MENUS FOR SCHEME

by

Morton Goldberg
3268 Alpine Drive
Ann Arbor, MI 48108

CompuServe: 72346,565
ARPANET: goldberg@vmb.brl.mil

Programs with menu-driven user interfaces are quite common these days, and a large body of end-users ranging from novices to expert users show a preference for such user interfaces. In the MS-DOS world, Microsoft is attempting to establish its MS Windows package as the standard for such user interfaces but its use is by no means universal. In particular, Texas Instruments (TI) does not provide any kind of support for MS Windows with its PC Scheme language software. TI does provide its own window facility with PC Scheme but doesn't extend the facility to the menus, alerts, and dialogs commonly provided by menu-driven user interfaces such as MS Windows. ***Menus for Scheme*** is my attempt to fill this void.

Menus for Scheme is implemented using the SCOOPS (Scheme Object-Oriented Programming System), an extension to Scheme provided by TI. A basic familiarity of the terminology of object-oriented programming as it applies to SCOOPS is assumed in the body of this article. A glossary is supplied for those readers who are not familiar with such terminology.

THE FACILITIES PROVIDED BY *Menus for Scheme*

Menus for Scheme provides horizontal and vertical menus, alerts, and a simple form of dialog box.

An alert is a popup window presenting a message requiring no particular action on the part of the user except the action required to close the window. ***Menus for Scheme*** provides text-only alerts called popup text windows.

A dialog box is a popup window allowing the user to interact with a program in a more complex way than making a simple selection from a menu. ***Menus for Scheme*** provides only a very simple form of dialog box in which a message and a type-in area are presented to the user. Nominally the message prompts for some information that the user will then enter into the type-in area. In ***Menus for Scheme***, these simple dialogs are called popup query windows.

A menu is a window that pops-up on the screen and presents the user with two or more choices, called menu items. When one of the menu items is selected, it determines the next action taken by the program. The user may decide to cancel the menu without selecting any of the menu items.

Menus for Scheme implements two classes of menus: horizontal ones and vertical ones. The two classes differ only in their appearance on the screen. Horizontal menus present all their menu items on one line of the screen. Vertical menus present each of their menu items on a separate line of the screen. They are somewhat easier to lay out than horizontal menus, and they can hold more menu items.

Selecting Menu Items

A menu has a bar cursor can be moved to cover a menu item. The bar is moved by the arrow keys, but a selection is not made until the enter key is pressed. Selection can also be made by pressing a selector key, which can be any key on the keyboard that generates a single ASCII character. The selector is usually the first capital letter appearing in the text describing a menu item on the screen, but ***Menus for Scheme*** allows arbitrary association between a selector and a menu item. It even allows for multiple selector keys to

be associated with a single menu item. Pressing the escape key will always abort the item selection process and close a menu. In this case no selection is made.

Combining Dialogs and Alerts with Menus

When selecting a menu item results in an action that generates a user message requiring no response from the user, a popup text window is used to display the message. The text window is erased when the user presses the escape key.

When selecting a menu item results in an action requiring additional input from the user, this may be solicited by means of a subsidiary menu or by a popup query window, which is a popup window containing a query message and an input field for receiving the user's response. The input field is a string of spaces which may be displayed with text attributes that differ from the text attributes of the rest of the query window. A cursor appears within the input field, signaling its readiness to receive input. Destructive backspace is supported so the user can perform simple editing. A query window is erased when the user's response satisfies the query, or when the user presses the escape key.

AN EXAMPLE OF PC SCHEME CODE USING *Menus for Scheme*

At this point you might like to see the kind of PC Scheme a programmer would write when using ***Menus for Scheme***. Listing 1 shows an example—a toy program that demonstrates all the main capabilities of ***Menus for Scheme***. Figure 1 shows some of the screen output produced during the execution of Listing 1.

A BRIEF LOOK AT HOW *Menus for Scheme* IS IMPLEMENTED

Figure 2 shows the relationships among the classes forming the menu system. If you ignore class **key-monitor** for now, what you see is simple class inheritance the Smalltalk-80 kind. The most general class is **basic-popup-window**. This is a component class, i.e., it is not instantiable. It defines the instance variables and methods common to all types of popup windows.

Class **basic-popup-window** is has two major subclasses, **popup-text-window** and **basic-menu**. The only subclass of **popup-text-window** is the more specialized **popup-query-window**. **Basic-menu** is another component class, but its two subclasses, **vertical-menu** and **horizontal-menu** are instantiable. Now what about class **key-monitor**? Well, it is a class that can function both as a component class and as an instantiable class. It is not used as an instantiable class in ***Menus for Scheme***, but I have instantiated it in applications to provide command loops with single-key command accelerators.

Both class **key-monitor** and class **basic-popup-window** are declared as mixins to class **basic-menu**. This means menus are as much key monitors as they are popup windows, and they inherit behavior from both. To put it somewhat whimsically, consider menus to be the hybrid offspring of a mating between popup windows and key monitors. They show their key-monitor-like traits when they do input and their popup-window-like traits when they do output.

Menu System Windows

The menu system is built on PC Scheme's window system, which supports two kinds of window objects: graphics screens and text windows. The menu system uses text windows exclusively. These are output ports delivering character data to a constrained rectangular area of a display, which must be in text mode. A Scheme text window may be a popup window, a window which is expected to be on the screen for a short time only. In this case, Scheme saves the text that it will be overwritten, just before the popup window appears, and restores it when the popup window disappears. Class **basic-popup-window** provides the interface between the menu system and the Scheme-supplied popup window facility.

Popup Text Windows

SCOOPS supports active values for instance variables. Class **popup-text-window** is one of two places in the menu system where this feature is used. **Text**, the instance variable which stores a popup text window's message, has an active set-method. Whenever the contents of **text** are modified by a **set-text** message, the method **adjust-size** is called to resize the window to conform to its new text.

Popup Query Windows

Class **popup-query-window** uses class **popup-text-window** as a component. An instance of the class has an input field which accepts a response to the message it presents. When creating a new instance, an application needs to put entries for the instance variables **cursor-row**, **cursor-col**, and **input-width** on the init-list it passes to the popup query window constructor. These instance variables tell the new instance where to put the input field, how big it should be, and where to put the input cursor.

Class **popup-query-window** has a instance variable **text** which is similar to the instance variable of the same name in class **popup-text-window**. It holds the text of the message presented by the query window, and it has an active set-method. But **text** isn't inherited from class **popup-text-window**. An instance of class **popup-query-window** needs to call a different method, **add-input-field**, when the contents of **text** are modified. **Add-input-field** locates the string in **text** which goes on the line **cursor-row**, appends a string of spaces of width **input-width** to it, and calls the method **adjust-size**, inherited from class **popup-text-window**, to size the window to fit the modified version of the text.

When a popup query window appears on the screen, the instance variables **cursor-row** and **cursor-col** are accessed to position the input cursor at the beginning of the input field, and then the auxiliary procedure **readln** is called to handle the user's response. **Readln** keeps the characters typed-in by the user in a list maintained as a push-down stack. This makes for easy destructive backspace editing. When the user presses the enter key, the list is reversed and converted into a string. The string returned by **readln** is stored in the gettable instance variable **response**. The application retrieves it by sending a **get-response** message to the popup query window. Because of its very general utility, I wrote **readln** as a normal Scheme procedure rather than making it a method or a local procedure of a method.

Key Monitors

Key monitors are keyboard filters. The basic idea behind them is to have a method, **look-for-key**, that looks for key codes and a mapping that translates the key codes into filter actions. In class **key-monitor**, the mapping is represented by a translation vector of procedures, and the key codes serve as indexes into the vector. Scheme make this representation attractive because in Scheme procedures are data objects and may be stored in vectors just like any other kind of data object. When **look-for-key** gets a key code, it simply calls the procedure at the vector index corresponding to the key code.

That's the theory. In practice there is a complication. PC Scheme gets some key codes as a pair of bytes, the first of which is zero. This is dealt with by actually representing the key code mapping by two 128-byte translation vectors, stored int the instance variables **actions-for-ASCII-keys** and **actions-for-special-keys**, rather than one 256-byte translation vector. So **look-for-key** has been written to accept the translation vector it uses as an argument. **Look-for-key** is first called with the argument **actions-for-ASCII-keys**. When it sees a zero, it calls the procedure installed at index zero of **actions-for-ASCII-keys**. This procedure simply calls **look-for-key** again, this time passing it the argument **actions-for-special-keys** so that the next code **look-for-key** sees will be translated by **actions-for-special-keys** rather than **actions-for-ASCII-keys**.

Class **key-monitor** supplies the methods, **install-ASCII-key** and **install-special-key**, for installing procedures in the translation vectors. This serves to protect the translation vectors.

Menus

The only difference between a horizontal menu and vertical menu is that a horizontal one goes across the screen on a single line and a vertical one goes down the screen in many lines. This is an ideal situation for SCOOPS, one in which its inheritance feature can be used to maximum advantage. The component class **basic-menu** and its methods contain all the format-independent code, which is practically everything. Only a constructor and a format-dependent initialization method need be added for each of the instantiable classes. In the case of class **horizontal-menu**, the format-dependent initialization method requires an additional instance variable, **label-spacing**, from which it gets the amount of spacing to insert between two menu labels.

The init method of class **basic-menu** does a lot work. First, it translates the menu description received from the application into an internal representation. This gets stored in the instance variable **item-table**. Next, using instance variables and methods inherited from class key-monitor, it builds a keyboard filter with the following properties:

- The arrow keys drive the bar cursor from menu item to menu item.
- The return key selects the menu item highlighted by the bar cursor.
- The escape key cancels the menu.
- The application-specified selector keys select their proper menu items.
- All other keys are ignored.

Finally, the init method calls on the format-specific init method of either class **horizontal-menu** or class **vertical-menu** to finish the initialization. The init method doesn't need to know which of the two instantiable menu classes it is working for when it makes this call. As long as both of the instantiable classes use the same name for their init method—and they do, of course—SCOOPS makes sure everything comes out right.

Menu Items

Class **menu-item** is not part of the class ensemble shown in Figure 2. Because it stands by itself, it must be instantiable. It has no methods, except the SCOOPS generated get-methods and set-methods, so for all practical purposes a menu item can be considered a pure data object.

Menu items are created by the constructor **make-menu-item**. A programmer does not deal directly with this constructor. When an application calls on one of the menu constructors to create a menu, it passes an item list to the menu constructor. After the menu constructor has created the new menu, but before it returns to the application, it sends the new menu an init message with the item list as its argument. The init method, which the menu inherits from of class **basic-menu**, calls on the method **make-item-table**, also inherited from class **basic-menu**, to construct menu items and install them in the instance variable **item-table**, a vector of menu items.

Make-item-table also constructs a procedure that simulates the bar cursor method of selecting the menu item. This procedure is bound to the selector key character(s) for that menu item by the **install-ASCII-key method** inherited from class **key-monitor**. Consequently, at the menu level, there is really only one way a to select a menu item even though it appears to the user that there are two ways to do it.

A GLOSSARY OF SCOOPS TERMS

ACTIVE VALUE. An INSTANCE VARIABLE has an active value if its SET-METHOD calls a procedure rather than performing its normal function of assigning a new value to the instance variable. The procedure gets passed the argument of the set-method, and whatever is returned by the procedure is assigned to the instance variable. Similarly, an instance variable also has an active value if its GET-METHOD calls a procedure rather than merely retrieving the current value of the instance variable. In this case, the procedure is passed the current value, and the get-method returns whatever the procedure returns. An instance variable that is both **gettable** and **settable** is doubly active. Active values can transform the relatively tame operations of getting or setting into monsters with really hairy side-effects. See the discussions under INSTANCE VARIABLE, GET-METHOD, and SET-METHOD, for further explanation.

CLASS. A collection of private variables (CLASS VARIABLES and INSTANCE VARIABLES) and procedures (METHODS) that together function as an abstract data type. The methods comprise the set of operations applicable to any object belonging to the class. The private variables are visible to the methods, but not elsewhere, while the methods are generally visible. When a new class is defined, it may incorporate the variables and methods of an existing class, in which case the new class is said to inherit from the existing class. SCOOPS provides a defining form, **define-class**, for the creation of new classes, and a debugging tool, **describe**, to permit examination of a class or an INSTANCE of a class. Unlike Smalltalk-80, SCOOPS does not provide any pre-defined classes for programmers to build on.

CLASS VARIABLE. A variable that is private to a CLASS, but shared by all the INSTANCES of the class. Class variables may be declared as having options. In SCOOPS, class variables must be declared as members of the **classvars** list of a class definition. See the INSTANCE VARIABLE for a discussion of options.

COMPONENT CLASS. A CLASS defined to serve as a prefabricated subassembly from which other, more elaborate classes can be built. Contrast this with INSTANTIABLE CLASS. Sometimes a class can be both a component class and an instantiable class.

GET-METHOD. A **method** automatically generated by SCOOPS for an INSTANCE VARIABLE in response to a **gettable** declaration in the options list of its CLASS definition. A get-method returns the value of its instance variable. For example, if **john** is an instance of some class which has a **gettable** instance variable **name** and the copy of **name** belonging to **john** has the string **"John Q. Public"** as its value, the message

```
(send john get-name)
```

returns the string **"John Q. Public"**.

INHERITANCE. When a CLASS inherits from a MIXIN, it absorbs all the CLASS VARIABLES, INSTANCE VARIABLES, and METHODS of the mixin as if they were defined in the inheriting class itself. Permitting multiple mixins adds a complication to SCOOPS which doesn't come up in object-oriented languages permitting only a single mixin (e.g., Smalltalk-80). In SCOOPS, it is possible for two or more mixins to define methods or variables with conflicting names. Even worse, mixins can have mixins, and some of these could also define methods or variables with conflicting names. SCOOPS resolves such conflicts by giving priority to the first object (method or variable) it sees as it makes a depth-first search along the class inheritance graph. Any other objects with the same name are shadowed.

INSTANCE. Conceptually an instance is a data object which derives its type from its CLASS. The class imposes a structure and a set of operations on it. In SCOOPS, instances are represented by environments and are created by calling the special form **make-instance** with a class name and (optionally) initial values for any **inittable** INSTANCE VARIABLES it has.

INSTANCE VARIABLE. A variable that is private to an INSTANCE of a CLASS. An instance variable may be declared as having options. In SCOOPS, instance variables must be declared as members of the **instvars** list of a class definition, and may have any of the options **gettable**, **settable**, or **inittable** (spelled thus in SCOOPS—an alternate spelling is *initable*). If an instance variable is declared **gettable**, it becomes accessible to an application by means of a GET-METHOD, which is automatically generated by SCOOPS. Similarly, if it is **settable**, it is modifiable by means of a SET-METHOD, and if it is **inittable**, it may be initialized when it is created by **make-instance**.

INSTANTIABLE CLASS. A CLASS from which useful instances can be constructed is said to be instantiable. Contrast this with COMPONENT CLASS.

MESSAGE. A special kind of procedure call. It passes an INSTANCE and possibly other arguments to a method. In SCOOPS, messages are special forms distinguished by the initial keyword **send**. Examples of messages can be found under GET-METHOD and SET-METHOD.

METHOD. An operation that an application may perform on an INSTANCE of a CLASS. Methods are represented in SCOOPS by procedures and it provides a defining form, **define-method**.

MIXIN. When a CLASS is used as a component of another class, the first class is said to be a mixin of the second. See INHERITANCE.

SET-METHOD. A method automatically generated by SCOOPS for an INSTANCE VARIABLE in response to a **settable** declaration in the options list of its class definition. A set-method assigns its argument to its instance variable. For example, if **john** is an INSTANCE of some CLASS which has a settable instance variable **name**, then the message

```
(send john set-name "John Q. Public")
```

assigns the string **"John Q. Public"** to the copy of **name** belonging to **john**. Set-methods are used for effect; they return nothing useful.

AVAILABILITY OF *Menus for Scheme*

Menus for Scheme can be downloaded from Data Library 13 of the AI Expert Forum on CompuServe. Residents of the U.S., Canada, and Mexico may obtain ***Menus for Scheme*** directly from the author for a small fee. Contact the author by electronic or paper mail at one of addresses given at the beginning of this article for full details.

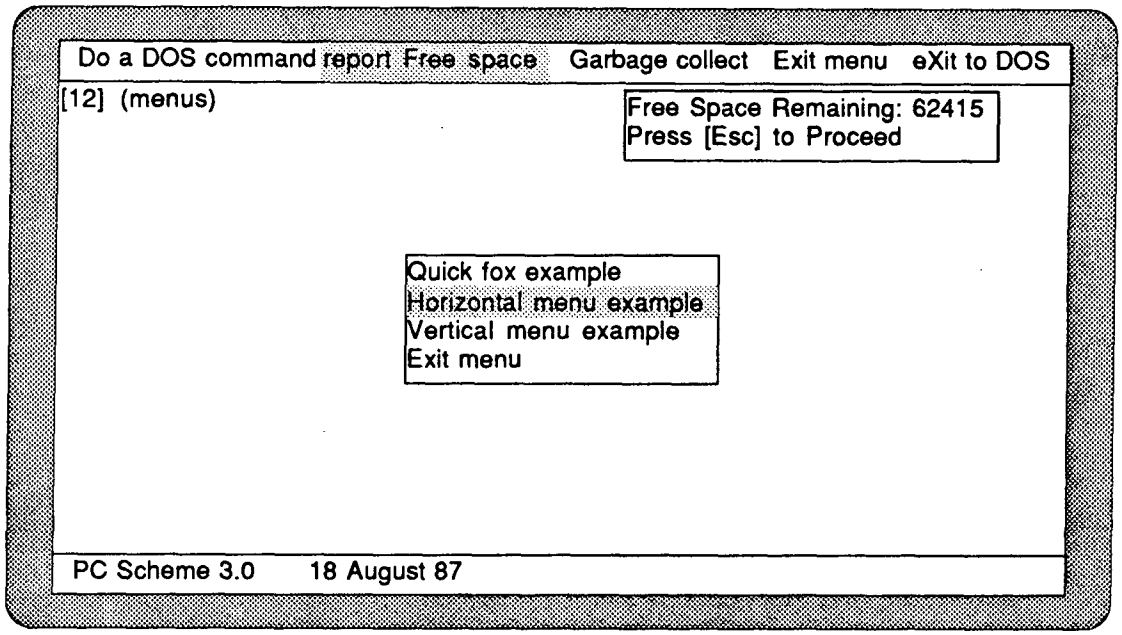

Figure 1. Output from Example Program

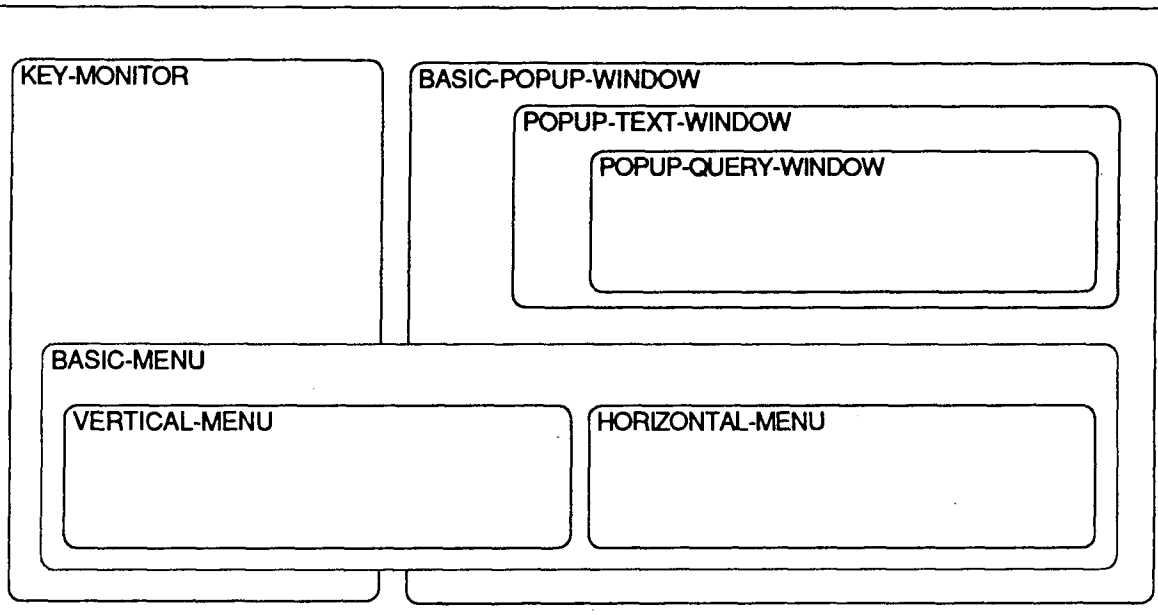

Figure 2. Menus for Scheme Class Hierarchy

```
;;; ------------------------------------------------------
;;; EXAMPLES
;;;
;;; Copyright (c) 1987 by Morton Goldberg.  Permission is
;;; granted to make copies of this program text for
;;; personal, non-commercial use provided this notice of
;;; copyright appears in all copies and derived works.  You
;;; may also distribute unmodified copies of this text as
;;; shareware.  All other rights reserved.
;;; ------------------------------------------------------
;;; Implementation Language: Texas Instruments' PC SCHEME
;;; (Version 3.0)
;;; ------------------------------------------------------

;;; ------------------------------------------------------
;;; DISPLAY ATTRIBUTES FOR TEXT WINDOWS
;;; ------------------------------------------------------

(define *rgb-monitor* #f) ; change to #t for color display

(define *white-black* #x07)
(define *white-blue*  #x17)
(define *white-red*   #x47)
(define *black-white* #x70)
(define *black-green* #x20)

;;; ------------------------------------------------------
;;; EXAMPLE 1 -- THE "QUICK BROWN FOX" POPUP TEXT WINDOW
;;; ------------------------------------------------------
;;; This example creates a popup text window and moves it
;;; around on the screen.  The procedure v-justify for
;;; performing vertical (top, center, and bottom) and
;;; h-justify for performing horizontal (left, center, and
;;; right) justification of a popup window are supplied with
;;; Menus for Scheme.

;;; Procedure for demonstrating the "quick brown fox"
;;; example.  After typing (do-fox) to start the demo, you
;;; must type [Esc] five times to complete it.  The window
;;; will move to a new position each time [Esc] is typed,
;;; excepting the 5-th [Esc], which ends the demo.
;;;
(define (do-fox)
  (v-justify 'TOP fox-popup)
  (h-justify 'LEFT fox-popup)
  (send fox-popup popup)
  (v-justify 'TOP fox-popup)
  (h-justify 'RIGHT fox-popup)
  (send fox-popup popup)
  (v-justify 'BOTTOM fox-popup)
  (h-justify 'RIGHT fox-popup)
  (send fox-popup popup)
  (v-justify 'BOTTOM fox-popup)
  (h-justify 'LEFT fox-popup)
  (send fox-popup popup)
  (v-justify 'CENTER fox-popup)
  (h-justify 'CENTER fox-popup)
  (send fox-popup popup))

;;; This is the init-list for the "quick brown fox" popup
;;; text window; it determines the visual properties (screen
;;; position, colors, format) of the window.
;;;
(define *fox-data*
  (let ((attr1 (if *rgb-monitor*
                   *black-green*
                   *black-white*)))
    '('w-top 1
      'w-left 1
      'n-attr ,attr1
      'border? #t)))

;;; This is text of the "quick brown fox" popup text window.
;;; It is supplied to the window as a vector of text
;;; strings, one string for each line of displayed text.
;;;
(define *fox-text*
  '#("The quick brown fox jumped"
     "over the lazy dog.  The quick"
     "brown fox jumped over the"
     "lazy dog."
     ""
     "Press [Esc] to Proceed"))

;;; Create the "quick brown fox" popup text window.
;;;
(define fox-popup
  (make-popup-text-window *fox-data* *fox-text*))

;;; ------------------------------------------------------
;;; EXAMPLE 2 -- A VERTICAL AND A HORIZONTAL MENU
;;; ------------------------------------------------------
;;; This example creates a horizontal menu and a vertical
;;; menu, both offering the same choices.  The menus invoke
;;; commonly used Scheme facilities such as the garbage
;;; collector.  To experiment with a menu, type (do-h) to
;;; get the horizontal menu or (do-v) to get the vertical
;;; one.

;;; Procedure for demonstrating the vertical menu.  This
;;; menu will appear with its 2-nd item highlighted.
;;;
(define (do-v)
  (send vertical set-item-index 2)
  (send vertical popup))
```

```
  (send vertical popup))

;;; Procedure for demonstrating the horizontal menu.
;;;
(define (do-h)
  (send horizontal popup))

;;; ------------------------------------------------------
;;; EXECUTE A DOS COMMAND
;;;
;;; The following code creates a two-line popup query window
;;; which prompts for a DOS command.  To activate the window,
;;; type (do-dos).  After the command is entered and [Enter]
;;; is pressed, the DOS command is executed and the window
;;; is erased.

;;; This is the init-list for the DOS-command popup query
;;; window; it determines the visual properties (screen
;;; position, colors, format) of the window.
;;;
(define *query-data*
  (let ((attr1 (if *rgb-monitor*
                   *white-blue*
                   *black-white*))
        (attr2 (if *rgb-monitor*
                   *black-white*
                   *white-black*)))
    `('w-top 4
      'w-left 35
      'cursor-row 1
      'cursor-col 0
      'input-width 40
      'n-attr ,attr1
      'hl-attr ,attr2
      'border? #t)))

;;; This is the vector of text strings for the DOS-command
;;; popup query window.  Note that the 2-nd element is an
;;; empty string; it serves as a place holder for the
;;; type-in area.
;;;
(define *query-text*
  '#("Enter a DOS Command or Press [Esc] to Cancel" ""))
;;; Create the DOS-command popup query window.
;;;
(define dos-cmd
  (make-popup-query-window *query-data* *query-text*))

;;; Activate the DOS-command popup query window.  Call DOS
;;; with the string typed-in by the user.
;;;
```



```
(define (do-dos)
  (send dos-cmd popup)
  (let ((cmd (send dos-cmd get-response)))
    (if (> (string-length cmd) 0)
      (dos-call "" cmd 16384))))

;;; ------------------------------------------------------
;;; REPORT ON FREE SPACE
;;;
;;; The following code creates a two-line popup text window
;;; which shows the amount to free space remaining to Scheme
;;; at the time the window is exposed.  To activate the
;;; window type (do-freesp).  Press [ESC] to erase the
;;; window.

;;; This is the init-list for the report-free-space popup
;;; text window; it determines the visual properties (screen
;;; position, colors, format) of the window.
;;;
(define *freesp-data*
  (let ((attr1 (if *rgb-monitor*
                   *white-blue*
                   *black-white*)))
    `('w-top 4
      'w-left 41
      'n-attr ,attr1
      'border? #t)))

;;; Create the report-free-space popup text window.
(define freesp-rpt
  (let ((dummy-text '#(" ")))
    (make-popup-text-window *freesp-data*
                            dummy-text)))

;;; Activate the report-free-space popup text window.
;;;
(define (do-freesp)
  (let ((line-1 (string-append "Free Space Remaining: "
                               (integer->string (freesp)
                                                10)))
        (line-2 "Press [Esc] to Proceed"))
    (send freesp-rpt set-text (vector line-1 line-2))
    (send freesp-rpt popup)))

;;; ------------------------------------------------------
;;; DO A COMPACTING GARBAGE COLLECTION
;;;
(define (do-gc)
  (gc #t))
```

```
;;; ------------------------------------------------------
;;; EXIT FROM THE MENU -- SAME A PRESSING [ESC]
;;;
(define (do-abort)
  'USER-ABORT)

;;; ------------------------------------------------------
;;; EXIT TO DOS
;;;
(define (do-exit)
  (exit))

;;; ------------------------------------------------------
;;; MENUS PROVIDING THE SERVICES DEFINED ABOVE

;;; This is the item-list for both menus; it determines the
;;; selections provided to the user by the menus.
;;;
(define *item-list*
  '(("Do a DOS command" (#\d #\D) ,do-dos)
    ("report Free space" (#\f #\F) ,do-freesp)
    ("Garbage collect" (#\g #\G) ,do-gc)
    ("Exit menu" (#\e #\E) ,do-abort)
    ("eXit to DOS" (#\x #\X) ,do-exit)))

;;; This is the init-list for the vertical form of the menu.
;;; It determines the visual properties (screen position,
;;; colors, format) of the menu.
;;;
(define *vertical-menu-data*
  (let ((attr1 (if *rgb-monitor*
                   *white-blue*
                   *black-white*))
        (attr2 (if *rgb-monitor*
                   *white-red*
                   *white-black*)))
    '('w-top 1
      'w-left 1
      'n-attr ,attr1
      'hl-attr ,attr2
      'border? #t)))

;;; Create the vertical version of the menu.
;;;
(define vertical
  (make-vertical-menu *vertical-menu-data*
                      *item-list*))

;;; This is the init-list for the horizontal form of the
;;; menu.
;;;
```



```
(define *horizontal-menu-data*
  (let ((attr1 (if *rgb-monitor*
                   *white-blue*
                   *black-white*))
        (attr2 (if *rgb-monitor*
                   *white-red*
                   *white-black*)))
    '('w-top 1
      'w-left 1
      'n-attr ,attr1
      'hl-attr ,attr2
      'border? #t
      'label-spacing 2)))

;;; Create the horizontal version of the menu.
;;;
(define horizontal
  (make-horizontal-menu *horizontal-menu-data*
                        *item-list*))

;;; ------------------------------------------------------
;;; Just for fun, here is a menu that runs all the examples
;;; in this file.
;;;
(define *examples*
  '(("Quick fox example" (#\q #\Q) ,do-fox)
    ("Horizontal menu example" (#\h #\H) ,do-h)
    ("Vertical menu example" (#\v #\V) ,do-v)
    ("Exit menu" (#\e #\E) ,do-abort)))

(define *demo-menu* '())

;;; This procedure constructs the demo menu the first time
;;; it is called, but only brings it back to the screen
;;; after the first time.  Because there is no reason for
;;; this menu to be different, *vertical-menu-data* is used
;;; for the init-list.
;;;
(define (menus)
  (if (null? *demo-menu*)
    (begin
      (set! *demo-menu*
            (make-vertical-menu *vertical-menu-data*
                                *examples*))
      (v-justify 'CENTER *demo-menu*)
      (h-justify 'CENTER *demo-menu*)))
  (send *demo-menu* popup))

;;; ------------------------------------------------------
```