These all work by displaying some arrangement of choices in a window. By pointing to one with the mouse the user can select it. The details (how the choices are specified, what the user interaction looks like, and what happens when a choice is selected) vary widely, which is why there are several separate facilities.

Each choice facility is implemented as a family of window flavors, providing several variations on the basic facility. For those who don't want to create their own window, each facility provides an easy-to-use function interface which temporarily pops up a window of the appropriate flavor. The function interfaces will be described first in each section. Following the function interfaces there is documentation on how to create and use a window which has the facility.

This document does not cover how to modify these facilities to provide your own specialized versions, except in the simplest ways. That is certainly a reasonable thing to want to do. In order to do it you will need to read some of the code that implements the facility in question, for instance to learn about window instance variables and about internal messages that you might want to redefine or put daemons on.

This document does not attempt to explain how the mouse is controlled by the window system. Certain obscure details of these facilities may not be obvious to those who are not familiar with the way the window system deals with the mouse internally. However, these facilities are supposed to shield the user from such details, and can be effectively used with no knowledge of how they are implemented internally. It is sometimes useful to know that there is an asynchronous process, the "mouse process", which handles interaction with the mouse. Some portions of these facilities execute in the process that calls them, while other portions execute in the mouse process. All Lisp evaluation with which the user is concerned takes place in the user's process when using the facilities described in this document, with a very small number of exceptions which are noted when they occur. Thus the user may freely use side-effects (both special variables and *throw) and need not worry about errors in his program "clobbering the system."

1.1 Menu Facility

A menu is an array of choices, each identified by a word or short phrase. You can select one of the choices by moving the mouse near it, which causes it to be highlighted (a box appears around it), and then clicking any mouse button.

What happens when you select one of the choices depends on the particular type of menu. Typically the choices in a menu might be commands to some program or choices for what a command should operate upon.

The system automatically chooses the arrangement of the choices and the size and shape of the window. Naturally there are ways for the user to control this if necessary.

For an example of a menu, click the right-hand mouse button twice, causing the System Menu to appear.

1.1.1 Menu Items

A menu has a list of items; each item represents one of the choices you have. An item tells the menu what to display and what to do if the user selects (clicks on) it. "What to do" specifies both what value to return and a possible side-effect.

Response to selection of an item is implemented by the :execute message, which is always sent in the user process (rather than the mouse process). Thus side-effects occur in the appropriate process. The returned value comes back to the user from tv:menu-choose, :choose, or :execute depending on how the menu is used. This will be explained in detail later.

An item can take any of the following forms:

a string or a symbol
: The string or symbol is both what is displayed and what is returned. There are no side-effects.

a cons
: This is like an assq-list entry. The car is a string or symbol to display and the cdr is what to return. The cdr must be atomic to distinguish this case from the remaining ones. There are no side-effects.

a list (*name value*)
: Another form of assq-list entry. *name* is a string or a symbol to display, and *value* is any arbitrary object to return. There are no side-effects.

a list (*name type arg option1 arg1 option2 arg2...*)
: This is the most general form. *name* is a string or a symbol to display. *type* is a keyword symbol specifying what to do, and *arg* is an argument to it. The *options* are keyword symbols specifying additional features desired, and the *args* following them are arguments to those options.

The types of menu item are:

:value
: *arg* is what to return. There are no side-effects.

:eval
: *arg* is a form to be evaluated. Its value is returned.

:funcall	*arg* is a function of no arguments to be called. The value it returns is returned.
:no-select	This item cannot be selected. Moving the mouse near it will *not* cause it to be highlighted. This is useful for putting comments, headings, and blank spaces into menus. *arg* is ignored, but must be present to make the item be the form that has a *type* keyword in it.
:kbd	*arg* is sent to the selected window via the :force-kbd-input message. Typically it is either a character code which is to be treated as if it was typed in from the keyboard, or a list which is a command to the program. See section 1.1.5, page 8.
:menu	*arg* is a new menu to choose from; it is sent a :choose message and the result is returned. Normally *arg* would be a pop-up menu. If *arg* is a symbol it gets evaluated.
:buttons	*arg* is a list of three menu items. The item actually chosen (i.e. the item to be executed) is one of these three, depending on which mouse button was clicked. The order in the list is (*left middle right*).
:window-op	*arg* is a function of one argument. The argument is a list of three elements: the window the mouse was in before this menu was popped-up and the X and Y coordinates of the mouse at that time. See page 6.

The menu item modifier keywords are:

:font	This keyword is followed by a font or a symbol which is the name of a font. The item is displayed in that font instead of the menu's default font.
:documentation	This keyword is followed by a string which briefly describes this menu item. When the mouse is pointing at this item, such that it is highlighted, the documentation string will be displayed in the documentation line at the bottom of the screen.

1.1.2 Easy Menu Interface

tv:menu-choose *item-list* &optional *label near-mode default-item*

item-list is a list of items as described above. It can also be thought of as a Lisp a-list. This function pops up a menu and allows the user to make a choice with the mouse. When the choice is made, the menu disappears and the chosen item is executed. The value of that item is returned.

If the user moves the mouse out of the menu and far away, it pops down without making any choice and this function returns nil.

label is a string to be displayed at the top of the menu, or nil (the default) to specify the absence of a label.

near-mode is where to put the menu. It defaults to the list (:mouse) and must be an acceptable argument to tv:expose-window-near.

default-item is the item over which the mouse should be positioned initially. This allows the user to select that item without moving the mouse. If *default-item* is nil or unspecified, the mouse is initially positioned in the center of the menu.

1.1.3 Geometry

A menu has something called its *geometry*, which is what controls the size and shape of the menu and the arrangement of the displayed choices. The creator of a menu may specify some aspects of the geometry explicitly, while leaving other aspects free to be chosen by the system according to its esthetic sense.

There are two ways the choices can be displayed. They can be in an array of rows and columns, or they can be "filled", that is, as many to a line as will fit with a reasonable amount of white space in between. Filled format is specified by giving zero as the number of columns.

The geometry is represented as a list of six elements:

columns	The number of columns (0 for filled format).
rows	The number of rows.
inside width	The inside-width of the window, in bits. If the user explicitly sets the size or edges of the window, it will be remembered here and act as a constraint on the menu from then on.
inside height	The inside-height of the window, in bits. If the user explicitly sets the size or edges of the window, it will be remembered here and act as a constraint on the menu from then on.
maximum width	The maximum width of the window, in bits. The system will prefer to choose a tall skinny shape rather than exceed this.
maximum height	The maximum height of the window, in bits. The system will prefer to choose a short fat shape rather than exceed this. If both the maximum width and the maximum height are effective, the system will display only some of the menu items and enable scrolling to make the rest accessible.

If an element of the geometry is nil, this means that it is unspecified and the system may choose it. The default geometry is all nil. The default shape is an upright golden rectangle, using rows-and-columns form with as many columns as required. Most small menus will have only one column.

When the size, shape, or item-list of a menu is changed, the unspecified portions of the geometry will be recomputed. Explicit setting of the size or shape (by sending the standard messages for this purpose) is remembered in the geometry.

The following init-plist options to a menu will initialize the geometry:

`:geometry` *list* (Init Option for tv:menu)
Sets the complete geometry.

`:rows` *n-rows* (Init Option for tv:menu)
Sets the number of rows.

`:columns` *n-columns* (Init Option for tv:menu)
Sets the number of columns.

`:fill-p` *t-or-nil* (Init Option for tv:menu)
Specifies whether to use filled format.

`:default-font` *font* (Init Option for tv:menu)
Sets the default font, the font in which items which do not specify a font are displayed.

The following messages may be sent to any flavor of menu to manipulate its geometry:

`:geometry` (to tv:menu)
Returns a list of six things, the menu's geometry. These are the constraints, with nil in unspecified positions; contrast :current-geometry.

`:current-geometry` (to tv:menu)
Returns a list of six things, which are the geometry corresponding to the actual current state of the menu. Only the *maximum width* and *maximum height* can be nil. Constrast this with :geometry.

`:set-geometry` &optional *columns rows inside-width inside-height max-width max-height* (to tv:menu)
Note that this takes six arguments rather than a list of six things as you might expect. This is because you frequently want to omit most of the arguments. The geometry is set from the arguments, which can cause the menu to change its shape and redisplay. An argument of nil means to make that aspect of the geometry unconstrained. An omitted argument or an argument of t means to leave that aspect of the geometry the way it is.

`:fill-p` (to tv:menu)
`:set-fill-p` *t-or-nil* (to tv:menu)
Get or set the menu's fill mode, t if it displays in filled form rather than columnar form. This is a special case of the :geometry/:set-geometry messages.

`:set-default-font` *font* (to tv:menu)
Sets the default font, the font in which items which do not specify a font are displayed. This recomputes the geometry.

1.1.4 Ordinary Menus

These are the *basic* and *mixin* flavors for the ordinary kinds of menus. They cannot be instantiated themselves but are useful to know about. Other kinds of menus are discussed in later sections.

tv:basic-menu *Flavor*
Everything else is built on this.

tv:basic-momentary-menu *Flavor*
This is a kind of menu which is only momentarily on the screen, often referred to as a "pop up" menu. A choose operation on a menu of this flavor causes it to position itself where the mouse is. When the user selects an item in the menu, or alternatively moves the mouse far away from the menu, the menu disappears and deactivates.

tv:window-hacking-menu-mixin *Flavor*
Provides for the :window-op item type.

These are the interesting instantiatable menu flavors:

tv:menu *Flavor*
This is tv:basic-menu with borders and a label on top. The default is for there to be no label but you can specify one with the :label init-plist option or the :set-label message.

tv:momentary-menu *Flavor*
This is tv:basic-momentary-menu mixed with the right other flavors. Momentary menus were described at the beginning of this section.

tv:pop-up-menu *Flavor*
This is *not* what is usually meant by a pop-up menu. It is a combination of tv:menu and tv:temporary-window-mixin, but does not have the automatic expose and deexpose features of tv:momentary-menu.

It is appropriate to use a pop-up menu rather than a momentary menu when you want to pop a menu up and make several choices from it before popping it back down, or if you don't want to allow the user the option of choosing nothing by moving the mouse out of the window.

tv:momentary-window-hacking-menu *Flavor*
A momentary menu with the window-hacking mixin.

tv:momentary-menu &optional (*superior* tv:mouse-sheet) (Resource)
This is a resource of momentary menus. tv:menu-choose allocates a window from this resource.

The following messages are useful to send to any flavor of menu. Also listed are init options which are useful with any flavor of menu. In addition to these general-purpose messages and init options, those which specifically have to do with the shape and arrangement of the menu are listed in the Geometry section (section 1.1.3, page 4).

:item-list (to tv:menu)
:set-item-list *item-list* (to tv:menu)
Get or set the list of items (choices). Setting the item-list recomputes the geometry and redisplays the menu.

:item-list *items* (Init Option for tv:menu)
The item-list can be set when the menu is created.

:choose (to tv:menu)
Exposes the menu if it is not already exposed, then waits for a selection to be made with the mouse. The selection is :execute'd and the resulting value is returned. A momentary menu will return nil from :choose if the mouse is moved far out of it, and in any case will pop down before returning.

:execute *item* (to tv:menu)
Given an item that was selected, performs the appropriate side-effects and returns the appropriate value. For most kinds of menus, this message is sent automatically as part of the :choose message, but command menus (see below) require the user program to send :execute explicitly.

:move-near-window *window* (to tv:menu)
Exposes the menu above or below the specified window, giving it the same width.

:center-around *x y* (to tv:menu)
Exposes the menu, putting its center or the center of the last item chosen at those coordinates in the superior. If this would cause the menu to stick outside of its superior, it is offset slightly to keep it inside. The actual coordinates of the center of the appropriate item are returned (you might want to put the mouse there). Momentary menus use this to put the menu in such a place that the mouse will be right over the last item chosen.

:current-item (to tv:menu)
Get the item the mouse is currently pointing at (nil if none). In most cases if you are using this message you are doing something wrong.

:chosen-item (to tv:menu)
:set-chosen-item *item* (to tv:menu)
Get or set the item which has been chosen by the mouse and is being communicated back to the controlling process. In most cases if you are using these messages you are doing something wrong.

:last-item (to tv:menu)
:set-last-item *item* (to tv:menu)
Get or set the item which was chosen by the mouse the last time this menu was used. When a momentary menu is exposed near the mouse by the :choose message, it will put the mouse over this item so that it easy to choose it again.

:column-row-size (to tv:menu)
Returns two values: the width of a column in bits and the height of a row in bits.

:item-cursorpos *item* (to tv:menu)
Returns two values like :read-cursorpos giving the coordinates of the center of the displayed representation of *item*. The result is nil if the item is scrolled off the display.

:item-rectangle *item* (to tv:menu)
Returns four values, the coordinates of the rectangle enclosing the displayed representation of the specified item. The result is nil if the item is scrolled off the display. Note that the returned coordinates are *inside* coordinates and that they include a 1-pixel margin around the item.

:menu-draw (to tv:menu)
Draws the menu's display. This is a message so that daemons can be put on it. This message is automatically sent by the system when required, and should not be sent explicitly by the user.

1.1.5 Command Menus

tv:command-menu-mixin *Flavor*
This kind of menu is not operated by the :choose message. Instead, when the user selects an item, a command is sent to the controlling process through an io-buffer. The command is a list, (:menu *item button-mask window*). The controlling process should (funcall *window* ':execute *item*). This is useful for a menu which does not stand alone but is part of a frame. The controlling process can be looking in its io-buffer for commands from several windows as well as keyboard input.

tv:command-menu *Flavor*
This is tv:command-menu-mixin mixed with tv:menu to make it instantiatble.

:io-buffer (to tv:command-menu)
:set-io-buffer *io-buffer* (to tv:command-menu)
These messages get or set the io-buffer to which a command-menu sends a command when an item is selected.

:io-buffer *buf* (Init Option for tv:command-menu)
The io-buffer to be used by a command menu is usually specified when it is created.

tv:command-menu-abort-on-deexpose-mixin *Flavor*
When a command menu built on this flavor is deexposed, it automatically clicks on its ABORT button. In other words, when such a menu receives the :deexpose message, it searches its item list for an item whose displayed representation is "ABORT". If such an item is found, a blip is sent to the io-buffer indicating that that item was clicked upon with the Left button.

1.1.6 Dynamic Item List Menus

tv:dynamic-item-list-mixin *Flavor*

Provides for a form which is evaluated to get the menu's item-list, kept in the tv:item-list-pointer instance variable. This form is evaluated at appropriate times (for instance when the :choose message is sent) to check whether the item-list should change.

:update-item-list (to tv:dynamic-...-menu)

This message is only accepted by menus with the dynamic item-list mixin. It sends a :set-item-list if one is necessary. The menu sends itself this message automatically at appropriate times.

:item-list-pointer *form* (Init Option for tv:dynamic-...menu)

form is saved and evaluated periodically to get the item-list for the menu. *form* is usually a special variable but any Lisp form is legal. The evaluation may occur in an arbitrary process, so only global variables should be accessed.

These are menu flavors which are just combinations of this with other flavors:

tv:dynamic-momentary-menu *Flavor*

A momentary menu with the dynamic item-list mixin.

tv:dynamic-momentary-window-hacking-menu *Flavor*

A momentary menu with both the dynamic item-list mixin and the window-hacking mixin.

tv:dynamic-pop-up-menu *Flavor*

A pop-up menu with the dynamic item-list mixin.

tv:dynamic-pop-up-command-menu *Flavor*

A command menu with the pop-up and dynamic item-list mixins.

tv:dynamic-pop-up-abort-on-deexpose-command-menu *Flavor*

A command menu with the pop-up, abort-on-deexpose, and dynamic item-list mixins.

1.1.7 Multiple Menus

tv:menu-highlighting-mixin *Flavor*

Provides for some of the menu items to be highlighted with inverse video. This is typically used with menus of "modes", where the modes currently in effect are highlighted. The menu items corresponding to modes will typically be set up so that when executed, they adjust the highlighting to reflect the enabling or disabling of a mode.

tv:multiple-menu-mixin *Flavor*

Gives a menu the ability to have multiple items "selected". Selected items are highlighted with inverse video, using the above highlighting mixin. Clicking on an item merely complements its selected state and does not execute it nor return from the :choose message.

In addition, at the top of the menu, in italics, are displayed some "special choices" which cannot be highlighted. Clicking on one of these behaves the same as clicking on an item of an ordinary menu. By default the only special choice is *Do It*, which returns a list of the results of executing all the highlighted choices (i.e. the result of the :highlighted-values message). You can define your own special choices with the :special-choices init-plist option, or get rid of them entirely by giving nil as the argument to this option.

tv:multiple-menu *Flavor*
A menu which behaves as described above. This is a combination of tv:multiple-menu-mixin with tv:menu.

tv:momentary-multiple-menu *Flavor*
A multiple-menu which is momentary.

The following messages and init-plist options pertain to these flavors of menus:

:highlighted-items (to tv:menu-highlighting-mixin)
:set-highlighted-items *list* (to tv:menu-highlighting-mixin)
Get or set the list of items which are highlighted. These messages are accepted only by menus with the menu-highlighting mixin.

:highlighted-items *items* (Init Option for tv:menu-highlighting-mixin)
When a menu with the menu-highlighting mixin is created, the list of items to be initially highlighted may be specified. The default is nil.

:add-highlighted-item *item* (to tv:menu-highlighting-mixin)
:remove-highlighted-item *item* (to tv:menu-highlighting-mixin)
These messages, accepted only by menus with the highlighting mixin, are used to highlight or un-highlight an item.

:highlighted-values (to tv:menu-highlighting-mixin)
:set-highlighted-values *list* (to tv:menu-highlighting-mixin)
:add-highlighted-value *value* (to tv:menu-highlighting-mixin)
:remove-highlighted-value *value* (to tv:menu-highlighting-mixin)
These messages are similar to the preceding four, except that instead of referring to items directly you refer to their values, i.e. the result of executing them. For instance if your item-list is an association list, with elements (*string* . *symbol*), these messages use *symbol*. This only works for menu items that can be executed without side-effects, not the :eval, :funcall, etc. kinds.

:special-choices *choice-list* (Init Option for tv:menu-highlighting-mixin)
Each element of *choice-list* specifies a menu item for a multiple-menu. These are the items which behave like normal menu items; the items from the :item-list init option behave as on/off switches as described above. An element of *choice-list* may be any form of menu item.

1.2 Multiple Choice Facility

The *Multiple Choice* facility provides a window containing a bunch of items, one per text line. For each item, there can be several yes/no choices for the user to make. The window is arranged in columns, with headings at the top. The leftmost column contains the text naming each item. The remaining columns contain small boxes (called *choice boxes*). A "no" box has a blank center, while a "yes" box contains an "X". Pointing the mouse at a choice box and clicking the left button complements its yes/no state. Each choice can be initialized by the program to yes or no as appropriate for a default. Note that some items may not allow some choices, so there can be blank places in the array of choice boxes.

There can be constraints among the choices for an item. For example, if they are mutually exclusive then clicking one choice box to "yes" will automatically set the other choice boxes on the same line to "no".

For an example of a multiple-choice window, try the Kill or Save Buffers operation in the editor menu.

There are several parameters associated with a multiple-choice window:

The *item-name* is a string which is the column heading for the leftmost column.

The *item-list* is a list of representations of items. Each element is a list, (*item name choices*). *item* is any arbitrary object. *name* is a string which names that object; it will be displayed on the left on the line of the display devoted to this item. *choices* is a list of keywords representing the choices the user can make for this item. Each element of *choices* is either a symbol, *keyword*, or a list, (*keyword default*). If *default* is present and non-nil, the choice is initially "yes"; otherwise it is initially "no".

The *keyword-alist* is a list defining all the choice keywords allowed. Each element takes the form (*keyword name*). *keyword* is a symbol, the same as in the *choices* field of an *item-list* element. *name* is a string used to name that keyword. It is used as the column heading for the associated column of choice boxes.

An element of *keyword-alist* can have up to four additional list elements, called *implications*. These control what happens to other choices for the same item when this choice is selected by the user. Each implication can be nil, meaning no implication, a list of choice keywords, or t meaning all other choices. The first implication is *on-positive*; it specifies what other choices are also set to "yes" when the user sets this one to "yes". The second implication is *on-negative*; it specifies what other choices are set to "no" when the user sets this one to "yes". The third and fourth implications are *off-positive* and *off-negative*; they take effect when the user sets this choice to "no". The default implications are nil t nil nil, respectively. In other words the default is for the choices to be mutually exclusive.

If the implications are not present, the defaults are rplacd'ed into the *keyword-alist* element.

The *finishing-choices* are the choices to go in the bottom margin. When the user clicks on one of these he is done. The variable tv:default-finishing-choices contains a reasonable default for this, providing *Do It* and *Abort* choices.

This is the easy interface to the multiple choice facility:

tv:multiple-choose *item-name item-list keyword-alist* &optional *near-mode maxlines*

Pops up a multiple-choice window and allows the user to make choices with the mouse. The dimensions of the window are automatically chosen for the best presentation of the specified choices. If there are too many choices, scrolling of the window is enabled.

item-name, *item-list*, and *keyword-alist* are as described above. *finishing-choices* cannot be specified and is always the default.

When the user clicks on one of the two finishing choices in the bottom margin (*Do It* and *Abort*) the window disappears and tv:multiple-choose returns. If the user finishes by choosing *Abort* the returned value is nil. If the user chooses *Do It*, the returned value is a list with one element for each item. Each element is a list whose car is the *item* (that arbitrary object which the user passed in in the *item-list* argument) and whose cdr is a list of the keywords for the "yes" choices selected for that item.

near-mode tells the window where to pop up. It is a suitable argument for tv:expose-window-near. The default is the list (:mouse). *maxlines*, which defaults to twenty, is the maximum number of choices allowed before scrolling is used.

These are the grubby details:

tv:basic-multiple-choice *Flavor*

This is the *basic* flavor which makes a window implement the multiple-choice facility. Like most basic mixins, it is not itself instantiatable but it does commit any window that incorporates it to being a multiple-choice rather than any different sort of window. tv:basic-multiple-choice is built out of tv:text-scroll-window.

tv:multiple-choice *Flavor*

This is a reasonable window with the multiple-choice facility in it. It has borders and a label area on top which is used for the column headings.

tv:temporary-multiple-choice-window *Flavor*

This is a multiple-choice window which is equipped to pop up temporarily.

tv:temporary-multiple-choice-window &optional (*superior tv:mouse-sheet*) (Resource)

This is a resource of pop-up multiple-choice windows, used by the tv:multiple-choose function.

The following messages are useful to send to a multiple-choice window:

:setup *item-name keyword-alist finishing-choices item-list* &optional *maxlines*
(to tv:multiple-choice)

This message sets up all the various parameters of the window. Usually one sends this message while the window is deexposed. The window decides what size it should be and whether all the items will fit or scrolling is required, then draws the display into its bit-array. Thus when the window is exposed the display will appear instantaneously.

maxlines is the maximum number of lines the window may have; if there are more items than this only some of them will be displayed and scrolling will be enabled. *maxlines* defaults to 20.

:choose &optional *near-mode* (to tv:multiple-choice)

Moves the window to the place specified by *near-mode*, which defaults to the list (:mouse), and exposes it. Then waits for the user to make a finishing choice and returns the window to its original activate/expose status before the :choose. This message returns the same value as the function tv:multiple-choose.

1.3 Choose Variable Values Facility

This facility presents the user with a display of a bunch of Lisp variables and their values. The user may change the value of some of the variables. When the values are to his liking he may indicate that he is done.

Each line of the display corresponds to one variable. The name of the variable, a colon, and the value of the variable are displayed. Pointing the mouse at the value causes a box to appear around it. Clicking the left mouse button at that point allows the value to be changed.

For an example of a choose-variable-values window, try the Frame option of the Split Screen command in the system menu.

Each variable has a *type* which controls what values it may take on. The way the value is displayed and the way the user enters a new value depend on the type. The type mechanism is extensible and is described in detail later. The types fall into two categories, those with a small number of legal values and those with a large or infinite number of legal values. The first kind of type displays all the choices, with the one which is the current value of the variable in boldface. Pointing at a choice and clicking the mouse sets the variable to that value. Those types with a large number of legal values display the current value. Pointing at the value and clicking the mouse allows a new value to be entered from the keyboard. Rubbing out more characters than typed in restores the original value instead of changing it.

All variables whose values are to be chosen must be declared special, so that they are represented by Lisp symbols and can be accessed non-locally to the user's program. The syntax for input and output is controlled by the binding of base, ibase, *nopoint, prinlevel, prinlength, package, and readtable as usual.

Each line of the display is represented by an *item*, which can be one of the following:

a string — The string is simply displayed. This is useful for putting headings and blank separating lines into the display.

a symbol — The symbol is a variable whose type is :sexp; that is, its value may be any Lisp object. The name of the variable on the display is simply its print-name.

a list (*variable name type args...*)
This is the general form. *variable* is the variable whose value is being chosen. *name* is optional; if it is omitted it defaults to the print-name of *variable*. If *name* is supplied it can be a string, which is displayed as the name of the variable, or it can be nil, meaning that this line should have no variable name, but only a value. *type* is an optional keyword giving the type of variable; if omitted it defaults to :sexp. *args* are possible additional specifications dependent on *type*. It is possible to omit *name* and supply *type* since one is always a string and the other is always a symbol.

For clarification of this, refer to the examples on page 17.

The following are the types of variables supported by default, along with any *args* that may be put in the item after the *type* keyword:

:sexp — The value is any Lisp expression (sometimes called an S-expression), printed with prin1, read with read.

:princ — Same as :sexp except that the value is printed with princ rather than prin1.

:string — The value is a string, printed with princ, read with readline.

:number — The value is a number (either fixed or floating). It is printed with prin1 and read with read, but only a number is accepted on type-in.

:number-or-nil
The value may be either a number or nil.

:date — The value is a universal date-time. It is printed with time:print-universal-time and read with readline and time:parse-universal-time.

:character — The value is a fixnum which is a character code. It is printed as the character name (using the ~:@C format operator), and is read as a single keystroke.

:character-or-nil
Like :character but nil is also allowed as the value. nil displays as "none" and can be input via the Clear Input key.

:string-list — The value is a list of strings, whose printed representation for input and output consists of the strings separated by commas and spaces.

:choose *values-list print-function*
The value of the variable must be one of the elements of the list *values-list*. Comparison is by equal rather than eq. All the choices are displayed, with the current value in boldface. A new value is input by pointing to it with the mouse and clicking. *print-function* is the function to print a value; it is optional and defaults to princ.

:assoc *values-list print-function*
Like :choose but car of each element of *values-list* is what to display, while cdr is the value that goes in the variable.

:menu-alist *item-list*
Like :choose, but instead of a list of values there is *item-list*, which is a list of menu items (see section 1.1, page 2). The usual menu mechanisms for specifying the string to display, the value to return, and the mouse documentation work with this.

:boolean — The value of the variable is either t or nil. The choices are displayed as yes and no.

:documentation *doc type args...*
This is not really a variable type, but goes in the place where a type would normally be expected. The real type is *type*; it and its *args* are optional as usual. *doc* is a string which is displayed in the mouse documentation line when the mouse is pointing at this item. The default if no documentation is supplied in this way depends on the type, and generally is something like "Click left to input a new value from the keyboard."

A choose-variable-values window optionally may have an associated function, which is called whenever a variable's value is changed. This function can implement constraints among the variables. It is called with arguments *window*, *variable*, *old-value*, and *new-value*. The function should return nil if just the original variable needs to be redisplayed, or t if no redisplay is required; in this case it would usually setq several of the variables then send a :refresh message to the window.

The system chooses the dimensions of the window, and enables scrolling if there are too many variables to fit in the chosen height.

tv:choose-variable-values *variables* &rest *options*

This is the easy-to-use function interface to the choose-variable-values facility. It pops up a window displaying the values of the specified variables and permits the user to alter them. One or more choice boxes (as in the multiple-choice facility) appear in the bottom margin of the window. When the user clicks on the *Exit* choice box the window disappears and this function returns. The value returned is not meaningful; the result is expressed in the values of the variables.

variables is a list whose elements can be special variables or the more general items described above. See the examples below.

options is the usual list of alternating option keywords and argument values. The following option keywords are allowed:

:label	The argument is a string which is the label displayed at the top of the window. The default is "Choose Variable Values".
:function	The function to be called if the user changes the value of a variable. The default is nil (no function).
:near-mode	Where to position the window. This is a suitable argument for tv:expose-window-near. The default is the list (:mouse).
:width	Specifies how wide to make the window. This can be a number of characters, or a string (it is made just wide enough to display that string). The default is to make it wide enough to display the current values of all the variables, provided that isn't too wide to fit in the superior.
:extra-width	When :width is not specified, this specifies the amount of extra space to leave after the current value of each variable of the kind that displays its current value (rather than a menu of all possible values). This extra space allows for changing the value to something bigger. The extra space is specified as either a number of characters or a character string. The default is ten characters. If :width is specified, then :extra-width is ignored.
:margin-choices	The argument is a list of specifications for choice boxes to appear in the bottom margin. Each element can be a string, which is the label for the box which means "done", or a cons of a label string and a form to be evaluated if that choice box is clicked upon. Since this form is evaluated in the user process it can do such things as alter the values of variables or *throw out. The default for :margin-choices is ("Exit").
:superior	The argument is the window to which the pop-up choose-variable-values window should be inferior. The default is the value of tv:mouse-sheet, or the superior of *w* if *near-mode* is (:window *w*).

Here are some examples of how to call tv:choose-variable-values. The simplest sort of thing you can do is:

```
      (tv:choose-variable-values '(base ibase *nopoint)
                                 ':label "Number format parameters")
```

which displays the three variables' names and values and lets the user change them. The same example can be done with nicer formatting with:

```
      (tv:choose-variable-values
              '((base "Output Base" :number)
                (ibase "Input Base" :number)
                (*nopoint "Decimal Point"
                          :assoc (("Yes" . nil)
                                  ("No" . t))))
              ':label "Number format parameters")
```

The entry for *nopoint would have been simply

```
              (*nopoint "No Decimal Point" :boolean)
```

except that we wanted to reverse the sense of t and nil. We might even have used

```
              (*nopoint :boolean)
```

if we wanted to use the name of the variable as the label rather than spelling it out.

For a hokier example, consider a grocery store. Suppose we have variables *cuts-of-beef*, *cuts-of-pork*, *cuts-of-lamb*, and *lettuce-types* which contain lists of strings indicating what is available, *squash-type* which indicates whether we stock summer squash or winter squash, and *milk-price* which contains a floating-point number which is the current price of a gallon of milk. Then the following expression would display the inventory and allow it to be modified, using several different kinds of items:

```
      (tv:choose-variable-values
              '("Meat Department"
                (*cuts-of-beef* "Beef" :string-list)
                (*cuts-of-pork* "Pork" :string-list)
                (*cuts-of-lamb* "Lamb" :string-list)
                ""
                "Produce"
                (*lettuce-types* "Lettuce" :string-list)
                (*squash-type* "Squash" :choose ("Summer" "Winter"))
                ""
                "Dairy"
                (*milk-price* "Milk"
                      :documentation
                          "Click left to raise the price of milk"
                      :number)))
```

Note the use of strings to provide labels for the sections, and null strings to separate the sections with blank lines.

1.3.1 User Option Facility

There is a facility, based on the Choose-Variable-Values facility, for keeping track of options to a program of the sort that a user would specify once and keep in his init file. Special forms are provided for defining options, and there are functions for putting all the options into a choose-values window so that the user can alter them, for writing the current state of the options into an init file, and for resetting all the options to their default initial values.

define-user-option-alist *Special Form*

(define-user-option-alist *name*) defines *name* to be a global variable whose value is a "user option alist", something which may be used by the other functions below. This alist will keep track of all of the option variables for a particular program.

(define-user-option-alist *name constructor*) also specifies the name of a constructor macro to be defined, which provides a slightly different way of defining an option variable from define-user-option. The form (*constructor option default type name*) will define an option in this user-option-alist. The arguments are the same as to define-user-option.

define-user-option *Special Form*

(define-user-option (*option alist*) *default type name*) defines the special variable *option* to be an option in the *alist*, which must have been previously defined with define-user-option-alist. The variable is declared and initialized via (defvar *option default*). The value of the form *default* is remembered so that the variable can be reset back to it later.

type is the type of the variable for purposes of the choose-variable-values facility. It is optional and defaults to :sexp.

name is the name of the variable to be displayed in the choose-variable-values window. It is optional and defaults to a string which is the print-name of the variable except with hyphens changed to spaces and each word changed from all-upper-case to first-letter-capitalized. If the first and last characters of the print-name are asterisks, they are removed. E.g. the default name for so:*sunny-side-up* would be "Sunny Side Up".

choose-user-options *alist* &rest *options*

Displays the values of the option variables in *alist* to the user and allows them to be altered. The *options* are passed along to tv:choose-variable-values.

reset-user-options *alist*

Each of the option variables in *alist* is reset to its default initial value.

write-user-options *alist stream*

For each option variable in *alist* whose current value is not equal to its default initial value, a form is printed to *stream* which will set the variable to its current value. The form uses login-setq so it is appropriate for putting into an init file.

1.3.2 Defining Your Own Variable Type

:decode-variable-type *kwd-and-args* (to tv:basic-choose-variable-values)

The system sends this message to a choose-variable-values window when it needs to understand an item. *kwd-and-args* is a list whose car is the keyword for the item and whose remaining elements, if any, are the arguments to that keyword. Six values are returned; these values are described below. The default method for :decode-variable-type looks for two properties on the keyword's property list:

tv:choose-variable-values-keyword
: The value of this property is a list of the six values described below. Unnecessary values of nil may be omitted at the end.

tv:choose-variable-values-keyword-function
: The value of this property is a function which is called with one argument, *kwd-and-args*. The function must return the six values.

You may add a new variable type to the standard set by putting one of the above properties on the keyword. You may define your own flavor of choose-variable-values window and give it a :decode-variable-type method to make it not use the standard variable types. This method must take care of implementing the :documentation keyword, which can appear in an item where a variable type would normally appear.

The six magic values are:

print-function A function of two arguments, object and stream, to be used to print the value. prin1 is acceptable.

read-function A function of one argument, the stream, to be used to read a new value. read is acceptable. If nil is specified, there is no read-function and instead new values are specified by pointing at one choice from a list. If the *read-function* is a symbol, it is called inside a rubout-handler, and over-rubout will automatically leave the variable with its original value. If *read-function* is a list, its car is the function, and it will be called directly rather than inside a rubout-handler.

choices A list of the choices to be printed, or nil if just the current value is to be printed.

print-translate If there are choices, and this function is supplied non-nil, it is given an element of the choice list and must return the value to be printed.

value-translate If there are choices, and this function is supplied non-nil, it is given an element of the choice list and must return the value to be stored in the variable.

documentation A string to display in the mouse documentation line when the mouse is pointing at this item. This string should tell the user that clicking the mouse will change the value of this variable, and any special information (e.g. that the value must be a number).

Alternatively, this can be a symbol which is the name of a function. It will be called with one argument, which is the current element of *choices* or the current value of the variable if *choices* is nil. It should return a documentation string or nil if the default documentation is desired. This can be useful when you want to document the meaning of a particular choice, rather than simply saying that clicking the mouse on this choice will select it. Note that the function should return a constant string, rather than building one with **format** or other string operations, because it will be called over and over as long as the mouse is pointing at an item of this type. The function is called by the who-line updating in the scheduler, not in the user process.

1.3.3 Making Your Own Window

tv:basic-choose-variable-values *Flavor*

This is the *basic* flavor which makes a window implement the choose-variable-values facility. It is built out of tv:text-scroll-window.

There are two ways to use this. One can create a window giving all of the parameters in the init-plist, or one can create a window without specifying the parameters then send the :setup message (see below).

The following init-plist options are relevant:

:function *fcn* (Init Option for tv:basic-choose-variable-values)

The function called when the value of a variable is changed. The default is **nil** (no function).

:variables *item-list* (Init Option for tv:basic-choose-variable-values)

The list of variables whose values are to be chosen. These can be either symbols which are variables, or the more general *items* defined above.

:stack-group *sg* (Init Option for tv:basic-choose-variable-values)

The stack group in which the variables whose values are to be chosen are bound. The window needs to know this so that it can get the values while running in another process, for instance the mouse process, in order to update the window display when it is refreshed or scrolled. This option is required, unless you use the :setup message.

:name-font *font* (Init Option for tv:basic-choose-variable-values)

The font in which names of variables are displayed. The default is the system default font.

:value-font *font* (Init Option for tv:basic-choose-variable-values)

The font in which values of variables are displayed. The default is the system default font.

:string-font *font* (Init Option for tv:basic-choose-variable-values)

The font in which items which are just strings (typically heading lines) are displayed. The default is the system default font.

:unselected-choice-font *font* (Init Option for tv:basic-choose-variable-values)

The font in which choices for a value, other than the current value, are displayed. The default is a small distinctive font.

:selected-choice-font *font* (Init Option for tv:basic-choose-variable-values)
The font in which the current value of a variable is displayed, when there is a finite set of choices. This should be a bold-face version of the preceding font. The default is the bold-face version of the default unselected-choice font.

If no dimensions are specified in the init-plist, the width and height will be automatically chosen according to the other init-plist parameters. The height is dictated by the number of elements in the *item-list*. Specifying a height in the init-plist, using any of the standard dimension-specifying init-plist options, overrides the automatic choice of height.

tv:choose-variable-values-window *Flavor*
This is a choose-variable-values window with a reasonable set of features, including borders, a label at the top, stream i/o, the ability to be scrolled if there are too many variables to fit in the window, and the ability to have choice boxes in the bottom margin.

This additional init-plist option is allowed:

:margin-choices *choice-list* (Init Option for tv:choose-variable-values-window)
The default is a single choice box, labelled "Done". See page 26 for the details of what you can put here. Note that specifying nil for this option will suppress the margin-choices entirely.

tv:choose-variable-values-pane *Flavor*
A tv:choose-variable-values-window that can be a pane of a constraint-frame. It will not change its size automatically; the size is assumed to be controlled by the superior.

tv:temporary-choose-variable-values-window *Flavor*
A tv:choose-variable-values-window that is equipped to pop up temporarily.

The following messages are useful to send to a choose-variable-values window:

:setup *items label function margin-choices* (to tv:choose-variable-values-window)
Changes the list of items (variables), the window label, the constraint function, and the choices in the bottom margin and sets up the display. Also remembers the current stack-group as the stack-group in which the variables are bound. If the window is not exposed this chooses a good size for it.

:set-variables *item-list* &optional *dont-set-height* (to tv:choose-variable-values-window)
Changes the list of items (variables) and redisplays. Unless *dont-set-height* is supplied non-nil, the height of the window will be adjusted according to the number of lines required. If more than 25. lines would be required, 25. lines will be used and scrolling will be enabled. The :setup message uses :set-variables to do part of its work.

:appropriate-width &optional *extra-space* (to tv:choose-variable-values-window)
Returns the inside-width appropriate for this window to accomodate the current set of variables and their current values. Send this message after a :setup and before a :expose, and use the result to do a :set-inside-size. The returned width will not be larger than the maximum that will fit inside the superior.

If *extra-space* is supplied, it specifies the amount of extra space to leave after the current value of each variable of the kind that displays its current value (rather than a menu of all possible values). This extra space allows for changing the value to something bigger. The extra space is specified as either a number of characters or a character string. The default is to leave no extra space.

:redisplay-variable *variable* (to tv:choose-variable-values-window)
Redisplays just the value of that variable.

In the simplest mode of operation, you call the tv:choose-variable-values function which takes care of creating the window and all necessary communication with it. When you make your own choose-variable-values window, you need to handle the communication yourself, using the information given below. An example of a situation in which this is necessary is when you have a frame, some panes of which are choose-variable-values windows.

A choose-variable-values window has an io-buffer, which it uses to send commands (also known as "blips") back to its controlling process. As usual these commands are lists, to distinguish them from keyboard characters which are numbers. If all panes send to the same io-buffer, then when one of these commands arrives it can be processed in the appropriate pane. At the same time, the controlling process can be looking in the io-buffer for other commands from other panes and for input from the keyboard. Compare this with command-menus (see section 1.1.5, page 8).

:io-buffer *buf* (Init Option for tv:choose-variable-values-window)
The io-buffer to be used.

The following io-buffer commands are used:

(:variable-choice *window item value line-no*)
Indicates that the user clicked on the value of a variable, expressing the desire to change it.

(:choice-box *window box*)
Indicates that the user clicked on one of the choice boxes in the bottom margin.

tv:choose-variable-values-process-message *window command*
This function implements the proper response to the above commands. It should be called in the process and stack-group in which the variables being chosen are bound. The function returns t if the command indicates that the choice operation is "done", otherwise it performs the appropriate special action and returns nil. If *command* is a character, it is ignored unless it is #\clear-screen, in which case the choose-variable-values window is refreshed.

tv:temporary-choose-variable-values-window &optional (*superior tv:mouse-sheet*)
(Resource)
This is a resource of windows, from which tv:choose-variable-values gets a window to use.

1.4 Mouse-Sensitive Items

The mouse-sensitive items facility is a feature somewhat related to the choice facilities described above. It is similar in its appearance to the user, but quite different in the way it is interfaced to by a program. Mixing tv:basic-mouse-sensitive-items into a window flavor equips the window with mouse-handling according to the paradigm described in this section. Mouse-sensitive items are something you use when defining your own window, rather than a complete, stand-alone facility and consequently do not have an "easy to use" functional interface.

For an example of mouse-sensitive items, try the c-X c-B (List Buffers) command in the editor. Try moving the mouse over the list of buffers and clicking the right-hand button.

The word "typeout" appears here and there in the mouse-sensitive items facility for historical reasons. Often mouse-sensitive items are typed out on top of some other display, such as an editor buffer. However, the mouse-sensitive-item facility has nothing to do with the typeout-window facility (which is not in this document). At this point it would be a fairly big incompatible change to fix this.

tv:basic-mouse-sensitive-items *Flavor*

Mixing this flavor into a window provides for areas of the screen which are sensitive to the mouse. Moving the mouse into such an area highlights the area by drawing a box around it. At this point clicking the mouse performs a user-defined operation. This flavor is called *basic* because it usurps the handling of the mouse by the window; it will not work to mix it with another flavor that also expects to use the mouse. However it is less basic than many basic flavors in that it does not do anything special with the displayed image of the window.

A mouse-sensitive item has a *type*, which is a keyword which controls what you can do to it, an *item*, which is an arbitrary Lisp object associated with it, and a rectangular area of the window. Typically something is displayed in that area at the same time as a mouse-sensitive item is created, using normal stream output to the window. Unlike things such as menu items, these mouse-sensitive items are not a permanent property of the window; they are just as ephemeral as the displayed text and go away if you clear the window or if typeout wraps around and types over them. Of course, if you don't type out more items and text than fit in the window, and never clear the window, then they will be permanent.

Associated with each type is a set of operations that are legal to perform on items of that type. One of these operations is selected as the default. The tv:item-type-alist instance variable is an a-list which defines these. The car of each element is a *type* keyword, the cadr is the default operation, the caddr is a documentation string, and the cdddr is the list of all the operations (the default doesn't necessarily have to be a member of this list). The cdddr is actually a list of menu items, so typically each element is (*name* . *operation*) where the user sees the string *name* but the program identifies the operation by the symbol *operation*. In most cases *operation* is a function to be called, but it can be any atom. The tv:item-type-alist instance-variable can be initialized via the init-plist when the window is created, or it can be set by sending a message, in the usual way.

Clicking mouse-left on a mouse-sensitive item performs the default operation on it. Clicking mouse-right pops up a menu of all the operations, and if you select one performs it. Clicking mouse-right-twice calls the system menu. Other mouse clicks and clicking on an item whose type is not in the type alist are errors and cause a beep.

What performing an operation means is that a command is sent to the controlling process through the :force-kbd-input message to the window. This command is a list, (:typeout-execute *operation item*), where *operation* is the operation and *item* is the arbitrary object remembered by the mouse-sensitive item. What this really means, and how the *operation* is performed, is up to the application program using this facility.

tv:add-typeout-item-type *Special Form*

The special form

```
        (tv:add-typeout-item-type alist type name function
                                   default-p documentation)
```

is used to declare information about a mouse-sensitive item type by adding an entry to an a-list kept in a special variable. This a-list can then be put into the item-type alist of a mouse-sensitive window, for instance using the :item-type-alist init-plist option. Note that each possible operation on a particular mouse-sensitive item type is defined with a separate tv:add-typeout-item-type form; this allows each operation to be defined at the place in the program where it is implemented, rather than collecting all the operations into a separate table. It also allows new operations to be added in a modular fashion.

alist is the special variable which contains the a-list. You should defvar it to nil before defining the first item type. Each program that uses mouse-sensitive items has its own a-list of item types, so that there is no conflict in the names of the types. *type* is the keyword symbol for the type being defined. *name* is the string which names the operation and *function* is the representation of the operation, for instance the function to be called. *default-p* is optional; if it is supplied and non-nil, it means that this operation is the default performed when you click the left button on an item of this type. *documentation* is optional but highly recommended; it is a string which documents what *function* does. When the user points the mouse at an item of this type, the documentation line at the bottom of the screen will give the documentation for the default function (reachable by the left button) and a list of the functions in the menu (reachable by the right button). If the user clicks right, calling for a menu, then the documentation for whichever function in the menu he points the mouse at will be displayed.

alist, *type*, and *function* are not evaluated. *name*, *default-p*, and *documentation* are evaluated.

When *function* is a function, the tv:add-typeout-item-type form is typically placed right before the definition of *function* in the program source file.

The following messages are useful to send to a window with mouse-sensitive items:

:item *type item* &rest *format-args* (to tv:basic-mouse-sensitive-items)

Creates and displays a mouse-sensitive item of type *type* with associated object *item*. If *format-args* are supplied, they are a format control-string and arguments used to generate the display for this item. If *format-args* are not supplied, the display is generated by princ'ing *item*.

:primitive-item *type item left top right bottom* (to tv:basic-mouse-sensitive-items)

Creates a mouse-sensitive item of type *type* with associated object *item*. This does not display anything in the window. *left*, *top*, *right*, and *bottom* are the coordinates of a rectangular area of the window assumed to contain the display.

:item-list *type list* (to tv:basic-mouse-sensitive-items)

Creates and displays several mouse-sensitive items, all of the same type *type*. The items are displayed in a regular array with as many columns on a line as will fit. If the elements of *list* are atoms then they are the items and the display is generated by princ'ing them. Otherwise car of each element is the string to be displayed and cdr of each element is the *item*, i.e. *list* is an a-list.

:item-type-alist *alist* (Init Option for tv:basic-mouse-sensitive-items)

Remembers *alist* as the set of item types allowed in this window. *alist* should be created by tv:add-typeout-item-type.

1.5 Margin Choices

A window can be augmented with choice boxes (see page 11) in its bottom margin using the flavor tv:margin-choice-mixin. These give the user a few labelled mouse-sensitive points which are independent of anything else in the window. Thus margin-choices can be added to any flavor of window in a modular fashion.

Margin choices are not a complete, stand-alone choice facility and consequently do not have an "easy to use" functional interface.

For an example of a window with margin choices (as well as choice boxes in its interior), try the Kill or Save Buffers operation in the editor menu.

tv:margin-choice-mixin *Flavor*

Puts choice boxes in the bottom margin, according to a list of choice-box descriptors which can be specified with the :margin-choices init-plist option or the :set-margin-choices message. A choice-box descriptor is a list, (*name state function x1 x2*). It is legal to use a longer list as a choice-box descriptor and store your own data in the additional elements.

name is a string which labels the box. *state* is t if the box has an "X" in it, nil if it is empty. *x1* and *x2* are used internally to remember where the choices boxes are; it always spreads them out evenly.

function is a function which is called in a separate process if the user clicks on the choice box; it receives three arguments: the choice-box descriptor for the choice box, the "margin region" which contains the choice boxes, and the Y position of the mouse relative to this window. You probably want to ignore the last two arguments. When *function* is called, self is bound to the window and all its instance variables are bound to special variables. The structure access functions tv:choice-box-name and tv:choice-box-state may be of use inside *function* (they are just more specific names for car and cadr). If *function* changes the state of the choice box, it will need to refresh the choice boxes by doing

```
(funcall (tv:margin-region-function region) ':refresh region)
```

where *region* is its second argument, which is why that argument is passed.

tv:margin-choice-mixin is built on tv:margin-region-mixin; the position of the latter in the list of component flavors controls where in the margins the choice boxes appear. The default leaves tv:margin-region-mixin last, which puts the choice boxes inside any other margin elements such as borders, which is generally what you want.

:margin-choices *choices* (Init Option for tv:margin-choice-mixin)

choices is a list of choice-box descriptors, described above. A line of choice-boxes will appear in the bottom margin of the window. If *choices* is nil, there will be no choice boxes and no space for them in the bottom margin; however the window will still be capable of accepting the :set-margin-choices message to create a line of choice boxes later.

:set-margin-choices *choices* (to tv:margin-choice-mixin)

Changes the set of margin choices according to *choices*, which is nil to turn them off or a list of choice-box descriptors, described above. If the choice boxes are turned on or off, the size of the window's bottom margin will change accordingly.

Index