PDF pour 18215 scan Data & Graphiques

Le présent fichier PDF **Data & Graphiques** concerne le document

ExperLisp,
For the MacIntosh

Il s'agit d'extraits d'un Manuel de 220 pages pour un logiciel créé en 1985-1986 par la société Californienne **ExperTelligence, Inc**.
Le directeur de cette société est Denison Bollay (voir 3ème page de couverture) et l'Auteur du manuel est Dean A. Ritz
A l'intérieur de ce Manuel, deux disquettes de 3'' pour la version 1.5 du logiciel ExperLisp, un langage de programmation LISP.

Cet ouvrage est disponible sous le numéro **AC 18215** dans l'inventaire de la collection de l'ACONIT, Association pour un conservatoire de l'informatique et de la télématique, Grenoble.

Auparavant il appartenait aux collections de la Bibliothèque Municipale de Grenoble, d'où la référence MA 2001 007 0003 qui apparaît sur la page de couverture.

Le présent PDF fournit un scan d'extraits de cet ouvrage, avec copie de :

la page de couverture du Manuel,
-les pages 27 à 32 décrivant le format des données «(Data),
-les pages 33 à 49 sur les Graphiques utilisables avec le logiciel ExperLisp dans l'environnement Windows de l'ordinateur MacIntosh,
-les pages 51 à 58 pour la création des « menus ».

Fiche rédigée le 24/6/2020 par Jean Ricodeau.

ExperLisp™ for the Macintosh™

by ExperTelligence™

LIS

ExperLisp Data Types

An ExperLisp object can take many forms. Among them are: integer numbers, floating point numbers, symbols, strings, lists, pointers, handles, arrays, streams, and functions. In this section each data type is followed by a general description notating all that is generally necessary for its use. Some notations optionally include a more detailed description for those interested. Comprehension of the detailed description is not necessary to use ExperLisp. The reference section of this manual has more information about functions used in the examples.

Every ExperLisp object is stored somewhere in memory. Each type of object has a special number assigned to it to identify its type. This number is known as its "type bits." This enables ExperLisp to quickly discern a legal argument to a function `(+ 1 4)` VS. an illegal argument `(+ '(1) "four")`.

☞ System use of these type bits is generally transparent.

☞ For this section, all Lisp objects imbedded in the text will be in a monospaced font and underlined => `'DOG` -or- `23.45`

Integer Numbers

An integer number is any number which does not contain a decimal point in its printed representation. `1963` is an integer number; `1963.` is not. Integer numbers lie within the ranges of -32769 and +32768. Numbers which exceed this range, whether they have a decimal point or not, are converted to floating point numbers (see below). The predicate FIXP (which stands for "fixnum") returns the object if the object is an integer number.

Extra-detail: ExperLisp will read hexadecimal numbers, though all numbers are printed in decimal. They must be preceded with a `$` as in `$3C`

```
-327
   -> -327
3428632
   -> 3428632.
```

Floating Point Numbers

A floating point number is any number which contains a decimal point in its printed representation. `1963.` is a floating point number, `1963` is not. The predicate FLOATP returns the object if the object is a floating point number.

Extra-detail: ExperLisp supports up to 80bits of precision for floating point numbers. Floating point numbers are printed in exponential notation when they exceed the ranges of -99999999 or 99999999.

```
1.0000
   ->  1.
(FLOATP 34)
   ->   NIL
1000340010
  ->   1.00034001E9
(FLOATP 1000340010)
  ->   1.00034001E9
```

Symbols

A symbol is any contiguous collection of printing characters which does not include delimeters used by other ExperLisp objects. `Catfood` is a symbol, `cat"fo()d` is not (It includes both a double quote used to delimit strings, and an opening and closing parenthesis used to delimit lists - see below).

ExperLisp does not care whether a symbol is in upper case letters, lower case letters, or a mixture of both. ExperLisp will always print a symbol using the same case in which it was first entered. ExperLisp has already entered many symbols before you have even begun to type. Therefore, don't be surprised if ExperLisp prints some of your symbols differently.

Extra-detail: Every unique symbol is placed into a "symbol table." Each entry into the symbol table includes the following information associated with the symbol: printed representation, property lists, and value. ExperLisp symbols have a single value cell.

```
'ApPLe
   ->   ApPLe
'apple
   ->   ApPLe
```

Strings

A string is any collection of printing characters delimitted by double quotes. "Pickled beets in a barrel" is a string, 3435 is not. Strings are used to store the printed representation of every unique symbol. The function GETPREP (get-printed-representation) can be used to retrieve the printed representation of a symbol (those entered in the symbol table). The predicate STRINGP returns the object if the object is a string. Strings can be up to 255 characters in length.

Extra-detail: Currently, string space (the block of memory used to store strings) is limited to 32767 characters. The printed representation of every symbol is stored in string space. Every unique symbol has a unique string associated with it. Every string is unique. A string is internally represented as an offset into the ExperLisp string space plus a length byte (number of characters beyond that offset). The function UPRSTRING can be used to convert to uppercase letters the printed representation of a symbol.

```
(STRINGP 'COCOLAT)
   ->  NIL
(STRINGP "COCOLAT")
   ->   "COCOLAT"
(STRINGP (GETPREP 'COCOLAT))
   ->   "COCOLAT"
```

Extra-detail:

```
'PeaNutButter
   ->   PeaNutButter
'PEANUTBUTTER
   ->   PeaNutButter
(UPRSTRING (GETPREP 'peanutbutter) T)
   ->   T
```

```
'PeaNutButter
   ->   PEANUTBUTTER
```

Lists

A list is any collection of Lisp objects delimitted by an opening parenthesis "(" and a closing parenthesis ")". `(A B 123)` is a list; `"(A B 123)"` is not (it is a string which happens to include some parentheses as elements). Lists are dynamic structures that use cons cells to store their elements (see the Touretzky book for more information). The predicate LISTP returns the T if the object is a list or NIL.

Extra-detail: All lists are stored in cons space. A cons cell is 8 bytes in length. The first four bytes points to the Lisp object in the CAR position of the cell. The last four bytes points to the Lisp object in the CDR position of the cell (usually another cons cell). Cons space is fixed at 16K bytes + 1/8 of memory.

```
(LISTP  '(A B (C D E) F))
   ->  T
(LISTP (FIRST '(A B (C D E) F)))
   ->  NIL
(LISTP (THIRD '(A B (C D E) F)))
   ->  T
```

Pointers

A pointer is the address of an absolute location in memory. Pointers are used to represent pen patterns, and bunnies. Unless you deliberately desire to fiddle with memory, you need not be concerned with why or how to use pointers. If the desire is there, consult the Memory Management section of this manual.

Handles

A handle is a pointer to a memory location whose contents point at a relocatable block of memory. Handles are used to represent other Lisp objects such as: arrays, bunnies, windows, menus, regions, etc. It is a convenient way to pass around large unprintable objects. Consult the Memory Management section of this manual for more information.

Arrays

An array is a Lisp object with components arranged according to a Cartesian coordinate system. Its components can be any Lisp object, including other arrays. Arrays must be created before they can be used. ExperLisp arrays can be multi-dimensional. The total number of dimensions of an array is called its "rank". The array created below has a rank of 2.

Extra-detail: Arrays are handles whose type bits are $58. The memory block of the handle contains the array header information and the structure necessary to reference its objects. The size of an array in bytes is 4 times the product of its dimensions, plus the 54 bytes of header information.

```
(+ 54 (* 4 (APPLY '* ARRAY-DIMENSIONS ARRAY ).
```

☞ The contents of an array are garbage collected, though the arrays themselves are not. See the section on Memory Management for more information.

```
(SETQ MYARRAY (MAKE-ARRAY '(3 3)))
   ->  ARRAY #58011D80
(SETQ (AREF MYARRAY 0 0) 'FISHFOOD)
  ->  FISHFOOD
(AREF MYARRAY 0 0)
   ->  FISHFOOD
```

Streams

A stream is a source or destination of data (a path in which the data can flow). This data is typically characters. It has no special delimeters. Currently, stream objects pertain only to the ExperLisp file system.

Extra-detail: Streams are handles whose type bits are $54. The memory block of the handle contains a file structure. See the section on Memory Management for more information.

Functions

A function is an object whose value points at executable code (anything that can legally be given to FUNCALL or APPLY). This value is stored in the value cell of the function's name (a symbol). The predicate FUNCTIONP returns the object if the object is a function.

```
(FUNCTIONP MEMBER)
   ->  FUNCTION #40017608
(FUNCTIONP 'MEMBER)
   ->  NIL
```

Extra-detail: All ExperLisp functions are compiled. See the section on Saving Source Code for Functions.

ExperLisp Graphics

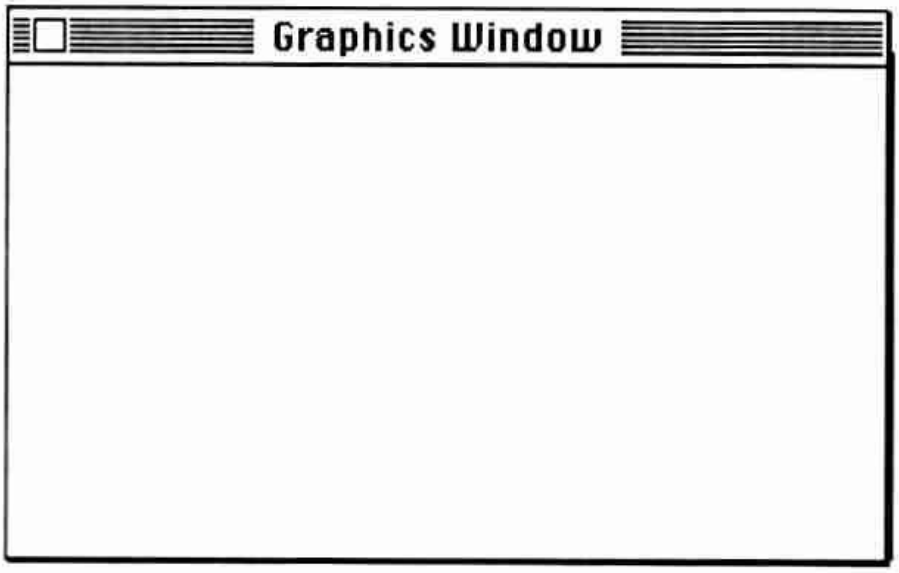
The graphics system can be divide into two catagories: bunny graphics and Quickdraw graphics.

Quickdraw Graphics - The Quickdraw graphics routines are supplied in the toolbox of the Macintosh {ROM}. These routines enable the programmer to create complex graphics easily and quickly. Most Quickdraw routines rely on Cartesian coordinates.

Bunny Graphics - ExperLisp bunny graphics are similar to turtle graphics. Bunny graphics center around an object called a 'bunny.' Commands such as FORWARD, RIGHT, LEFT, SETPOS, PITCH, etc., allow bunnies to move to positions in relative space. ExperLisp supports multiple bunnies of three types: 2-dimensional, 3-dimensional, and spherical.

All graphics commands are displayed in a Graphics Window. Using any graphics command creates a Graphics Window (STD_GRAF) if one does not already exist.

```
(CLEARSCREEN)
   ->  T
```



The system special variable STD_GRAF is now bound to a handle to this window. ExperLisp creates a new Graphics Window only if the system variable STD_GRAF is bound to NIL. All graphics operations are sent to the

STD_GRAF window, even if it is hidden behind another window. Making a window active brings it to the front of the screen. Make the Graphics Window active before drawing if you wish to see the results.

Window Fun

All of the Graphics Windows can be manipulated under program control, providing the ability to size, move, and select windows.

(1) Execute a graphics function. This will create a STD_GRAF window if one does not already exist.

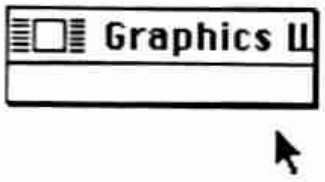
```
(CLEARSCREEN)
   ->  T
```

(2) Send the STD_GRAF window a message to move it to a different screen location.

```
(SEND STD_GRAF 'MOVEWINDOW 50 50 T)
   ->  T
(DOTIMES (K 40) (SEND STD_GRAF 'MOVEWINDOW K (* 6 K) T))
   ->  NIL
```

(3) Send the STD_GRAF window a message to change its size.

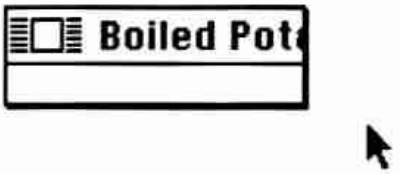
```
(SEND STD_GRAF 'SIZEWINDOW 100 12)
   ->  T
```



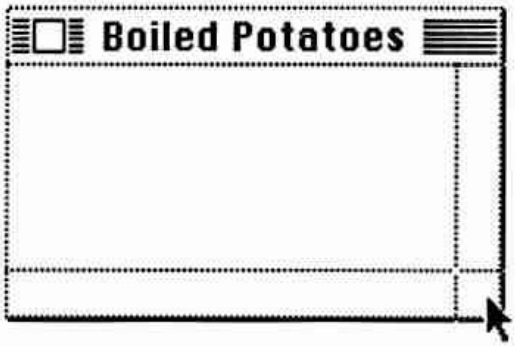
(4) Try some other messages to the window.

☞ See the section More On Graphics for more information on window management. Also see Windows in the listing by category found in the Reference Section.

```
(SEND STD_GRAF 'GETWTITLE)
   ->  "Graphics Window"
(SEND STD_GRAF 'SETWTITLE "Boiled Potatoes")
   ->  "Boiled Potatoes"
```



(5) Resizing the Boiled Potates window enables the whole title to come into view.



2-Dimensional Bunny Graphics

Initially, ExperLisp is equiped with a single 2-dimensional bunny. This bunny is known as the default bunny. A pointer to this default bunny is stored in the system variable CURBUN. Unless otherwise directed, all bunny commands are carried out by CURBUN. Dispose of the current Graphics Window by clicking the mouse in the hide box of the Graphics Window (found in the upper left corner of the window).

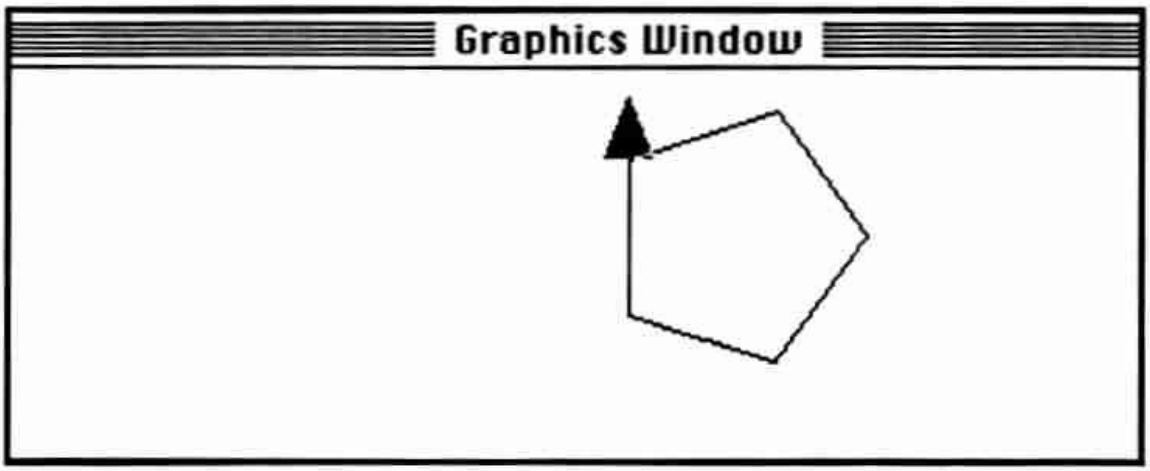
Every Graphics window has a 'home' position. This is the center of the window at the time it was created. Resizing the window does not move this home position. This home position has the Cartesian coordinate of (0, 0). The section More On Graphics has more information on the Graphics Window.

(1) Make the bunny CURBUN visible. This should slow the graphics enough to make its movement visible. Only bunnies which are 2-dimensional can be made visible.

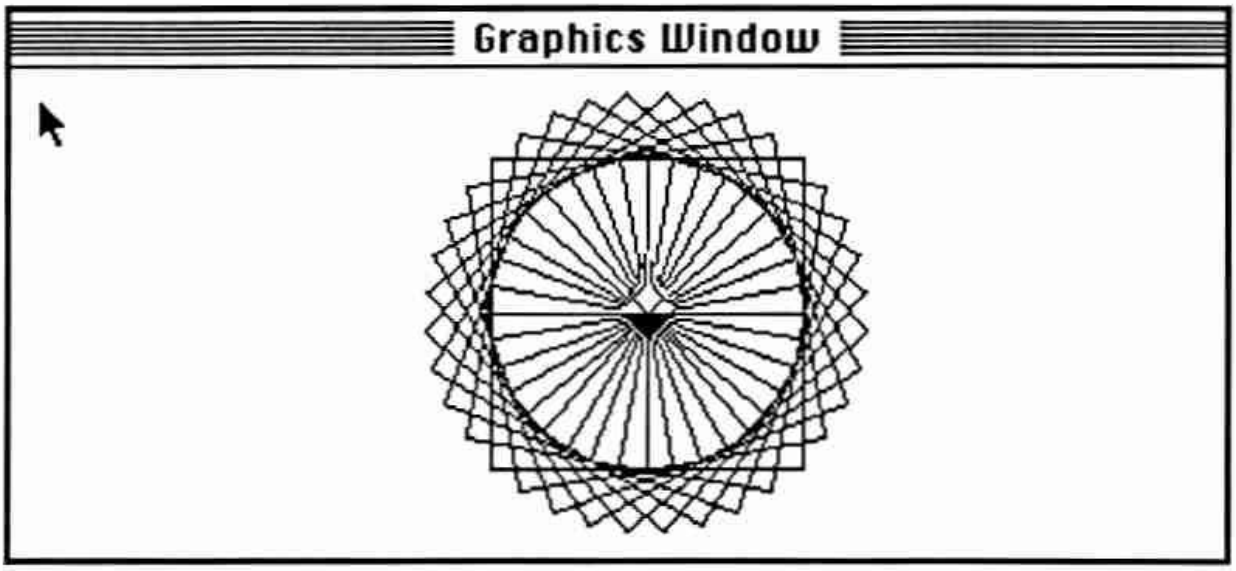
```
(SHOWBUNNY)
   ->  T
```

(2) Test drive the bunny with the following lines.

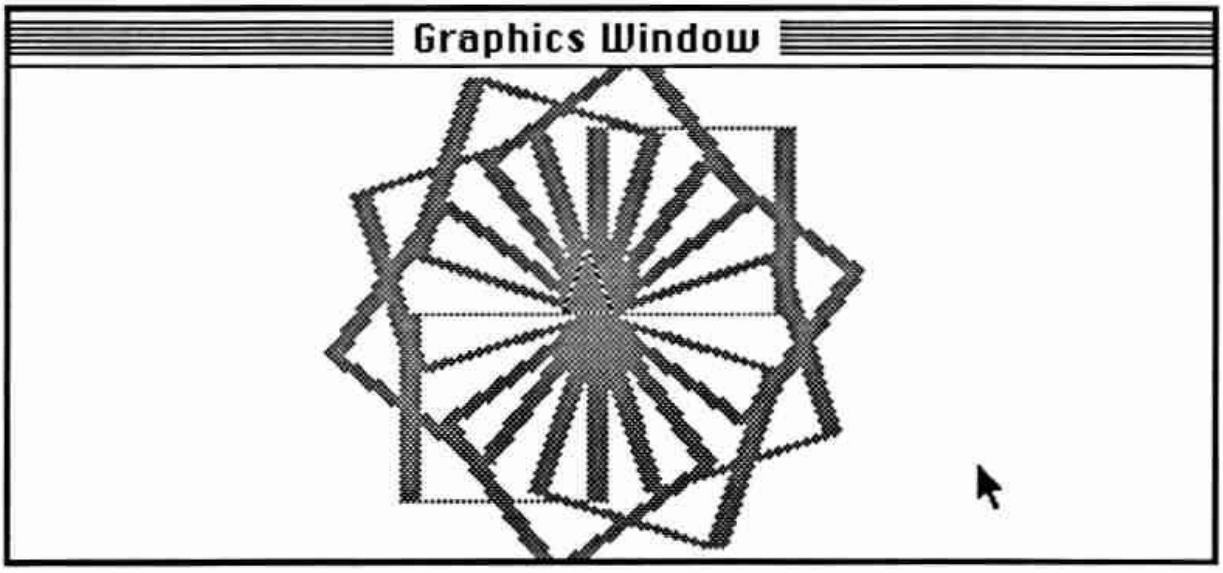
```
(FORWARD 50)
   ->  50
(DOTIMES (K 5) (RIGHT 72) (FORWARD 50))
   ->  NIL
```




```
(HOME)
   ->   T
(CLEARSCREEN)
   ->  T
(DEFUN SQUARE (X) (DOTIMES (K 4) (FORWARD X) (RIGHT 90)))
   ->  SQUARE
(DOTIMES (K 36) (SQUARE 50) (LEFT 10))
   ->  NIL
```




```
(CLEARSCREEN)
   ->  T
(PENPAT GRAY)
   ->  T
(PENSIZE 7 1)
   ->  T
(DOTIMES (K 10) (SQUARE 60) (LEFT 36))
   ->  NIL
```



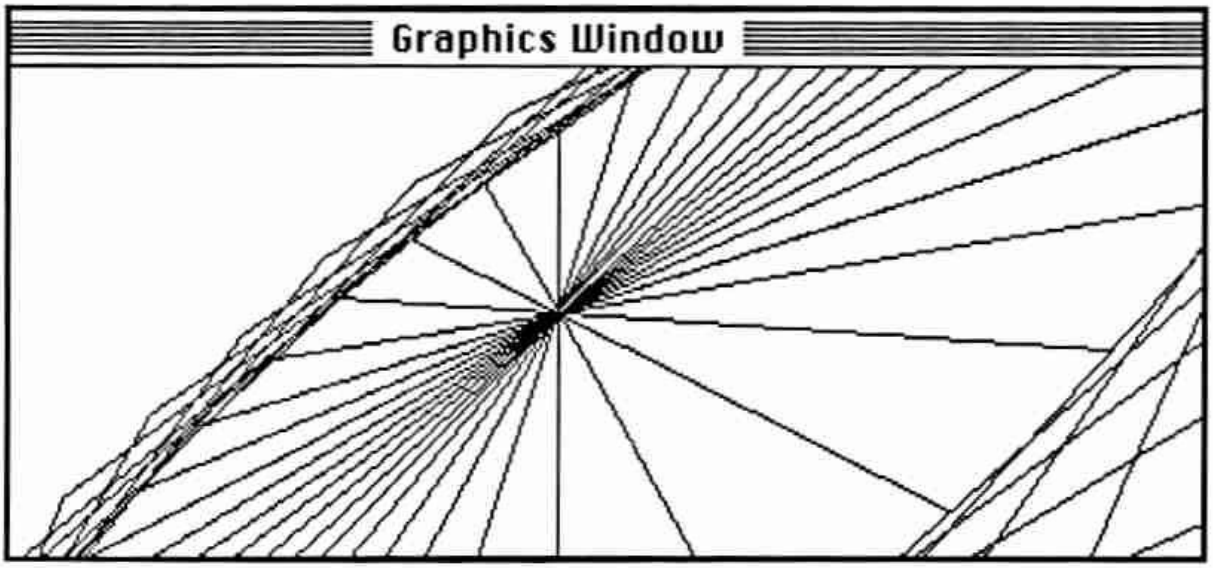
3-Dimensional Bunny Graphics

(1) Change the CURBUN bunny to a new 3-dimensional bunny. 3-dimensional bunnies are never visible.

```
(SETQ CURBUN (NEW3DBUN))
   ->  ADDRESS #68011E06
```

(2) Clear the screen and make way for some 3-dimensional graphics. The 3-dimensional bunny responds to commands such as: ROLL, PITCH, and YAW. See 3-D Bunny Graphics in the listing by category found in the Reference Section for more information on 3-dimensional bunny commands.

```
(CLEARSCREEN)
   ->  T
(PENNORMAL)
   ->  T
(PITCH -70)
   ->  -70
(ROLL 35)
   ->  35
(DOTIMES (K 36) (SQUARE 250) (LEFT 10))
   ->  NIL
```



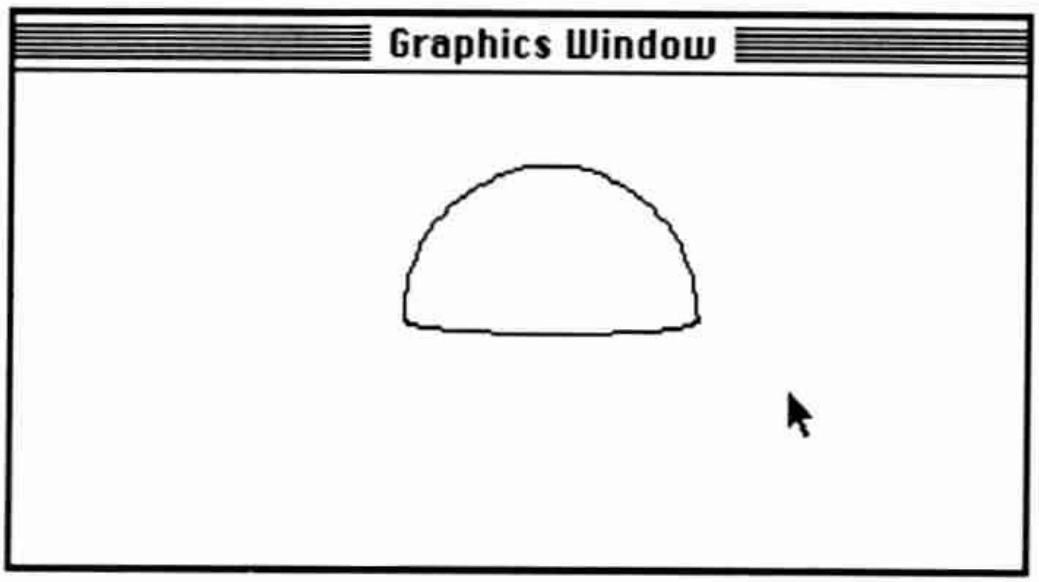
Spherical Bunny Graphics

(1) Change the CURBUN bunny to a new spherical bunny. Spherical bunnies are never visible.

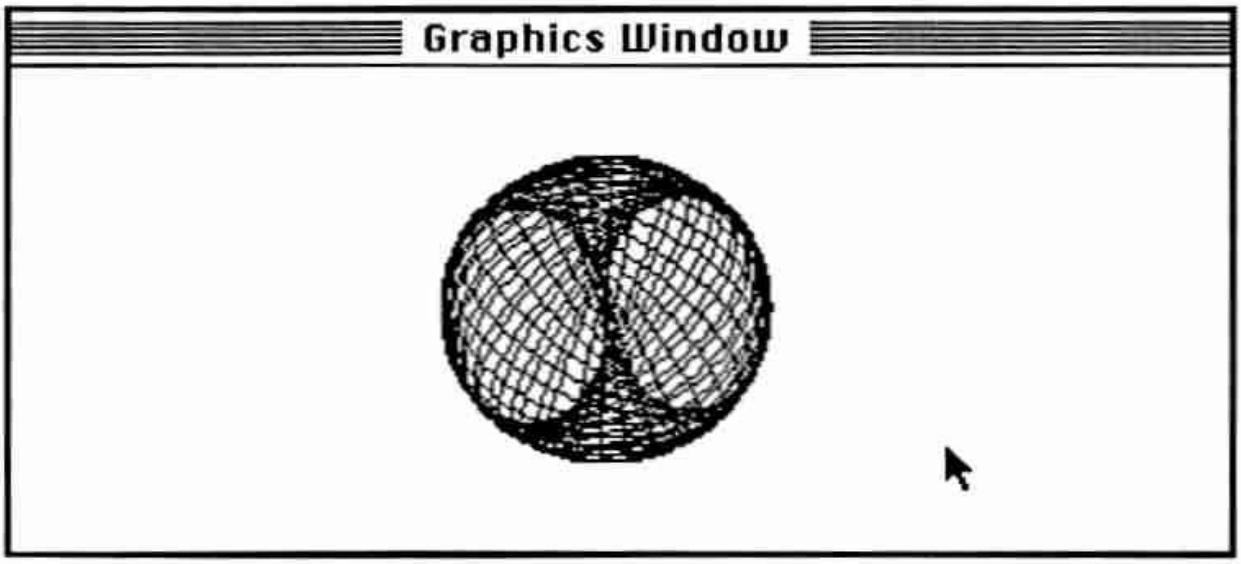
```
(SETQ CURBUN (NEWSPBUN))
   ->  ADDRESS #68011DFA
```

(2) Clear the screen and make way for some spherical graphics. The spherical bunny responds to commands such as: FORWARD, BACK, LEFT, and RIGHT. It moves in polar coordinates. Therefore, a (FORWARD 90) always moves the bunny 90° around the sphere - regardless of sphere size. The Spherical Bunny Graphics in the listing by category found in the Reference Section has more information on spherical bunny commands.

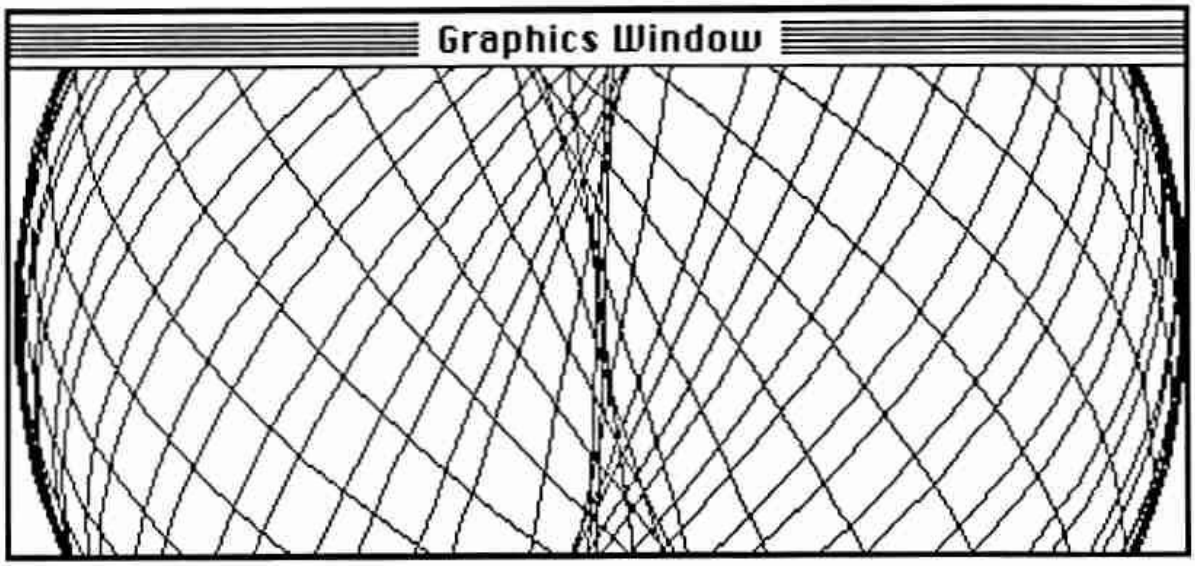
```
(CLEARSCREEN)
   ->  T
(FORWARD 180)
   ->  180
(LEFT 85)
   ->  85
(FORWARD 180)
   ->  180
```



```
(HOME)
   ->  T
(CLEARSCREEN)
   ->  T
(DOTIMES (K 36) (FORWARD 370) (RIGHT 10))
   ->  NIL
```



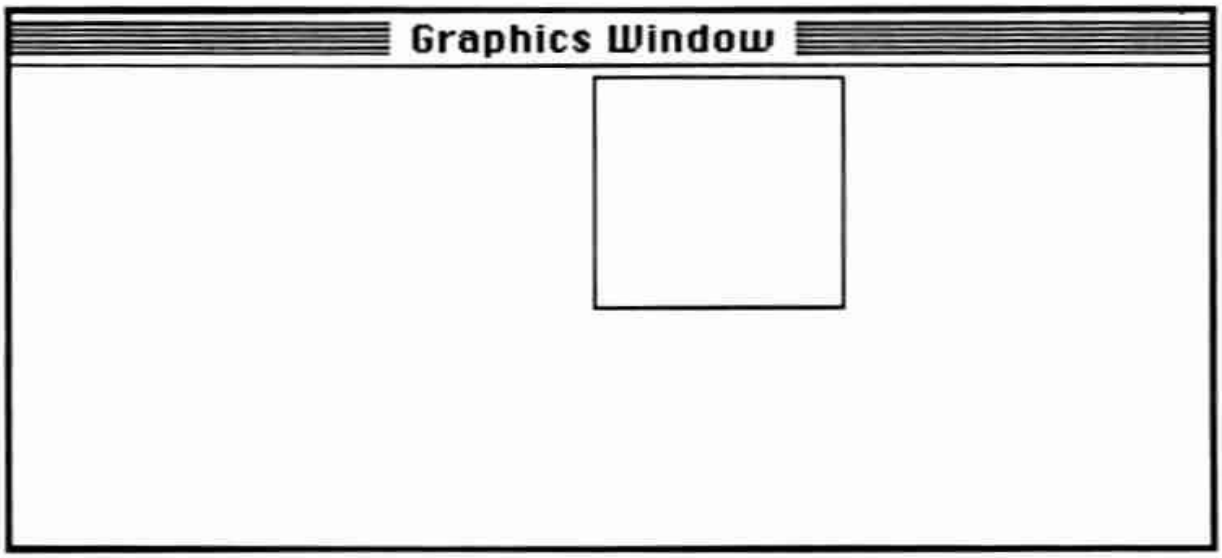

```
(SETRADIUS 200)
   ->  200
(HOME)
   ->  T
(CLEARSCREEN)
   ->  T
(DOTIMES (K 36) (FORWARD 370) (RIGHT 10))
   ->  NIL
```



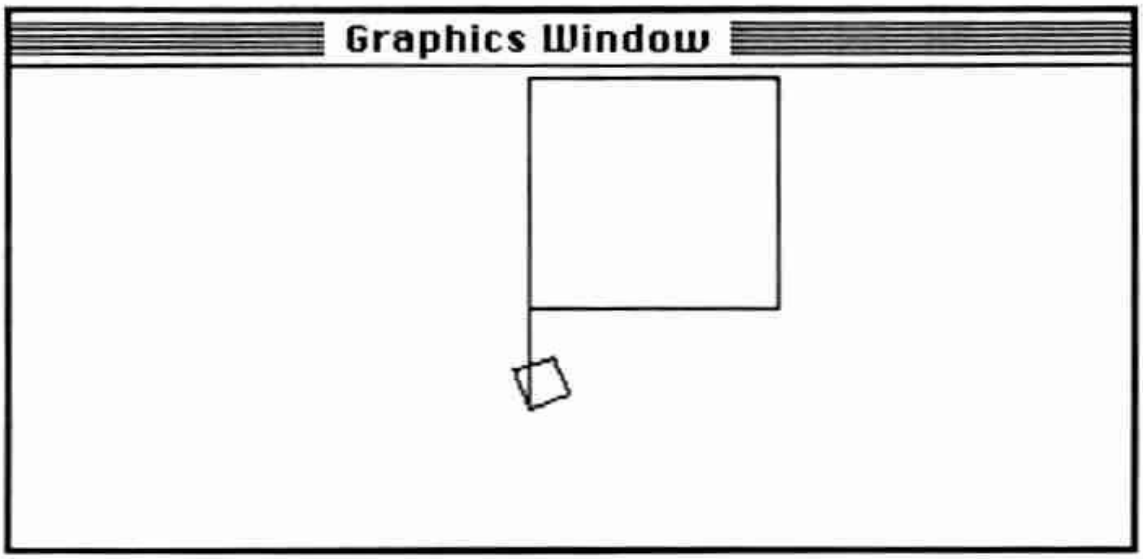
Hutch Managenent

As previously mentioned, there 3 types of bunnies: 2-dimensional, 3-dimensional, and spherical. CURBUN refers to but one bunny. Additional bunnies can be created to cohabitate with CURBUN. The macro function TELL facilitates the sending of commands to a particular bunny.

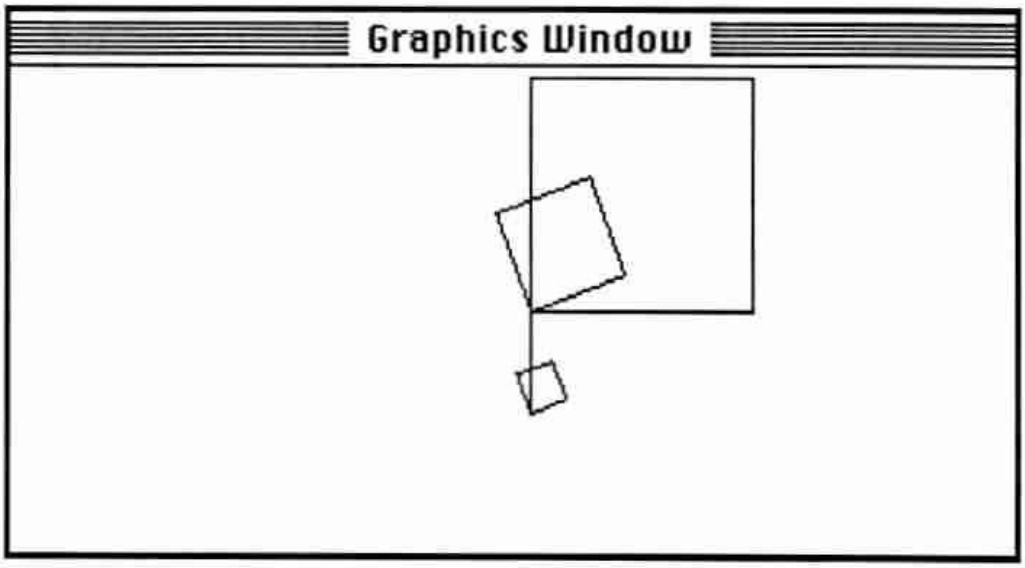
```
(CLEARSCREEN)
   ->  T
(SETQ TWOD (NEW2DBUN))
   ->  ADDRESS #68011E0A
(SETQ TWODTOO (NEW2DBUN))
   ->  ADDRESS #68011E02
(TELL TWOD (SQUARE 76))
   ->  NIL
```




```
(TELL TWODTOO (BACK 33) (LEFT 20) (SQUARE 13))
   ->  NIL
```




```
(TELL TWOD (LEFT 20) (SQUARE 34))
   ->  NIL
```



It is recommended that a bunny of each of the three types be created and saved away, rebinding CURBUN to the value of each bunny as needed, or using TELL. Letting the bunnies run rampant and multiply can result in reduced performance.

☞ In other words, do not do a `(DOTIMES (J 400) (NEW3DBUN))`

☞ Depressing the command-period `(.)` keys simultaneously breaks the execution of any ExperLisp program. The command key lies to the immediate left of the space bar.

Quickdraw Graphics

Quickdraw functions, like those of bunny graphics, carry out commands through a graphical object. This object is called the pen. Changing the characteristics of the pen affects the images that Quickdraw creates, as well as some of the images that bunnies create. In a previous example the Quickdraw function PENSIZE was used to modify the line drawn by a bunny.

Experience will show that there are many close relations between the Quickdraw graphics and the bunny graphics. See Quickdraw in the listing by category, found in the Reference Section, for more information on Quickdraw commands.

Example 1

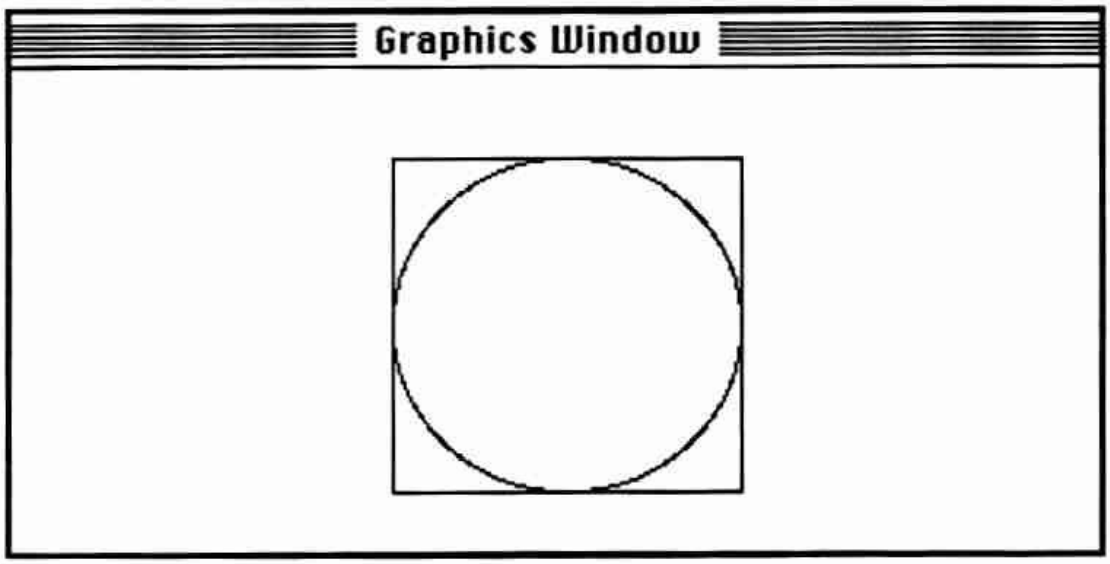
Quickdraw makes it very easy to paint large sections of a Graphics Window with a variety of shapes.

(1) Clear the current Graphics Window.

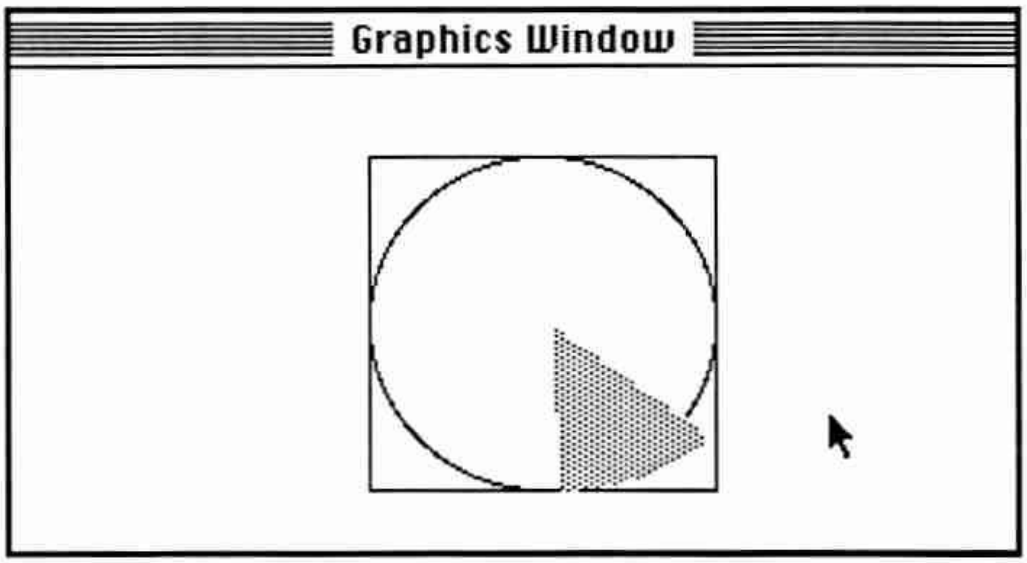
```
(CLEARSCREEN)
   ->  T
```

(2) Frame some shapes. These functions take a boundary list as its argument. The boundary list contains 4 coordinates providing the positions of the top, left, bottom, and right sides.

```
(FRAMERECT '(-50 -50 60 60))
   ->  T
(FRAMEOVAL '(-50 -50 60 60))
   ->  T
```




```
(FILLARC '(-50 -60 60 87) 132 45 LTGRAY)
   ->  T
```



Quickdraw Regions and Polygons

Rectangles and ovals are among the predefined shapes manipulated by Quickdraw. Quickdraw also allows the creation of areas of any closed shape. These arbitrary shapes are divided into two types: regions and polygons. Polygons allow the definition of an arbitrary closed shape. Commands to fill, paint, erase, frame, and offset polygons are supported. Regions posess the same features as polygons plus the ability to define new regions by performing the following operations on two given regions: intersection, union, difference, and exclusive-or.

(1) Define some functions to help us quickly create shapes. Define these in the Edit Buffer.

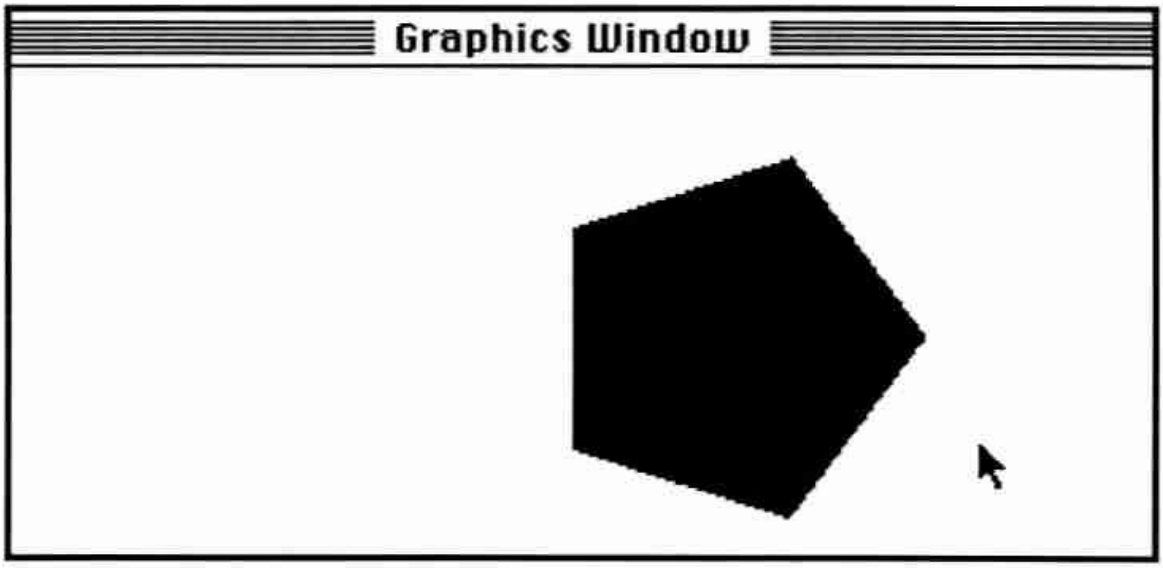
```
(DEFUN POLY (SIDES &AUX (SIZE (/ 360 SIDES)))
   (DOTIMES (INDEX SIDES)
            (FORWARD SIZE)
            (RIGHT SIZE)))

(DEFUN STARZ (SIZE)
   (DOTIMES (INDEX 5)
            (FORWARD SIZE)
            (RIGHT 144)))
```

(2) NEWRGN returns a handle for a region definition. OPENRGN create a temporary storage area for a region definition. The definition is assigned to the region whose handle is supplied to CLOSERGN. A regions definition is defined by all of the graphics commands executed between an OPENRGN and a CLOSERGN.

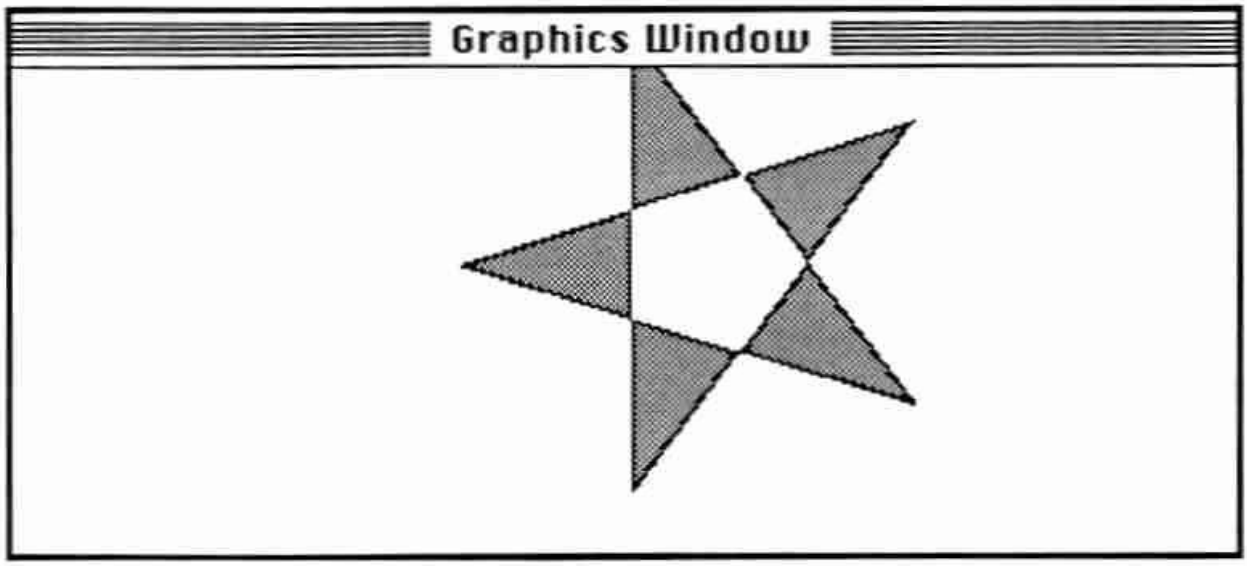
☞ Notice that the bunny commands executed when defining a region do not result in anything visible. They merely define the region. The defined region becomes visible when a command to draw the region is executed (such as PAINTRGN). The documentation of example functions is in the Reference Section.

```
(CLEARSCREEN)
   -> T
(SETQ CURBUN TWOD)
   -> ADDRESS #68011DDE
(SETQ RGN1 (NEWRGN))
   -> ADDRESS #68011DDE
(HOME)
   -> T
(BACK 45)
   -> 45
(OPENRGN)
   -> T
(POLY 5)
   -> NIL
(CLOSERGN RGN1)
   -> ADDRESS #68011DDE
(PAINTRGN RGN1)
   -> T
```

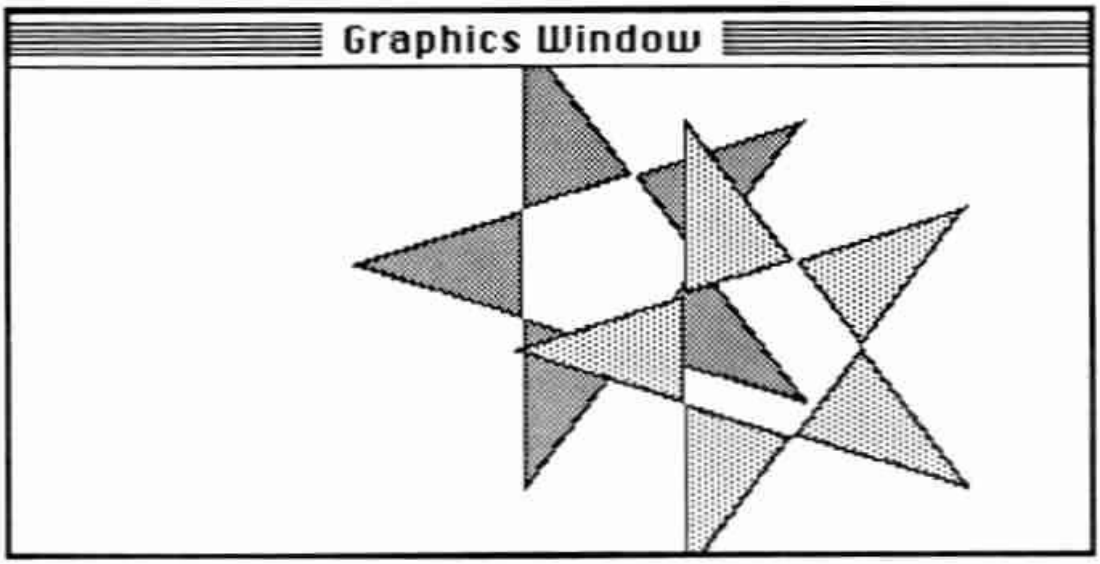


(3) The handle used to represent a region definition can be recycled to store other definitions.

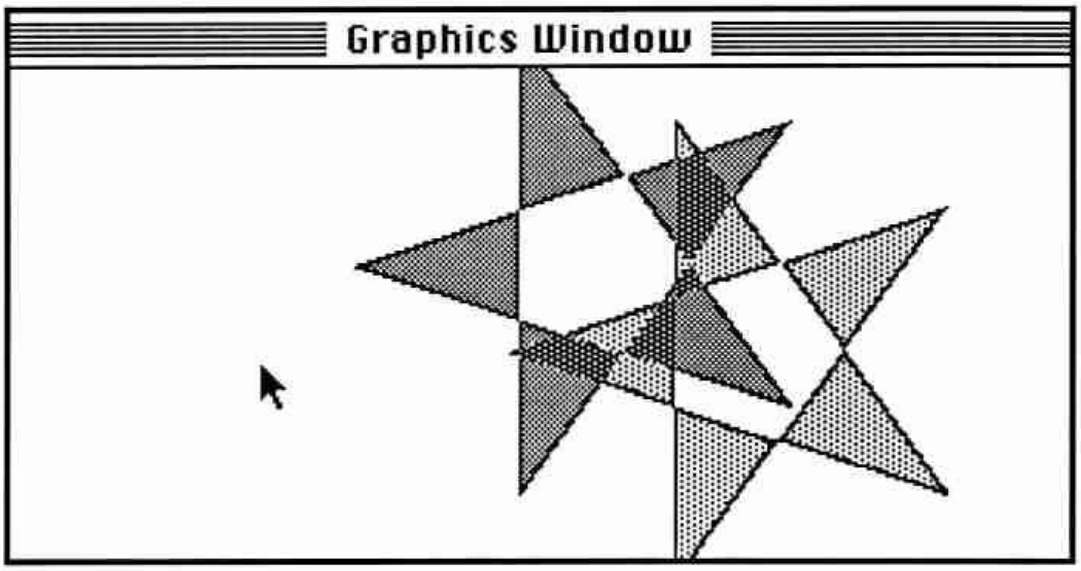
```
(CLEARSCREEN)
   ->  T
(HOME)
   ->  T
(OPENRGN)
   ->  T
(BACK 60)
   ->  60
(STARZ 150)
   ->  NIL
(CLOSERGN RGN1)
   ->  ADDRESS #68011DDE
(FILLRGN RGN1 GRAY)
   ->  T
(FRAMERGN RGN1)
   ->T
```




```
(SETQ RGN2 (NEWRGN))
   ->  ADDRESS #68011DE2
(COPYRGN RGN1 RGN2)
   ->  ADDRESS #68011DE2
(OFFSETRGN RGN2 50 38)
   ->  T
(FILLRGN RGN2 LTGRAY)
   ->  T
(FRAMERGN RGN2)
   ->  T
```



```
(SECTRGN RGN1 RGN2 RGN2)
   ->  T
(FILLRGN RGN2 DKGRAY)
   ->  T
```



(4) Because regions takes up a great deal of memory, it is recommended that old region definitions be disposed of when when the fun is done. Polygons are slightly more memory efficient. Use the function DISPOSERGN to dispose of a region; and KILLPOLY to dispose of a polygon.

```
(DISPOSERGN RGN1)
   ->  T
(DISPOSERGN RGN2)
   ->  T
```

Menu Management

ExperLisp allows the creation of user defined menus. You can add your menus to the existing menu bar, or create a new menu bar. The examples below illustrate how to define a new menu whose items run other Lisp programs. Consult the Reference Section for more information on individual menu functions.

Example 1

(1) Create a new menu using NEWMENU. NEWMENU uses the integer number as the menu ID, and the string as the menu's title. It returns a handle to the menu. Notice that this does not add the menu to the menu bar. This will come later.

☞ The menu ID number must be greater than 9.

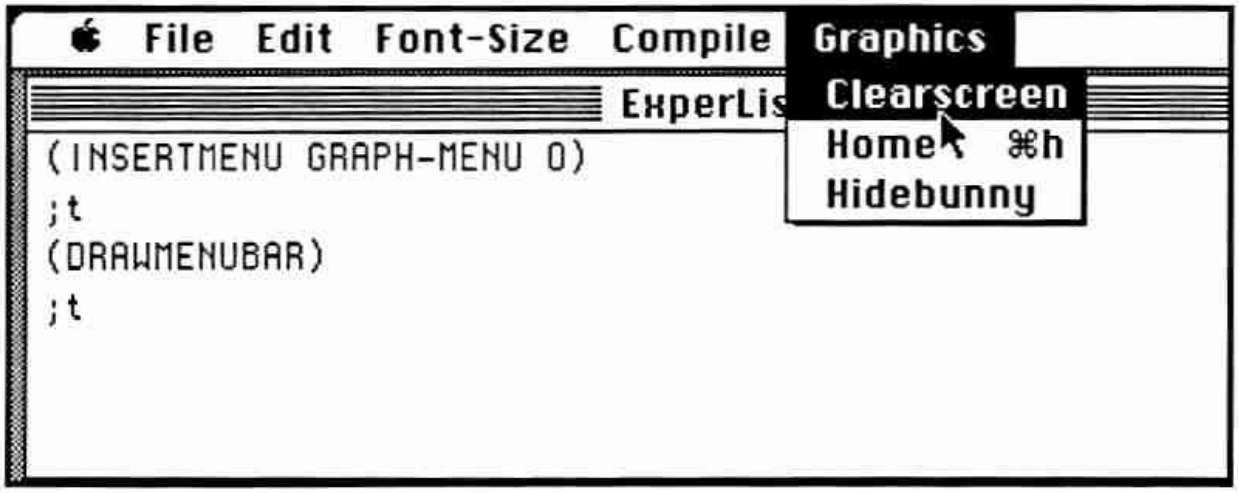
```
(SETQ GRAPH-MENU (NEWMENU 10 "Graphics"))
   ->  ADDRESS #68011EE8
```

(2) Add item to the menu using APPENDMENU. APPENDMENU takes the menu handle and a string containing the names of the menu items (separated by a semi-colon), plus additional menu information. In the example below, the menu item "Home" is given the keyboard equivalent of Command-h. The Reference Section documentation for the function APPENDMENU has more information on menu item options.

```
(APPENDMENU GRAPH-MENU "Clearscreen;Home/h;Hidebunny")
   ->  T
```

(3) Insert the menu into the system by using INSERTMENU. Then make the menu visible by using DRAWMENUBAR. INSERTMENU takes a menu handle and an integer number to position the menu. 0 (zero) tells INSERTMENU to place the menu to the right of the last menu on the menu bar (see illustration below).

```
(INSERTMENU GRAPH-MENU 0)
   ->  T
(DRAWMENUBAR)
   ->  T
```



(4) ExperLisp now has the menu, but it doesn't know what to do when an item from that menu is selected. This information is provided to the system through the global system variable ᵃMENULIST (the ᵃ character can be typed by depressing the "Option" and "9" keys simultaneously). Each element of the menu list contains a 'key' for matching, and then a Lisp form to compile if this key is matched. Type the following into the Edit Buffer, select the text, and then select **Compile Selection** from the **Compile** menu. Using APPEND to add a list of elements prevents the destruction of the current ᵃMENULIST. This simplifies adding and deleting elements.

```
(SETQ ᵃMENULIST (APPEND '( ((10 1) (CLEARSCREEN))
                           ((10 2) (HOME))
                           ((10 3) (TOGGLE-BUNNY)) )
                        ᵃMENULIST))
```

Below lies the disection of an element of ᵃMENULIST. When an item is selected from a menu, ExperLisp looks for an element of ᵃMENULIST whose key contains the menu ID of the selected menu, and the menu item of the selected item. It then compiles and runs the form of the element.

`((10 1) (CLEARSCREEN))`	is an element of ᵃMENULIST.
`(10 1)`	is the key of the element.
`10`	is the menu ID of a menu.
`1`	is the menu item of the menu.
`(CLEARSCREEN)`	is the form to compile when this menu item is selected.

(5) The last step is insuring that we have something for the menu to do. The function TOGGLE-BUNNY is called when the item **Hidebunny** is selected from our menu (menu ID of 10, menu item of 3). TOGGLE-BUNNY refers to a new global variable (so sorry) to find the current bunny state.

If TOGGLE-BUNNY is called and the BUNNY-STATE is 'S (visible), then hide the bunny, reset the BUNNY-STATE, and modify the menu to say "Showbunny" rather than "Hidebunny."

If TOGGLE-BUNNY is called and the BUNNY-STATE is 'H (hidden), then show the bunny, reset the BUNNY-STATE, and modify the menu to say "Hidebunny" rather than "Showbunny."

Define TOGGLE-BUNNY from the Edit Buffer.

```
(DEFUN TOGGLE-BUNNY ()
   (COND ((EQ BUNNY-STATE 'S)
          (HIDEBUNNY)
          (SETQ BUNNY-STATE 'H)
          (SETITEM GRAPH-MENU 3 "Showbunny"))
         (T
          (SHOWBUNNY)
          (SETQ BUNNY-STATE 'S)
          (SETITEM GRAPH-MENU 3 "Hidebunny"))))
```

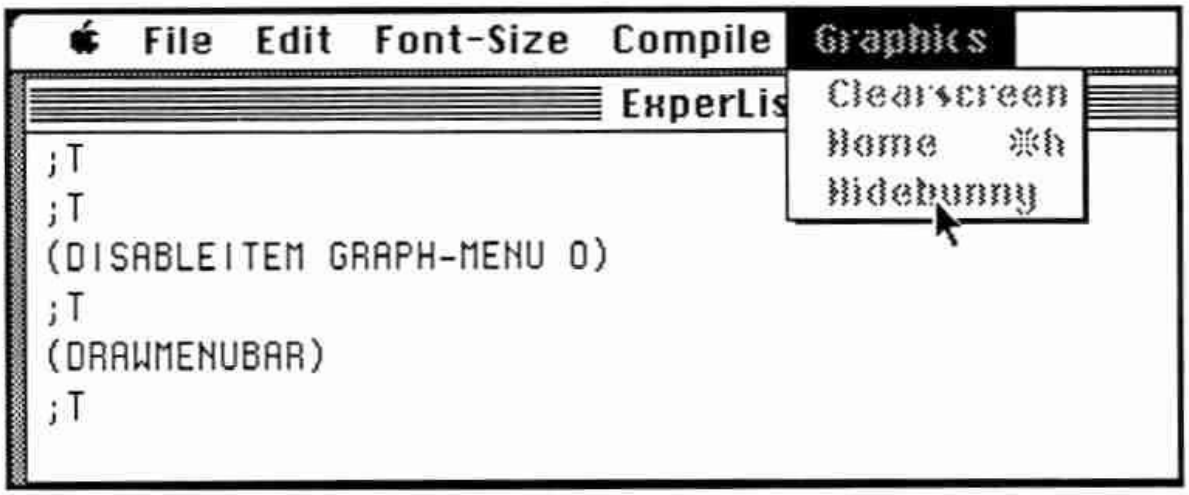
(6) Bind the global variable BUNNY-STATE to 'S.

```
(SETQ BUNNY-STATE 'S)
   ->  S
```

(7) The Graphics menu should be already to go. But before that, experiment with some other menu commands. DISABLEITEM prevents individual menu items from being selected. Providing DISABLEITEM with 0 (zero) prevents the selection of any items from a menu. A DRAWMENUBAR must be done to reflect the changes in the menu's appearance.

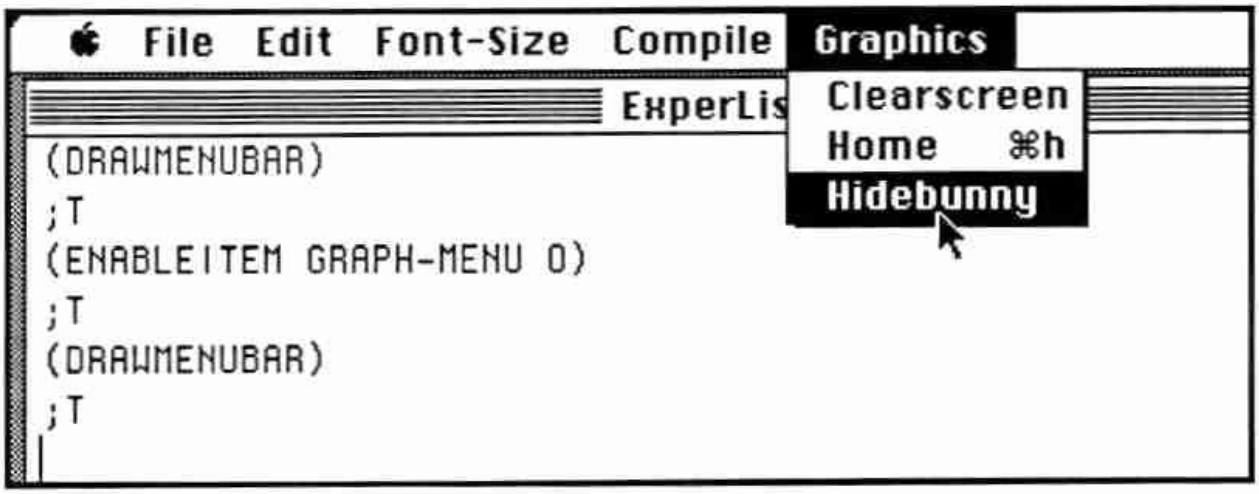
```
(DISABLEITEM GRAPH-MENU 0)
   ->  T
```

```
(DRAWMENUBAR)
   ->  T
```



(8) ENABLEITEM enables a menu item. Though in this case it enables an entire menu.

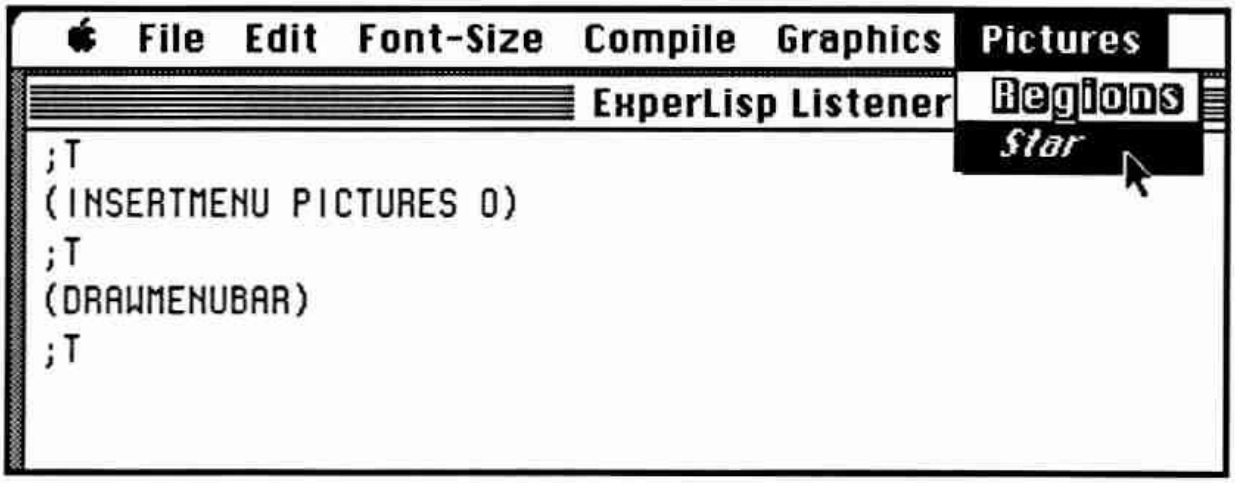
```
(ENABLEITEM GRAPH-MENU 0)
   ->  T
(DRAWMENUBAR)
   ->  T
```



Example 2

(1) Create a new menu. The string given to APPENDMENU has some new format commands which alter the style of its menu items. Consult APPENDMENU in the reference section for more information.

```
(SETQ PICTURES (NEWMENU 11 "Pictures"))
   ->  ADDRESS #68011EB0
(APPENDMENU PICTURES "Regions<S;Star<I")
   ->  T
(INSERTMENU PICTURES 0)
   ->  T
(DRAWMENUBAR)
   ->  T
```



(2) Compile the following functions. These are the functions that our new menu Pictures will call when an item from that menu is selected.

```
(DEFUN STAR (SIZE)
   (DOTIMES (I 36)
            (SQUARE SIZE)
            (LEFT 10)))
```

```
(DEFUN SQUARE (SIZE)
   (DOTIMES (I 4)
            (FORWARD SIZE)
            (RIGHT 90)))

(DEFUN REGION-PICT (&AUX RGN)
   (SETQ RGN (NEWRGN))
   (OPENRGN)
   (DOTIMES (I 5)
            (PENTAGON 70)
            (RIGHT 72))
   (CLOSERGN RGN)
   (FILLRGN RGN LTGRAY)
   (FRAMERGN RGN)
   (DISPOSERGN RGN))

(DEFUN PENTAGON (SIZE)
   (DOTIMES (I 5)
            (FORWARD SIZE)
            (RIGHT 72)))
```

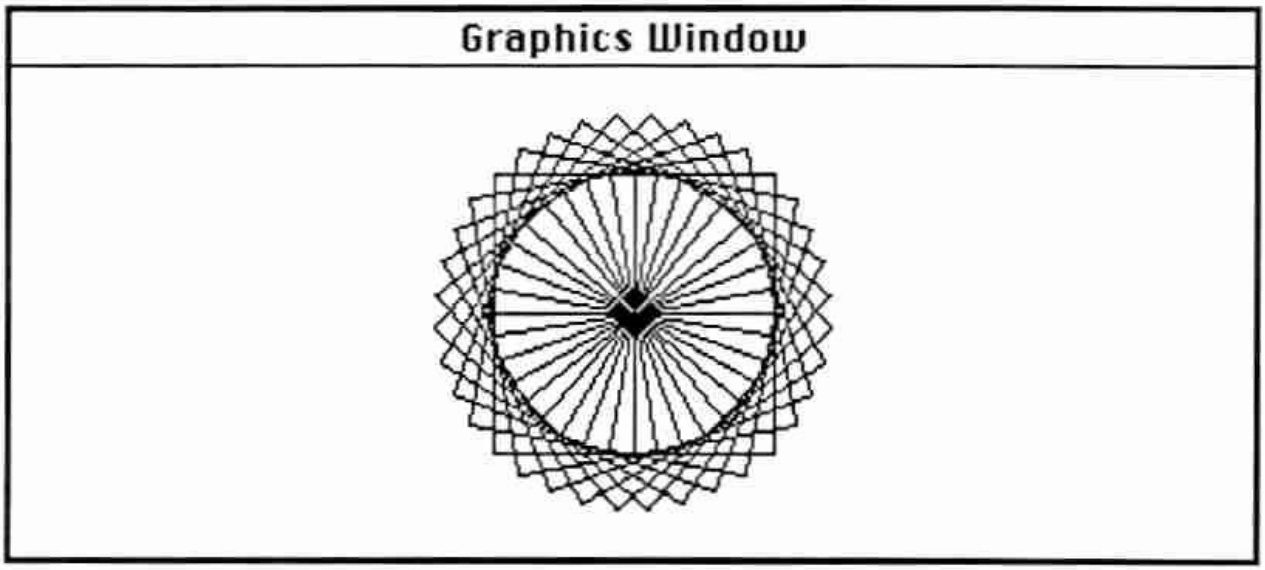
(3) Add some elements to the *MENULIST. Notice that the forms to be compiled (within an element) can call several lisp functions. You can put any single form in that position - perhaps a DOTIMES full of dozens of Lisp forms. The longer the form, the more time the system will take to react.

☞ Each form is compiled each and every time that menu item is selected.

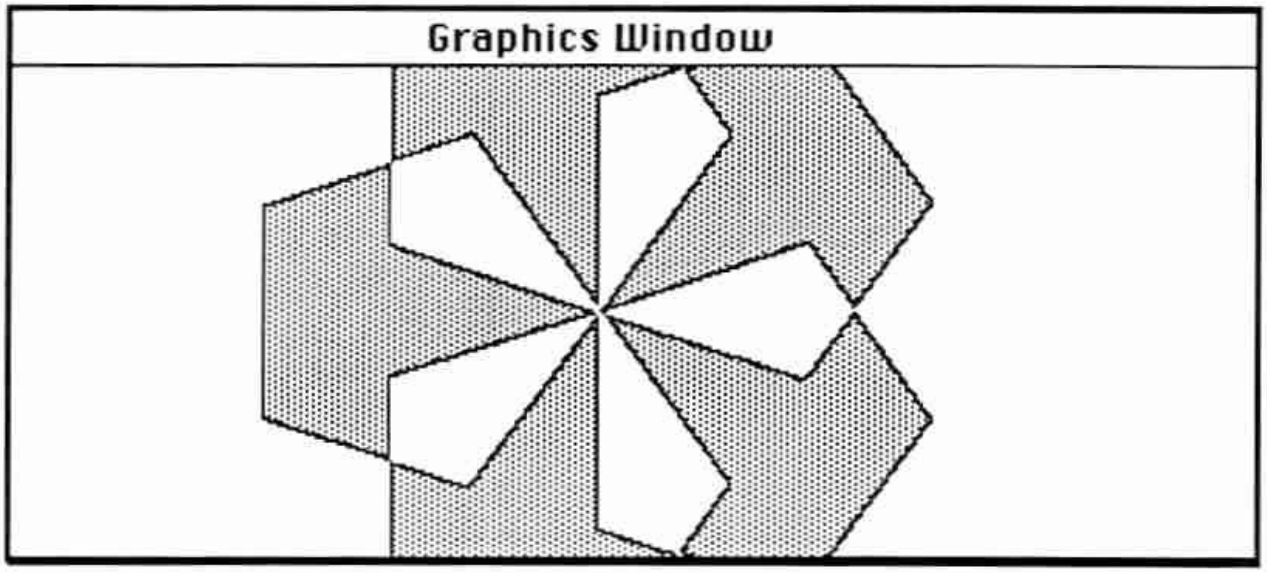
```
(SETQ *MENULIST (APPEND '( ((11 1) (REGION-PICT))
                           ((11 2) (STAR (RANDOM 200))) )
                        *MENULIST))
```

(4) Enjoy yourself (see illustrations below). These were created by selecting items from the **Pictures** menu.

This was created by selecting **Star** from the **Pictures** menu. Your Star may not look exactly like this because Star is given a random number as its argument for SIZE.



This was created by selecting **Regions** from the **Pictures** menu.



This was created for your enjoyment.