Advanced AutoLISP Programming Correspondence Course

by Tony Hotchkiss, Assistant Professor, State University College at Buffalo, New york. He may be reached at HOTCHKAJ@SNYBUFAA.CS.SNYBUF.EDU. At the time this course was developed he was an Assistant Professor in the Department of Engineering Professional Development, University of Wisconsin-Madison.

Study Guide, part **1** of **8**

Lesson One - AutoLISP Programming Techniques

Solutions to the homework questions:

1. Question: Using a program format similar to the prototype structure in the lesson, how do you determine which variables are to be designated as 'local' in each module of the program?

Answer: The prototype program format, includes an error handler, an initial system variable setting function, an input section, the calculation/processing function(s), an output section, a system variable re-setting function, and the main command function that executes all of the above. For a program called 'PROG.LSP', with a main command called C:PROG, the prototype lisp program format is:

```
PROG.LSP - a typical program structure
;;; PROG.LSP  A program to solve any problem!
(defun err (s)
   (prompt "\nThe new error handler goes here..")
); end of error function
(defun setting ()
   (prompt "\nThis is where you set sysvars...")
); end of setting system variables
(defun input ()
   (prompt "\nThis is the input section...")
); end of defun
(defun calc ()
   (prompt "n\This is the calculation section...")
); end of defun
(defun output ()
   (prompt "\nThis is the results section...")
); end of defun
(defun resetting ()
   (prompt "\nReset sysvars to previous values...")
); end of defun
(defun C:PROG ()
   (setting)
   (input)
   (calc)
   (output)
   (resetting)
   (princ)
); end of defun C:PROG
```

There may be user defined functions in your program that are not called in the main C:PROG section. For instance, a function to convert degrees to radians, which may be called by the (calc) function, does not appear in the list of functions called by C:PROG. In general, any group of statements that are used more than once in a program may be written as a separate defined function, and so your programs will quite possibly contain a great many defined functions that do not appear in the list called by C:PROG.

As another example, the error function is not explicitly called by C:PROG, but is called automatically by AutoLISP under the name *error*. All you need to do to ensure that your error function is automatically called, is to rename it in the setting function as shown in the lesson, by saving the old error function and renaming the new one.

With so many (defuns...) all over the program, the use of local variables must be addressed. You use local variables to save memory and to avoid the risk of overwriting global variables. Local variables do not appear in the atomlist, so they can not be used by other defined functions in your program, unless they are passed as arguments of the function.

When many defined functions use the same variable, its value must be retained by allowing it to be global. This means that if a modular programming technique is used, you will generally create more global variables than you would need to if a single defined function is used for the entire program. Defined functions 'return' values which can be assigned to variables by the 'calling' expression, so you can minimise the generation of global variables.

Use your judgement when you separate your programs into modules. In our example of the FASTFACE.LSP, it is clearly not sensible to insist on separate (input), (calc), and (output) functions, because the first two of these functions only acquire the name of a file, so instead of:

```
(defun input ()
   (setq fname (getstring "\nTXT file name: "))
); end of input
(defun calc ()
   (setq fname (strcat fname ".txt"))
); end of calc
(defun output (/ f1 numlines p px py pz pp)
   (setq f1 (open fname "r"))
   (if f1    ;test if file is open
      (progn  ;then read the first line of data
```

We could have:

```
(defun output (/ fname f1 numlines p px py pz pp)
   (setq fname (getstring "\nTXT file name: "))
   (setq fname (strcat fname ".txt"))
   (setq f1 (open fname "r"))
   (if f1    ;test if file is open
      (progn  ;then read the first line of data
```

In the second case, we can include the name of the file 'fname' as a local variable. The important concept here is that there is always some compromise or trade-off involved in optimizing programs for efficient use of the computer resource. Making all of your variables local minimizes memory use for variable storage, and using modules saves repetition and improves execution speed, but may require more global variables, unless you are creative in your use of 'returned' values from the (defun)'s.

2. **Question:** If you wanted to create a '3dmesh' instead of a 'pface' mesh, what would be a suitable format of the external data file? Look in the AutoCAD Reference Manual to verify the required input.

Answer: The AutoCAD 3dmesh defines a three-dimensional polygon mesh that is topologically rectangular. Being topologically rectangular has the effect of starting with a rectangular elastic sheet, and forming it by stretching and shrinking it onto a molded shape. If you drew a set of lines that were parallel to each of the edges of the original rectangular sheet, each small square on the sheet would be distorted on the molded form, but they would still retain four corners connected by edges.

The edges of the original rectangle are termed the 'M' and 'N' edges, and the number of parallel divisions are the mesh M and N size, so that a typical surface mesh would resemble that of figure SG1-1. The mesh is similar to the pface mesh, except that it can be edited more conveniently as a single entity, rather than as a collection of individual faces. The 3dmesh can be 'closed' in the M and/or N directions by editing with the pedit command.

The vertex points in figure SG1-1 represent the order in which the points are supplied, and they appear as (M, N) pairs.

The 3dmesh command requires that the Mesh Msize and the Mesh Nsize be specified with the prompts:

```
Command:  3dmesh
Mesh M size:
Mesh N size:
```

Integer values between 2 and 256 are allowed for each M and N size. The total number of vertices of a 3dmesh surface is M x N, and that is the required number of points in your external data file.

After you supply the M and N size, the 3dmesh command prompts you to enter the vertices in the order of M, N, such that all of the N values for a given M are entered in turn. The prompts ask for the vertex points in order, numbering from zero for each of the M and N directions as follows in this example of a 4 x 5 mesh:

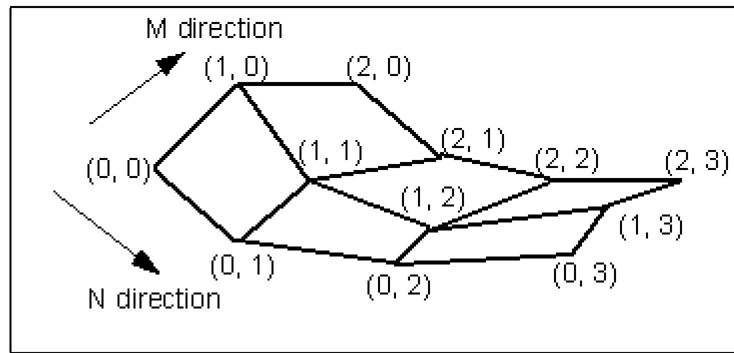

Figure SG1-1 The 3D Mesh

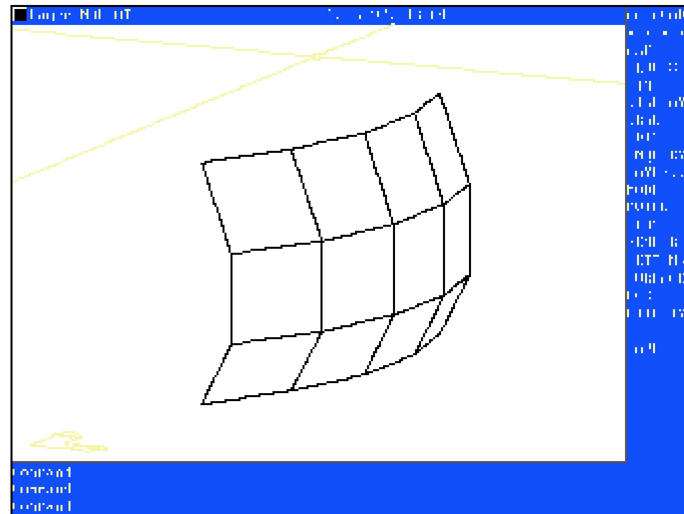

Figure SG1-2 A 3D mesh created from vertices supplied by an external file

```
Command:  3dmesh
Mesh M size: 4
Mesh N size:  5
Vertex (0, 0):  5.000  0.000  3.000
Vertex (0, 1):  6.613  0.000  2.139
Vertex (0, 2):  8.381  0.000  1.673
Vertex (0, 3):  10.20  0.000  1.626
Vertex (0, 4):  12.00  0.000  2.000
Vertex (1, 0):  4.000  1.250  2.000
Vertex (1, 1):  5.613  1.250  1.139
Vertex (1, 2):  7.381  1.250  0.674
Vertex (1, 3):  9.209  1.250  0.626
Vertex (1, 4):  11.00  1.250  1.000
Vertex (2, 0):  4.000  2.750  2.000
Vertex (2, 1):  5.613  2.750  1.139
Vertex (2, 2):  7.381  2.750  0.674
Vertex (2, 3):  9.209  2.750  0.626
Vertex (2, 4):  11.00  2.750  1.000
Vertex (3, 0):  5.000  4.000  3.000
Vertex (3, 1):  6.613  4.000  2.139
Vertex (3, 2):  8.381  4.000  1.673
Vertex (3, 3):  10.20  4.000  1.626
Vertex (3, 4):  12.00  4.000  2.000
Command:
```

The mesh created by the above input is shown in figure SG1-2.

The external data file should give the M and N values, followed by the vertex points in the order required, so for the example of figure SG1-2, the input data file will contain the data of table 1-1.

I used scientific notation in my external data file because I generated the file automatically from the vertex points of 3D polylines, and I wanted to ensure that the numbers appeared in columns with a field width of 14 characters each, including two spaces separating the columns.

In the lesson, in order to generate the pface mesh, the external data file had the format of three fields of 14 characters each, with a header line containing the number of vertices. In this case, I have supplied two header lines, containing the number of points in the M and N directions (the M and N sizes) respectively.

Table SG1-1 The 3D mesh data file

```
4
5
5.000000E+00  0.000000E+00  3.000000E+00
6.613774E+00  0.000000E+00  2.139993E+00
8.381966E+00  0.000000E+00  1.673762E+00
1.020998E+01  0.000000E+00  1.626250E+00
1.200000E+01  0.000000E+00  2.000000E+00
4.000000E+00  1.250000E+00  2.000000E+00
5.613774E+00  1.250000E+00  1.139993E+00
7.381966E+00  1.250000E+00  6.737621E-01
9.209976E+00  1.250000E+00  6.262503E-01
1.100000E+01  1.250000E+00  1.000000E+00
4.000000E+00  2.750000E+00  2.000000E+00
5.613774E+00  2.750000E+00  1.139993E+00
7.381966E+00  2.750000E+00  6.737621E-01
9.209976E+00  2.750000E+00  6.262503E-01
1.100000E+01  2.750000E+00  1.000000E+00
5.000000E+00  4.000000E+00  3.000000E+00
6.613774E+00  4.000000E+00  2.139993E+00
8.381966E+00  4.000000E+00  1.673762E+00
1.020998E+01  4.000000E+00  1.626250E+00
1.200000E+01  4.000000E+00  2.000000E+00
```

3. **Question:** Write a program that will create a 3dmesh using the prototype program structure of this month's lesson. Specify the required format of the external data file in your program, and include an error handler that will reset any system variables back to their initial values in the event of an AutoLISP error.

Answer: The program MESH3D.LSP is a solution:

```
;;;  MESH3D.LSP A program by Tony Hotchkiss
;;;  MESH3D CREATES A 3D MESH USING
;;;  VERTEX POINTS FROM A DATA FILE.
;;;  THE FORMAT OF THE DATA FILE IS:
;;;   m     (the total M size)
;;;   n     (the total N size)
;;;  x  y  z   (three fields of 14 characters)
;*************************************
(defun err (s)
   (if (= s "Function cancelled")
      (princ "\nMESH3D Function cancelled")
      (progn (princ "\nMESH3D Error: ") (princ s) (terpri))
   )
   (resetting)
   (princ "\nSYSVARS have been reset...")
   (princ)
)
(defun setv (systvar newval)
   (set 'x (read (strcat systvar "1")))
   (set x (getvar systvar))
   (setvar systvar newval)
)
(defun setting ()
```

```
    (setq oerr *error*)
    (setq *error* err)
    (setv "CMDECHO" 0)
    (setv "BLIPMODE" 0)
 ); end of setting
```

This opening sequence has a new defined function 'setv' for saving old system variables and setting new values. This demonstrates a use of the (set) function for creating a new variable which gets its name from the concatenation of the system variable and the number 1, which is supplied as a string character in **(set 'x (read (strcat systvar "1")))**

The (read) function returns its argument converted to the corresponding data type, which in this case is a string. However, the string returned does NOT have double quotes around it, so the value that is assigned to the quoted variable 'x appears as a symbol. For instance, if the systvar is "CMDECHO", then 'x is set to CMDECHO1.

The next statement **(set x (getvar systvar))** sets the value of the new symbol CMDECHO1 indirectly, by using (set) without a quoted argument. In other words, x contains the symbol CMDECHO1, so it is CMDECHO1 that takes the assigned value returned from **(getvar "CMDECHO")**. Here we are continuing with the example of systvar being substituted with "CMDECHO".

Finally, the new value of the system variable is assigned with, (setvar systvar newval), where systvar and newval are supplied as the arguments of the function (setv). The setting function sets system variables, whose old values are saved, and whose new values are set simply by a function call of the form **(setv "CMDECHO" 0)**. The old value of the system variable is stored in the new variable which created by adding the character "1" to the end of the system variable name. This naming convention allows us to reset the system variables back to their old values as we show later in the resetting function.

The input function starts by clearing and displaying the text window with (textpage). A series of prompts then describes the required external data file format, and asks the user to enter the file name, which is concatenated with the file extension .txt in the (calc) function.

```
 (defun input ()
    (textpage)
    (prompt "\nMESH3D.LSP uses a .TXT file with the following format: ")
    (prompt  "\nm  (the M size) ")
    (prompt  "\nn  (the N size) ")
    (prompt  "\nx  y  z  (fields of 14 characters including spaces) ")
    (prompt "\n\nEnter the file name: ")
    (setq fname (getstring) )
    (graphscr)
 ); end of input
 (defun calc ()
    (setq fname (strcat fname ".txt"))
 ); end of calc
```

Now we come to the output function which reads the data from the external file and constructs the 3D mesh. This section of the program is actually simpler than the FASTFACE.LSP program that is given in the lesson for drawing pface meshes, because after supplying the M and N sizes, the set of vertex points is supplied, and the mesh is automatically created with no need to specify any more vertex or face ordering. The variable 'numverts' is the product of the M and N sizes, and it controls the repeat loop for processing the vertex points. Note that the repeat loop is identical to that of the FASTFACE.LSP program.

```
 (defun output (/ f1 msize nsize numverts p px py pz pp)
    (setq f1 (open fname "r"))
    (if f1                              ;test if file is open
       (progn                       ;then read the first 2 lines of data
          (setq msize (read (read-line f1)))
          (setq nsize (read (read-line f1)))
          (setq numverts (*  msize  nsize))
          (command "3dmesh" msize nsize)
          (repeat numverts          ; read in the vertices
             (setq p (read-line f1)
                px (read (substr p 1 14))
                py (read (substr p 15 14))
                pz (read (substr p 29 14))
                pp (list px py pz)
             )
```

```
            (command pp)
            (princ)
         ); end of repeat for list of vertices
      ); progn
      (princ  (strcat "\nCannot open file " fname))
   ); if
   (close f1)
);  end of defun output
```

In the repeat loop, although the setq function returns the value of its last expression (in this case the vertex point pp), the point pp is dumped out to the command line by the (command pp) function, and not by the setq. The (princ) following the (command pp) inhibits the 'nil' that is returned by the command function.

As in the lesson example for FASTFACE.LSP, all of the action of the output section takes place inside an 'if' statement to test for the open file. If the file does not exist, the message "Cannot open file ...." is printed.

The next section of the program is the resetting function, which uses the new defined function (rsetv). The new function takes one argument, the name of the system variable to be reset. We use the set function with a quoted variable to create a symbol name as we did in the setting function, with the statement **(set 'x (read (strcat systvar "1")))**. In this case, the symbol already exists and has a value assigned to it, so we evaluate the symbol with **(setvar systvar (eval x))**. Note here that the symbol x contains the unevaluated symbol CMDECHO1 or BLIPMODE1 etc., so use the (eval) function to extract its value, which we already set to be the old value of the system variable.

```
(defun rsetv (systvar)
   (set 'x (read (strcat systvar "1")))
   (setvar systvar (eval x))
)
(defun resetting ()
   (rsetv "CMDECHO")
   (rsetv "BLIPMODE")
   (setq *error* oerr)
); end of resetting
(defun C:MESH3D ()
   (setting)
   (input)
   (calc)
   (output)
   (resetting)
   (princ)
); end of defun
```

Finally, the function C:MESH3D executes each function in turn to run the program.

Reference literature for AutoLISP

My favorite books for AutoLISP programming are:

1. "Maximizing AutoCAD: Inside AutoLISP Volume II" J. Smith & R. Gesner, New Riders Publishing, 1991 Tel: 1-800-541-6789

This book is a good general reference that covers just about all aspects of AutoLISP very thoroughly. It is useful for the expert as well as for the relative beginner, and comes with a disk containing some useful LISP programs as well as a structured approach to help you with many exercises. I learned a lot about AutoLISP from a previous version of this volume by the same authors, entitled "Customizing AutoCAD", and I use "Maximizing AutoCAD: Vol. II" mainly for reference.

2. "AutoLISP in Plain English, 4th Edition" George O. Head, Ventana Press, P.O. Box 2468, Chapel Hill, NC 27515, 1992 Tel: 919-942-0220 FAX: 1-800-877-7955

This is an excellent book for the novice to intermediate AutoLISP user, and is very easy to read and understand. It is subtitled 'A Practical Guide for Non-Programmers', and it will certainly get you started in writing your first AutoLISP programs. There are some useful program listings that will give you instant productivity gains in several areas. The example LISP listings are very well annotated.

Advanced AutoLISP Programming Correspondence Course

This book also has a floppy disk containing programs from the book and ready to run sample programs.

Other sources for AutoLISP are the magazines CADalyst and CADENCE.

Advanced AutoLISP Programming Correspondence Course

by Tony Hotchkiss, Assistant Professor, State University College at Buffalo, New york. He may be reached at HOTCHKAJ@SNYBUFAA.CS.SNYBUF.EDU. At the time this course was developed he was an Assistant Professor in the Department of Engineering Professional Development, University of Wisconsin-Madison.

Study Guide, part **2** of **8**

Lesson Two - System Variables and Interfacing with AutoCAD

Solutions to the homework questions:

1. Question: Write a function C:LISTDXF that will return a DXF list for any object you draw. The choices for the objects should include lines, arcs, circles, and polylines. Your program should default to the line, so the user prompt might read "Draw: Circle/Arc/Polyline/<Line>: ". The DXF list should be in the form of one group code per line, and the listing should be written to an external file called LISTDXF.TXT. Use the append mode so that all of the entity types can be listed in the same file.

Answer: The program LISTDXF.LSP starts off with the error handler, which is similar to the standard form that we suggested in lesson one of this series, except that the "Function cancelled" message is suppressed because we have deliberately set up an 'infinite loop' in the program, so that a "Cancel" is necessary to terminate it. I have also added a release 12 (alert) function that displays an alert dialogue box on the graphic screen. An alternative to this is a 'print' function to display the message in the command/prompt area.

```
;;;  LISTDXF.LSP   - by Tony Hotchkiss
;**********************************(defun err (s)
   (if (/= s "Function cancelled")
      (progn
         (princ "\nLISTDXF Error: ")
         (princ s)
      ); progn
   ); if
   (resetting)
   (alert "SYSVARS have been reset")
   (princ)
); err
(defun setting ()
   (setq oerr *error*)
   (setq *error* err)
   (setq ocmd (getvar "CMDECHO"))
   (setvar "CMDECHO" 0)
   (setq oblip (getvar "BLIPMODE"))
   (setvar "BLIPMODE" 0)
); setting
```

The program input section opens the text file "LISTDXF.TXT" in write mode, then writes a heading before closing the file. This is done to initialize the file each time the program is run, otherwise the later "append" modes would make the file grow each time the program is used.

```
(defun input ()
   (setq f1 (open "LISTDXF.TXT" "w"))
   (write-line "DXF LISTING FOR VARIOUS ENTITY TYPES" f1)
   (close f1)
); end of input
```

The input section is followed by the output section, which sets the "CMDECHO" system variable on, so that the subsequent prompts for the line, arc, circle, etc are displayed properly. The required prompt for the program is supplied by the (getkword) and (initget) combination, where the "Line" choice is offered as the default. The responses are controlled by a (cond) function as shown, which tests for the entity types "Circle", "Arc", and "Polyline", and takes the default 'none-of-the-above' by forcing a 't' and drawing the line. The simplest version of each of the entities is allowed by the number of 'pause' expressions in each of the (command) functions as shown.

Following the (cond), the program gets the dxf listing for t he last entity created, then opens the LISTDXF.TXT file, and prints each associated dxf pair on separate lines of the .txt file using a combination (foreach) and (print). The append mode is used here to place the dxf list under the heading made in the input function.

```
(defun output ()
   (setvar "CMDECHO" 1)
   (initget "Circle Arc Polyline Line")
   (setq type (getkword "\nDraw: Circle/Arc/Polyline/<Line>: "))
   (cond
      ((= type "Circle") (command "circle" pause pause))
      ((= type "Arc") (command "arc" pause pause pause))
      ((= type "Polyline") (command "pline" pause pause ""))
      (t  (command "line" pause pause ""))
   )
   (setq elist (entget (entlast)))
   (setq f1 (open "LISTDXF.TXT" "a"))
   (foreach item elist (print item f1))
   (close f1)
   (output)
); output
```

The text file is closed, and the output function then calls itself recursively with the (output) statement, so the effect is to maintain an 'infinite loop', which will remain in effect until the user cancels the program with a ^C (ctrl-C). An alternative to this is to place an (exit) or (quit) function in the (cond) function so that the user can quit instead of cancel, in which case, the (err) function would still allow the program to end quietly by resetting the system variables as before.

The program ends with the (resetting) function as shown next, followed by the C:LISTDXF function that allows the execution to take place as a new AutoCAD command:

```
(defun resetting ()
   (setvar "CMDECHO" ocmd)
   (setvar "BLIPMODE" oblip)
   (setq *error* oerr)
); resetting
(defun C:LISTDXF ()
   (setting)
   (input)
   (output)
   (resetting)
   (princ)
); C:LISTDXF
```

The program is shown executing in figure SG2-1, which shows the 'alert' box which appears after the 'cancel' is issued. In figure SG2-1, after the program has been loaded, the new command 'listdxf' is given, and the prompt for "Draw: Circle/Arc/Polyline/<Line> is displayed, and I started my input with the 'c' to get the circle choice.

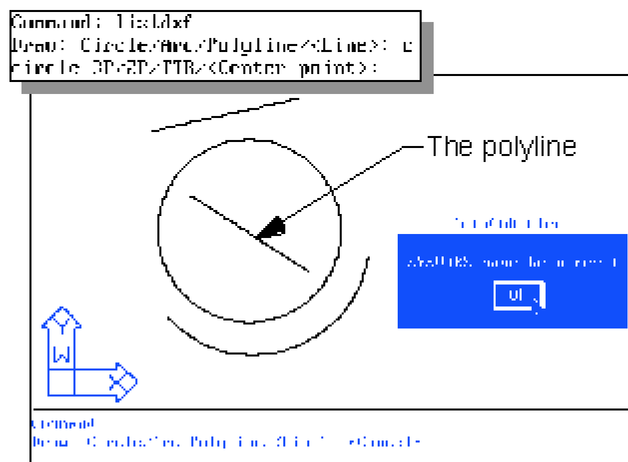

Figure SG2-1 Execution of LISTDXF.LSP

After drawing the circle, each of the other entities was created in turn, and the command prompts are shown in the text screen of figure SG2-2:

```
Circle 3P/2P/TTR/<Center point>: Diameter/<Radius> <1.0732>:
Command:
Draw: Circle/Arc/Polyline/<Line>: a
arc Center/<Start point>:
Center/End/<Second point>:
End point:
Command:
Draw: Circle/Arc/Polyline/<Line>:
line From point:
To point:
To point:
Command:
Draw: Circle/Arc/Polyline/<Line>: p
pline
From point:
Current line width is 0.0000
Arc/Close/Halfwidth/Length/Undo/Width/<Endpoint of line>:
Arc/Close/Halfwidth/Length/Undo/Width/<Endpoint of line>:
Command:
Draw: Circle/Arc/Polyline/<Line>: xcancelx

Command:
```

Figure SG2-2 The text screen for LISTDXF.LSP

The resulting text file LISTDXF.TXT is created as follows, in which the dxf listing is displayed with one dotted pair per line as required. The meanings of the group codes are given briefly in italics that I added to the file. You can refer to the appropriate AutoCAD user manuals for complete codes for all of the entity types (AutoCAD Customization Manual, chapter 11 for release 12, and the AutoCAD Reference Manual appendices for earlier releases).

```
LISTDXF.TXT - the text file
DXF LISTING FOR VARIOUS ENTITY TYPES
entity dxf code associated pair     description
(-1 .  <Entity name: 600003fc>)    entity name
(0 .  "CIRCLE")                    entity type
(8 .  "1")                         layer
(10 4.07884 3.40217 0.0)           circle center point
(40 .  1.72198)                    radius
(210 0.0 0.0 1.0)                  orientation vector
(-1 .  <Entity name: 6000041e>)    entity name
(0 .  "ARC")                               entity type
(8 .  "1")                         layer
(10 4.15051 3.24538 0.0)           arc center point
(40 .  2.17114)                    radius
(50 .  3.88764)                    start angle
(51 .  6.13457)                    end angle
(210 0.0 0.0 1.0)                  orientation vector
(-1 .  <Entity name: 60000440>)    entity name
(0 .  "LINE")                      entity type
(8 .  "1")                         layer
(10 2.25152 5.27174 0.0)           start point
(11 4.9925 5.88043 0.0)            end point
(210 0.0 0.0 1.0)                  orientation vector
(-1 .  <Entity name: 60000462>)    entity name
(0 .  "POLYLINE")                  entity type
(8 .  "1")                         layer
(66 .  1)                          vertices follow flag
(10 0.0 0.0 0.0)                   polyline elevation
(70 .  0)                          polyline optional flag
(40 .  0.0)                        starting width
(41 .  0.0)                        default ending width
(210 0.0 0.0 1.0)                  orientation vector
(71 .  0)                          polygon mesh M count
(72 .  0)                          polygon mesh N count
(73 .  0)                          M-surface density
(74 .  0)                          N-surface density
(75 .  0)                          curve/surface type
```

Note that the dxf group codes for the polyline do not show any "vertices", only the group code 66 that flags the fact that there are vertices to follow this initial section of the listing. A later lesson (five) will cover complex entities and how to display the complete dxf list for each of the individual entities that are included in the complex entity. Group code 210, the orientation vector, is also called the 'extrusion' vector.

2. Question: Write a function that can test whether a particular layer name exists, and if the layer does exist, make it the current layer. If the layer does not exist, create the layer and make it the current layer.

Answer: This is a typical use for tblsearch, as shown in the following defined function (chklyr). The function can be called with **(chklyr *string*)**, where the string argument is the layer-name.

```
  (defun chklyr (curlyr / ans)
     (setq ans (tblsearch "LAYER" curlyr))
          (if (= ans nil);    Check to see if the layer exists,
                (command "LAYER" "M" curlyr ""); if not, create and make it
current.
                (command "LAYER" "S" curlyr ""); if it exists, make it current
          ); if
     (princ)
  ); chklyr
```

The test is whether 'ans' is equal to nil, because (tblsearch) returns nil if the layer name is not found. The (princ) is not necessary if this function is called from another section of a program that uses a prototype form that ends with its own (princ) for clean endings.

Advanced AutoLISP Programming Correspondence Course

by Tony Hotchkiss, Assistant Professor, State University College at Buffalo, New york. He may be reached at HOTCHKAJ@SNYBUFAA.CS.SNYBUF.EDU. At the time this course was developed he was an Assistant Professor in the Department of Engineering Professional Development, University of Wisconsin-Madison.

Study Guide, part **3** of **8**

Lesson Three - Entity Selection for AutoLISP

Solutions to the homework questions:

1. Question: When using filter lists, as in other cases in AutoLISP, it is permissible to use variable names provided that the data type represented by the variable is appropriate for its intended use. If you wanted to substitute symbols (variables) in filter lists, would it be correct to use the quoted form '((filter-list..., or the list function form (list (cons...? Give reasons for your choice.

Answer The specific examples of filter lists given in the lesson are all of one type or the other (quoted or using list/cons). For the example of filtering all red circles on layer "1", each of the formats is given thus:

```
(setq ss3 (ssget "X" '((0 .  "CIRCLE") (8 .  "1") (62 .  1)) ))
or alternatively:
(setq ss3 (ssget "X" (list  (cons  0  "CIRCLE") (cons  8  "1") (cons  62  1))
))
```

If the layer name is given as a variable 'lyrname', by having previously assigned the variable with **(setq lyrname "1")**, the first (quoted) format will look like:

(setq ss3 (ssget "X" '((0 . "CIRCLE")(8 . lyrname)(62 . 1))))

This is not appropriate because the quote means that the following items in the list are not evaluated, and the items in the list will be taken literally. The symbol 'lyrname' will be subject to 'type' checking when the filter list is used, and the value associated with the dxf group code 8 must be of type 'string'. An error of the form 'bad argument type' will result if you attempt to use this format.

The alternative format will look like:

(setq ss3 (ssget "X" (list (cons 0 "CIRCLE")(cons 8 lyrname)(cons 62 1))))

In this case, because the expression (cons 8 lyrname) gets evaluated (it is not quoted), lyrname will be evaluated as "1", which is the appropriate type 'string'. Incidentally, if you use numbers for your layers, they are actually interpreted as strings because layer names must be of type 'string', as we see in this example, where "1" is a string and not an integer or a real number.

The entire filter list could be supplied as a symbol, then your programs would be able to select from a number of preset filter lists. You could write something like:

```
(setq lyrname "1")
(setq f-list1 (list (cons 0 "CIRCLE")(cons 8 lyrname)(cons 62 1)) )
(setq ss3 (ssget "X" f-list1))
```

It is actually possible to 'mix' the formats to have both (quote..) and (cons..), as in:

```
(setq f-list2 (list '(0 .  "CIRCLE") (cons 8 lyrname) '(62 .  1)) )
(setq ss3 (ssget "C" pt1 pt2 f-list2))
```

which gives a more concise format. Note here that the entire filter list begins with the (list..) function, and each individual dotted pair is quoted, but the (cons..) expression is not quoted because it is supposed to be evaluated when the filter list is formed. Filter lists can be used with any (ssget..) format, as we show here by changing the "X" with the "C" definition with the addition of the crossing points pt1 and pt2, that should be defined earlier in the program.

To show how the filter lists are evaluated, create a layer "1", and draw some circles that are colored red (not red 'bylayer'!), then enter the following at the AutoCAD command prompt and you will see:

```
Command:  (setq lyrname "1")
"1"
```

```
Command:  (setq flst (list '(0 .  "CIRCLE")'(62 .  1)(cons 8 lyrname)))
((0 .  "CIRCLE") (62 .  1) (8 .  "1"))
Command:  (setq ss1 (ssget "X" flst))
<Selection set:  1>
Command:
```

The variable 'ss1' now contains the selection set 1 <Selection set: 1> as shown, and the functions (sslength) and (ssname) can be used with ss1, so if 3 circles met the filter list criteria, then (sslength ss1) returns 3, and (ssname ss1 0) returns the name of the first of the circles in the selected set in the form <Entity name: 60000066>.

You could take the creation of filter lists a stage further by writing a program that creates filter lists based on input supplied by the user, as in the following defined function 'filtlst', that returns a filter list for entity type, layer and/or color in any combination. The usage of the function is

(setq flist (filtlst)) (setq ss1 (ssget "X" flist))....

```
;DEFINED FUNCTION 'filtlst'- returns a filter-list
; function by Tony Hotchkiss
(defun filtlst (/ flist fltype flval)
   (setq flist nil)
   (setq more t)
   (while more
      (initget "Entity-type Layer-name Color")
      (setq fltype (getkword
               "\nItem type Entity-type/Layer-name/Color/<exit>: "))
      (if (/= fltype nil)
         (progn
            (setq flval (getstring "\nItem value: "))
            (setq item
               (cond
                  ((= fltype "Entity-type") (cons 0 flval))
                  ((= fltype "Layer-name") (cons 8 flval))
                  ((= fltype "Color") (cons 62 (atoi flval)))
                  (t nil)
               ); cond
            ); setq
            (setq flist (append flist (list item)))
            (setq more t)
         ); progn
         (setq more nil)
      ); if
   ); while
   flist
); filtlst
```

FILTLST starts by setting the list flist to 'nil', and the looping test symbol 'more' to 't'. The function works inside a while loop which depends on the value of the test 'more'. The choices for items to include in the filter list are given by the combination of (initget) and (getkword). You can add to these if you like. The 'default' choice of Item-type is <exit>, and if a <return> is given, (getkword) returns nil. An (if) expression tests that the user did not enter a <return>.

The 'then' (progn) creates the filter list by setting the variable 'item' to the appropriate dotted pair with the (cond) function that tests the Item-type as shown. The item-value is obtained with (getstring), so the variable 'flval' is of type 'string'. If necessary, flval is converted to another type as shown in the "color" choice, where an integer is required, and flval is converted to an integer with (atoi flval). The appropriate dxf code is supplied in the (cons) expressions of the (cond) statement. Finally, 'item' is appended as a list to the 'flist' with (setq flist (append flist (list item))). **flist** is the last statement of the function, and that forces the function to return the list flist.

A typical sequence of operations at the command-prompt area is:

```
Command:  (setq fl1 (filtlst))
Item type Entity-type/Layer-name/Color/<exit>:  e
Item value:  arc
Item type Entity-type/Layer-name/Color/<exit>:  c
Item value:  2
Item type Entity-type/Layer-name/Color/<exit>:  L
Item value:  1
```

```
Item type Entity-type/Layer-name/Color/<exit>:
((0 .  "arc") (62 .  2) (8 .  "1"))
Command:  (setq ss1 (ssget "X" fl1))
<Selection set: 3>
```

Any of the items can be selected to be included in the filter list

2. Question: Write a program that will erase all of the yellow objects from any layer, where the layer name is supplied by the user in response to a getstring function.

Answer The program YRASE.LSP is a simple solution, based on the above discussions.

```
(defun yrase ()
      (setq lyrname (getstring "\nLayer name: "))
      (setq flist (list (cons 62 2) (cons 8 lyrname)))
      (setq ss1 (ssget "X" flist))
      (command "erase" ss1 "") (princ)
); yrase
```

Variations on this program are:

```
(defun yrase ()
      (setq lyrname (getstring "\nLayer name: "))
      (setq flist (list '(62 .  2) (cons 8 lyrname)))
      (setq ss1 (ssget "X" flist))
      (command "erase" ss1 "") (princ)
); yrase
```

or

```
(defun yrase ()
     (setq lyrname (getstring "\nLayer name: "))
     (setq ss1 (ssget "X" (list (cons 62 2) (cons 8 lyrname))))
     (command "erase" ss1 "") (princ)
); yrase
```

or

```
(defun yrase ()
       (setq lyrname (getstring "\nLayer name: "))
       (setq ss1 (ssget "X" (list '(62 .  2) (cons 8 lyrname))))
       (command "erase" ss1 "") (princ)
); yrase
```

Advanced AutoLISP Programming Correspondence Course

by Tony Hotchkiss, Assistant Professor, State University College at Buffalo, New york. He may be reached at HOTCHKAJ@SNYBUFAA.CS.SNYBUF.EDU. At the time this course was developed he was an Assistant Professor in the Department of Engineering Professional Development, University of Wisconsin-Madison.

Study Guide, part **4** of **8**

Lesson Four - AutoLISP Data Base Manipulations

Solutions to the homework questions:

1. Question: The function (arccenrad) takes an entity name and an intersection point as its arguments, as in the call to the function in (xint) of the XHATCH.LSP program, (arccenrad ename xpt). Write the function, with a defun expression (defun arccenrad (en parc).....) where en is the name of the arc, and parc is a point near to one end of the arc. The function should create global variables for the center point of the arc 'cen', the radius 'rad', and the angle 'ang' between the center point and the point on the arc, parc.

Answer: A solution is the following 'arccenrad' function, which takes the arc 'en' and the point 'parc' as input.

```
(defun arccenrad (en parc / angs ange ang1 ang2)
   (setq cen (dxf 10 en))
   (setq rad (dxf 40 en))
   (setq ang (angle cen parc))
   (setq en1 en)
); arccenrad
```

The function creates the required variables as shown, using the (dxf) defined function from the lesson. The DXF group codes 10 and 40 are associated with the center point and the radius respectively. The angle between the arc center and the initial point on the arc 'parc' is returned by the (angle) function.

In addition to the above, I have also created a new global variable 'en1', by assigning the current entity name 'en', so that I can refer to the entity in the proposed (arcmov) function. This follows the same procedure as (lineang) from the lesson.

2. Question: The function (arcmov) works like linmov, except that the search direction is along an arc instead of in a straight line. For this, we need a delta angle instead of a delta distance, where the delta angle results in the actual distance moved along the arc to be similar to the distance delta. What is the relationship between the delta angle 'deltang', 'delta', and the arc radius 'rad'? Write an expression (setq deltang ( )) that sets the value of deltang in terms of delta and rad.

Answer: With reference to figure SG4-1, the arc length, s, is equal to the radius multiplied by the angle in radians. (Remember that the radius R and the circumference S of a circle are related by S/R = 2[[pi]]).

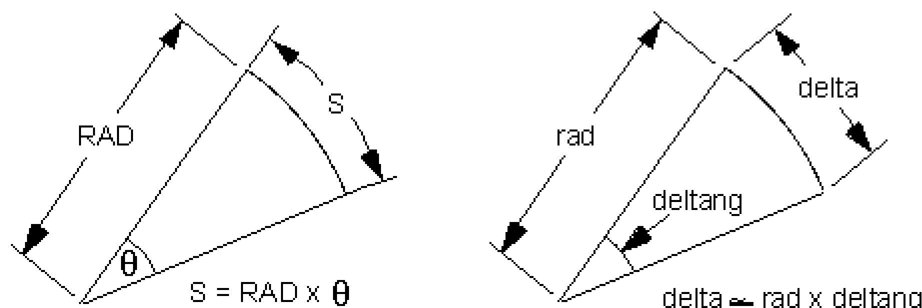

Figure SG4-1, arc length and delta for small angles

If the incremental distance to be stepped out along the arc is equal to 'delta', the equivalent angle 'deltang' is given by

delta = rad x deltang, or **deltang = delta/rad**

from which the appropriate AutoLISP statement is

(setq deltang (/ delta rad))

3. Question: Using the while loop of the function (linmov) as a model, write the function (arcmov) that has a single argument for the start point (that is the first intersection point on the arc). The defun will look like (defun arcmov (parc2)......), and the function should return a point on the next boundary entity that adjoins the arc. Assume that you have the delta angle 'deltang', the arc center point 'cen', the arc radius 'rad', and the start angle 'ang' between the center point and the start point of the arc.

Answer: The defined function (linmov) is repeated here for convenience:

```
(defun linmov (p4 a / p2 ss)
   (setq delta (* (/ (float (getvar "APERTURE"))
                     (car (getvar "SCREENSIZE")))
                  (- (car (getvar "LIMMAX"))
                     (car (getvar "LIMMIN"))))
   ); delta
   (while (= ss nil)
      (setq p2 (polar p4 a delta))
      (setq ss (ssget "C" p4 p2))
      (setq p4 p2)
   ); while
   (osnap p2 "NEA")
); linmov
```

In (linmov), the increment 'delta' is defined as a global variable, and is related to the size of the aperture, but converted to AutoCAD units instead of pixels by using its ratio to the screen size and the drawing limits. In the while loop, a crossing box is defined by the two points p4 and p2, where p4 is supplied as an argument to (linmov), and is a point near to one end of the current line. The angular direction of the search for the next object has already been defined by (lineang). These parameters are shown in figure SG4-2.

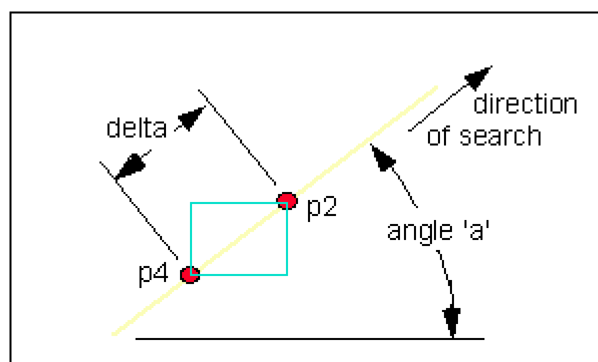

Figure SG4-2 Parameters of the (linmov) function

For the equivalent (arcmov), the direction of search is to be established, as this was not done in (arccenrad). Note that before either (linmov) or (arcmov) are called, the current entity is deleted (see the (xint) function which is the calling function for linmov and arcmov).

The defined function (arcmov) begins by creating deltang according to the statement of question 2 above. The direction of motion of (arcmov) is toward the farthest end of the arc from the start angle 'parc2', so the start and end angles of the arc (the current entity) must be found. To do this, we first undelete the entity with (entdel en1), and find the start and end angles using the group codes 50 and 51 respectively in the (dxf) function.

```
(defun arcmov (parc2 / ss p2)
   (setq deltang (/ delta rad))
   (entdel en1)
   (setq angs (dxf 50 en1))
   (setq ange (dxf 51 en1))
   (setq ang1 (abs (- angs ang)))
   (setq ang2 (abs (- ange ang)))
   (entdel en1)
   (if (> ang1 ang2)
         (setq deltang (- deltang))
   )
   (setq pmid (polar cen (+ ang deltang) rad))
```

```
    (while (= ss nil)
       (setq p2 (polar cen (+ ang deltang) rad))
       (setq ss (ssget "C" parc2 p2))
       (setq ang (+ ang deltang))
       (setq parc2 p2)
    ); while
    (osnap p2 "NEA")
); arcmov
```

The difference between the start and end angles and the angle returned by (arccenrad) are compared and assigned to ang1 and ang2 as shown. These parameters are shown in figure SG4-3.

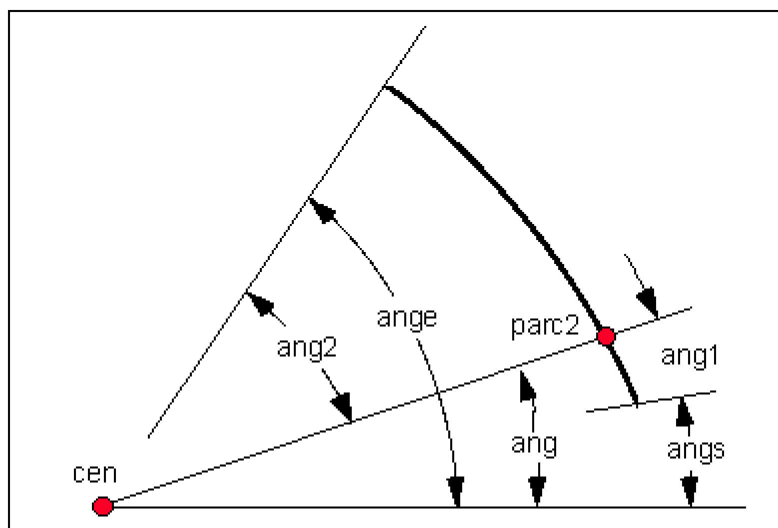

Figure SG4-3 The angles of (arcmov)

Next, after deleting the arc with (entdel en1), we use the (if) statement to determine the sign (positive or negative) of the deltang. This is very similar to the procedure used in (lineang) to find the line endpoint furthest away from the initial point. Again, the assumption is made that the start and end of the entity are nearest and furthest respectively from the initial point. If that assumption is correct, then the direction of travel is assumed correctly, otherwise the sign of deltang is reversed.

XHATCH.LSP creates new entities around the boundary with the defined function (newent), which we will see later, and in order to construct the arc segments of the boundary, we will use the '3-point' method of creating arcs. For this reason, we create an intermediate point 'pmid' as shown with:

(setq pmid (polar cen (+ ang deltang) rad))

Finally, arcmov moves the cursor position, searching along the arc in thw while loop, using parc2 and p2 as the corner points of the crossing box used in (ssget). The movement around the arc is caused by re-assigning parc2 to p2, and then continuing the while loop until an entity is found in the crossing box. Figure SG4-4 shows a typical set of parameters during the search along the arc. Note that the arc (current entity) is actually deleted during the search, so that the criterion for the selection set 'ss' returned by (ssget) can be used as the while loop test (= ss nil).

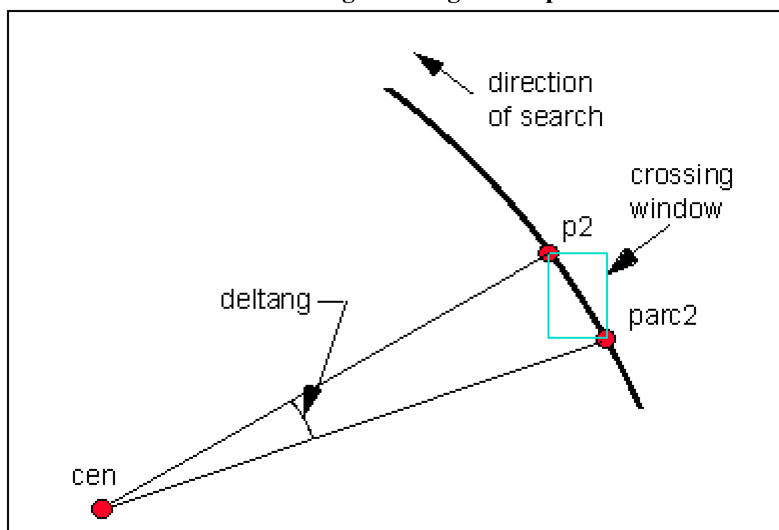

Figure SG4-4 Search parameters for (arcmov)

In the lesson, I indicated that the defined function (newent) creates new boundary entities, either lines or arcs, which are added to the entity list 'elst' with (ssadd) in the while loop of the (calc) function. The (newent) function takes the intersection points int1 and int2, each end of the new entity to be created. If the 'old entity' (the current entity) is a line, then the new entity is created by

(command "line" p1 p2 "")

and if the old entity is an arc, then the new entity is created as

(command "arc" p1 pmid p2)

where p1 and p2 are the arguments of (newent), which are set to the intersection points int1 and int2 by the calling function (calc). The defined function (newent) returns the name of the new entity, and is simply:

```
(defun newent (p1 p2)
   (command "layer" "s" "noplot" "")
   (cond
      ((= (dxf 0 en1) "LINE") (command "line" p1 p2 ""))
      ((= (dxf 0 en1) "ARC") (command "arc" p1 pmid p2))
      (t nil)
   )
   (ssname (ssget "L") 0)
); newent
```

I changed the layer to the 'noplot' layer before creating the new entities. It would more sophisticated to check that this layer exists, then create it if necessary. the (tblsearch) function could be used for that purpose as in the following:

```
(if (=  (tblsearch "layer" "noplot") nil)
    (command "layer" "M" "noplot" "")
    (command "layer" "S"  "noplot" "")
); if
```

The complete XHATCH.LSP program, with the automatic layer checking is:

```
;;; ******************************************
;;;  XHATCH.LSP   - a lisp program by Tony Hotchkiss
;;;   Copyright (C) 1993 by Tony Hotchkiss
;;;*******************************************
;;;   DESCRIPTION
;;;
;;;   XHATCH.LSP Cross-hatches a boundary of lines and
;;;   arcs which are not necessarily end to end.  The direction
;;;   of travel around the boundary.  (either clockwise or ccw)
;;;   is in the direction of the endpoint furthest from any
;;;   intersection of boundary objects.
;;;
;;;   To use the program, pick an area enclosed by line and arc
```

```
;;;   entities.  The program then automatically creates a closed
;;;   polyline on a layer called "NOPLOT", and hatches it.
;;;
;;; ********************************************
(defun err (s)
   (if (= s "Function cancelled")
      (princ "\nYou cancelled the function.")
      (progn (princ "\nXHATCH: ") (princ s) (terpri))
   )
   (resetting)
   (princ "SYSVARS have been reset\n")
   (princ)
); err
(defun setting ()

   (setq oerr *error*)
   (setq *error* err)
   (setq ocmd (getvar "CMDECHO"))
   (setvar "CMDECHO" 0)
   (setq oblip (getvar "BLIPMODE"))
   (setvar "BLIPMODE" 0)
   (setq olyr (getvar "CLAYER"))
); end of setting
(defun calc (/ pt p int1 int2 epsilon enold)
   (setq pt (getpoint "\nPick area to be hatched: "))
   (setq p (linmov pt pi))
   (setq int1 (xint p))
   (setq p0 int1)
   (setq elst (ssadd)
         int2 pt
         epsilon 0.001
   ); setq
   (while (> (distance int2 p0) epsilon)
      (setq enold en1)
      (entdel enold)
      (setq int2 (xint int1))
      (entdel enold)
      (setq ent (newent int1 int2))
      (setq elst (ssadd ent elst))
      (entdel ent)
      (setq int1 int2)
   ); while
); calc

(defun dxf (code entname)
   (cdr (assoc code (entget entname)))
); dxf
(defun xint (xpt / etype ename)
   (setq ename (ssname (ssget xpt) 0))
   (setq etype (dxf 0 ename))
   (cond
      ((= etype "LINE")
         (progn (lineang ename xpt)
            (entdel ename) (setq xpt (linmov xpt ang))
      )); lines
      ((= etype "ARC")
         (progn (arccenrad ename xpt)
            (entdel ename) (setq xpt (arcmov xpt))
      )); arcs
      (t nil)
   ); cond
   (entdel ename)
   (osnap xpt "INT")
); xint
```

```
(defun lineang (en p3 / p1 p2 ang1 ang2 dis1 dis2)
   (setq p1 (dxf 10 en))
   (setq p2 (dxf 11 en))
   (setq dis1 (distance p3 p1))
   (setq dis2 (distance p3 p2))
   (if (> dis2 dis1) (setq ang (angle p1 p2))
                     (setq ang (angle p2 p1))
   ); if
   (setq en1 en)
); lineang
(defun linmov (p4 a / p2 ss)
   (setq delta (* (/ (float (getvar "APERTURE"))
                        (car (getvar "SCREENSIZE")))
                     (- (car (getvar "LIMMAX"))
                        (car (getvar "LIMMIN"))))
   ); delta
   (while (= ss nil)
      (setq p2 (polar p4 a delta))
      (setq ss (ssget "C" p4 p2))
      (setq p4 p2)
   ); while
   (osnap p2 "NEA")
); linmov
(defun arccenrad (en parc / angs ange ang1 ang2)
   (setq cen (dxf 10 en))
   (setq rad (dxf 40 en))
   (setq ang (angle cen parc))
   (setq en1 en)
); arccenrad
(defun arcmov (parc2 / ss p2)
   (setq deltang (/ delta rad))
   (entdel en1)
   (setq angs (dxf 50 en1))
   (setq ange (dxf 51 en1))
   (setq ang1 (abs (- angs ang)))
   (setq ang2 (abs (- ange ang)))
   (entdel en1)
   (if (> ang1 ang2)
         (setq deltang (- deltang))
   )
   (setq pmid (polar cen (+ ang deltang) rad))
   (while (= ss nil)
      (setq p2 (polar cen (+ ang deltang) rad))
      (setq ss (ssget "C" parc2 p2))
      (setq ang (+ ang deltang))
      (setq parc2 p2)
   ); while
   (osnap p2 "NEA")
); arcmov
(defun newent (p1 p2)
(if (=  (tblsearch "layer" "noplot") nil)
    (command "layer" "M" "noplot" "")
    (command "layer" "S"  "noplot" "")
); if
   (cond
      ((= (dxf 0 en1) "LINE") (command "line" p1 p2 ""))
      ((= (dxf 0 en1) "ARC") (command "arc" p1 pmid p2))
      (t nil)
   )
   (ssname (ssget "L") 0)
); newent
(defun output ()
   (setq n (sslength elst))
   (setq i (- 1))
```

```
   (repeat n (entdel
      (ssname elst (setq i (1+ i)))))
   (command "PEDIT"
      (ssname elst 0) "Y" "J" elst "" "X")
(if (=  (tblsearch "layer" "hatch") nil)
    (command "layer" "M" "hatch" "")
    (command "layer" "S"  "hatch" "")
); if
   (command "hatch" "ANSI31" "" "" "L" "")
   (princ)
); output
(defun resetting ()
   (setvar "CMDECHO" ocmd)
   (setvar "BLIPMODE" oblip)
   (setvar "CLAYER" olyr)
   (setq *error* oerr)
); end of resetting
(defun C:XHATCH ()
   (setting)
   (calc)
   (output)
   (resetting)
   (princ)
); end of defun C:XHATCH
```

Advanced AutoLISP Programming Correspondence Course

by Tony Hotchkiss, Assistant Professor, State University College at Buffalo, New york. He may be reached at HOTCHKAJ@SNYBUFAA.CS.SNYBUF.EDU. At the time this course was developed he was an Assistant Professor in the Department of Engineering Professional Development, University of Wisconsin-Madison.

Study Guide, part **5** of **8**

Lesson Five - AutoCAD Complex Entities

Solutions to the homework questions:

1. Question: Write a program that makes blocks with (entmake), with similar prompts to the regular AutoCAD BLOCK command, and allowing any form of entity selection.

Answer: In the lesson (five), the program MKBLKS.LSP was shown as:

```
MKBLKS.LSP   A program to make blocks with entmake.
(defun mkblks (/ blkname inspt yn esel)
   (setq blkname (getstring "\nBlock name: "))
   (setq inspt (getpoint "\nIndicate insertion point: "))
   (entmake (list (cons 0 "BLOCK") (cons 2 blkname)
        (cons 70 64) (cons 10 inspt)))
   (prompt "\nFirst entity...  ")
   (setq yn "Yes")
   (while (= yn "Yes")
      (entmake (cdr (entget (car (setq esel (entsel))))))
      (redraw (car esel) 3)
      (initget 1 "Yes No")
      (setq yn (getkword "\nAdd another entity?  (Yes or No) "))
   ); while
   (entmake '( (0 .  "ENDBLK")))
 (princ)
); mkblks
```

MKBLKS.LSP only allows objects to be selected one at a time, and to fix this, the code following the block header must be changed, so we could write:

```
MKBLKS2.LSP   A program to make blocks with entmake.
(defun mkblks (/ blkname inspt yn esel)
   (setq blkname (getstring "\nBlock name: "))
   (setq inspt (getpoint "\nInsertion base point: "))
   (entmake (list (cons 0 "BLOCK") (cons 2 blkname)
        (cons 70 64) (cons 10 inspt)))
;************************************
  NEW CODE TO BE INSERTED HERE
;************************************
   (entmake '( (0 .  "ENDBLK")))
 (princ)
); mkblks
```

We need a prompt to select entities, and this can be supplied by the (ssget) function as in

(setq ss1 (ssget))

If you only want to select simple entities (not complex entities such as inserts with attributes or polylines), each entity in the selected set can be added to the block with entmake in a repeat loop where the number of repetitions is given by (sslength), as in

```
(repeat (sslength ss1)
   (entmake (cdr (entget (ssname ss1 n))))
   (setq n (1+  n))
); repeat
```

You would need to initialize the value of 'n' to zero before starting the repeat loop, so for simple entities the new program is:

```
MKBLKS2.LSP   A program to make blocks with entmake.
(defun mkblks (/ blkname inspt ss1 n)
   (setq blkname (getstring "\nBlock name: "))
   (setq inspt (getpoint "\nInsertion base point: "))
   (entmake (list (cons 0 "BLOCK") (cons 2 blkname)
         (cons 70 64) (cons 10 inspt)))
   (setq ss1 (ssget)  n  0)
   (repeat (sslength ss1)
      (entmake (cdr (entget (ssname ss1 n))))
      (setq n (1+  n))
   ); repeat
   (entmake '( (0 .  "ENDBLK")))
 (princ)
); mkblks
```

Note that MKBLKS2.LSP allows you to select INSERT entities, even nested inserts, as they are identified by a single DXF list returned by (entget). The DXF list returned by complex entities such as polylines are not in a suitable format for a single (entmake) statement, because they need a header to include the entity type 'POLYLINE' and an ending statement containing 'SEQEND', so you would need to include a test in your programs to deal with these entities. For instance, in the repeat loop, you could have the (if) statement to test for polylines:

```
   (repeat (sslength ss1)
      (setq elist (entget (ssname ss1 n)))
      (if (= (cdr (assoc 0 elist)) "POLYLINE")
         (make_pline elist)
         (entmake (cdr elist))
      ); if
      (setq n (1+  n))
   ); repeat
```

Here, if the selected entity is a polyline, we call a defined function (make_pline) which takes 'elist' as input, and returns the entmake statements for the selected polyline.

A complete program, MKBLKS3.LSP including the (make_pline) function allows you to select polylines:

```
;MKBLKS3.LSP   A program to make blocks with entmake.
(defun mkblks (/ blkname inspt ss1 n elist)
   (setq blkname (getstring "\nBlock name: "))
   (setq inspt (getpoint "\nInsertion base point: "))
   (entmake (list (cons 0 "BLOCK") (cons 2 blkname)
         (cons 70 64) (cons 10 inspt)))
   (setq ss1 (ssget)  n  0)
   (repeat (sslength ss1)
      (setq elist (entget (ssname ss1 n)))
      (if (= (cdr (assoc 0 elist)) "POLYLINE")
         (make_pline elist)
         (entmake (cdr elist))
      ); if
      (setq n (1+  n))
   ); repeat
   (entmake '( (0 .  "ENDBLK")))
 (princ)
); mkblks
(defun make_pline (elist / ename)
   (setq ename (cdr (assoc -1 elist)))
   (entmake (cdr (entget ename)))
   (while (/= (cdr (assoc 0 (entget ename))) "SEQEND")
      (setq ename (entnext ename))
      (entmake (cdr (entget ename)))
   ); while
); make_pline
(defun mkblks3 ()
   (mkblks)
); mkblks3
```

The defined function (make_pline) takes the 'elist' that is passed as an argument from the (mkblks) function, and the entity name (assigned to 'ename') is extracted from the 'elist', which must be the dxf list of a polyline, otherwise (make_pline) would not have been called. The first 'header' line of the polyline is established with **(entmake (cdr (entget ename)))**, and a while loop is used to obtain all of the other (entmake) statements which take each sub-entity in turn of the polyline by 'stepping' through successive entities using (entnext) as shown.

In order to make the appropriate function name appear at the command/prompt area on loading, a final function (mkblks3) is defined, and this simply calls (mkblks).

2. Question: Write a program that will count the number of instances of any given block reference, in situations where there may be several different block references, each having an arbitrary number of instances (the blocks are inserted many times in your drawing).

Answer: The program BCOUNT.LSP asks the user to indicate a block, and prints the number of blocks with the same name as the one selected.

```
(defun bcount ()
   (prompt "\nSelect block reference")
   (setq bname (cdr (assoc 2 (entget (car (entsel))))))
   (setq ss1 (ssget "X" '((0 .  "INSERT")) ))
   (setq numblk 0 i 0)
   (repeat (sslength ss1)
      (setq ename (ssname ss1 i))
      (setq elist (entget ename))
      (if (= (cdr (assoc 2 elist)) bname)
         (setq numblk (1+ numblk))
      )
   (setq i (1+ i))
   ); repeat
   (princ (strcat "\nThere are " (itoa numblk)
         " references of block " bname))
   (princ)
); bcount
```

The program starts by asking the user to select a block reference, then assigns the name of the selected block to 'bname', using (entsel), (entget), and (assoc) as follows:

```
   (prompt "\nSelect block reference")
   (setq bname (cdr (assoc 2 (entget (car (entsel))))))
```

Next, the program selects all objects in the drawing that are of type "INSERT", using (ssget) with a filter list:

(setq ss1 (ssget "X" '((0 . "INSERT"))))

After initializing the variables 'numblk' and 'i' to zero, a repeat loop does the job of counting the blocks having the same name as the selected block. The repeat loop uses (sslength) to determine the number of repetitions:

```
   (setq numblk 0 i 0)
   (repeat (sslength ss1)
```

In the repeat loop, the name of the 'ith' object in the selection set is assigned to 'ename', and the DXF list of that object is returned by (entget) and assigned to 'elist'. The first time the loop is entered, 'i' has a value of zero, so the entity 'ename' is the first entity to be examined. Now the name of the block is tested with an (if) statement to check that it is equal to the selected block name 'bname'. If the name matches, a counter 'numblk' is incremented by one, and the if statement terminates. The last action of the repeat loop is to add one to the index 'i', so that the next object can be tested, and so on until all of the items of the selected set are tested.

```
      (setq ename (ssname ss1 i))
      (setq elist (entget ename))
      (if (= (cdr (assoc 2 elist)) bname)
         (setq numblk (1+ numblk))
      )
   (setq i (1+ i))
   ); repeat
```

The program ends by printing out the number of blocks matching the name of the selected block:

```
   (princ (strcat "\nThere are " (itoa numblk)
         " references of block " bname))
```

```
    (princ)
); bcount
```

To write this type of program, you need to know the DXF listing format for the 'insert' entity so that the block name can be tested by its association with the appropriate group code (2 in this case).

Advanced AutoLISP Programming Correspondence Course

by Tony Hotchkiss, Assistant Professor, State University College at Buffalo, New york. He may be reached at HOTCHKAJ@SNYBUFAA.CS.SNYBUF.EDU. At the time this course was developed he was an Assistant Professor in the Department of Engineering Professional Development, University of Wisconsin-Madison.

Study Guide, part **6** of **8**

Lesson Six - AutoLISP Anonymous Blocks

Solutions to the homework questions:

1. Question: Why can't we scan the drawing for inserted anonymous blocks instead of examining the block table when looking for the name of the last created anonymous block?

Answer: The short answer is that it is possible to scan the drawing for INSERT entities, but it would be cumbersome to establish the last name by using something like (ssget "X"...) with a filter list, followed by (sslength), because there may be multiple occurrences of any of the blocks, or some of the *U type names may be missing from the sequence of assigned names.

Filter lists for unnamed blocks can make use of the 'wild-card' asterisk as in *U*, but you need to make sure that the first asterisk is supplied as a literal character. To do that requires the use of the "escape" character `, found on the same key as the "tilde" character ~. The filter list would be:

(list (cons 2 "`*U*"))

The first asterisk is now taken as part of the string, and the second asterisk is a wild card that represents all following characters. The (ssget) statement is

(setq ss1 (ssget "X" (list (cons 2 "`*U*))))

These statements are useful for filtering all unnamed blocks, without regard for multiple instances of the blocks. In the table "block", there are no multiple occurrences, so a straightforward list of names can be established from which you can access any of the currently assigned names.

You can see the difference easily by using the BLOCK command with the query ? to show the list of blocks:

```
User    External        Dependent       Unnamed
Blocks  References      Blocks          Blocks
0       0               0               6
```

The number of unnamed blocks from this table is the number of unique blocks that are currently in the drawing, not counting any multiple instances. The table does not indicate any actual block names.

The 'uname' defined function returns the last name of the unnamed blocks:

```
(defun uname (/ ublist ts)
   (setq ublist nil ts (tblnext "block" t))
   (while ts
      (if (= (substr (cdr (assoc 2 ts)) 1 2) "*U")
         (setq ublist (append ublist (list (cdr (assoc 2 ts)))))
      ); if
      (setq ts (tblnext "block"))
   ); while
   (last ublist)
); uname
```

2. Question: Using the program ABLOCK.LSP, create some anonymous blocks in a new drawing. Undo everything in the drawing and make some more anonymous blocks. Use (entget (entlast)) occasionally as you create the blocks, and examine the names of the blocks. After a number of 'Undo/Back' commands, what is the effect on the names of the anonymous blocks?

Answer: For simply creating anonymous blocks, it is not necessary to have a function to find the last unnamed block, because the "ENDBLK" form of (entmake) returns the block name that is created by (entmake), so your ABLOCK program can be modified to ABLOCK2.LSP.

```
;;; ABLOCK2.LSP   A program to make and insert anonymous blocks
(defun C:ABLOCK2 (/ ins-pt num ss1 i n)
   (setq ins-pt (getpoint "\nInsertion point: "))
   (entmake (list '(0 .  "BLOCK")'(2 .  "*U")'(70 .  1)(cons 10 ins-pt)))
   (prompt "\nSelect entities")
   (setq ss1 (ssget) i (sslength ss1) n (- 1))
   (repeat i
      (entmake (cdr (entget (ssname ss1 (setq n (1+ n))))))
   )
   (setq num (entmake '((0 .  "ENDBLK")) ))
   (entmake (list '(0 .  "INSERT")(cons 2 num)(cons 10 ins-pt)))
   (command "move" "L" "" ins-pt pause "redraw")
   (princ)
); ablock2
```

After doing this exercise, you will notice that the block name returned by (entget (entlast)) continues to increase, but with each successive UNDO BACK, the number of blocks displayed by BLOCK ? starts again numbering from 1. This demonstrates that the intermediate unnamed block names are not discarded in the current drawing session. You can check the last block number with the (uname) function alone to confirm that it is the same as that returned by (entget (entlast)) while you are creating the blocks.

3. Question: After you have been through some cycles of creating anonymous blocks and using 'Undo/Back', record the name of the last anonymous block that you created. Save your drawing, and then exit the drawing session and return to the saved drawing. Now examine the name of the same anonymous block that you looked at previously (you can do this with (entget (car (entsel)))....). What has happened to the block name?

Answer: The block name will be reduced to a number in the new sequence, starting from *U0, because AutoCAD automatically reorders the names of the unnamed blocks. This exercise confirms that the names of anonymous blocks can change, so you cannot rely on using the name for identification. This is quite different from the use of ordinary blocks, for which the names remain fixed. If you save anonymous blocks in a drawing, and you need to refer to them without actually picking them with the cursor, you can identify them uniquely by using handles, as we show in the answer to the next question.

4. Question: Repeat the process of question 3 with entity handles enabled, then examine the dxf lists of the anonymous blocks to observe the handles (dxf group code 5).

Answer: Use the HANDLES command with the option ON to turn on handles, then all entities will be assigned a unique 'handle'. After creating some new unnamed blocks, the use of **(foreach x (entget (car (entsel))) (print x))** yields:

```
(-1 .  <Entity name: 600001ba>)
(0 .  "INSERT")
(8 .  "1")
(5 .  "1C")
(2 .  "*U1")
(10 6.45 1.59783 0.0)
(41 .  1.0)
(42 .  1.0)
(50 .  0.0)
(43 .  1.0)
(70 .  0)
(71 .  0)
(44 .  0.0)
(45 .  0.0)
(210 0.0 0.0 1.0)
```

The entity handle is "1C" in group code 5, and is used to identify the block. The function (handent) takes the single argument (the entity handle), and returns the entity name, thus:

```
Command:  (handent "1c")
<Entity name: 600001ba>
```

Note that the block name *U1 and the entity name <Entity name: 600001ba> can change between drawing sessions, so the entity handle is the only constant identifier. You may also notice that the letters in the handles can be upper or lower case. (handles are 'hexadecimal' numbers, which means having a base of 16.) It never ceases to amaze me that we need to combine so many counting systems in a single application!

To keep track of handles in an organized way, you might write a list of the blocks or other entities that you want to identify with their handles to an external file when they are created. You can then use the data in a spread-sheet or a data-base application, or interface with a data-base via the ASE for release 12, which is beyond the scope of this course.

5. Question: Write a program to select an anonymous block by using its handle as a selection criterion. Note that filter lists in (ssget) don't recognize handles (group code 5).

Answer: First, we will create a file HANDFILE.TXT that contains the handles of the anonymous blocks that we create. ABLOCK3.LSP relates the 'handle' to an 'item name'.

```
  (defun C:ABLOCK3 (/ f1 ins-pt num ss1 i n more hndl zzname)
     (if (/= (getvar "HANDLES") 1) (command "HANDLES" "ON"))
     (setq f1 (open "handfile.txt" "w")); create file in write mode
     (write-line "ITEM NAME AND HANDLE FILE" f1) ; write file header
     (close f1)
     (setq f1 (open "handfile.txt" "a")) ; open file in append mode
     (setq more t)
     (while more  ; start a loop to create several blocks
        (initget "eXit")  ; loop exit criteria
        (setq ins-pt nil)
        (setq ins-pt (getpoint "\nInsertion point/<eXit>: "))
        (if (or (= ins-pt nil) (= ins-pt "eXit"))
           (progn
              (setq more nil)  ; loop ends if more is nil
           ); progn
           (progn          ; this is where the entities are made
              (entmake (list '(0 .  "BLOCK")'(2 .  "*U")'(70 .  1)(cons 10 ins-
pt)))
              (prompt "\nSelect entities")
              (setq ss1 (ssget) i (sslength ss1) n (- 1))
              (repeat i
                 (entmake (cdr (entget (ssname ss1 (setq n (1+ n))))))
              ); repeat
              (setq num (entmake '((0 .  "ENDBLK")) ))
              (entmake (list '(0 .  "INSERT")(cons 2 num)(cons 10 ins-pt)))
              (command "move" "L" "" ins-pt pause "redraw")
              (setq hndl (cdr (assoc 5 (entget (entlast))))) ; get the entity
handle
              (setq zzname (getstring "\nItem name: ")) ; invent an 'item name'
              (write-line (strcat zzname " " hndl) f1) ; write the item & handle
           ); progn                                    to the file with a
space between them.
        ); if
     ); while
     (close f1)
     (princ)
  ); ablock3
```

ABLOCK3.LSP is a modification of ABLOCK2.LSP. The main differences are that ABLOCK3.LSP uses a while loop so that any number of new unnamed blocks can be created in sequence, and some block information can be written to an external file. The user is requested to supply an 'item name', which serves as the equivalent to a fixed block name. The item name is associated with the handle of the block in the sense that both data are written on the same line in the external file.

Note that the entity handle, associated with DXF group code 5, is a hexadecimal number, but its type is actually a string. It can therefore be concatenated as a string with the item name as shown in the statement **(write-line (strcat zzname " " hndl) f1)**. The two data items are separated by a space, provided as shown in the double quotation marks

The program HSELECT.LSP retrieves the handles from the file by reference to the item name.

```
  ;;;  HSELECT.LSP   - a lisp program to
```

```
;;;        select entities (ablocks) by their handles
(defun C:HSELECT ()
   (setq bname (getstring "\nItem Name: ")) ; get the item name
   (setq nlen (strlen bname))                                  ; of the block.
   (setq f1 (open "handfile.txt" "r")) ; open the handles file
   (if f1
      (progn
         (setq more t)
         (while more    ; use 'more' as a test
            (setq ln (read-line f1))
            (setq zzname (substr ln 1 nlen)); read name from file
            (cond ((= (strcase zzname) (strcase bname));compare name
               (progn              ; if the name matches the item name
                  (setq hndl (substr ln (+ nlen 2) 20)) ; read the handle
                  (setq ent (handent hndl)) ; note - the handle is a string
                  (redraw ent 3);   highlight the block
                  (setq more nil);  end the loop criteria
               )); progn and end first conditional statement
               (t nil)
            ); cond
         ); while
         (close f1)
      ); progn
      (princ (strcat "Cannot open file "))   ; no file identifier necessary
   ); if
   (princ)
); hselect
```

The item name and the block handle are written to the file "handfile.txt" on the same line, separated by a space, so to retrieve both of them, you can make use of the number of characters of the item name, as shown in the statement **(setq nlen (strlen bname))**.

When a line has been read from the file, the (substr) function is used to separate the two parts of the line. First, the statement **(setq zzname (substr ln 1 nlen))** is used to get the item name, assuming the number of characters in the item name is nlen. The (cond) function tests that zzname matches the user supplied item name, and if it does, the handle is extracted from the line by **(setq hndl (substr ln (+ nlen 2) 20))**. The block is then identified by **(handent)**, which takes the entity handle as an argument, and returns the entity name, which can change for different drawing sessions.

The file "handfile.txt", created by the ABLOCK3.LSP program, has the form:

```
ITEM NAME AND HANDLE FILE
one 3F
two 46
three 4E
four 56
five 64
```

In the program HSELECT.LSP, I arbitrarily allowed up to 20 characters for the handle. You can modify this, remembering that the handle is a hexadecimal number having a base of 16, so *nn* characters allows up to 16^{nn} unique handles.

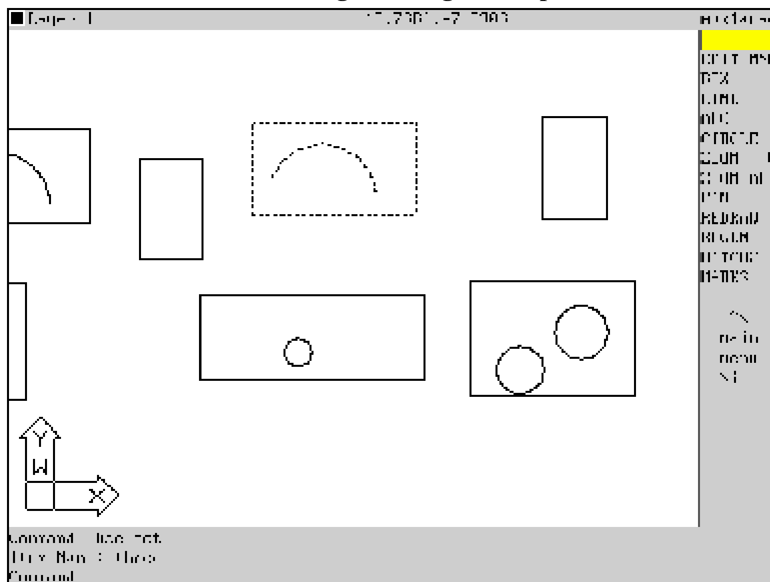

Figure SG6-1 HSELECT.LSP in operation

Advanced AutoLISP Programming Correspondence Course

by Tony Hotchkiss, Assistant Professor, State University College at Buffalo, New york. He may be reached at HOTCHKAJ@SNYBUFAA.CS.SNYBUF.EDU. At the time this course was developed he was an Assistant Professor in the Department of Engineering Professional Development, University of Wisconsin-Madison.

Study Guide, part **7** of **8**

Lesson Seven - Extended Entity Data

Solutions to the homework questions:

1. Question: Modify the ADDXDAT.LSP from this lesson so that it will attach xdata to an object, even if the object already has xdata.

Answer: The original ADDXDAT.LSP created the extended entity data that could subsequently be attached to any object. This time, because we are dealing with an existing object, we need to check whether or not the object already has a registered application name. Note that it is possible to attach xdata to an object under a number of different registered application names. I assume throughout that you have registered the appid's.

In the lesson, I indicated also that if you add xdata to an object that may already contain xdata, then you should check that there is room to add more.

The program ADDXDAT2.LSP is a simple example that modifies ADDXDAT.LSP to make these checks and to attach additional xdata to a block:

```
;;; ADDXDAT2.LSP   A function to create an xdata list
;;;            and add it to an existing block that may contain
;;;            some extended entity data
(defun addxdat2 (/ bname appname oldxdat app_exists yq
      xdat more xdname xdtype xdval item)
   (setq appname (getstring "\nApplication ID name: "))
;;;******************** new code ************
   (setq bname (car (entsel "Select a block: ")))
   (redraw bname 3)
   (if (not (setq oldxdat (assoc -3 (entget bname (list appname)))))
      (progn
         (setq app_exists nil)
         (prompt (strcat "\nApplication name " appname " not found"))
         (initget "Yes Quit")
         (setq yq (getkword " ....  Create it? Quit/<Yes> "))
         (if (= yq "Quit") (quit))
      ); progn
      (setq app_exists T)
   ); if
;;;***********************************************
```

The program begins as before by asking the user to supply an application name. The user selects a block, and the program tests that the application name does not exist, and the program offers a choice of creating the application name or terminating with (quit). The variable *app_exists* is set to nil or T accordingly. Note that we keep the xdata as the variable *oldxdat* for future reference.

The program continues with the section of ADDXDAT.LSP that creates all of the data contained in the {...} brackets:

```
    (setq xdat (list (cons 1002 "{")))
   (setq more t)
   (while more
      (setq xdname (getstring "\nItem name <exit>: "))
      (if (/= xdname "")
         (progn
            (setq xdname (cons 1000 xdname))
            (setq xdat (append xdat (list xdname)))
            (initget "String Distance Integer Real")
```

```
          (setq xdtype (getkword
             "\nItem type String/Distance/Integer/Real: "))
          (setq xdval (getstring "\nItem value: "))
          (setq item
             (cond
                ((= xdtype "String") (cons 1000 xdval))
                ((= xdtype "Distance") (cons 1041 (atof xdval)))
                ((= xdtype "Integer") (cons 1070 (atoi xdval)))
                ((= xdtype "Real") (cons 1040 (atof xdval)))
                (t nil)
             ); cond
          ); setq
          (setq xdat (append xdat (list item)))
          (setq more t)
       ); progn
       (setq more nil)
    ); if
 ); while
 (setq xdat (append xdat (list (cons 1002 "}"))))
```

After this, some more new code is required to check the size of any existing xdata and to attach the xdata. The new code is done in two parts, depending on whether the given application name exists or not. The test is (if app_exists... which is either nil or T from the beginning section of the program. If the application name was found, the (xdroom) and (xdsize) functions are tested, and if there is room for the additional xdata, this is appended appropriately to the old xdata using the application name as follows:

```
;;;************ new code from here on ********************
 (if app_exists
    (progn
       (setq newxdat (list -3 (cons appname xdat)))
       (if (> (xdroom bname) (xdsize newxdat))
          (progn
             (setq oldlist (cdadr oldxdat))
             (setq xdat (append oldlist xdat))
             (setq xdat (list -3 (cons appname xdat)))
             (setq elist (entget bname (list appname)))
             (setq newlist (subst xdat (assoc -3 elist) elist))
             (entmod newlist)
             (entupd bname)
          ); progn
          (prompt "\nNot enough room for the new xdata...")
       ); if
    ); progn
```

The xdata size test is made as suggested in the lesson, so the variable newxdat is defined as shown for that purpose. The technique for appending the old and new xdata is to append only the lists that are contained within the {} brackets, so the old list is defined as (cadr oldxdat) in order to strip off the -3 and the application name, leaving a list that starts with (1002 . "{"). The append function is a slightly modified form of the one given in the lesson, because the form of xdat is already an appropriate list in this case. The xdata is completed by adding the (-3 ("appid... to it with the statement (setq xdat (list -3 (cons appname xdat))). The new list is then created with the (subst) function as shown, before modifying and updating.

The 'else' part of the if...then...else function attaches the xdata to the block as was done in the lesson, just as though there were no other xdata attached:

```
    (progn
       (setq xdat (list -3 (cons appname xdat)))
       (setq elist (entget bname))
       (setq newlist (append elist (list xdat)))
       (entmod newlist)
       (entupd bname)
    ); progn
 ); if
);end of defun
```

In fact, it is quite possible that there is some existing xdata, but under a different application name. If this is the case, this last section of the program will still attach the new xdata properly, under a newly defined application name, but it would of course be appropriate to perform the checking for xdroom and xdsize. This time, you could test (xdroom bname), which will return 16383 if there is no existing xdata.

2. The program ABLOCK.LSP, from lesson six, uses (entmake) with (entget) to create anonymous blocks from simple entities. How can you modify this program to select objects that already contain xdata? The result will be to make nested anonymous blocks.

Answer: The following ABLOCK2.LSP is a simplified version of the earlier ABLOCK.LSP, and does exactly the same functions as ABLOCK.LSP. The difference is that ABLOCK2.LSP uses the fact that the ENDBLK parameter forces (entmake) to return the block name.

```
;;; ABLOCK2.LSP   A program to make and
;;;                              insert anonymous blocks
(defun C:ABLOCK2 (/ ins_pt num ss1 i n)
   (setq ins_pt (getpoint "\nInsertion point: "))
   (entmake (list '(0 .  "BLOCK")'(2 .  "*U")'(70 .  1)(cons 10 ins_pt)))
   (prompt "\nSelect entities")
   (setq ss1 (ssget) i (sslength ss1) n (- 1))
   (repeat i
      (entmake (cdr (entget (ssname ss1 (setq n (1+ n))))))
   )
   (setq num (entmake '((0 .  "ENDBLK")) ))
;*************** Insert the anonymous block ******
   (entmake (list '(0 .  "INSERT")(cons 2 num)(cons 10 ins_pt)))
   (command "move" "L" "" ins_pt pause "redraw")
   (princ)
); ablock2
```

Now compare ABLOCK2.LSP with the following XABLOCK.LSP, which can make anonymous blocks from component objects that can have xdata attached to them. The only difference here is that the user is asked to supply the appid, which is stored as the symbol app_id as shown. The app_id is then used later in the (entget) statement to make sure that the xdata is included in the definition of the block.

```
;;; XABLOCK.LSP   A program to make and insert anonymous blocks
;;;     whose components can have XDATA attached
(defun C:XABLOCK (/ ins_pt num ss1 i n app_id aname)
   (setq ins_pt (getpoint "\nInsertion point: "))
   (setq app_id (getstring "\nApplication name <*>: "))
   (if (= app_id "") (setq app_id "*"))
   (entmake (list '(0 .  "BLOCK")'(2 .  "*U")'(70 .  1)(cons 10 ins_pt)))
   (prompt "\nSelect entities")
   (setq ss1 (ssget) i (sslength ss1) n (- 1))
   (repeat i
      (entmake (cdr (entget (ssname ss1 (setq n (1+ n))) (list app_id))))
   )
   (setq num (entmake '((0 .  "ENDBLK")) ))
;*************** Insert the anonymous block ******
   (entmake (list '(0 .  "INSERT")(cons 2 num)(cons 10 ins_pt)))
   (command "move" "L" "" ins_pt pause "redraw")
   (princ)
); xablock
```

Note that the application name can be a 'wild-card' asterisk so that any extended entity data associated with any application name will be created as part of the new block.

3. Modify the program LISTBLK.LSP from lesson five to list blocks that contain extended entity data, even if the blocks are nested.

Answer: This problem is complicated by the fact that xdata is not attached to blocks, but only to the block references (inserts). Xdata can also be attached to other objects such as lines, arcs, etc., but not to items that are stored as tables in AutoCAD. The program LISTBLK.LSP lists the individual components of blocks by reference to the block table. The modified form LISTBLK2.LSP must use both table data and the insert entities separately, bearing in mind that the insert entity can in turn contain other nested block references. The program uses a defined function (do-item) that handles nested insert entities.

```
;;;LISTBLK2.LSP - A program to list block data
;;;           to an external file even if the nested blocks
;;;           contain extended entity data.
(defun init (/ fname iname bname il)
   (setq fname (getstring "\nEnter the .TXT output file name: "))
   (setq fname (strcat "\\acad12\\drwgs\\" fname ".txt"))
   (setq iname (car (entsel "\nSelect block reference: ")))
   (redraw iname 3)
   (setq f1 (open fname "w"))
   (setq bname (cdr (assoc 2 (entget iname)) ))
   (print (strcat "DXF LIST FOR BLOCK: " bname) f1)
   (prntxd iname bname)
   (setq il (list bname iname))
); init
(defun prntbl (bname / blist x)
         (setq blist (tblsearch "block" bname))
         (print (strcat "Block Table List for Block " bname) f1)
         (foreach x blist (print x f1))
         (print "End of Block Table List" f1)
); prntbl
(defun prntxd (iname bname / xdt x)
   (if (setq xdt (cdadr (assoc -3 (entget iname '("*")) )) )
      (progn
         (print (strcat "Block " bname " has xdata:") f1)
         (foreach x xdt (print x f1))
         (print "End of xdata list" f1)
      ); progn
   ); if
); prntxd
(defun getlist (blklist / ename b blist)
   (setq ename (cdr (assoc -2 blklist)))
   (setq blist nil)
   (while ename
      (setq blist (append blist (list (entget ename))))
      (setq ename (entnext ename))
   ); while
   (setq b blist)
); getlist
(defun do-item (elist / bname iname blist)
   (if (= (cdr (assoc 0 elist)) "INSERT")
      (progn
         (setq bname (cdr (assoc 2 elist)))
         (print (strcat "Sub-Block " bname) f1)
         (prntbl bname)
         (setq iname (cdr (assoc -1 elist)))
         (prntxd iname bname)
         (setq blist (getlist (tblsearch "block" bname)))
         (foreach item blist (do-item item))
      ); progn
      (foreach item elist (print item f1))
   ); if
); do-item
(defun listblk2 (/ inlist bname iname blist)
   (setq inlist (init) bname (car inlist) iname (cadr inlist))
   (setq blist (getlist (tblsearch "block" bname)))
   (prntbl bname)
   (foreach item blist
      (print (strcat "First-Level Item of " bname) f1)
      (do-item item)
   ); foreach
   (redraw iname) (close f1) (princ)
); listblk2
(defun C:LB2 () (listblk2))
```

Let's examine the main function (init) first. This function asks the user to supply a file name for the dxf listings, and then to select a block reference. It is assumed that an insert entity is selected, although of course it would be well to test that and to warn the user accordingly. The selected block reference is highlighted with the redraw statement as shown. The name of the block is extracted from the insert entity using the (assoc 2) format in the statement **(setq bname (cdr (assoc 2 (entget iname))))** after which the header for the external file is printed with the name of the selected block. (init) then calls the defined function (prntxd) which prints any extended entity data (xdata), if it exists for the chosen insert entity. The function (printxd) prints the xdata list for any application name, as suggested by the wild card in the (entget) statement of (printxd). Finally, (init) returns the list of the block name and the insert entity name via the last (setq) statement.

The (prntxd) function takes a block name and the corresponding insert name as its input, and tests to see if there is any xdata. Note how the xdata is extracted for any application name by using the wild card asterisk format in the (entget) function:

(setq xdt (cdadr (assoc -3 (entget iname '("*")))))

The xdata list is printed to the file on one line at a time using a (foreach) loop as shown, preceded and followed by some messages for easy file scanning later on.

The main function (listblk2) calls (init) and extracts the block and insert entity names with its first (setq) statement as shown. The defined function (getlist) is then called in order to obtain a dxf list of all of the first-level components of the selected block. Note that at this stage we are not interested in the 'nested' entities, as these will be dealt with later. (getlist) takes the list returned by (tblsearch "block"...) for the selected block as input, and returns the dxf lists of each major component of the block, and this is assigned to the variable 'blist' as shown in the second statement of (listblk2). Another defined function (prntbl) is then called in order to print the block table list for the selected block to the external file. (listblk2) continues with the foreach loop shown in order to deal with each of the components of the selected block in turn. For each entity (first level of nesting), a message is printed to the external file to let us know that this is the start of a new first-level component of the selected block.

The defined function (do-item) is called for each of these block components in turn, and we shall see that (do-item) really does all of the work of stepping through the component entities of the selected block, regardless of how many layers of nesting their may be, and of course, taking into account and listing any xdata that may be attached to any of the nested blocks.

(do-item) takes as its only argument the dxf list of one component entity of a block. That component may be an individual entity or a block. I have not catered for any other kind of complex entity such as a polyline, but these entities can be handled in just the same manner as we handle blocks (inserts), as follows. The function (do-item) tests that the dxf group code zero is of type "INSERT", and if it is, the first set of (progn) statements are executed, otherwise the dxf list is written to the external file one line at a time with the (foreach) loop shown as the 'else' section of the (if) statement.

If the entity is an "INSERT", the corresponding block name is captured with (assoc 2..) as shown, and the heading "Sub-Block ..." followed by the block name is written to the next line of the external file. The block table data is then written to the file using the defined function (prntbl). Next, the insert entity name is extracted from the dxf list and the xdata, if any, is written to the file using (prntxd). Now comes the interesting part: the function (getlist) is used on the sub-block just as it was used on the initially selected block so that a list of the sub-blocks' component parts can be returned and stored as shown in the local variable 'blist'. After this, the (foreach) loop takes each item of the current block component list in turn and executes the function (do-item) on it. This procedure continues using the process of looping by calling (do-item) for every nested block until the deepest nested components are single entities.

This method of looping is called 'recursion', which will be discussed further in the next and final lesson of this series.

Remember that (do-item) is always called in a (foreach) loop, so each of the nested blocks at any given level of nesting will be handled in turn, and no component item can be ignored or left out of the process. The result is that the external file contains listings of every component of every nested block with all of the xdata that has been attached to the blocks. I have not listed any xdata for non-block entities, but you can easily include that by writing and calling a function similar to (prntxd) for simple entities as part of the 'else' section of the (if) statement of (do-item). For this, the printed statement "Block so-and-so has xdata" can be changed to the "entity-type has xdata" where the entity type is extracted from group code zero of its dxf list.

I have included a partial print-out of a file TEXTL2.TXT of nested blocks which were selected from the shapes of figure SG7-1, of which two of the blocks contain some simple xdata just to give an example of some nested entities. I used the programs XABLOCK.LSP and ADDXDAT2.LSP to create the examples for this program. You can see that

the same few blocks have been collected together to form new blocks in different combinations and with different levels of nesting to produce the file TEXTL2.TXT.

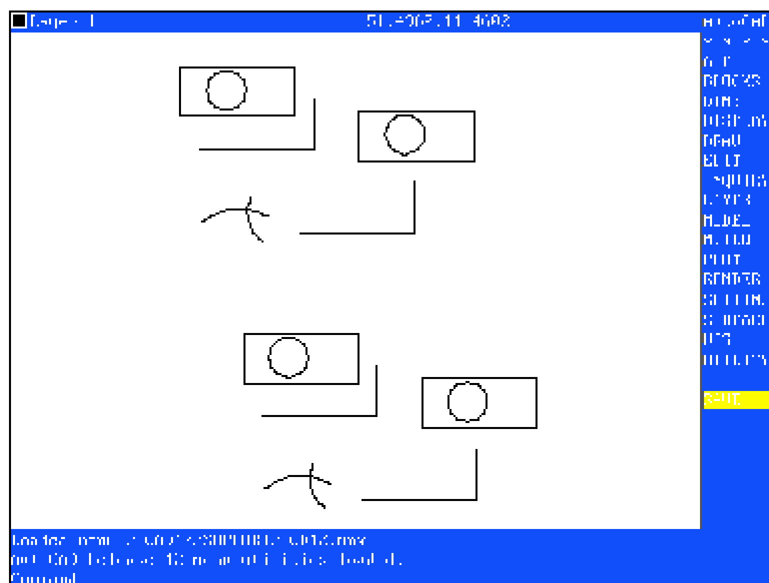

Figure SG7-1, some nested blocks for testing LISTBLK2.LSP

An extract from the file TEXTL2.TXT

```
"DXF LIST FOR BLOCK: *U7"
"Block Table List for Block *U7"
(0 .  "BLOCK")
(2 .  "*U7")
(70 .  65)
(10 23.1466 6.42278 0.0)
(-2 .  <Entity name: 4000058a>)
"End of Block Table List"
"First-Level Item of *U7"
"Sub-Block *U4"
"Block Table List for Block *U4"
(0 .  "BLOCK")
(2 .  "*U4")
(70 .  65)
(10 7.28314 5.81306 0.0)
(-2 .  <Entity name: 4000033e>)
"End of Block Table List"
"Sub-Block *U1"
"Block Table List for Block *U1"
(0 .  "BLOCK")
(2 .  "*U1")
(70 .  65)
(10 9.06046 5.94565 0.0)
(-2 .  <Entity name: 40000118>)
"End of Block Table List"
"Block *U1 has xdata:"
(1002 .  "{")
(1000 .  "it1_tony")
(1000 .  "tony_item-1")
(1002 .  "}")
(1002 .  "{")
(1002 .  "}")
(1002 .  "{")
(1000 .  "test-tony1")
(1000 .  "test-again-tony1")
(1002 .  "}")
"End of xdata list"
```

```
(-1 .  <Entity name: 40000118>)
(0 .  "LINE")
(8 .  "1")
(10 9.64781 5.55435 0.0)
(11 3.64376 5.55435 0.0)
(210 0.0 0.0 1.0)
(-1 .  <Entity name: 40000142>)
(0 .  "LINE")
(8 .  "1")
(10 9.64781 8.16304 0.0)
(11 9.64781 5.55435 0.0)
(210 0.0 0.0 1.0)
"Sub-Block *U0"
"Block Table List for Block *U0"
(0 .  "BLOCK")
(2 .  "*U0")
(70 .  65)
(10 6.60228 6.70652 0.0)
(-2 .  <Entity name: 40000022>)
"End of Block Table List"
"Block *U0 has xdata:"
(1002 .  "{")
(1000 .  "it1_fred")
(1070 .  23)
(1002 .  "}")
"End of xdata list"
(-1 .  <Entity name: 40000022>)
(0 .  "CIRCLE")
(8 .  "1")
(10 6.03668 6.94565 0.0)
(40 .  1.04441)
(210 0.0 0.0 1.0)
(-1 .  <Entity name: 40000044>)
(0 .  "LINE")
(8 .  "1")
(10 9.64781 5.55435 0.0)
(11 3.64376 5.55435 0.0)
(210 0.0 0.0 1.0)
(-1 .  <Entity name: 4000006e>)
(0.........etc.
```

In order to have a more organized external file, you might consider modifying the program as LISTBLK3.LSP as follows:

```
;;;LISTBLK3.LSP - A program to list block data to an
;;; external file.
(defun init (/ fname iname bname il)
   (setq fname (getstring "\nEnter the .TXT output file name: "))
   (setq fname (strcat "\\acad12\\drwgs\\" fname ".txt"))
   (setq iname (car (entsel "\nSelect block reference: ")))
   (redraw iname 3)
   (setq f1 (open fname "w"))
   (setq bname (cdr (assoc 2 (entget iname)) ))
   (write-line (strcat "DXF LIST FOR BLOCK: " bname) f1)
   (prntxd iname bname 1)
   (setq il (list bname iname))
); init
(defun prntbl (bname lvl / blist item)
   (setq blist (tblsearch "block" bname))
   (princ (strcat
                  (prfx lvl) "Block Table Data for Block " bname) f1)
   (foreach item blist
      (if (or
              (= (car item) 0)
              (= (car item) 2)
```

```
               (= (car item) -2)
            ); or
            (progn
              (princ (strcat (prfx lvl)) f1)
              (princ item f1)
            ); progn
        ); if
    ); foreach
    (princ (strcat (prfx lvl) "End of Block Table Data") f1)
); prntbl
(defun prntxd (iname bname lvl / xdt x)
    (if (setq xdt (cdadr (assoc -3 (entget iname '("*")) )) )
        (progn
            (princ (strcat (prfx lvl) "Block " bname " has xdata:") f1)
;           (foreach x xdt (print x f1))
;           (print (strcat (prfx lvl) "End of xdata list") f1)
        ); progn
    ); if
); prntxd
(defun getlist (blklist / ename b blist)
    (setq ename (cdr (assoc -2 blklist)))
    (setq blist nil)
    (while ename
        (setq blist (append blist (list (entget ename))))
        (setq ename (entnext ename))
    ); while
    (setq b blist)
); getlist
(defun prfx (lvl / x)
    (setq x "\n")
    (if (/= lvl 0)
        (repeat lvl
            (setq x (strcat x "   "))
        ); repeat
        (setq x "\n")
    ); if
); prfx
(defun do-item (elist lvl / bname iname blist)
    (foreach item elist
        (if (or
               (= (car item) 0)
               (= (car item) -1)
               (= (car item) 2)
            ); or
            (progn
               (princ (strcat (prfx lvl)) f1)
               (princ item f1)
            ); progn
        ); if
    ); foreach
    (if (= (cdr (assoc 0 elist)) "INSERT")
        (progn
            (setq bname (cdr (assoc 2 elist)))
            (princ (strcat (prfx (1+ lvl)) "Level "
                                     (itoa (1+ lvl)) " Sub-Block " bname) f1)
            (prntbl bname (1+ lvl))
            (setq iname (cdr (assoc -1 elist)))
            (prntxd iname bname (1+ lvl))
            (setq blist (getlist (tblsearch "block" bname)))
            (foreach item blist (do-item item (1+ lvl)))
            (princ (strcat (prfx (1+ lvl)) "End of Sub-Block " bname) f1)
        ); progn
    ); if
); do-item
```

```
(defun listblk3 (/ inlist bname iname blist)
   (setq inlist (init) bname (car inlist) iname (cadr inlist))
   (setq blist (getlist (tblsearch "block" bname)))
   (prntbl bname 0)
   (foreach item blist
      (princ (strcat (prfx 0) "First-Level Item of " bname) f1)
      (do-item item 0)
   ); foreach
   (redraw iname) (close f1) (princ)
); listblk3
(defun C:LB3 () (listblk3))
```

This time, in LISTBLK3.LSP, I have cut down on the amount that is printed for each entity, restricting the output to group codes 0, 2, and -2 together with some headings. I have also restricted the xdata to a heading. LISTBLK3.LSP has a system of indentation for each level of nesting, and this is provided by the additional defined function (prfx), which prefixes a number of spaces and a line feed to the lines to be printed. A new variable 'lvl' keeps a count of the level of nesting. Here is a sample output, TEXTL3.TXT, for the same example blocks of figure SG7-1:

```
DXF LIST FOR BLOCK: *U7
Block Table Data for Block *U7
(0 .  BLOCK)
(2 .  *U7)
(-2 .  <Entity name: 4000058a>)
End of Block Table Data
First-Level Item of *U7
(-1 .  <Entity name: 4000058a>)
(0 .  INSERT)
(2 .  *U4)
   Level 1 Sub-Block *U4
   Block Table Data for Block *U4
   (0 .  BLOCK)
   (2 .  *U4)
   (-2 .  <Entity name: 4000033e>)
   End of Block Table Data
   (-1 .  <Entity name: 4000033e>)
   (0 .  INSERT)
   (2 .  *U1)
      Level 2 Sub-Block *U1
      Block Table Data for Block *U1
      (0 .  BLOCK)
      (2 .  *U1)
      (-2 .  <Entity name: 40000118>)
      End of Block Table Data
      Block *U1 has xdata:
      (-1 .  <Entity name: 40000118>)
      (0 .  LINE)
      (-1 .  <Entity name: 40000142>)
      (0 .  LINE)
      End of Sub-Block *U1
   (-1 .  <Entity name: 400003a1>)
   (0 .  INSERT)
   (2 .  *U0)
      Level 2 Sub-Block *U0
      Block Table Data for Block *U0
      (0 .  BLOCK)
      (2 .  *U0)
      (-2 .  <Entity name: 40000022>)
      End of Block Table Data
      Block *U0 has xdata:
      (-1 .  <Entity name: 40000022>)
      (0 .  CIRCLE)
      (-1 .  <Entity name: 40000044>)
      (0 .  LINE)
      (-1 .  <Entity name: 4000006e>)
      (0 .  LINE)
```

```
      (-1 .  <Entity name: 40000098>)
      (0 .  LINE)
      (-1 .  <Entity name: 400000c2>)
      (0 .  LINE)
      End of Sub-Block *U0
   End of Sub-Block *U4
First-Level Item of *U7
(-1 .  <Entity name: 400005a6>)
(0 .  INSERT)
(2 .  *U0)
   Level 1 Sub-Block *U0
   Block Table Data for Block *U0
   (0 .  BLOCK)
   (2 .  *U0)
  (-2 .  <Entity name: 40000022>)
   End of Block Table Data
   Block *U0 has xdata:
   (-1 .  <Entity name: 40000022>)
   (0 .  CIRCLE)
   (-1 .  <Entity name: 40000044>)
   (0 .  LINE)
   (-1 .  <Entity name: 4000006e>)
   (0 .  LINE)
   (-1 .  <Entity name: 40000098>)
   (0 .  LINE)
   (-1 .  <Entity name: 400000c2>)
   (0 .  LINE)
   End of Sub-Block *U0
First-Level Item of *U7
(-1 .  <Entity name: 400005d9>)
(0 .  INSERT)
(2 .  *U5)
   Level 1 Sub-Block *U5
   Block Table Data for Block *U5
   (0 .  BLOCK)
   (2 .  *U5)
   (-2 .  <Entity name: 40000400>)
   End of Block Table Data
   (-1 .  <Entity name: 40000400>)
   (0 .  INSERT)
   (2 .  *U1)
      Level 2 Sub-Block *U1
      Block Table Data for Block *U1
      (0 .  BLOCK)
      (2 .  *U1)
      (-2 .  <Entity name: 40000118>)
      End of Block Table Data
      Block *U1 has xdata:
      (-1 .  <Entity name: 40000118>)
      (0 .  LINE)
      (-1 .  <Entity name: 40000142>)
      (0 .  LINE)
      End of Sub-Block *U1
   (-1 .  <Entity name: 40000463>)
   (0 .  ARC).......etc.
```

Advanced AutoLISP Programming Correspondence Course

by Tony Hotchkiss, Assistant Professor, State University College at Buffalo, New york. He may be reached at HOTCHKAJ@SNYBUFAA.CS.SNYBUF.EDU. At the time this course was developed he was an Assistant Professor in the Department of Engineering Professional Development, University of Wisconsin-Madison.

Study Guide, part 8 of 8

Lesson eight does not have an assignment to complete. Instead, Tony Hotchkiss has prepared sample programs you my use in any manner you would like. This gives you an opportunity to see different examples of his work. You may use these for ideas and as starting points for your own refinements. There are not restrictions on their use.

The table of contents follows and a disk is included in the course package for registered students.

TABLE OF CONTENTS

ABLOCK2.LSP Make and insert anonymous blocks
ABLOCK3.LSP Make, insert, & store handles of anonymous blocks in a file
ADDXDATA.LSP To create an extended entity data list
ADDXDAT2.LSP To add XDATA to a block (with checks for existing xdata)
ARRLINE.LSP Place arrowheads at the end of a line or arc
BASEPLAT.DAT Baseplate parameters for BASEPLAT.LSP
BASEPLAT.LSP Draw and dimension a baseplate with design parameters from a file
BOX1.LSP Draw a square with origin and length of side
BOX2.LSP Draw a square, dragging the length of side
BOX3.LSP Alternative draw square with one point variable
BOX31.LSP Draw and dimension a square with origin and length of side
CHATTXT.LSP Change attribute text height - user selects block
CHKLYR.LSP Check if layer exists and make it current anyway
CHLAYER.LSP Change layer of any set of entities to a selected entity target layer
CLYR2.LSP Short version of CHLAYER.LSP
CUSP1.LSP Calculate cusp height
CUSP2.LSP Calculate step-over distance
CUSP3.LSP Draw and dimension ball nose cutter given tool dia and cusp height
DLBRK.LSP Break double line at intersection with line or polyline
DLBRK2.LSP Break double line at intersection with a line
DXF.LSP Returns data element of associated pair, given dxf code and list
FASTFACE.LSP Draw a Pface mesh from external file of points
FIBON.LSP Returns nth number in fibonacci series (recursively)
FILTLIST.LSP Creates a filter list
FINALXX.LSP Stairs program with rolling ball animation
GRAPH1.LSP Draw polyline with entmake from external file points
GRAPH2.LSP Plot a graph with y-ordinate values from external file
GRAPH4.LSP Plot a scaled graph with y-ordinates from external file
HATCH2.LSP Hatch a non-contiguous boundary by indicating INT points
HSELECT.LSP Select anonymous blocks by their handles, from external file
IB3.LSP Draw and fillet an I-beam with Height argument
IBEAM1SG.LSP Draw but not fillet an I-beam

BEAMDATA.DAT Ibeam parameters for IBEAM4.LSP
IBEAM4.LSP Parametric I-beam with data from an external file
IBEAM7.LSP Draw an I-beam with design rules

IBEAM8.LSP Draw an I-beam with design rules in modular form
IDBLOCK.LSP Extract data from solid model by reading xdata of anon. blocks
INFLINES.LSP Infinite length lines in X or Y direction

INS1.LSP Insert any of six blocks using a slide and (getkword)
INSTNUM.LSP List the names of inserted anonymous blocks
LBEAM.LSP Draw an L-beam with rotation angle
LISTATT.LSP List attribute data to an external file

LISTBLK.LSP List block data to an external file
LISTBLK2.LSP List block data to an external file with more formatting
LISTBLK3.LSP List block data to an external file with full indentation
LISTDXF.LSP List dxf data for various entities to an external file LISTDXF.TXT
LISTPOLY.LSP List polyline data to an external file
LORP.LSP Draw a square as lines or a polyline
LSHAP.LSP Draw an L-beam with fillets

MESH3D2.LSP Draws a 3DMESH with vertices from an external file
MKBLKS2.LSP Makes blocks with entmake

PINECONE.LSP Draw a plan view of a pinecone using fibonacci series
PRINTEST.LSP Prints the file TEST.LSP
PRINTX.LSP Prints the file AUX1.MNU
PRNTESTC.LSP Prints the file TEST.C
PROTO.LSP Prototype LISP program
PROTOX.LSP Prototype LISP program using (set) for system variables
RECTAN2.LSP Draw a rectangle at an angle
RECTAN3.LSP Draw a rectangle from diagonal points with (getcorner)
RECTANG.LSP Draw a rectangle from diagonal points with (getpoint)
SAVE2.LSP Saves drawings on a network and to local C:\acad10\ directory
SAVE3.LSP Saves drawings on a network, version 2
SAVE4.LSP Saves drawings on a network, version 3
SAVTEST.LSP Copy .LSP file to another filename
SAVTESTC.LSP Copy .C file to another filename
SETVARS.LSP Uses (set) to set system variables
SHAFT.LSP Draws and dimensions a shaft parametrically
SUBTEXT.LSP Substitute all instances of a text string
XHATCH.LSP Hatch by giving internal point (BOUNDARY HATCH)
XPFACEX.LSP Draw a PFACE mesh from data file points (uses pointlist)
XUHATCH.LSP Boundary hatch for line/arcs with user hatch pattern
XXHATCH.LSP Boundary hatch with auto layers
XXPFACE.LSP Creates a PFACE mesh from external data file
YRASE.LSP Erase with filtering (simple filter exercise)

End of sample programs