1 User Interaction

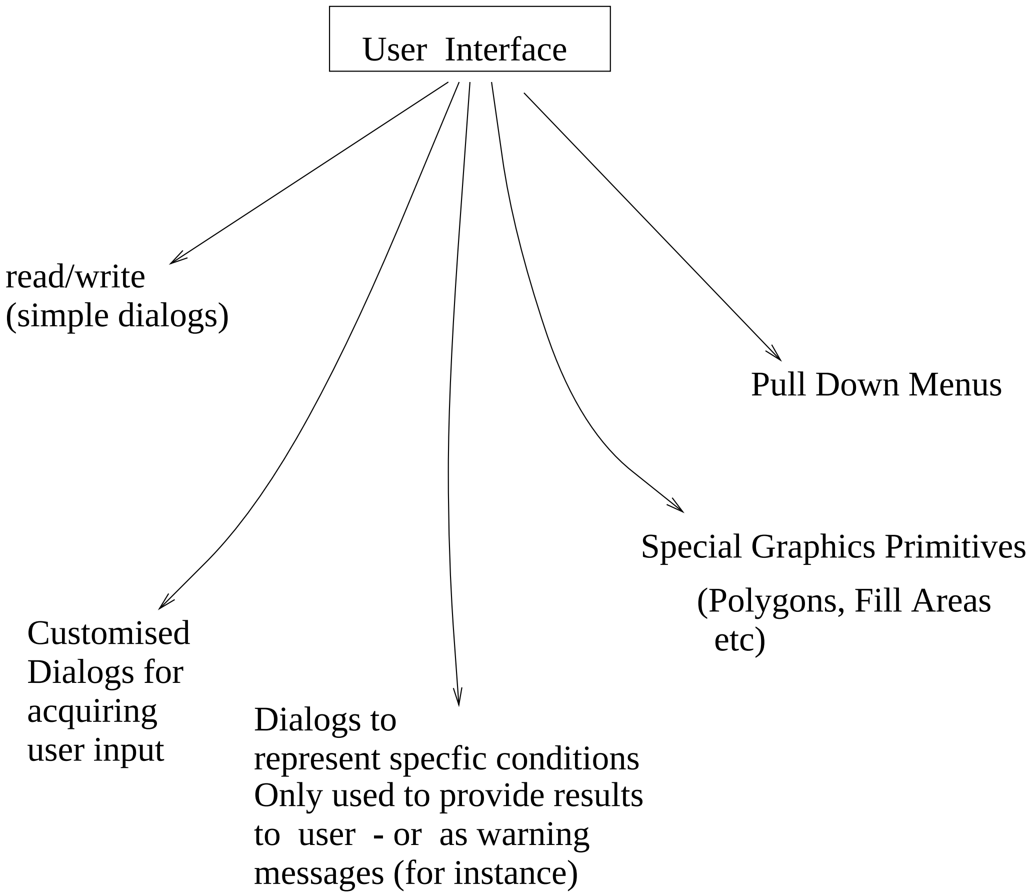

Figure 1: General User Interaction

- Basic window management : includes basic features such as *creating*, *deleting*, *highlighting*, *resizing*windows
- Menus - can be *pulldown* or *popup*
- Dialogs - range from *predefined dialogs*, *custom dialogs* - custom dialog composed of buttons - also *advanced dialogs* allow the use of *pictures*
- Buttons can be *radio*, *check*, *push* etc
- Control Windows - stand alone windows - need to be explicitly killed by the programmer

2 Windows

- A window in *MacProlog* can be a *program window* (prog), a *display window* (disp), a *control window* (ctrl), a *graphics window* (grap) or a *clipboard window* (clip)
- A *program window* may contain edit-able text and can be compiled while a *display window* cannot be edited or compiled (only for text)
- *Dialog* windows used to give information to, or accept input from the user during program execution. A dialog window may be of a *predefined* format, or can be customised - example of predefined is the *ask* in bank balance
- A *control* window may contain any number of items for interaction with the user, such as *buttons*, *check boxes*, *edit fields* and all items available in a dialog window
- MacProlog comes with a *Graphics Description Language* (GDL), used for the *placement* and *sizing* of graphical objects. These graphical objects are displayed in *graphic windows*

Windows are divided according to their functionality. The type of window active at any stage is determined by what is required of the user (or what needs to be displayed).

A *user interface* is a very important part of a program and should not be *tacked on* as an after thought. Rather, like any other aspect of your program, it should be *engineered*! (by considering user requirements, program specification and most importantly, program style).

Before we look at the different types of windows present in more details - consider the types of operations performable on all windows :

2.1 Window Management in MacProlog

- Create a new window :

  ```
  wcreate/[6,7]
  wcreate(Name, Visible, Top, Left, Depth, Width, Go_away)
  ```

  Note : Display windows created by MacProlog environment do not have Go_away boxes, to distinguish them from edit (program) windows

  ```
  wpcreate/8
  wpcreate(Name, Visible, Syntax, Mode, Top,
  Left, Depth, Width)
  ```

  Will create a new program edit window. The *syntax* specifies how the code in the window is to be compiled (e.g. EDINBURGH, KSL, PROLOG++), the Mode corresponds to whether the window is *Interpreted* or *Optimised*

- Window types :

  ```
  windows/2
  windows(Type, Window_list)
  ```

  where *Type* refers to *prog, disp, ctrl or grap*, the list is returned in the variable *Window_list*

  ```
  wtype/2
  wtype(Window, Type)
  ```

where *Window* is the name of a window and its type is returned in *Type*

- Displaying a window and removing it :

```
wkill/1
wkill(Window)
whide/1
whide(Window)
wshow/1
wshow(Window)
wvis/1
wvis(Window)
wfront/1
wfront(Window)
```

where *Window* is the name of a window. When a window is hidden, its contents are preserved in a buffer - and one can still read or write to it. When a window is killed, anything that it may contain is now lost.

```
mywin :-
 wcreate(win1, 1, 50, 60, 100, 200, 1),
 wpen(win1,yellow),
 wpaper(win1,magenta).
```

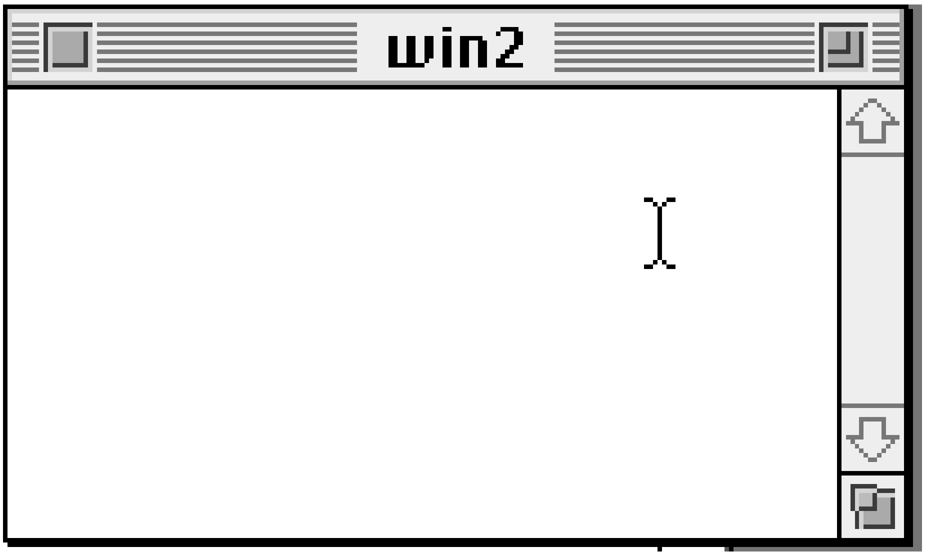

Figure 2: A window with a *Go-Away* button

- The look :

```
wsize/5
wsize(Window, Top, Left, Depth, Width)
wlen/2
wlen(Window, Length)
wpen/2
wpen(Window, Colour)
wpaper/2
wpaper(Window, Colour)
```

where *Window* is the name of a window. In the *wsize* definition, other than the window name, all other arguments may be variables - in which case they will be instantiated to the current values for the named window. The *Length* variable refers to the length of text in the window. *wpen* is the foreground (text) and wpaper the background colour of the window. *Colour* can be *white, black, red, blue, yellow, green, cyan, magenta*

Another very useful and important command for windows management is the *actdeact* call :

```
actdeact(Window, Prog)
```

- where *Window* is the name of a window, and *Prog* refers to a program name called when the window is activated (i.e. when the window becomes the *front* window) or deactivated (i.e. when the window is *closed* or ceases to be the *front* window).
- The *Prog* must contain three different scenarios :

```
Prog(activate, Window) :-
code for activation ...

Prog(deactivate, Window) :-
  code for deactivation ...

Prog(close, Window) :-
code for closing ...
```

- Especially useful for manipulating the *Menu Bar* - so that when certain windows are at the front, relevant menus can be made available
- No calls should be included in an *actdeact* program that might cause further window activations or deactivation - could lead to endless loops!!
- Windows should not be killed in their *activate* or *deactivate* modes.

3 Menu's and Sub-Menus

- Used to provide an interface for offering multiple options to the user
- Can have *pulldown*, *hierarchical* or *popup* menus.
- Pulldown menu's always appear at the top of the screen
- A menu can be created by the call :

  ```
  install_menu( Name, Item_list)
  ```

  where *Name* is the name of the menu as it appears on the bar and *Item_list* is a list of choices available for the menu.
- It is important to note that a predicate *Name* must also be produced in the program to handle the menu options. When a selection is made from the menu a call :

  ```
  Name(Item)
  ```

  is made, where *Item* is a member of *Item_list*

Consider the following example, which creates a four item menu with *aura* initially disabled, and a horizontal separating bar between *do-bop* and *tutu*.

```
install_menu('Miles',
 [sketches_of_spain, '(aura', amandala, dobop,
    '(-','tutu/P']).
```

This is infact equivalent to doing :

```
install_menu('Miles',
 ['sketches_of_spain;(aura;amandala;dobop;(-;tutu/P']).
```

There MUST be a program to deal with the menu also, and must take the general form of :

```
'Miles'(sketches_of_spain) :-
... code for sketches_of_spain option

'Miles'(aura) :-
... code for aura option

etc
```

Figure 3: The Miles Menu added to the Menu Bar

Figure 4: The Miles Menu

3.1 the Fruits menu - a multilevel menu

Similar to the Miles menu, we now consider the implementation of the *Fruits* menu, which contains multiple levels. It is implemented by the following call :

```
install_menu('Fruit',
  [apple([ 'Granny Smith',
 'Delicious'(['Red','Golden']),
 'Macintosh',
 'Cox']),
 pear,
 orange(['Jaffa',
 'Seville',
 'Navel'])]).
```

- This will install a menu called *Fruit* on the menu bar - with three items *apple, pear, orange*
- *apple* and *orange* contain submenus, and the *Delicious* option of apple contains a further submenu
- If the user selects *Golden*, the goal

  ```
  'Delicious'( 'Golden')
  ```

  will be called
- If the user selects *Seville*, then the Prolog goal

  ```
  orange( 'Seville')
  ```

  will be called

Figure 5: The Fruit Menu added to the Menu Bar

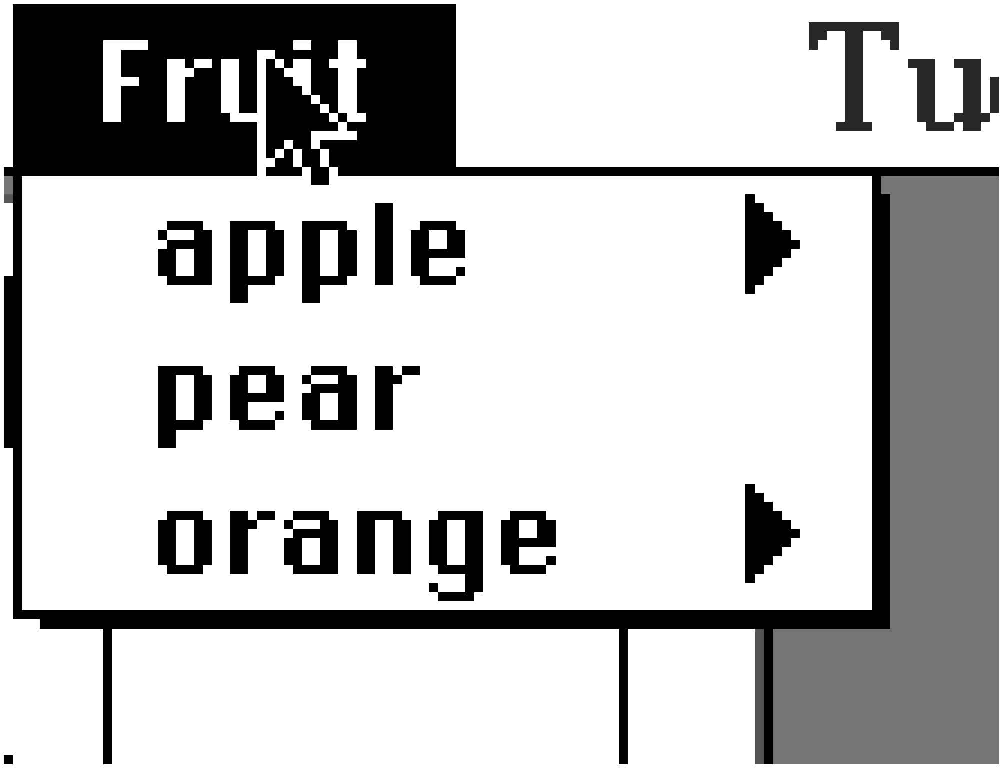

Figure 6: The Fruit Menu

- A menu can be removed with the

  `kill_menu(Menu)`

  command. If a menu has submenus, all of the submenus will also be deleted (removed). It is possible to remove a submenu from within a menu (like Fruit above) with this command - so that entries in the top level menu are automatically adjusted

  `clear_menu(Menu)`

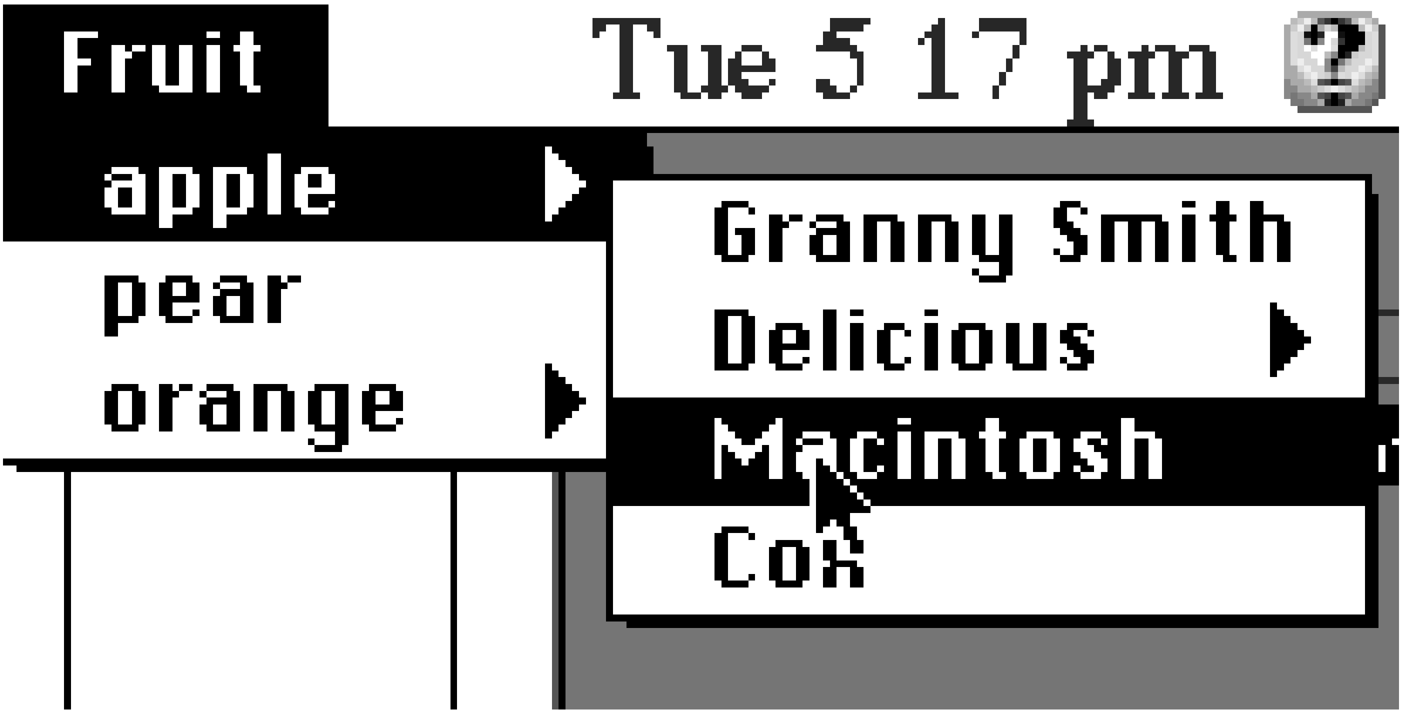

Figure 7: The Fruit Menu

removes all entries from a menu - although the menu will still exist - but it will contain no items

`disable_menu(Menu)`

The whole menu *Menu* is disabled - its colour on the menu bar (if it it a top level menu) is 'greyed out' - and all items in it also appear grey

`enable_menu(Menu)`

will enable a previously disabled *Menu*

- Checking menus

`is_menu(Menu)`

Is to check if *Menu* exists ... from within the program

Figure 8: The Fruit Menu

```
get_menus(Menus)
```

Returns a list of current menus in the variable *Menus* - no indication is provided of whether a menu is in the menu bar, a hierarchical submenu or a popup menu

Figure 9: The Fruit Menu - disabled

- It is also possible to add items to menus that already exist - this includes the default menus provided within the *MacProlog* environment. This is done with the

```
extend_menu(Menu,Items)
```

for instance, the call

```
extend_menu('Edit',['Make me coffee'])
```

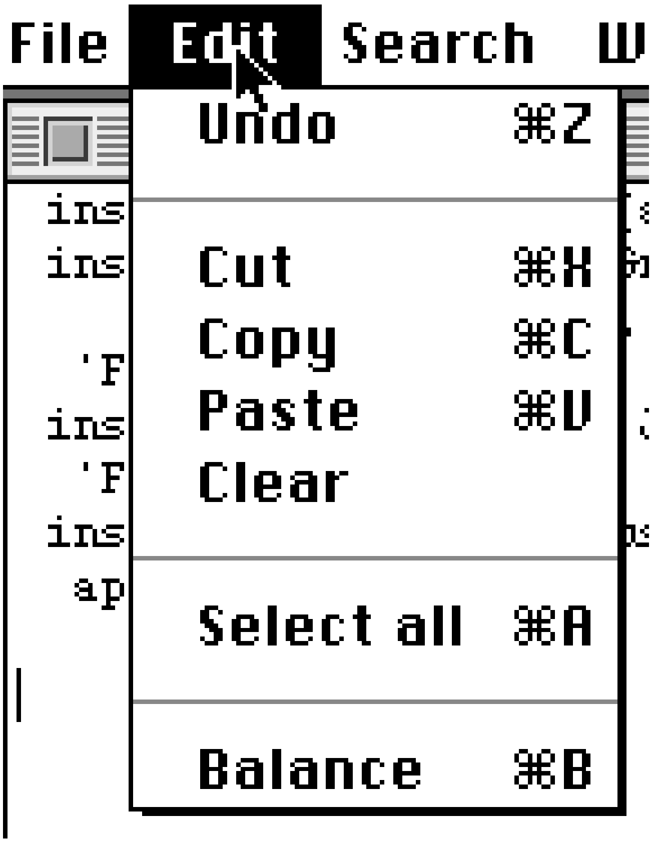

Figure 10: The Edit Menu

will add a new entry to the existing *Edit* menu

- Similar to enabling a menu, it is also possible to enable or disable an item within a menu, this is done with

```
disable_item(Menu,Item)
enable_item(Menu,Item)
```

- Further it possible to rename items within a menu

```
rename_item('Eval','Enter','  /T')
```

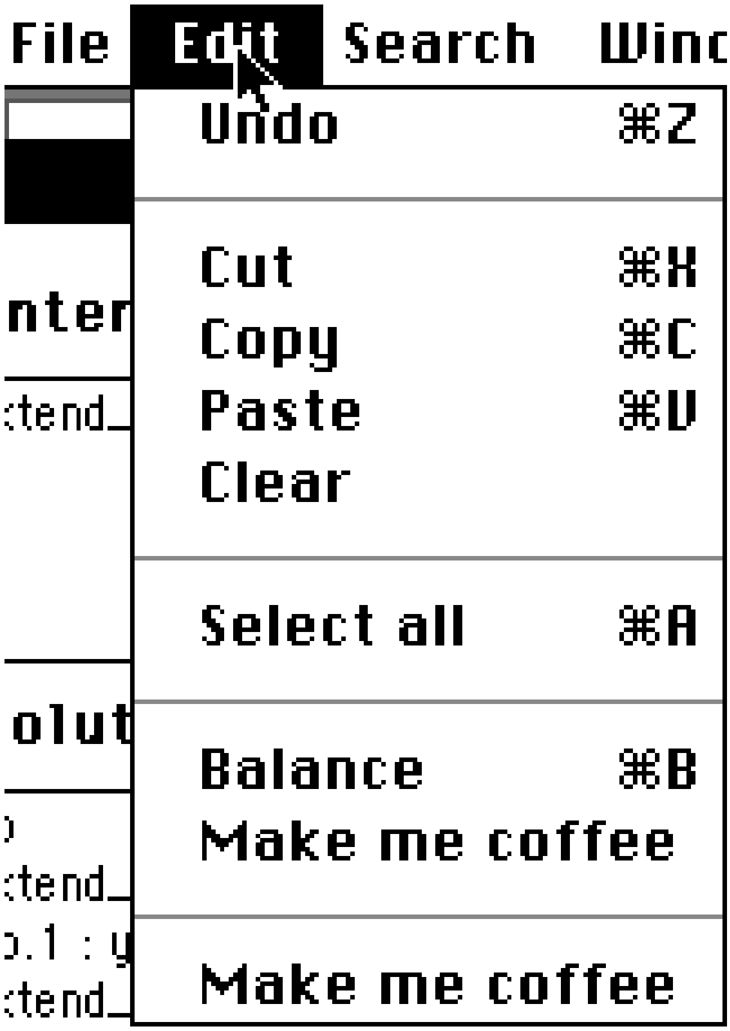

Figure 11: The Edit Menu - with the additional entry

3.2 Popup Menus

- Ability to *popup* a menu at a particular place on the screen
- Usually used with graphics primitives allowing one the ability to place a menu on any part of the screen
- The popup menu may either be a submenu (but not a pull down menu)
- An option in the *wcreate* allows the creation of a popup menu also

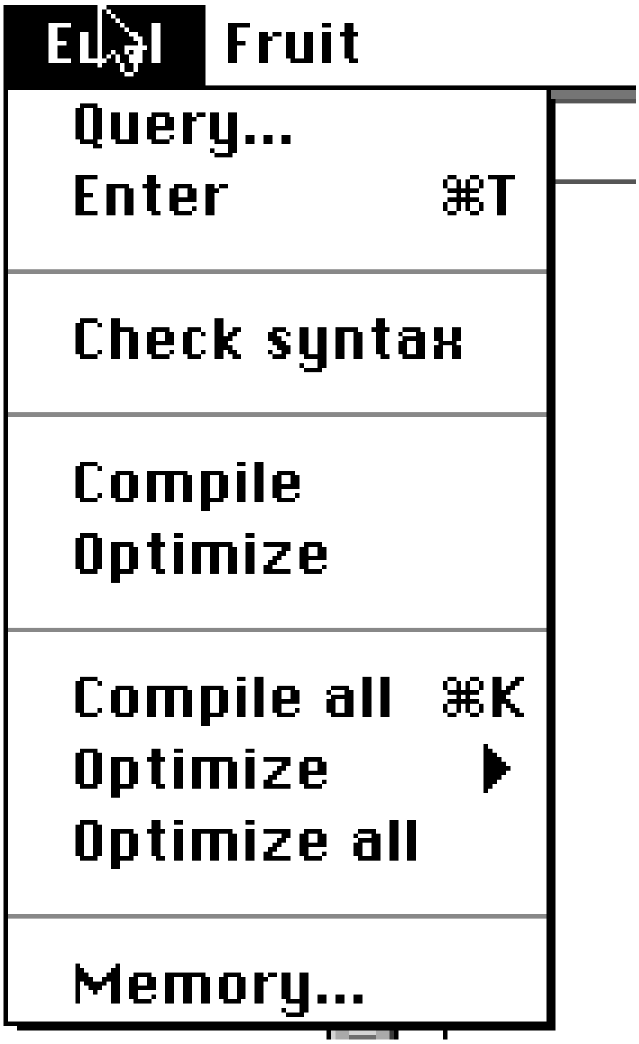

Figure 12: The Eval Menu

```
popup(Menu,Item, SMenu, SItem)
popup(Menu,Item, SMenu, SItem, (Y,X))
popup(Menu,Item, SMenu, SItem, GWin(Y,X))
```

where

- *Item* - which is the name of the menu item to be preselected,
- *SMenu* will be name of the selected menu and *SItem* is the name of the selected item,

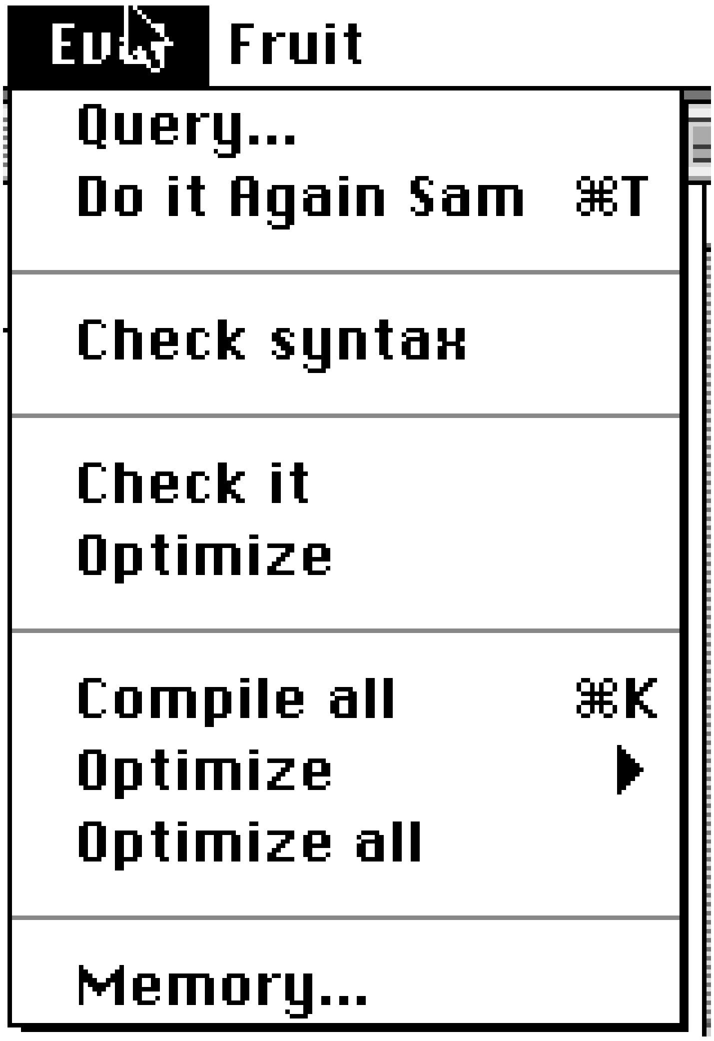

Figure 13: The Eval Menu - with the *compile* and *Enter* renamed

- *(Y,X)* is a pair of integer specifying the position where the menu appears - if not specified - menu appears at the position of the mouse
- *GWin* the name of the graphics window, and the *(Y,X)* specify the point where the window appears in the coordinate system of the graphics window

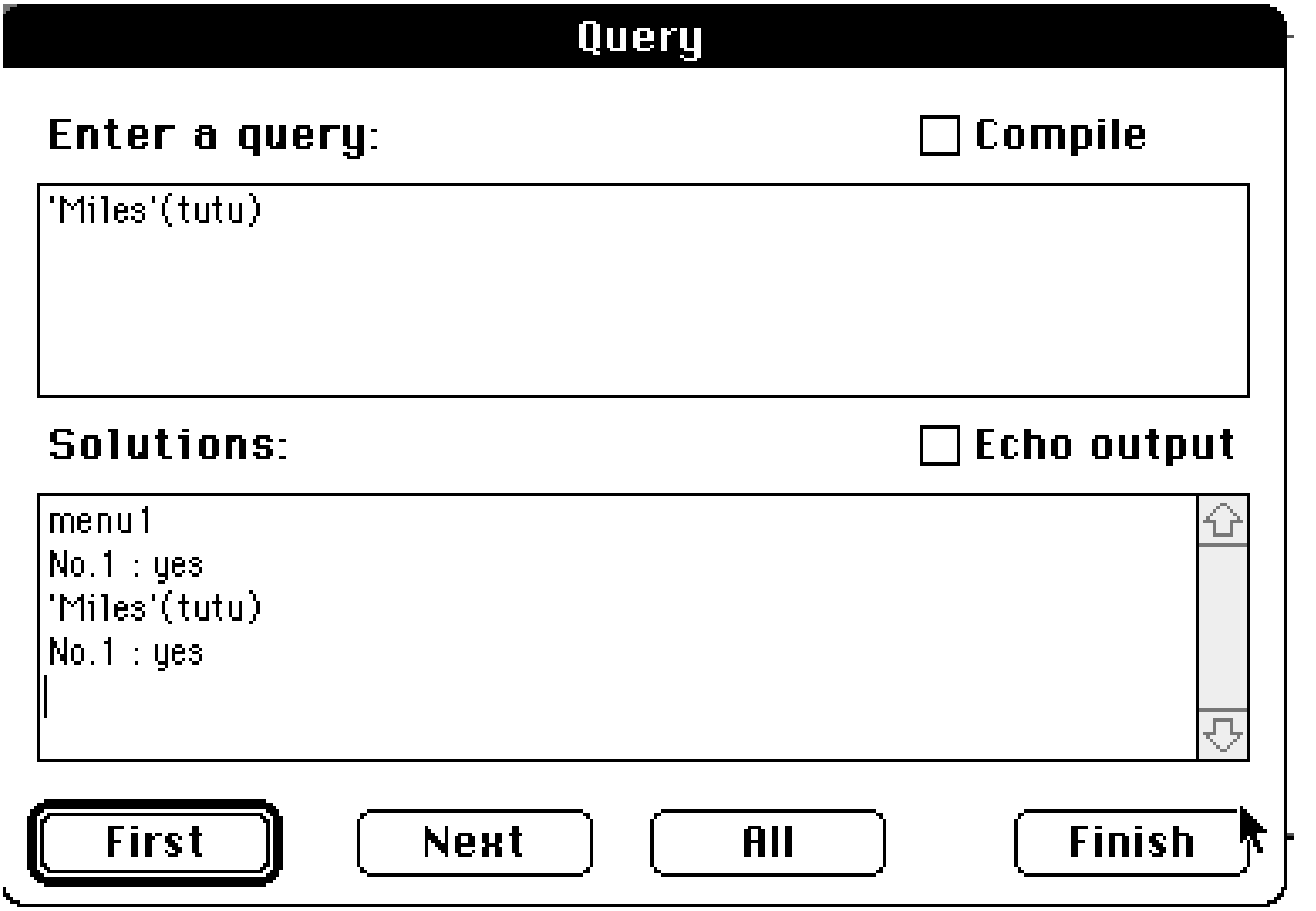

Figure 14: The Query Dialog