AutoLISP Programming Techniques

Lesson One in the CADalyst/University of Wisconsin Advanced AutoLISP Programming course covers AutoLISP program structure & development.

by
Anthony Hotchkiss, Ph.D, P.Eng.

About this course.

Advanced AutoLISP Programming is an eight-lesson independent study course, originally published in monthly installments in CADalyst magazine. Before you begin this course, it is assumed that you have reached a basic level of AutoLISP programming, equivalent to completing the preceding 'Introduction to AutoLISP Programming' course which was originally published in CADalyst, and is now available as an independent study course from the University of Wisconsin-Madison. On successfully completing an intermediate and a final mail-in project, registered students will receive certificates of completion and 5.0 CEU's (continuing education units).

For further information, contact:

Steve Bialek
Engineering Professional Development
University of Wisconsin-Madison
432 North Lake Street
Madison, Wisconsin 53706
(608) 262-1735

LESSON ONE - AUTOLISP PROGRAMMING TECHNIQUES

This lesson shows you how to structure your AutoLISP programs for systematic development. You will also learn some techniques for making your programs more efficient. The lesson shows you how to write your own error handler to deal with such error conditions as a cancel during program execution. The lesson ends with some homework questions to test your understanding of the text.

PROGRAM STRUCTURE FOR SYSTEMATIC DEVELOPMENT

A Prototype AutoLISP Program

A typical program contains an input section, a processing or calculation section, and an output section. Also, because your programs interact with AutoCAD, you can include a method for setting system variables and resetting them back to their initial conditions. You can do this with the additional defined functions **(setting)** and **(resetting)**, so the program may resemble the following PROG.LSP

PROG.LSP - a typical program structure

```
;;; PROG.LSP  A program to solve any problem!
(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
```

The program ends with the defined function C:PROG, which simply executes all of the other functions in turn. One advantage of using a format like this is that your final defined function contains the structure and executing order of your program. It helps you to plan the approach to problem solving, and allows you to develop your program with modules that can be debugged in sequence.

We will use this prototype structure as the basis for developing a program to draw a polyface mesh for general surface modeling such as may be used for landscaping, plastic molding, or any application requiring arbitrary shapes to be created.

Development of a Polyface Mesh Program

First, let us understand what AutoCAD needs as input in order to create the polyface mesh. The PFACE mesh requires a set of vertex points, followed by the order of the points for each vertex in turn. You can supply any number of vertices for each face, but we will restrict ourselves in this example to a constant number of 4 vertices per face (quadrilateral faces). You can modify the program for different numbers of vertices per face, but we assume that the number does not vary over the mesh. The command/prompt for the pface mesh is:

```
Command: pface
Vertex 1:
```

Here, you enter as many vertices as you like by supplying point coordinates. In this example we will use a number divisible by 4, as we will be constructing a mesh of quadrilateral faces. when the last vertex has been supplied, enter a null <return> to signal the end of vertex definitions. The next prompt is:

Face 1, vertex 1:

Here you enter the faces by entering vertex numbers for each face in turn, thus:

```
Face 1, vertex 1: 1
Face 1, vertex 2: 2
Face 1, vertex 3: 3
Face 1, vertex 4: 4
<return>
Face 2, vertex 1: 5
Face 2, vertex 2: 6
Face 2, vertex 3: 7
Face 2, vertex 4: 8
<return>
Face 3, vertex 1: 9
....etc.
```

Each group of four numbers is followed by a null <return> to signal the end of the face definition. Finally, when all the faces have been defined, another <return> is issued to terminate the process

It is convenient to read the data (vertex) points from an external file, and our program starts with a note to tell us about the format of the file. The first defun here is the **(setting)** function:

```
XPFACE.LSP   - the (setting) function
;;;  XPFACE.LSP A program by Tony Hotchkiss
;;;  XPFACE CREATES A PFACE MESH
;;;  FROM A DATA FILE.
;;;  THE FORMAT OF THE DATA FILE IS:
;;;   n     (the total number of data points)
;;;  x  y  z   (three fields of 14 characters)
;************************************
(defun setting ()
   (setq ocmd (getvar "CMDECHO"))
   (setvar "CMDECHO" 0)
   (setq oblip (getvar "BLIPMODE"))
   (setvar "BLIPMODE" 0)
); end of setting
```

The program input section asks the user to supply the name of the external data file:

```
XPFACE.LSP   - the (input) function
(defun input ()
   (setq fname (getstring "\nTXT file name: "))
); end of input
```

In the processing section (calc), the point coordinates are read in and assembled into a point list, ready to be supplied to the pface command in sequence. The vertex points are to be stored in a list called ptlst, and **(append)** is used to create the ptlst, so you should initialize ptlst to nil, in case it already contains any data. The file name supplied is concatenated with the .TXT file extension using **(strcat)**, and the file is then opened in 'read' mode. We use the **(read)** function with **(read-line)** to get the first line of the file which contains the single integer indicating the number of data points to follow. Note that **(read)** reads a character string (with no spaces), and returns the string converted to the corresponding data type, so if no decimal point exists, the string representing the number will be converted to an integer. We use **(read)** because **(read-line)** only returns a character string, which may contain spaces. It would also be possible to use **(atoi)** instead of **(read)** to convert the string into an integer.

The subsequent lines of the data file are manipulated somewhat differently, as the x, y, and z coordinates must be extracted as real numbers from a string.

```
XPFACE.LSP   - the (calc) function
(defun calc ()
   (setq ptlst nil)
   (setq fname (strcat fname ".txt"))
   (setq f1 (open fname "r"))
   (if f1   ;test if file exists
      (progn  ;then read and format the points
         (prompt "\nReading the data file....")
         (setq numlines (read (read-line f1)))
         (repeat numlines (setq p (read-line f1))
            (setq px (read (substr p 1 14))
                  py (read (substr p 15 14))
                  pz (read (substr p 29 14))
                  pp (list px py pz)
                  ptlst (append ptlst (list pp))
            ) ; setq
         ) ;end of repeat for list of vertices
         (close f1)
         (setq vxlst nil)
         (setq vlst nil)
         (setq i 1)
         (repeat (/ numlines 4)
            (setq vlst (list i (+ i 1) (+ i 2) (+ i 3)))
            (setq i (+ i 4))
            (setq vxlst (append vxlst (list vlst)))
         ); repeat
      ) ;progn
      (princ  (strcat "Cannot open file " fname))
   ) ; if
   (princ)
) ; end of defun calc
```

Having established the number of data points (numlines) to follow, a repeat loop is used to separate the individual x, y, and z components of the points before recombining them into a list, and finally appending the points to the point list. This rather cumbersome procedure is necessary, as there is no other way to read numbers separated by spaces. **(substr)** returns a partial string of its argument (in this case the argument is 'p'). The nth character in the string and the number of characters (field width) are the last two arguments of **(substr)**. The external data file is formatted in three columns with 14 characters including spaces, so the x coordinate starts in column 1, the y coordinate starts in column 15, and the z in column 29.

After all of the points have been read in, the data file is closed. Note the test used for the 'if' statement is simply f1, which is non-nil if the file exists, and the 'else' expression is the message that 'fname' cannot be opened.

The **(calc)** function continues by defining a list containing the sequence of vertex numbers in groups of 4. The variable vlst contains the numbers 1 through 4, then 5 through 8 etc., and vxlst appends vlst for each face to be created in turn, which is numlines divided by 4. Note that each face in the mesh has 4 unique vertex points, but for all of the internal mesh points, there will be 4 overlapping points, so our data file is much larger than it needs to be, but it simplifies the numbering process to a trivial collection of numbers in ascending order.

The (output) section of our program creates output in the form of the polyface mesh, and the (command) function is used with the appropriate AutoCAD subcommands as follows:

```
XPFACE.LSP - (output) draws the pface mesh
(defun output ()
   (command "pface")
   (foreach p ptlst (command p))
   (command "")
   (while (setq vlst (car vxlst))
      (while (car vlst) (command (car vlst))
         (setq vlst (cdr vlst))
      ); while
      (command "")
      (setq vxlst (cdr vxlst))
   ); while
   (command "")
);  defun
```

Note the simplicity of the output function for drawing the pface mesh of any arbitrary size and shape. The function starts with the pface command, which is entered as a single statement. The next required input is to supply the list of vertices, which is done by dumping out each point in turn from the point list with **(foreach)**, in the standard LISP manner.

After the vertices have been provided, a <return> is issued with **(command "")**. Next, the sets of four numbers have to be given, separated with a <return>, and this is accomplished as shown by taking the first item of the vxlst, and operating on it in a nested while loop. The nested while loop uses **(command)** to dump out the first item of the vlst, then resets the vlst to eliminate its first item, so that the next time round the loop, the first item is actually a new first item, and so on until there are no more items in the list. The outer while loop does exactly the same job on the vxlst, which contains each group of 4 numbers in turn.

Note that this code is actually independent of the number of items in either vlst (the set of numbers that define the vertices per face), or vxlst (the collection of lists vlst). This output section is therefore suitable for dealing with any number of faces with any number of vertices per face.

After each face is defined, a <return> is issued, and after all faces have been created, a final <return> ends the process.

All that remains is the final **(resetting)** section:

```
XPFACE.LSP - the (resetting) function
(defun resetting ()
   (setvar "CMDECHO" ocmd)
   (setvar "BLIPMODE" oblip)
); end of resetting
```

Putting all of this together, and making local variables, we have:

```
XPFACE.LSP   - the complete program
;;;  XPFACE.LSP A program by Tony Hotchkiss
;;;  XPFACE CREATES A PFACE MESH
;;;  FROM A DATA FILE.
;;;  THE FORMAT OF THE DATA FILE IS:
;;;   n     (the total number of data points)
;;;  x  y  z   (three fields of 14 characters)
;*************************************
(defun setting ()
   (setq ocmd (getvar "CMDECHO"))
   (setvar "CMDECHO" 0)
   (setq oblip (getvar "BLIPMODE"))
   (setvar "BLIPMODE" 0)
); end of setting
(defun input ()
   (setq fname (getstring "\nTXT file name: "))
); end of input
(defun calc (/ f1 numlines p px py pz pp i)
   (setq ptlst nil)
   (setq fname (strcat fname ".txt"))
   (setq f1 (open fname "r"))
   (if f1   ;test if file exists
      (progn  ;then read and format the points
         (prompt "\nReading the data file....")
         (setq numlines (read (read-line f1)))
         (repeat numlines (setq p (read-line f1))
            (setq px (read (substr p 1 14))
                  py (read (substr p 15 14))
                  pz (read (substr p 29 14))
                  pp (list px py pz)
                  ptlst (append ptlst (list pp))
             ) ; setq
         ) ;end of repeat for list of vertices
         (close f1)
         (setq vxlst nil)
         (setq vlst nil)
         (setq i 1)
         (repeat (/ numlines 4)
            (setq vlst (list i (+ i 1) (+ i 2) (+ i 3)))
            (setq i (+ i 4))
            (setq vxlst (append vxlst (list vlst)))
         ); repeat
      ) ;progn
      (princ  (strcat "Cannot open file " fname))
   ) ; if
   (princ)
) ; end of defun calc
(defun output ()
   (command "pface")
   (foreach p ptlst (command p))
   (command "")
   (while (setq vlst (car vxlst))
      (while (car vlst) (command (car vlst))
         (setq vlst (cdr vlst))
      ); while
      (command "")
      (setq vxlst (cdr vxlst))
   ); while
   (command "")
);  defun
(defun resetting ()
   (setvar "CMDECHO" ocmd)
   (setvar "BLIPMODE" oblip)
); end of resetting
(defun C:XPFACE ()
   (setting)
   (input)
   (calc)
   (output)
   (resetting)
   (princ)
); end of defun C:XPFACE
```

Further Program Development - Why is it so slow in execution?

I tested this program with data files containing thousands of points, and it took about twenty minutes to generate a somewhat complicated mesh. In fact, without the blipmode and command echo control, the time was nearly doubled again! This should give us a clue that there may be more effective ways to approach the problem. The traditional LISP function **(foreach)** is a very handy tool for dealing with lists of points, and therefore it is standard procedure in AutoLISP programs that process points to collect the points into a list as we have done here, and then to use **(foreach)** to deliver the points to the appropriate AutoCAD command.

FASTFACE.LSP shows an alternative to the use of (foreach) with a list of points. It is good practice in any situation to handle data as little as possible, and if you can read in the points from the data file and process them immediately, you will save a lot of memory and your programs will run faster. The approach in FASTFACE.LSP is based on a suggestion of a computer science major at UW-Madison, Troy Jacobson, who asked why we were assembling the points into a huge list and then processing the points for a second time by extracting them from the list.

The result is that FASTFACE.LSP has a very short (calc) section, limited to attaching the .TXT file extension to the data file name. All of the work is now done in the (output) section of the program.

```
FASTFACE.LSP   - a quick pface mesh
;;;  FASTFACE.LSP A program by Tony Hotchkiss
;;;  FASTFACE CREATES A PFACE MESH
;;;  FROM A DATA FILE.
;;;  THE FORMAT OF THE DATA FILE IS:
;;;   n     (the total number of data points)
;;;  x  y  z   (three fields of 14 characters)
;************************************
(defun setting ()
   (setq ocmd (getvar "CMDECHO"))
   (setvar "CMDECHO" 0)
   (setq oblip (getvar "BLIPMODE"))
   (setvar "BLIPMODE" 0)
   (setq i 1)
   ); end of setting
   (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
            (setq numlines (read (read-line f1)))
            (command "pface")
            (repeat numlines ; 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
            (command "")  ; end of vertex data
            (repeat (/ numlines 4) ; face definition
               (command i (+ i 1) (+ i 2) (+ i 3) "")
               (setq i (+ i 4))
               (princ)
            ); end of repeat for face data
            (command ""); end of faces
         );progn
         (princ  (strcat "Cannot open file " fname))
      ); if
      (close f1)
   );  end of defun output
   (defun resetting ()
      (setvar "CMDECHO" ocmd)
      (setvar "BLIPMODE" oblip)
   ); end of resetting
   (defun C:FASTFACE ()
      (setting)
      (input)
      (calc)
      (output)
      (resetting)
      (princ)
   ); end of defun C:FASTFACE
```

After opening the data file for reading, the **(output)** function reads the first line containing the number of points to follow, and immediately starts the AutoCAD 'pface' command. Each vertex point is read in and processed in turn in a repeat loop. The assembled points pp are supplied to the 'pface' by the **(command pp)** statement, which is followed by the **(princ)** inside the repeat loop. The **(princ)** is required to inhibit any other output from the repeat loop. A <return> is then given to signal the end of the vertex points, and another repeat loop supplies the face definition numbers in groups of 4, followed by a <return>, as in **(command i (+ i 1) (+ i 2) (+ i 3) "")**. A **(princ)** is again necessary at the end of the repeat loop to inhibit any unwanted input to the pface command.

The data input is terminated by a final <return> in the form of **(command "")** as shown, and the data file is closed.

Error handling

The programs we have listed here use an (if) statement to determine that the input data file has been successfully opened. The 'else' part of these statements print out a message that the file could not be opened, but our programs do not end there, because other modules are attempted to be executed by the last defined function which invokes all of the other sections of the program in turn.

For this reason, if you type in a non-existent file name, the programs print out the appropriate message, but they also terminate with errors because we have not ensured that no further processing is attempted.

In the case of the FASTFACE.LSP program a suitable remedy is to assign an 'abnormal end' variable after the message as in:

```
            (command ""); end of faces
         );progn
         (progn
         (princ  (strcat "Cannot open file " fname))
         (setq abend 1)
         ) ; progn
      ); if
      (close f1)
   );  end of defun output
```

The abend variable is then used as a test in the final executing function as follows:

```
   (defun C:FASTFACE ()
      (setting)
      (input)
      (calc)
      (if (not abend)
         (output)
      )
      (resetting)
      (princ)
   ); end of defun C:FASTFACE
```

You could initialize the value of abend in the **(setting)** section:

```
   (defun setting ()
      (setq ocmd (getvar "CMDECHO"))
      (setvar "CMDECHO" 0)
      (setq oblip (getvar "BLIPMODE"))
      (setvar "BLIPMODE" 0)
      (setq i 1)
      (setq abend nil)
   ); end of setting
```

This kind of error tracking lets the program terminate normally after resetting the system variables, instead of dumping the remainder of your LISP code to the screen.

Another error handling procedure is the *error* function of AutoLISP, which allows you to define an error message and procedure for what to do in the case of a user cancel of your program.

The standard error function is **(*error* *string*)**, and is called automatically by AutoLISP. The string argument receives a message "Function cancelled" when an executing program is cancelled with a control-C. For other error messages, you can refer to the AutoLISP programmer's Reference, from Autodesk. The appropriate message is automatically supplied as a string argument to the *error* function.

You can redefine the error function as follows. In the case of the FASTFACE program, we define the following function:

```
   (defun err (s)
    (if (= s "Function cancelled")
       (princ "\nYou cancelled the function.")
       (progn
          (princ "\nFASTFACE Error: ")
          (princ s)
        ) ; end of progn
    ) ; end of if
    (resetting)
    (princ "\nSYSVARS have been reset")
    (princ)
 ) ; end of error function
```

This error function shows how you can replace any of the standard messages with your own message. Remember, you will later redefine the standard ***error*** function with this one, so the message argument 's' will be supplied automatically by AutoLISP. The standard message is echoed to your terminal by using **(princ s)** as shown in the err function. Here the 'Function cancelled' message has been replaced by the message 'You cancelled the function.', and all other error messages are preceded by 'FASTFACE Error'. The function then calls **(resetting)** and the message 'SYSVARS have been reset' is printed.

The next step is to save the old error function to a variable, and then redefine ***error***. This can be included in the **(setting)** function:

```
 (defun setting ()
    (setq oerr *error*)
    (setq *error* err)
    (setq ocmd (getvar "CMDECHO"))
    (setvar "CMDECHO" 0)
    (setq oblip (getvar "BLIPMODE"))
    (setvar "BLIPMODE" 0)
    (setq i 1)
 ); end of setting
```

Finally, the **(resetting)** function resets the standard *error* handler:

```
 (defun resetting ()
    (setvar "CMDECHO" ocmd)
    (setvar "BLIPMODE" oblip)
    (setq *error* oerr)
 ); end of resetting
```

Now, if you enter a non-existent file name you do not need any other special precaution such as the 'abend' variable that we used earlier. The ending function can be the way it was originally:

```
 (defun C:FASTFACE ()
    (setting)
    (input)
    (calc)
    (output)
    (resetting)
    (princ)
 ); end of defun C:FASTFACE
```

This produces the following dialogue for an unknown file name:

```
 Command: fastface

 TXT file name:  hhh

 Cannot open file hhh.txt
 FASTFACE Error:  bad argument type
 SYSVARS have been reset

 Command:
```

The error message 'bad argument type' is caused by the program continuing to execute with no input data, but the main message you need to see is the fact that the system variables have been reset. In fact, you could suppress the 'bad argument type' message by using **(if (/= s "bad argument type") (princ s))** instead of **(princ s)** in the above, after the program is running otherwise satisfactorily. The message 'FASTFACE Error' would then appear to be caused by the file not being opened.

Homework:

1. 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?

2. 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.

3. 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.

In lesson two of this series, "System variables and interfacing with AutoCAD", you will learn some useful system variables for AutoLISP programming. The rest of the lesson then deals with the Drawing Exchange Format (DXF), and you will learn about AutoCAD tables and association lists. This is your introduction to delving into the data structure of AutoCAD, and it will prepare you for manipulating objects in later lessons. The lesson will end with some homework problems to test your understanding of the text.

System Variables and Interfacing with AutoCAD

Lesson Two in the CADalyst/University of Wisconsin Advanced AutoLISP Programming course covers some useful system variables for AutoLISP and the Drawing Exchange Format (DXF).

by
Anthony Hotchkiss, Ph.D, P.Eng.

About this course.

Advanced AutoLISP Programming is an eight-lesson independent study course, originally published in monthly installments in CADalyst magazine. Before you begin this course, it is assumed that you have reached a basic level of AutoLISP programming, equivalent to completing the preceding 'Introduction to AutoLISP Programming' course which was originally published in CADalyst, and is now available as an independent study course from the University of Wisconsin-Madison. On successfully completing an intermediate and a final mail-in project, registered students will receive certificates of completion and 5.0 CEU's (continuing education units).

For further information, contact:

Steve Bialek
Engineering Professional Development
University of Wisconsin-Madison
432 North Lake Street
Madison, Wisconsin 53706
(608) 262-1735

LESSON TWO - SYSTEM VARIABLES AND INTERFACING WITH AUTOCAD

In this month's lesson you will learn some useful system variables for AutoLISP programming. The rest of the lesson deals with the Drawing Exchange Format (DXF), and you will learn about AutoCAD tables and association lists. This is your introduction to delving into the data structure of AutoCAD, and it will prepare you for manipulating objects in later lessons. The lesson will end with some homework problems to test your understanding of the text.

SOME USEFUL SYSTEM VARIABLES FOR AUTOLISP

The system variables that you will use in your programs will obviously depend upon what your program is supposed to do. If a considerable amount of drawing or geometry is created, then the BLIPMODE will be an important consideration. We saw in lesson one of this series that blipmarks can take a while to be displayed, and can double the time taken to draw surface meshes.

The system variables that you might need to consider for any of your programs are:

BLIPMODE; 0 = OFF, 1 = ON

CMDECHO; 0 = Command function dialogue is not shown in the command/prompt area, 1 = command prompts are displayed.

MENUECHO; Useful in menu macro operation of lisp functions. 0 = Display all menu macro items and command prompts. This is the default setting. 1 = Menu macro text is not displayed, but command prompts are displayed. 2 = Menu macro text and commands are suppressed. 4 = the special character ^P which is used to toggle MENUECHO on and off is disabled. In menu macros, it is common to use ^P before and after a set of LISP statements. ^P toggles MENUECHO between the values 0 and 1. 8 = a debugging aid for DIESEL macros in release 12.

EXPERT; This affects some system command prompts, and can affect the responses to COMMAND functions in your LISP programs. The prompt controls are 0 = Display all AutoCAD prompts (default). 1 = Suppresses "About to regen.." and "Really...turn current layer off?" 2 = Suppresses all of the preceding prompts, and the Block & Wblock redesignating prompts. 3 = Suppresses all of the preceding prompts, and LINETYPE prompts for existing or already loaded linetypes. 4 = Suppresses all of the preceding prompts, and UCS save and VPORTS save prompts if the saved name already exists. 5 = Suppresses all of the preceding prompts, and DIM save and DIM Override if the Dimstyle name already exists.

HIGHLIGHT; 1 = When objects are selected they are highlighted, 0 = highlighting is suppressed. For release 12, if objects are selected with grips, highlight is not used.

OSMODE; Object snap mode control uses the sum of the bit-codes to set object snap (more than one mode may be requested): 0 = NONE, 1 = END, 2 = MID, 4 = CEN, 8 = NODE, 16 = QUA, 32 = INT, 64 = INS, 128 = PER, 256 = TAN, 512 = NEA, 1024 = QUI

TEXTEVAL; Used for text strings and attribute values. 0 = All responses to prompts are taken literally. 1 = Text starting with "(" or "!" is evaluated as an AutoLISP expression.

THE DRAWING EXCHANGE FILE (DXF) FORMAT

The DXF file is an ASCII (American Standard Code for Information Interchange) text file, in other words, human readable, but with none of the special characters for bold, italic, new paragraph etc. that are normally present in word processing documents. AutoCAD objects, such as lines, arcs, circles etc. are listed in a DXF file with the command DXFOUT. The format of the file is a series of numbers that are associated with objects and with the parameters of those objects, such as center point and radius for circles, and line type, layer and color. The numbers are called 'group codes', and are used in the DXF format as shown in table 1

Table 1 DXF group codes

Code range	**Data type**	**Typical uses**
-3, -2, -1	Various	Entity names,Extended entity flag
0 - 9	String	Entity type, named objects, etc.
10 - 37	Real	Coordinates

38, 39	Real	Elevation and Thickness
40 - 59	Real	Text height, scale factors, angles
60 - 79	Integer	Color, flags, counts, modes
210 - 230	Real	X, Y, Z components of extrusion
999	String	Comments
1000 +	Various	Extended entity data

As an example, the DXF codes for a line entity are:

Code	**Description**
-1	Entity name (example: 60000057)
0	Entity type ("LINE")
5	Entity handle (a hexadecimal number)
6	Line type (example: "BYLAYER")
8	Layer name (example: "OBJECTS")
10	Start X coordinate (example: 2.382041)
11	End X coordinate (example: 8.081538)
20	Start Y coordinate (example: 5.967391)
21	End Y coordinate (example: 3.75)
30	Start Z coordinate (example: 0.0)
31	End Z coordinate (example: 0.0)
62	Color number (example: 3)
67	Model space/Paper space flag (example: 0)
210	X component of extrusion direction (example: 0.0)
220	Y component of extrusion direction (example: 0.0)
230	Z component of extrusion direction (example: 1.0)

Not all of the group codes can be used for any entity (a line does not have a code for a 'center point'), and some of the 'default' group codes are not included in a 'DXFOUT' listing. The DXFOUT command can produce a file that will contain very much more data than we have shown here, as we are concerned with entities only. The DXF file also contains information about system variables, tables, and blocks. (The entities section of the DXF file can be output if the DXFOUT Entities option is used)

As a simple exercise, make a drawing that only contains a single line, and create DXF files with DXFOUT, with and without the 'Entities' option (give the files different file names) and examine the files with your text editor. If you do this exercise with menu selections, the sequence will be:

From the screen menu, select UTILITY, then from the Utility menu select DXF/DXB, then select DXFOUT. (You can short-cut all of the above by entering the command DXFOUT). The next prompt will be to enter a file name which will contain the DXF list. Release 12 displays a dialogue box for this purpose, as shown in figure 2-1, so you can easily choose disk drives and/or directories. After you enter the file name, the choices in the command/prompt area are:

Enter decimal places of accuracy (0 to 16)/Entities/Binary <6>:

Here, for the complete DXF file, simply <return> to accept the default number of decimal places. For the Entities choice, enter 'e' and you will be prompted to 'select objects' in the usual way.

For a single line, the full DXF file contains about 800 lines for release 12, most of which refer to the system variables. The 'entities only' DXF file is as follows:

```
0
SECTION
2
ENTITIES
0
LINE
8
OBJECTS
10
2.382041
20
5.967391
30
0.0
11
8.081538
21
3.75
31
0.0
0
ENDSEC
0
EOF
```

Here, the DXF group code is indented, and the value associated with the group code is written on a separate line following the code.

AUTOLISP ASSOCIATION LISTS

Association lists are groups of codes that are associated with entities and entity parameters in the same way that the DXF codes are associated. The general format of the association list is the 'dotted pair'. For instance, the layer name of an entity is associated with the group code 8, conforming to DXF format, and the 'dotted pair' is the list **(8 . "OBJECTS")** (following the example of the objects layer of the previous DXF listing). Note that the dotted pair is separated by a dot as shown.

Association lists are very similar to DXF codes. The only differences are that the coordinate data is associated with a single code, instead of with a code for each of the X, Y and Z coordinates. So we have the group codes 10 and 11 associated with X, Y and Z, instead of just X for the start and end of a line. The extrusion coordinates are also associated with the single code 210, instead of with 210, 220 and 230. In the cases where more than one item is associated with a group code, the 'dot' is missing from the list, thus the end point of a line would be given by the list **(11 8.08154 3.75 0.0)** in our line example, and the extrusion direction is expressed as **(210 0.0 0.0 1.0)**.

You can see the association lists for the last entity created, by entering the following at the command/prompt: **(setq elist (entget (entlast)))**

There are two new functions here, (entlast) and (entget). (entlast) returns the name of the last non-deleted entity created, and (entget) returns the association list for the named entity. Your screen will look like the following:

```
  Command:  (setq elist (entget (entlast)))
  ((-1 . <Entity name: 60000022>) (0 . "LINE") (8 . "OBJECTS") (10
2.38204 5.96739
    0.0) (11 8.08154 3.75 0.0) (210 0.0 0.0 1.0))
```

Now, because the DXF list is assigned to the variable named 'elist', you can make this list a little less confusing by writing each dotted pair on a separate line with the (foreach) function, so the result will be:

```
  Command:  (foreach item elist (print item))

  (-1 . <Entity name: 60000022>)
  (0 . "LINE")
  (8 . "OBJECTS")
  (10 2.38204 5.96739 0.0)
  (11 8.08154 3.75 0.0)
  (210 0.0 0.0 1.0) (210 0.0 0.0 1.0)
```

Remember that (print) places a new line before, and a space after the item to be printed, and (foreach) returns the result of the last expression, so the last line is repeated with the extrusion vector associated with group code 210.

TABLE ASSOCIATION LISTS

In addition to entities, AutoCAD also has TABLES that have association lists. The TABLES are:

VPORT, LTYPE, LAYER, BLOCK, STYLE, VIEW, DIMSTYLE, UCS, and APPID

The DIMSTYLE and APPID tables were introduced in release 11, and are retained in release 12 of AutoCAD. DIMSTYLE contains group codes associated with dimension variables, and APPID refers to extended entity data, which is associated with an 'application identification' name, hence the table name APPID. We will see more of this later in lessons dealing with extended entity data.

The information and group codes vary for tables as they do for entities, for example the DXF group codes for the LAYER table are as follows:

```
  Table association list for the LAYER table:

  (0 . "LAYER)                    Symbol type
  (2 . "OBJECTS")            Symbol name
  (70 . 0)                              Flags:
                                                0 - Thawed, not
created or used
                                                1 - Frozen, not
created or used
                                                64 - Thawed &
created or used
                                                65 - Frozen &
created or used
  (62 . 2)                              Color number, (or negative)
  (6 . "CONTINUOUS")    Line type name
```

There are two functions that return DXF lists for tables:

```
  (tblnext  table-name  rewind-option)
  (tblsearch  table-name table-entry setnext-option)
```

The 'table-name' for both of these functions is one of the valid TABLES of AutoCAD, such as "LAYER" or "BLOCK". There will normally be many entries in a table, and each entry contains the appropriate DXF group codes and their associated items, such as in the layer table example shown above for the 'objects' layer. Information about any particular layer or table-entry for other types of table is not directly accessible using (tblnext). Instead, you can list the first entry in a table by using the 'rewind' flag of (tblnext), thus, at the command prompt:

```
  Command: (tblnext "layer" t)
  ((0 . "LAYER")  (2 . "0") (70 . 0) (62 . 7) (6 . "CONTINUOUS"))
```

The symbol 't' is used here to denote a non-nil value for the rewind-option, and that forces the table to be rewound so that the first entry of the table (in this case, the layer '0') is listed.

Subsequent uses of (tblnext) without the rewind-option will return similar lists for the next entry in the table, so if you have a layer named '1', that would be the next layer to be listed as in:

```
  Command: (tblnext "layer")
  ((0 . "LAYER")  (2 . "1") (70 . 64) (62 . 1) (6 . "CONTINUOUS"))
```

It is possible to locate a table entry by name using (tblsearch), thus:

```
  Command: (tblsearch "layer" "noplot"  t)
  ((0 . "LAYER")  (2 . "NOPLOT") (70 . 64) (62 . 2) (6 .
"CONTINUOUS"))
```

If the optional 'setnext-option' argument is supplied and is non-nil, as in this example, then the table entry is set as a 'current' entry for the purpose of the (tblnext) function. In this case, execution of a (tblnext) will list the next table entry following that of the (tblsearch) containing the setnext-option argument.

You can test this behavior by entering successive (tblnext) and (tblsearch) functions both with and without using the rewind and setnext arguments.

To make your table lists more readable, you could use (foreach) as before, so your command prompt area would look like:

```
  Command: (progn (foreach x (tblnext "block" t) (print x)) (princ))

  (0 . "BLOCK")
  (2 . "*D0")
  (70 . 65)
  (10 0.0 0.0 0.0)
  (-2 . <Entity name: 4000001e>)

  Command:
```

This time, we listed the first item of the block table. The block listed here is a dimension, so it has the name "*Dn" where n is a number starting from 0. The block table was rewound, using the argument 't', so the first dimension block is named "*D0". The asterisk is a convention that is used for 'anonymous' blocks in AutoCAD. We will see more of that in later lessons dealing with creating and using anonymous blocks. In the block table, the -2 group code is associated with the name of the first entity in the block. (progn) was used here so that (princ) could be included to suppress printing of the last expression, and giving a 'cleaner' listing as shown.

Homework:

1. 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.

2. 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.

In lesson three of this series, "Entity Selection for AutoLISP", you will learn how to select entities and to use filters to separate out subdivisions of selected groups of objects. You will also learn how to use selection lists in preparation for manipulating the data structure of AutoCAD.

Entity Selection for AutoLISP

Lesson Three in the CADalyst/University of Wisconsin Advanced AutoLISP Programming course shows you how to select entities in preparation for data base manipulation.

by
Anthony Hotchkiss, Ph.D, P.Eng.

About this course.

Advanced AutoLISP Programming is an eight-lesson independent study course, originally published in monthly installments in CADalyst magazine. Before you begin this course, it is assumed that you have reached a basic level of AutoLISP programming, equivalent to completing the preceding 'Introduction to AutoLISP Programming' course which was originally published in CADalyst, and is now available as an independent study course from the University of Wisconsin-Madison. On successfully completing an intermediate and a final mail-in project, registered students will receive certificates of completion and 5.0 CEU's (continuing education units).

For further information, contact:

Steve Bialek
Engineering Professional Development
University of Wisconsin-Madison
432 North Lake Street
Madison, Wisconsin 53706
(608) 262-1735

LESSON THREE - ENTITY SELECTION FOR AUTOLISP

In this lesson you will learn how to select entities and to use filters to separate out subdivisions of selected groups of objects. The AutoLISP functions for selecting and filtering are explained with examples that show you how to use them. You will also learn how to use selection lists in preparation for manipulating the data structure of AutoCAD. As usual the homework assignment will test your understanding of the text.

ENTITY SELECTION FUNCTIONS

In conventional AutoCAD design/drafting, entities are selected with the cursor, either singly, or with a window or crossing box. It is also possible to select the 'previously selected set' of entities, and this implies that there is a mechanism in AutoCAD for remembering what that set of entities is. AutoLISP uses this fact to record selected sets, using functions that have the prefix 'ss' for 'Selection Set'.

The ss-functions are ssget, ssadd, ssdel, ssname, sslength, and ssmemb:

(ssget) is the most general way of selecting entities, and the function with no arguments prompts the user to select entities, and collects those entities into a selection set, so that, for instance,

(setq ss1 (ssget))

prompts the user to select objects in the usual manner, using window, crossing, single, etc., then saves the selected set of entities and assigns the name of the set to 'ss1'. It would then be possible to perform some editing on that selected set, as we will discover later.

(ssget) has a number of optional arguments, such as the 'mode' of selecting objects. You can obtain the previously selected set with **(ssget "P")**, and you select the last entity with **(ssget "L")**. To make a single entity selection, you can simply supply a point in any of the standard forms, such as **(ssget pt1)**, or **(ssget '(2.5 3.625))**. The entity that passes through the point will be the selected set. The last example shows another optional argument in the form of a single point which is supplied with no 'mode' argument.

Other selection modes, such as "W" or "C", for window or crossing boxes, require corresponding pairs of points to be supplied to define the selection box, as in:

(ssget "W" pt1 pt2), or **(ssget "C" '(2.5 3.25) '(4.75 5))**

To put these commands into practice, let's try a simple example. First, insert a block into your AutoCAD drawing, then set your view and UCS parallel to the view, as shown in figure 1, in which I have inserted a block called "BRKTWIRE".

Now pick some points ready to use as a crossing or window box with the following sequence:

```
Command:  (setq pt1 (getpoint "\npt1: "))

pt1:  (-3.64302  4.54373  0.0)

Command:  (setq pt2 (getpoint "\npt2: "))

pt2:  (-1.16044  6.57232  0.0)
```

Your screen will resemble figure 1, with your points crossing the object.

Try the next sequence, and you will see that a crossing mode gives a selection set, but a window mode returns NIL, because your points describe a box that does not completely include the object:

```
Command:  (setq ss1 (ssget  "C"  pt1  pt2))
<Selection set: 1>

Command:  (setq ss2 (ssget "W" pt1 pt2))
nil
```

You can make use of this in your programs for checking whether something was selected or not, as in this extract from a program which is intended to trim all objects inside a crossing box:

```
Partial CTRIM.LSP program - use of ssget.
    (setq ss1 nil)
    (while (=  ss1  nil)
       (setq ll (getpoint "\nLower left corner: "))
       (setq ur (getcorner "\nUpper right corner. " ll))
       (command "erase" "w" ll ur "") ; erase with window
       (setq ss1 (ssget "C" ll ur))
```

```
        (if  (= ss1 nil) (prompt "\nNo objects to trim. "))
     ); end of while ...
```

In this program segment, we want to clear out all of the objects from a box, defined with the lower left and upper right points. All objects that are entirely inside the box are erased, and any objects that cross the box will be included in the selection set ss1. If no objects cross the box, the selection set ss1 will be NIL, and the user is requested to select another box in the while loop.

If there are objects that cross the box, they should be trimmed, and that is the main purpose of the program. We will return to this example later to illustrate other selection set functions.

We showed in the previous lesson (2) that you can get a DXF list of an entity with (entget) if you supply the name of the entity as an argument to (entget). The ss function that returns the name of entities is (ssname), which requires arguments to specify the selection set and the number of the required item in the selection set. It is of course possible to select many items in a selection set, and the particular item whose DXF list is required is identified by an index number that starts with zero, so the third item in the selection set will be identified by the number 2. In the case of our block, that will appear as a single item, so the number 0 will be appropriate, as in **(ssname ss1 0)**, which will return the name of the first item of the selection set 'ss1'. Counting items from zero up is confusing, but computer scientists have learned to know and love this crazy system, which appears to them to be perfectly logical.

You can use (foreach) to print each item of the list, as we did in lesson 2, in order to get a clearer DXF listing. Toggle your text screens, and type: (foreach item (entget (ssname ss1 0)) (print item)).

Your screen will then look like:

```
Command:  (foreach item (entget (ssname ss1 0)) (print item))

(-1 . <Entity name:  60000022>)
(0 . "INSERT")

(8 . "1")
(2 . "BRKTWIRE")
(10 0.0 0.0 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)  (210 0.0 0.0 1.0)

Command:
```

The DXF codes for each type of entity are given in the User Reference for AutoCAD release 11 and earlier, and in the AutoCAD Customization Manual, chapter 11, for release 12. When a block is inserted into a drawing , an INSERT entity is created, so the DXF listing will show INSERT to be the entity type. The other DXF codes for inserts are:

10; insertion point
41, 42 and 43; X, Y, and Z scale factors
50; rotation angle
70 and 71; number of columns and rows for multiple inserts
44 and 45; column and row spacing for multiple inserts

The other DXF codes are common to all objects, such as -1, the entity name, 8, the layer name, and 210, the orientation vector.

If the block has attributes, a flag code 66 indicates that there are attributes, and the attribute entities will then follow the Insert, but more of that later.

Selection set modes and other arguments.

The complete set of optional arguments for the release 12 ssget function is:

(ssget *mode point1 point2 point-list filter-list*)

We have seen that point1 and point2 are used together with Window and Crossing modes, and point1 is used on its own with no mode in order to select an entity which passes through the point. In release 12, there are some new selection modes called Polygons and Fences, and there is a method of selecting objects which is similar to using the Select command in order to select entities before editing them, thus creating an Implied selection set. All of these modes can be used with ssget. The Polygon and Fence modes need a list of points to define the fence or polygon, and that list is provided by the new argument, point-list.

The modes of ssget are:

(ssget "W" point1 point2) Window mode with two corner points
(ssget "C" point1 point2) Crossing mode with two corner points
(ssget "P") The most recent previously selected set mode
(ssget "I") Implied selection set mode
(ssget "L") The last entity created and not frozen
(ssget "X") Selects all of the objects in the drawing
(ssget "F" point-list) Fence mode, uses a point list to define the fence
(ssget "WP" point-list) Window Polygon mode with a list of polygon points
(ssget "CP" point-list) Crossing Polygon mode with a point list to define the polygon

The modes "I", "F", "WP", and "CP" are all release 12 modes

Filter lists for selection sets

It is possible to constrain the selected set to objects that meet certain criteria, such as belonging to a particular layer, or entity type, by using a filter list that takes the form of the DXF associated pairs. A filter list that will select only the red circles on layer 1 would take the form:

((0 . "CIRCLE") (8 . "1") (62 . 1))

The DXF codes are 0 for entity type, 8 for layer, and 62 for color. The filter list is contained in a set of parentheses, hence the double open parentheses to start this list. A typical use in a program is:

(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))))

Note that if the object is colored red because it is assigned "bylayer", it will not be selected with this filter. It would have to be specifically given the color assignment as "red", and not as "bylayer".

Filter lists in versions of AutoCAD prior to release 12 are quite restricted in their uses of the DXF group codes. The available codes for release 11 are 0, 2, 3, 6, 7, 8, 38, 39, 62, 66, and 210. Release 12 allows all codes except -1, 5, and codes greater than 1000 (extended entity data). It is possible to use -3 to filter those entities having extended entity data with a particular registered application name. Extended entity data is a subject for future lessons in this series.

In release 12, it is also possible to use the relational operators to filter out, for instance, all circles with radii (code 40) greater than some value, as in:

(setq ss4 (ssget "X" '((0 . "CIRCLE") (-4 . ">") (40 . 2.5))))

The new -4 group code for release 12 indicates that a relational operator is to be used as a test for the associated pair that follows the -4 code.

Filter lists can be used to erase all objects from a particular layer, as in this example in which temporary objects are created on a 'noplot' layer. The objects are erased with:

```
(setq ss1 (ssget "X" '((8 . "NOPLOT"))))
(if  ss1
    (command "erase" ss1 "")
    (princ "\nNo objects found on NOPLOT layer")
); end of if
```

Note that the filter list is still contained within parentheses, even though only one dotted pair is given. The strings supplied for the layer names, entity types, etc. may be upper or lower case.

As well as the relational functions for filter lists, release 12 also allows the logical functions such as (and), (or), which are used in groups surrounded by the < > brackets (grouping operators) associated with group code -4 as in the following examples:

```
(setq ss2 (ssget "X" (list (cons 8 "1")
        (cons -4  "<or") (cons 0 "ARC") (cons 0 "CIRCLE") (cons -4
"or>")) ))
```

assigns to ss2 all arcs or circles on layer 1. Note the use of the new 'grouping operators' < and > that are used with the logical functions. The logical functions can be written in upper or lower case, and can be nested as shown in the next example:

```
(setq ss3
   (ssget "C" pt1 pt2  '((-4 . "<OR")
      (-4 . "<AND") (0 . "CIRCLE") (62 . 3) (-4 . "AND>")
      (0 . "ARC") (-4 "OR>))
  ); ssget
); setq
```

This example assigns to ss3 objects that are in or cross the window defined by the points pt1 and pt2, that are arcs or green circles. The group code 62 is the 'color' code, and the color numbers for the first seven colors are 1, red; 2, yellow; 3, green; 4, cyan; 5, blue; 6, magenta; and 7, white.

This example also shows that filter lists can be used with any of the selection set modes, not just with the "X" mode.

(ssadd *ename sset*) returns a new selection set depending on the optional arguments ename and sset, as in these examples:

(ssadd) with no arguments creates a null selection set that has no items in it, and is used to initialize a selection set, as in **(setq ss1 (ssadd))**.

(ssadd ent1) returns a selection set that contains only the entity named ent1, and is used in:

```
(setq  ent1 (entget (entlast)))
(setq ss2 (ssadd ent1))
```

which assigns to ss2 the selection set containing the name of the last entity created. With this form, the selection set is effectively initialized to contain the single entity ent1, so you would not need to initialize ss2 with **(setq ss2 (ssadd))** before making this assignment.

If both the entity name and the existing selection set are supplied as arguments, then **(ssadd ent2 ss2)** will add the entity named ent2 to the selection set ss2, and will return the newly formed selection set.

(ssdel *ename sset*) deletes the entity named 'ename' from the selection set 'sset', and returns the newly formed selection set. Note that the entity is not erased from the drawing by this command, but is simply removed from the selection set. (ssadd) and (ssdel) are equivalent to the 'ADD' and 'REMOVE' options in conventional AutoCAD entity selection.

(ssname *sset n*) returns the nth entity name of the selection set 'sset', as in our earlier example in which the nth item is numbered from zero.

(sslength *sset*) returns the number of items in the selection set 'sset'. This is used to count numbers of entities, and to determine the number of loops that may be required to perform editing operations on each item of a selection set as in this extract from the CTRIM.LSP program

```
Partial CTRIM.LSP program - use of sslength.
.
.
(setq len (sslength  ss1))
(setq index 0)
(repeat  len
   (setq  dxflst  (entget  (ssname ss1 index)))
    .
    .
   (setq index (1+ index))
); repeat
```

in this extract from CTRIM.LSP, the selected objects are operated on in a repeat loop. There are many operations to perform, including identifying the type of each entity, and then trimming with the appropriate pick points for each entity, although these are not shown here. Each entity is dealt with in turn before the index counter is incremented so that the next selected object can be trimmed. In later lessons you will see how to extract data from the DXF lists returned by entget.

(ssmemb *ename sset*) checks whether the entity named ename is part of the selected set 'sset', and returns the name of the entity if it is a member of the set, otherwise the function returns NIL.

Homework:

1. 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.

2. 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.

In lesson four of this series, "AutoLISP Data Base Manipulations", you will learn about the entity functions and how to use association and substitution to make changes directly to the AutoCAD data entity data structure. Programming examples will be used to show you how to apply the functions, and homework assignments will test your understanding of the lesson.

AutoLISP Data Base Manipulations

Lesson Four in the CADalyst/University of Wisconsin Advanced AutoLISP Programming course shows you use entity functions for data base manipulation.

by
Anthony Hotchkiss, Ph.D, P.Eng.

About this course.

Advanced AutoLISP Programming is an eight-lesson independent study course, originally published in monthly installments in CADalyst magazine. Before you begin this course, it is assumed that you have reached a basic level of AutoLISP programming, equivalent to completing the preceding 'Introduction to AutoLISP Programming' course which was originally published in CADalyst, and is now available as an independent study course from the University of Wisconsin-Madison. On successfully completing an intermediate and a final mail-in project, registered students will receive certificates of completion and 5.0 CEU's (continuing education units).

For further information, contact:

Steve Bialek
Engineering Professional Development
University of Wisconsin-Madison
432 North Lake Street
Madison, Wisconsin 53706
(608) 262-1735

LESSON FOUR - AUTOLISP DATA BASE MANIPULATIONS

In this lesson you will learn about the entity functions and how to use association and substitution to make changes directly to the AutoCAD entity data structure. Programming examples will be used to show you how to apply the functions, and homework assignments will test your understanding of the lesson.

ENTITY FUNCTIONS

We have already used the entity functions, (entget), for retrieving the dxf list of a named entity, and (entlast), for getting the name of the last entity created (and not deleted). in this lesson, we add the functions (entsel), (entdel), and (entmod). We will also cover the functions (assoc) and (subst), which are used to return dxf group associated pairs, and substitute those pairs for new pairs. With all of these functions, you will have the power to make automatic changes to the 'simple' AutoCAD entities. We use the term 'simple' here because the next lesson (five) deals with manipulating blocks and polylines, which contain more than one entity, and are called 'complex' entities.

(entsel *"prompt"*), has an optional prompt, just like the (get..) functions, and it allows the user to select any entity from the screen. The name of the selected entity is returned along with the point of selection. If the optional prompt is not supplied, (entsel) gives the default prompt, "Select object:"

Try entering **(entsel "\nPick any object)** at your command/prompt area, but make sure there is at least one object to select on your AutoCAD screen. You should see a list that looks like **(<Entity name: 60000022> (4.5831 6.3788 0.0))**, which is the name of the selected entity plus the list of the point at which the entity was selected.

To extract the name only, enter **(setq ent1 (car (entsel)))**. This time you will see the default prompt for (entsel), and you will also see the name of the entity only, which is returned and assigned to the variable 'ent1'

(entdel *ename*) deletes the entity whose name is supplied as the argument 'ename'. After an entity has been deleted, (entdel) will undelete the entity to restore it to the data base. This is not the same as 'undo', but is similar to 'OOPS'. Now, enter **(entdel ent1)**, and you will see the object that you picked earlier disappear. Enter **(entdel ent1)** again to see the object return to your screen.

(assoc *item list*) returns the value that is associated with 'item', together with 'item' in a list of pairs of associated items. A common usage of (assoc) is the example of the association list returned by (entget). For instance, if you select a line entity, using **(setq elist (entget (car (entsel))))**, and **(foreach x elist (print x))**, as we have done in previous lessons, your text screen will display the dxf list as:

```
(-1 . <Entity name:  60000022>)
(0 . "LINE")
(8 . "OBJECTS")
(10 2.38204 5.96739 0.0)
(11 8.08154 3.75 0.0)
(210 0.0 0.0 1.0)
```

You can get the value associated with any pair of these dxf group codes using the (assoc) function. Enter **(setq endpoint (assoc 10 elist))**. This will return a list that contains the group code 10, plus the associated point list, as in (10 2.38204 5.96739 0.0), and similarly **(setq lyr (assoc 8 elist))** returns the entity layer associated pair (8 . "OBJECTS").

To return the value associated with the dxf group code, you need to use (cdr), which returns all except the first item of the list, so **(setq p (cdr (assoc 10 elist)))** returns the point (2.38204 5.96739 0.0) in the above example.

We can use some of these new functions to provide a new editing feature to change the layer of any selected objects to the layer of another selected 'target' object. This is useful if, like me, you forget to change layers when you need to, for instance, if you put your dimensions on different layers from those of your geometry, you might have a "DIM1" layer, but returning to the drawing later to add new dimensions, you forget to change layers to the "DIM1" layer. Here is a defined function, C:CLYR2 that will do the trick:

```
FUNCTION C:CLYR2
(defun C:CLYR2 ()
   (princ "\nSelect object on target layer: ")
   (setq elist (entget (car (entsel))))
   (setq lyrname (cdr (assoc 8 elist)))
   (princ "\nSelect objects to be put on the target layer: ")
   (setq ss1 (ssget))
   (command "CHPROP" ss1 "" "LA" lyrname "")
   (princ)
); C:CLYR2
```

The function asks you to select an object on the target layer, that is the layer that you want to change to. The selection is made with (entsel), using the form (setq elist (entget (car (entsel)))). The prompt to select the object has already been made with the preceding (princ) statement. The layer name is extracted from the entity dxf list and assigned to 'lyrname' with (assoc), as shown. The objects to be placed on the selected layer are then selected with the general form of (ssget), and the selected set is assigned to 'ss1'.

Note how the selected set can be used directly in the editing command "CHPROP", by simply entering the variable ss1 as shown in the expression (command "CHPROP" ss1 "" "LA" lyrname "").

It is also possible to use the alternative form of entity selection:

```
(command "SELECT"  "auto"  pause)
(command "CHPROP" "p" "" "LA" lyrname)
```

Here, the "p" for "previous" selected set is used instead of the earlier selection set 'ss1'. You can save more than one selection set at a time using a series of (ssget) statements, but it is only possible to retain one (the most recent) selected set using the "select" and "p" options, so (ssget) is more flexible.

An example program that I wrote some years ago to show the application of (entget), (entdel) and (assoc), cross-hatches an enclosed area by letting the user pick a point anywhere inside an enclosed area. This method of hatching is now a feature of release 12, although the method of approach is clearly different from mine (and somewhat faster!). It is quite common that the programs you write today become the standard features of a future AutoCAD release. The hatching program XHATCH.LSP is still useful for those of you who have not yet upgraded to release 12, and it also shows some nice techniques for developing programming codes and algorithms.

XHATCH.LSP takes an initial point inside of the area to be hatched. The main function (calc) searches for a boundary object using a defined function (linmove) that takes point and angle arguments as the start conditions, and returns a point on the boundary object. The returned point is then used as an argument in another defined function (xint), which locates an intersection point between the first boundary object and an adjoining (though not necessarily sharing a common endpoint) object. (xint) returns the intersection point, which is assigned to int1. The (calc) function is:

```
XHATCH.LSP - the (calc) function
(defun calc (/ pt p int1 int2 epsilon enold)
   (setq pt (getpoint "\nPick hatch area: "))
   (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
```

The algorithm from here on is simply a search, in a logically 'end-to-end' direction, for the next entity that has an intersection with the 'current entity'. This is done with the (while) loop that locates the next intersection point int2, creates a new entity between intersection points int1 and int2 (using the defined function (newent), then adds the new entity to a selection set 'elst'. The point int1 is then assigned the value of int2, and the loop begins again and continues until the next intersection point int2 is coincident with the very first intersection.

Coincident here allows for some numerical 'fuzziness' by using the technique of comparing the distance between two points with an arbitrarily small value 'epsilon', rather than insisting on a zero comparison. This technique is especially useful in other cases where the compared points are taken from screen pick positions, instead of well defined end or intersection points.

We imposed the restriction that the hatch boundary entities should be logically end-to-end, which means that the direction of travel is toward the farthest end point of the next boundary object from the intersection point, but not necessarily coincident with the intersection. A more general algorithm would be to follow a logical 'clockwise' or 'counter-clockwise' direction, or perhaps to select all entities in a region and test each one against the rest to find intersection points on the hatch boundary.

In the while loop, there is a 'current entity' en1, which is renamed to enold and then deleted with (entdel) before the search for the next intersection begins. The entity is deleted because the search for the next intersection (xint) looks for entities, and only if there are none to be found, searches in the direction of the old entity until the next object is discovered. After the next object has been found, (xint) un-deletes the old entity so that the true intersection point can be found. The last statement in (xint) is (osnap p "INT"), which returns the appropriate point. It is this statement that effectively forces (xint) to return the point, which is assigned to int2 as shown in the while loop with the statement (setq int2 (xint int1)).

The actual hatching is performed by the (output) function as follows:

```
 XHATCH.LSP - the (output) function
 (defun output ( / n i)
    (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")
    (command "hatch" "ANSI31" "" "" "L" "")
 (princ)
 ); output
```

The number of boundary elements, 'n' is found from (setq n (sslength elst)), where elst is the final selection set of the boundary objects. These objects are un-deleted with (entdel) in a repeat loop before being joined together as a polyline with the "Join" option of "PEDIT". Note how the repeat loop increments at the same time as the 'ith' entity name is given, by using (ssname elst (setq i (1+ i))), where i is initialized as -1.

This is a short-hand method which has the same effect as (ssname elst i) followed by (setq i (1+ i)).

A benefit of collecting entities in a selection set is that the selection set can be supplied to the AutoCAD (command) editing functions such as "PEDIT", shown in (output), to provide the entities to be joined to a polyline. The first entity of the selection set is supplied in response to the PEDIT prompt to pick a polyline. The selection set is elst, and the first entity in the set is provided with (ssname elst 0) as shown. The first entity is not a polyline, so we answer "Y" to turn it into one, before responding with "J" to join other entities to the polyline. For this, we supply the entire selection set 'elst', including the first entity that is now ignored by AutoCAD, so you don't need to play around with any fancy routines to supply all except the first entity.

Figures 1 through 4 show the XHATCH.LSP program in operation with line and arc boundary elements.

It is a good programming technique to invent defined functions as required, such as (linmov), (xint), and (newent), so that the (calc) function can really look like the algorithm for the problem. For instance, I wanted a function to search along a straight line in some direction, and return a point that lies on an entity (in this case, a boundary object). The (linmov) function does that job, with a call that looks like **(setq p (linmov *point angle*))**. 'Point' and 'angle' are the arguments representing the start point of the search, and the direction. The first time (linmov) is called, I wanted the direction to be to the left, in the negative x direction, so I supplied PI as the angle.

A word of warning here is to remember to document your programs well, because when you return to the program to update it after a long absence, you might be confused. Looking at programs that you did not write is a salutary lesson, if you are not totally familiar with all of the functions of AutoLISP, you may not know what is a standard function and what is an invented (defined) function at first sight. There is often confusion about functions that look as though they should be built-in, such as the popular (dtr) degrees-to-radians, and the (dxf) function that we will see later in this lesson, that returns the value of an associated pair of dxf group codes.

Remember that defined functions return the result of the last expression evaluated, so if (linmov) returns a point that lies on the object, its last expression must return a point, and for this I used (osnap p2 "NEA") as shown in the listing of (linmov).

The (linmov) function starts by defining a 'delta' increment that I related to the size of the aperture, as a ratio of the horizontal drawing limits, so no matter what the drawing limits are, delta will always be controlled by the APERTURE system variable. Note that this method increases the 'delta' size if you zoom into the drawing. If this is not suitable, you could also use the VIEWSIZE system variable, which contains the actual height of the current viewport. Note the use of (float) to convert the integer APERTURE value to a real (floating-point) number before dividing by another integer, the value of SCREENSIZE.

It would be more efficient to create 'delta' in an (input) or (setting) section of the program so that it is calculated only once, instead of every time that (linmov) is executed.

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

The function executes in a while loop, setting the position of a point p2 at an angle and distance (delta) from the start point (plin), then tests to see if there is an object within a crossing window defined by the start point and the point p2 as shown. The points are then changed, so the start point moves to the p2 point, and the loop continues until an entity is found. The loop condition is that the entity name ename is nil. After the boundary entity has been found, a point on the entity is returned with (osnap p2 "NEA").

Now let's have a look at (xint), which returns an intersection point between the 'current' entity and the next adjoining entity.

(xint) takes a start point as its argument, then searches along the entity that the start point is on, and returns the intersection point between the current entity and the next entity.

```
 XHATCH.LSP - the (xint) function
 (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
```

Note that we assume there is only one entity at the start point, so its name is obtained with the first statement (setq ename (ssname (ssget xpt) 0)). XHATCH.LSP allows the boundary elements to be lines or arcs, so the entity type is checked with the next expression (setq etype (dxf 0 ename)). Here, we have invented the defined function (dxf), that takes the dxf group code and the name of the entity as its arguments, and returns the value associated with the group code. The (dxf) defined function is:

```
 XHATCH.LSP - the (dxf) function
 (defun dxf (code entname)
    (cdr (assoc code (entget entname)))
 )
```

The code for 'entity type' is 0, and (dxf) uses (assoc) with (entget) to return the entity type. This function will be used many times in XHATCH.LSP, so it worth defining as a separate function.

To continue, (xint) tests for entity type with (cond) as shown. If the entity type is a line, the defined function (lineang) is used to obtain the angular direction of the line, then (linmov) is used as before to return the next intersection point.

If the entity is of type "ARC", two functions, (arccenrad) and (arcmov), are defined to obtain the next "NEArest" entity point. The functions (lineang) and (arccenrad) do not have to return anything, but they create some new global variables that can be used by (linmov) and (arcmov) respectively. The point 'xpt' that is returned by (linmov) or (arcmov) is used by (xint) to determine and return the next intersection point with the last expression (osnap xpt "INT").

(lineang) creates the variable 'ang', which is the angle that is used as the direction required by (linmov). The first time we used (linmov), there was no entity to follow, so the angle was supplied as PI, so that the search direction was forced to be in the negative x-direction, or 180 degrees. Subsequent calls to (linmov) supply the angle 'ang', created by (lineang) as follows:

```
XHATCH.LSP - the (lineang) function
(defun lineang (en p3 / p1 p2 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
```

The start and end points of the line are found using (dxf) with group codes 10 and 11 respectively. The intersection point, p3 is closer to one end of the line than the other end, and this is determined by testing the distance of each end from p3. The direction in which we want to search for the next entity is worked out as follows.

First, assume that the start end of the line is closest to the intersection point that we already have (p3). If this is the case, then the distance 'dis2' should be greater than the distance 'dis1'. Note that we have already defined these distances in the expressions (setq dis1 (distance p3 p1)) and (setq dis2 (distance p3 p2)). If the assumption is correct, then the angle we need is simply (angle p1 p2), otherwise the angle must be given by (angle p2 p1). The 'if' expression of (lineang) states just that, so the angle 'ang' will always be the appropriate one, regardless of which is the start end of the line boundary entities. Finally, the current line entity is assigned the name 'en1', so that it can be deleted as required by the program.

Note that the entity name 'en1' and the angular direction 'ang' are assigned as global variables. If you really want all of your symbols to be local, and you use a function to produce many variables, then you can have your function return a list of those variables. In that case, the calling function would assign the list of variables. For instance, if we wanted (lineang) to return the angle 'ang' and the entity name 'en1', we could use the form **(setq linlist (lineang ename xpt))**, and then (lineang) would contain statements such as:

```
    (if (> dis2 dis1) (setq ang (angle p1 p2))
                            (setq ang (angle p2 p1))
    ); if
    (setq en1 en)
    (list ang en1)
); end of lineang
```

Now, because the (list) is the last expression, the list of (ang en1) will be returned.

We will leave the remaining functions for you to ponder in the homework assignment, so the full listing of XHATCH.LSP will be in the study guide that accompanies this lesson in the correspondence course.

SUBSTITUTION AND MODIFICATION OF DXF GROUP CODES

Any item of a list can be substituted with a new item, with function (subst), that requires arguments for the new item, the old item, and the list, as in **(subst *new-item old-item list*)**. This is commonly used to replace items such as the dotted pairs of entity association lists. The example we gave earlier to change the layer of any objects to a new layer could be done with (subst) instead of the change property command, as in the CLYR3.LSP program. After the substitution has been made, the objects are updated in the AutoCAD data base with the (entmod) function:

```
C:CLYR3 - changing layers with (subst)
(defun C:CLYR3 ( / elist targetlyr ss1 num index oldlist
      newlist oldlyr)
   (princ "\nSelect object on target layer: ")
   (setq elist (entget (car (entsel))))
   (setq targetlyr (assoc 8 elist))
   (princ "\nSelect objects to be put on the target layer: ")
   (setq ss1 (ssget))
   (setq num (sslength ss1))
   (setq index 0)
   (repeat num
      (setq oldlist (entget (ssname ss1 index)))
      (setq oldlyr (assoc 8 oldlist))
      (setq newlist (subst targetlyr oldlyr oldlist))
      (entmod newlist)
      (setq index (1+ index))
   ); repeat
   (princ)
); C:CLYR3
```

This is a more cumbersome program than CLYR2.LSP, but there are some benefits for checking whether or not the changes made are correct, and if not, then the old list can be reverted to with (entmod oldlist) as in the next partial program example, CLYR4.LSP, which allows the user to have a change of mind or to correct an error in selection:

```
FUNCTION C:CLYR4 - changing layers with error checking
(defun C:CLYR4 ( / elist targetlyr ss1 num index oldlist
      newlist oldlyr)
;  the first part of the program is the same
;  as CLYR3.LSP
      (repeat num
      (setq oldlist (entget (ssname ss1 index)))
      (setq oldlyr (assoc 8 oldlist))
      (setq newlist (subst targetlyr oldlyr oldlist))
      (entmod newlist)
```

```
        (setq yn (getstring "\nIs this okay?<Y>: "))
        (if (= (strcase yn) "N")
           (entmod oldlist)
        ); if
        (setq index (1+ index))
    ); repeat
    (princ)
 ); C:CLYR4
```

Naturally, for this to be effective, there must be some visual clue that the layer has been changed, such as a color or linetype change. You can also invent new layer names by using the form (setq targetlyr (cons 8 "NEWLAYERNAME")), instead of choosing an entity on the target layer, and the new layer will automatically be created. This approach can be used to substitute and modify any of the group codes except entity type or entity handle.

Homework:

1. 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.

2. 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.

3. 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.

In lesson five of this series, "AutoCAD Complex Entities", you will learn about complex entities such as blocks and inserts, and you will learn how to create blocks and other entities with the 'entity make' functions. You will discover how to list and sort through the data structure of blocks and inserts. Programming examples will be used to show you how to apply the functions, and homework assignments will test your understanding of the lesson.

AutoCAD Complex Entities

Lesson Five in the CADalyst/University of Wisconsin Advanced AutoLISP Programming course shows you use entity functions for data base manipulation.

by
Anthony Hotchkiss, Ph.D, P.Eng.

About this course.

Advanced AutoLISP Programming is an eight-lesson independent study course, originally published in monthly installments in CADalyst magazine. Before you begin this course, it is assumed that you have reached a basic level of AutoLISP programming, equivalent to completing the preceding 'Introduction to AutoLISP Programming' course which was originally published in CADalyst, and is now available as an independent study course from the University of Wisconsin-Madison. On successfully completing an intermediate and a final mail-in project, registered students will receive certificates of completion and 5.0 CEU's (continuing education units).

For further information, contact:

Steve Bialek
Engineering Professional Development
University of Wisconsin-Madison
432 North Lake Street
Madison, Wisconsin 53706
(608) 262-1735

LESSON FIVE - AUTOCAD COMPLEX ENTITIES

Lesson five covers complex entities such as blocks and polylines, and you will learn how to create blocks and other entities with the 'entity make' functions. You will discover how to list and sort through the data structure of complex entities. Programming examples will be used to show you how to apply the functions, and homework assignments will test your understanding of the lesson.

BLOCKS AND INSERTS

When you create a block in AutoCAD, whether it is an internal BLOCK or an external WBLOCK, the method for using the block is to 'insert' the block with the INSERT command. It is then common terminology to refer to what is inserted as a 'block'. In fact, the entity displayed on your screens is actually an 'INSERT' entity that contains a reference to the block from which it has been inserted.

This distinction is vital if you need to list and manipulate the entity data base of AutoCAD. For instance, if there is an INSERT entity that you have just inserted from a block called "FRED", the dxf list of the insert entity can be obtained with (entget (entlast)), as we have seen in previous lessons. Your command/prompt area (after toggling the 'text' screen) will look like:

```
  Command:  (entget (entlast))
  ((-1 . <Entity name: 60000082>) (0 . "INSERT") (8 . "1") (2 .
"FRED")
  (10 9.5095 2.46123 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)
```

There is no direct indication of any entities that form the group of objects in the insert entity because the insert entity is simply a reference to a block, and contains information about the insertion point (group code 10), X, Y, and Z scale factors (group codes 41, 42 and 43), rotation angle (group code 50), multiple insert options (group codes 70, 71, 44, and 45), and the orientation vector, code 210. In addition, the reference to the block name is shown as group code 2.

The INSERT entity is a single entity that contains the reference to the complex entity (the block from which it is inserted). If Attributes are attached to the block, these will be referenced in the insert entity as a group code 66, which is a flag for 'attributes to follow', and the insert entity will then become a 'complex' entity that contains 'subentities'.

If we look at the block table, the block name and listing from (tblsearch) will appear in the command/prompt area of the screen as:

```
Command:  (tblsearch "block" "fred")
((0 . "BLOCK") (2 . "FRED") (70 . 64) (10 4.67331 2.98173 0.0)
(-2 . <Entity name>: 40000020>))
```

Here, the group code 10 is the block base point, code 70 is the block type (the bit-value 64 means it is referenced), and the code -2 is the name of the first entity of the following entities that comprise the block. In order to list all of the elements of the block, there is a standard function (entnext), which returns the name of the first entity in the data base. If an optional entity name is supplied as an argument, as in (entnext entname), this will return the name of the first entity following the entity named in the argument.

If the listing of (tblsearch) is saved as a list using (setq blklist (tblsearch "block" "fred")), the name of the first entity in the block can be found from (setq entname (cdr (assoc -2 blklist))). The dxf list for this first entity is then obtained with (setq elist (entget entname)).

For dxf listings of subsequent entities in the block, the entity name of the following item is returned with (setq entname (entnext entname)) which re-names the entity assigned to 'entname'. The expression (setq elist (entget entname)) then returns the dxf list of this, the next entity in the block. The program LISTBLK.LSP uses these expressions with a loop to list the entities in a block to an external file:

```
LISTBLK.LSP - A program to list block data to an external file.
(defun C:LISTBLK (/ fname insname elist f1 blkname blklist entname)
   (setq fname (getstring "\nEnter the .TXT output file name: "))
   (setq fname (strcat "\\acad12\\drwgs\\" fname ".txt"))
   (setq insname (car (entsel "\nSelect block reference: ")))
   (setq f1 (open fname "w"))
   (setq blkname (cdr (assoc 2 (entget insname))))
   (print (strcat "DXF LIST FOR BLOCK: " blkname) f1)
   (setq blklist (tblsearch "block" blkname))
   (setq entname (cdr (assoc -2 blklist)))
   (while entname
      (setq elist (entget entname))
```

```
        (foreach item elist (print item f1))
        (setq entname (entnext entname))
     ); while
     (close f1)  (princ)
  ); listblk
```

The significant feature here is the change of entity name to the next entity name with **(setq entname (entnext entname))** in the while loop. This is the mechanism to list successive 'subentities' contained within the block. Note that it is the block data, and not the block reference (insert) data that is being listed.

Figures 5-1 and 5-2 show the execution of this program with the "happy" block. The resulting file is as follows:

```
  "DXF LIST FOR BLOCK: HAPPY"
  (-1 . <Entity name: 40000be5>)
  (0 . "CIRCLE")
  (8 . "1")
  (10 1.0 1.0 0.0)
  (40 . 0.337287)
  (210 0.0 0.0 1.0)
  (-1 . <Entity name: 40000c07>)
  (0 . "ARC")
  (8 . "1")
  (10 0.0 0.5 0.0)
  (40 . 2.5)
  (50 . 3.78509)
  (51 . 5.63968)
  (210 0.0 0.0 1.0)
  (-1 . <Entity name: 40000c39>)
  (0 . "CIRCLE")
  (8 . "1")
  (10 0.0 0.0 0.0)
  (40 . 3.0)
  (210 0.0 0.0 1.0)
  (-1 . <Entity name: 40000c5b>)
  (0 . "CIRCLE")
  (8 . "1")
  (10 -1.0 1.0 0.0)
  (40 . 0.344201)
  (210 0.0 0.0 1.0)
```

The base point for the block is the zero point, and all of the other entities are in positions relative to the base point (which is at the center of the 3 inch radius circle, as shown in entity 40000c39)

LISTING ATTRIBUTES

When a block has attributes attached to it, the attribute definitions become part of the block, but it is the insert entity (block reference) that contains the attribute values. If we use the LISTBLK.LSP program on a block with attributes, the first part of the listing contains "ATTDEF" entities, as shown in this partial listing of a title block "TBS". Figure 5-3 shows the attribute values being attached via the dialog box (obtained with the system variable ATTDIA set to 1 prior to inserting the block), and figure 5-4 shows the title block insert with the attribute text filled in.

The partial listing for the ATTDEF entities include group codes 1, 2 and 3, representing the default attribute text value, the attribute tag, and the attribute prompt respectively. Refer to your AutoCAD Reference Manual (release 11 and earlier), or to the AutoCAD Customization Manual, chapter 11 (release 12) for explanations of the other group codes.

```
PARTIAL LISTING OF "TBS"
"DXF LIST FOR BLOCK: TBS"
(-1 . <Entity name: 4000001f>)
(0 . "ATTDEF")
(8 . "TEXT")
(10 -4.00027 0.872627 0.0)
(40 . 0.11066)
(1 . "")
(3 . "COMPONENT #:")
(2 . "COMPNO")
(70 . 0)
(73 . 0)
(50 . 0.0)
....etc
```

Instead of listing the block dxf codes, you will want to examine the actual attribute tags and values that are part of the INSERT entity, and a modified version of the listing program is appropriate, as shown in LISTATT.LSP, in which the block reference is listed instead of the block itself.

```
LISTATT.LSP - A program for listing attribute data
(defun C:LISTATT (/ fname entname elist f1 blkname)
   (setq fname (getstring "\nEnter the .TXT output file name: "))
   (setq fname (strcat "\\acad12\\drwgs\\" fname ".txt"))
   (setq entname (car (entsel "\nSelect block reference: ")))
   (setq f1 (open fname "w"))
   (setq blkname (cdr (assoc 2 (entget entname))))
   (print (strcat "DXF LIST FOR INSERT: " blkname) f1)
   (while entname
      (setq elist (entget entname))
      (foreach item elist (print item f1))
      (setq entname (entnext entname))
   ); while
   (close f1)
   (princ)
); listatt
```

The output from the LISTATT.LSP program is shown in the partial listing of the INSERT-TBS, which starts with the "INSERT" entity, that has all the group codes shown in the block reference "FRED" earlier. In addition, the insert "TBS" contains the group code 66 with a value of 1 that indicates "attributes to follow".

The entity that follows the "INSERT" entity is of type "ATTRIB" (associated with the group code 0), which has the group code descriptions shown in italics, although of course these descriptions are not part of the output to the external file. All of the attributes are listed in this way until the end of the "ATTRIB" entities is reached. After the last "ATTRIB" entity, the final entity in the list is the "SEQEND" entity, which indicates the end of the sequence of attributes.

The addition of attributes converts the INSERT entity into a 'complex' entity which must be listed with successive (entnext) expressions.

```
PARTIAL LISTING OF INSERT-TBS
"DXF LIST FOR INSERT: TBS"
(-1 . <Entity name: 60000022>)
(0 . "INSERT")
```

```
(8 . "TITLE_BLOCK")
(66 . 1)
(2 . "TBS")
(10 8.36434 2.59783 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)           DESCRIPTION
(-1 . <Entity name: 60000044>)entity name
(0 . "ATTRIB")              entity type
(8 . "TEXT")                layer name
(10 4.36406 3.47045 0.0)    text start point
(40 . 0.11066)              text height
(1 . "930001")              attribute value
(2 . "COMPNO")              attribute tag
(70 . 0)                    attribute flag
(73 . 0)                    field length
(50 . 0.0)                  text rotation angle
(41 . 1.0)                  x-scale factor
(51 . 0.0)                  text oblique angle
(7 . "STANDARD")            text style
(71 . 0)                    text generation flag
(72 . 0)                    horizontal justification
(11 0.0 0.0 0.0)            text alignment point
(210 0.0 0.0 1.0)           orientation vector
(74 . 0)                    vertical justification
(-1 . <Entity name: 60000066>)
(0 . "ATTRIB")
(8 . "TEXT")
(10 5.36055 3.45955 0.0)
(40 . 0.125)
(1 . "ATTRIBUTE EXAMPLE")
(2 . "TITLE")
....etc
.
.
(-1 . <Entity name: 60000154>)
(0 . "SEQEND")
(8 . "TITLE_BLOCK")
(-2 . <Entity name: 60000022>)
```

POLYLINES

Polylines are complex entities that contain a set of VERTEX subentities. A suitable program for listing the polyline data is shown in LISTPOLY.LSP.

```
LISTPOLY.LSP - A program for listing polyline data
(defun C:LISTPOLY (/ fname pname elist f1 entname)
   (setq fname (getstring "\nEnter the .TXT output file name: "))
   (setq fname (strcat "\\acad12\\drwgs\\" fname ".txt"))
   (setq entname (car (entsel "\nSelect polyline: ")))
   (setq f1 (open fname "w"))
   (while entname
      (setq elist (entget entname))
      (foreach item elist (print item f1))
      (setq entname (entnext entname))
```

```
    ); while
    (close f1)
    (princ)
  ); listpoly
```

The polyline data starts with the "POLYLINE" entity, and ends with the "SEQEND" entity, with vertex entities in between as shown in the "partial listing of polyline".

```
  PARTIAL LISTING OF POLYLINE
                                DESCRIPTION
  (-1 . <Entity name: 60000022>)Entity name
  (0 . "POLYLINE")              Entity type
  (8 . "1")                     Layer
  (66 . 1)                      Vertices follow flag
  (10 0.0 0.0 0.0)              Polyline elevation
  (70 . 1)                      Flag (1 = closed)
  (40 . 0.0)                    Starting width default
  (41 . 0.0)                    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)                      Surface M density
  (74 . 0)                      Surface N density
  (75 . 0)                      Curves & surface type
  (-1 . <Entity name: 60000044>)
  (0 . "VERTEX")
  (8 . "1")
  (10 1.77293 3.29348 0.0)      Vertex location point
  (40 . 0.0)                    Start width
  (41 . 0.0)                    End width
  (42 . 0.0)                    Bulge tangent (polyarcs)
  (70 . 0)                      Entity type flag
  (50 . 0.0)                    Fit curve tangent direction
  (71 . 0)                      Surface edge visibility
  (72 . 0)                      ''
  (73 . 0)                      ''
  (74 . 0)                      ''
  .
  .
  (-1 . <Entity name: 600000ee>)
  (0 . "SEQEND")
  (8 . "1")
  (-2 . <Entity name: 60000022>)Name of 'parent' entity
```

An examination of the group codes shows that polylines are the entity types for the Mesh surfaces of AutoCAD, as well as for two and three dimensional polylines.

MODIFYING COMPLEX ENTITIES

The complex entities are blocks (block definitions), inserts with attributes, and polylines (these include ellipses, polygons, polyface surface meshes and 'donuts')

The technique for manipulating the dxf codes of complex entities is by using (subst) and (entmod) in the list of the individual subentity, and then using the function (entupd) to update the entire complex entity as follows in this example of changing the text height of any specified attribute of an insert entity:

```
  CHATTXT - A function for changing attribute text height
  (defun CHATTXT (/ attname ht entname elist newlst)
     (setq attname (getstring "\nAttribute to change: "))
     (setq ht (getreal "\nNew text height: "))
```

```
      (setq entname (car (entsel "\nSelect block reference: ")))
      (setq elist (entget entname))
      (while (and (/= "SEQEND" (cdr (assoc 0 elist)))
                  (/= (strcase attname) (cdr (assoc 2 elist)))
              ); and
         (setq entname (entnext entname))
         (setq elist (entget entname))
      ); while
      (setq newlst (subst (cons 40 ht)  (assoc 40 elist) elist))
      (entmod newlst)
      (entupd entname)
      (princ)
  ); chattxt
```

This program asks the user for the attribute name for which the text height is to be changed. The attribute name is then used as a search criteria in the while loop. The while loop test is set up so that each subentity in turn is skipped over by stepping through the subentities with (entnext), as long as the subentity is not a "SEQEND" entity, and its name does not match the input name.

When the required attribute name is found, the while loop ends and the text height is substituted with (subst) and (entmod) as we saw in lesson 4. This time, for complex entities, the additional step of using (entupd) is necessary to update the entire complex entity.

To search your drawings for entities containing attributes, you can create a selection set that has filters for the combination of "INSERT" entity type, and group code 66 with a value of 1, for example:

(setq attset (ssget "X" '((0 . "INSERT") (66 . 1))))

will create the appropriate selection set. It would be possible to perform a series of such updates on all block references in the drawing.

Another useful function for complex entities is (nentsel), which works like (entsel), except that it returns the name of the subentity of a block reference or polyline. Like (entsel), (nentsel) also has an optional prompt argument. When simple entities or attributes are selected, (nentsel) returns the same information as (entsel), that is the entity name and the pick point. When non-attribute entities are selected in a block reference, (nentsel) returns the subentity name, the pick point, a transformation matrix relating the model to the world coordinate system, and the names of any nested blocks including the block that contains the selected entity.

MAKING BLOCKS

Release 11 of AutoCAD introduced a new function (entmake), which can be used to create your own entities, both simple and complex. **(entmake)** requires a single argument in the form of a list similar to the list returned by (entget). The list need only contain the bare minimum of information required to define the entities, as most of the dxf group codes are optional, and will be attached to the (entmake) entity by default.

For instance, try entering this simple definition of a circle. All you need is the entity type, center point and the radius:

(entmake '((0 . "CIRCLE") (10 4.0 3.0 0.0) (40 . 2.5)))

The circle will be created on the current layer in the current XY plane, with the default color and linetype. You can now enter (entget (entlast)) to see that the regular dxf list is returned with the default values added.

Another way to supply the list is to use (entget) as follows, by selecting an existing entity to reproduce:

(entmake (cdr (entget (car (entsel)))))

This method of using (entget) is useful for creating your own blocks from selected objects. You might well ask "why bother to make your own blocks and entities with entmake, rather than use AutoCAD commands via the (command) function?". The answer is that it is generally more efficient to have your LISP program do the job of manipulating data, so your programs will run faster.

Also, as we shall see, you can create blocks that are not automatically erased, and in our next lesson (six), we will deal with 'anonymous' blocks which can represent groups of objects that do not conflict with the block name list in your drawings, just like dimensions and hatch patterns. This means that you do not have to 'purge' the blocks that you create in case of conflicts and similar names.

To make blocks, you need a series of (entmake) expressions that define the start of the block, the subentities in the block, and an end of block entity. Here is an example of a block of three concentric circles:

```
  (entmake '((0 . "BLOCK") (2 . "FRED") (70 . 64) (10 4.0 5.0 0.0)))
  (entmake '((0 . "CIRCLE") (10 4.0 4.0 0.0) (40 . 1.0)))
  (entmake '((0 . "CIRCLE") (10 4.0 4.0 0.0) (40 . 2.0)))
  (entmake '((0 . "CIRCLE") (10 4.0 4.0 0.0) (40 . 3.0)))
  (entmake '((0 . "ENDBLK")))
```

This could also be written in the alternative form as

```
  (entmake (list (cons 0 "BLOCK") (cons 2 "FRED") (cons 70 64) (cons
10 4.0 5.0 0.0)))
  (entmake (list (cons 0 "CIRCLE") (cons 10 4.0 4.0 0.0) (cons 40
1.0)))
  (entmake (list (cons 0 "CIRCLE") (cons 10 4.0 4.0 0.0) (cons 40
2.0)))
  (entmake (list (cons 0 "CIRCLE") (cons 10 4.0 4.0 0.0) (cons 40
3.0)))
  (entmake (list (cons 0 "ENDBLK")))
```

The alternative form allows for symbols (variable names) to be used as the values in the expressions. The group code 70 is a required bit coded number where 64 means that it is a referenced block.

We use a mixture of all of the above forms in the example program, MKBLKS.LSP that creates block definitions with (entmake):

```
 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
```

This program only allows you to select objects one at a time, but you can fix that as one of our homework questions by extending the selection set capacity of the program to allow all kinds of entity selection.

The block program starts by getting the block name and insertion point from the user, then the (entmake) block header expression is written using the (list (cons... format so that the user supplied values can be used as variables. Only the (entmake) expressions are used in the construction of the block, and they do not need to be supplied one after the other, so the other AutoLISP functions can be placed as shown.

I used (redraw (car esel) 3) to highlight the objects as they are selected to make the selection process more intuitive. The while loop allows any number of entities to be selected for the block, and the final ENDBLK entity is supplied after the while loop ends.

The block that is created can now be inserted in the normal way, and the selected entities are not erased by this program.

Homework:

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

2. 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).

In lesson six of this series, "AutoLISP Anonymous Blocks", you will learn how to make 'anonymous' complex entities that behave like dimensions that are inserted but not given names in the block list. You will learn some new naming conventions for anonymous blocks, and you will be introduced to the concept of 'extended entity data' that allows you to attach intelligent attributes to your geometry. Programming examples will be used to show you how to apply the functions, and homework assignments will test your understanding of the lesson.

AutoLISP Anonymous Blocks

Lesson Six in the CADalyst/University of Wisconsin Advanced AutoLISP Programming course shows you how to create anonymous blocks and introduces extended entity data.

by
Anthony Hotchkiss, Ph.D, P.Eng.

About this course.

Advanced AutoLISP Programming is an eight-lesson independent study course, originally published in monthly installments in CADalyst magazine. Before you begin this course, it is assumed that you have reached a basic level of AutoLISP programming, equivalent to completing the preceding 'Introduction to AutoLISP Programming' course which was originally published in CADalyst, and is now available as an independent study course from the University of Wisconsin-Madison. On successfully completing an intermediate and a final mail-in project, registered students will receive certificates of completion and 5.0 CEU's (continuing education units).

For further information, contact:

Steve Bialek
Engineering Professional Development
University of Wisconsin-Madison
432 North Lake Street
Madison, Wisconsin 53706
(608) 262-1735

LESSON SIX - AUTOCAD ANONYMOUS BLOCKS

In this lesson, you will learn how to make 'anonymous' complex entities that are inserted but not given names in the block list. You will learn some new naming conventions for anonymous blocks, and you will be introduced to the concept of 'extended entity data' that allows you to attach intelligent attributes to your geometry. Programming examples will be used to show you how to apply the functions, and homework assignments will test your understanding of the lesson.

THE STRUCTURE OF ANONYMOUS BLOCKS

In lesson two of this series, we listed the DXF group codes of a block that was a dimension. The DXF list was:

```
(0 . "BLOCK")
(2 . "*D0")
(70 . 65)
(10 0.0 0.0 0.0)
(-2 . <Entity name: 4000001e>)
```

We noted that the use of the asterisk in the name of the block was a convention for naming anonymous blocks. The 'D' in the name signifies that this anonymous block is a Dimension (actually an associative dimension). Another type of anonymous block is the hatch pattern, which has a name of the form *X*nnn*, where *nnn* is a number. The anonymous blocks that we are concerned with here are the 'User' anonymous blocks that take names of the form *U*nnn*, where *nnn* is a number that is assigned automatically by AutoCAD.

Anonymous blocks can be used to create groups of entities that don't get recorded as named blocks in the drawing block list, and they are automatically purged from the drawing if they are not referenced (inserted). For this reason, the number portion of the anonymous block name can change between drawing sessions, just like the names of any other entity, but you can attach 'handles' to them if you want a constant identifier.

The use of the asterisk as the first character of the name usually has the effect of exploding inserted blocks. You can not insert anonymous blocks with the INSERT command, because AutoCAD will assume you intend to insert and explode a block with the name of U*nnn.* Instead, you need to use (entmake) to create the INSERT entity.

Another aspect of using anonymous blocks is that you can attach extended entity data (xdata) to them, as you can with any entity or complex entity. The extended entity data has dxf codes that can relate distances and positions to the parent object (that is the object to which the xdata are attached), so if the parent object is scaled, the extended entity data are automatically scaled with it. You can use this property to attach 'smart' attributes to entities in a much more flexible way than you can by using the regular attributes of AutoCAD. If anonymous blocks are used to attach the intelligent attributes via xdata, you don't have to create standard blocks that need to be purged if they are not used.

The dxf list of an anonymous block is obtained from (tblsearch) thus;

```
Command: (foreach x (tblsearch "block" "*U1") (print x))
(0 . "BLOCK")
(2 . "*U1")
(70 . 65)
(10 8.38609 6.1413 0.0)
(-2 . <Entity name: 40000d00>)
```

The program LISTBLK.LSP from the previous lesson (five) of this series can be used to expand the list of sub entities in the anonymous block with (entnext), just as you can with regular block and insert entities. Two things are different from regular blocks in this list; the name begins with the asterisk-U, and the flag bit code 70 has a '1' in it to signify the anonymous block. In this case the '1' has been added to the flag bit code '64', which means that the block definition is referenced.

The listing from (entget) of the INSERT looks like:

```
(-1 . <Entity name: 600000ee>)
(0 . "INSERT")
(8 . "1")
(2 . "*U1")
(10 4.25287 9.20652 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 dxf group codes are the same as those for regular blocks

CREATING ANONYMOUS BLOCKS

If you have a new drawing, the first anonymous block name will be "*U0", so it would be safe to type in the following to create a simple anonymous block using (entmake):

```
(entmake '((0 . "BLOCK") (2 . "*U0") (70 . 64) (10 4.0 5.0 0.0)))
(entmake '((0 . "CIRCLE") (10 4.0 4.0 0.0) (40 . 1.0)))
(entmake '((0 . "CIRCLE") (10 4.0 4.0 0.0) (40 . 2.0)))
(entmake '((0 . "CIRCLE") (10 4.0 4.0 0.0) (40 . 3.0)))
(entmake '((0 . "ENDBLK")))
```

The number '0' in the name *U0 is actually ignored by AutoCAD, so you could equally well write "*U" in the group code '2' instead.

Now, in order to insert the new anonymous block, you must use (entmake), and not the command 'INSERT'. The (entmake) statement to insert the block at coordinates 6,6 is:

(entmake '((0 . "INSERT") (2 . "*U0") (10 6.0 6.0 0.0)))

As this is the first anonymous block created in your drawing, it is safe to assume that the name has been assigned to be *U0, but if you created an anonymous block in a drawing that already had some anonymous blocks, you would need to know the number of the last anonymous block so that you can insert the one that you just created, and not another one.

Without proper care, it is also possible to redefine existing anonymous blocks by giving them the same name as an existing block, just as you can with conventional blocks. (entmake) automatically returns the block name if the entity type is "ENDBLK", so you can make use of this when inserting the block.

ABLOCK.LSP is a program for creating anonymous blocks by selecting objects just like you would for conventional blocks.

```
  ;;; ABLOCK2.LSP   A program to make and insert anonymous blocks
  ;;;           Program by Tony Hotchkiss
  (defun C:ABLOCK2 (/ ins_pt num ss1 i n)
  ;(defun test () ; ** this line is used for development only
     (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
```

The program starts by getting the insertion point of the block, which is used in the (entmake) statement for the block header as **(entmake (list '(0 . "BLOCK")'(2 . "*U")'(70 . 1)(cons 10 ins_pt)))**. The list for entmake contains a combination of quoted and unquoted expressions, because the variable ins_pt must be evaluated when the list is supplied to (entmake). The block name is supplied as "*U" because AutoCAD will use its own numbers. If you really want to redefine an existing user block, you can supply the appropriate number.

Entity selection is made in the simplest way with the general form of (ssget). The number of entities selected is recorded as 'i', to be used for the following repeat loop count, and the value of 'n' is set to -1. The entities of the block are made with the single (entmake) statement of the repeat loop, in which each entity of the selected set is in turn listed with (entget). Note how the counter 'n' is automatically incremented by 1 as it is used.

After all entities have been made, the "ENDBLK" entity is included to signal the end of the anonymous block, which is now ready to be inserted. The block name is assigned to 'num'.

The "INSERT" entity is created with (entmake) as shown, using the block name variable and the previous insertion point. I didn't erase the entities that make up the anonymous block, implying that the "INSERT" entity will be inserted exactly over the old entities, so I included the 'MOVE' command with a base point at the insertion point. The block is attached to the cursor and can be placed at any desired positions, as shown in figure 6-1.

EXTENDED ENTITY DATA - AN INTRODUCTION

Extended entity data (Xdata) are values associated with dxf group codes currently in the range 1000 to 1071. Unlike the regular dxf codes, the xdata group codes may appear many times, so organization of data is important. Sets of extended entity data are associated and grouped by an 'application' name. Application names are stored in a symbol table called 'APPID'.

Before you attach extended entity data, it is necessary to provide the appid (application name/identifier) associated with group code 1001. For a single set of xdata (a single appid), the dxf code 1001 does not actually appear in the xdata list. The appid must be 'registered' using the function (regapp) as follows:

(regapp *namestring*)

For instance, type (regapp "myapp"), followed by (tblsearch "appid" "myapp"), and you will see the symbol table entry for the APPID as

((0 . "APPID") (2 . "MYAPP") (70 . 0))

The group code 70 is a flag that is common to the other symbol tables to flag the reference status.

EXTENDED ENTITY DATA GROUP CODES

The xdata group codes are shown in table 6-1.

TABLE 6-1 XDATA GROUP CODES

CODE	ENTITY TYPE AND DESCRIPTION
1000	String. Up to 255 bytes long plus a reserved null character.
1001	Application name; a string up to 31 bytes long (plus a null character).
1002	A special control string; can be either "{" or "}", used for conveniently grouping data.
1003	Layer name associated with the xdata.
1004	Binary data - hexadecimal digits, used by ADS, not AutoLISP.
1005	Handles of AutoCAD entities.
1010	3 real values, stored as X, Y, Z. In a DXFout list, each real is associated with 1010, 1020, and 1030 respectively.
1011	1011, 1021, 1031 A world space position. These coordinates are moved, scaled, mirrored, rotated and stretched with the 'parent' entity.
1012	1012, 1022, 1032 Another 3D point that is scaled, rotated and mirrored with the parent entity, but not moved or stretched.
1013	1013, 1023, 1033 Another 3D point that is only rotated and mirrored with the parent.
1040	A real value.
1041	A distance (a real value that is scaled with the parent entity).
1042	A scale factor (also a real value that is scaled with the parent entity).
1070	An integer. A '16-bit' integer that can be signed (plus/minus)
1071	A 'long-integer'. A 32-bit integer, used by ADS, not AutoLISP

As with the regular group codes, the 3D points can have each coordinate grouped separately, or as a single code. For instance, a group 10 point can be (10 2.0, 3.0, 0.0) or (10 . 2.0) (20 . 3.0) (30 . 0.0), and the same is true for groups 1010 thru 1013. A DXFOUT listing always makes the separate code format.

The significance of having values scaled etc. with the parent entity (the entity to which the xdata are attached) will become clear with a very simple example that you can try.

A SIMPLE XDATA EXAMPLE

For this example, we will use a 'fastener' application, in which we can imagine that there are some mechanical fasteners such as nuts and bolts. We represent a bolt by drawing a circle, and we will attach some xdata to the circle to represent the bolt type, thread form, and the bolt length and diameter. This example shows the 'intelligence' of these xdata as compared with regular attributes. Extended entity data is all grouped under the dxf code -3, and this is followed by the subgroup under the registered application name, (the APPID). The rest of the data is grouped in dotted pairs with the xdata group codes.

First make sure that you have a circle (I used a radius of 0.375), register the application with (regapp "fasteners"), and get the regular dxf group listing of your circle as shown in this command/prompt sequence, which assigns the circle dxf list to the symbol 'elist':

```
  Command:  (regapp "fasteners")
  "FASTENERS"
  Command:  (setq elist (entget (car (entsel))))
  Select object:  ((-1 . <Entity name: 6000001a>) (0 . "CIRCLE") (8 .
"1")
   (10 3.71912 1.87589 0.0) (40 . 0.375) (210 0.0 0.0 1.0)
  Command:
```

Now let's make some xdata with (setq...) as in this sequence (don't worry about the prompt for missing parentheses, shown as 3>, just enter the data as shown here):

```
  Command:  (setq xdat '(-3 ("fasteners" (1000 . "socket-head") (1000
. "16-UNF")
  3>   (1002 . "{") (1041 . 0.75) (1041 . 1.5) (1002 . "}"))))
```

Remember that the 3> is a prompt of the AutoLISP interpreter, not data to be entered, and use the quoted format for the list. AutoCAD will then display the xdat list as it is returned by (setq). In this example, we use the numbers 0.75 and 1.5 associated with group code 1041 to represent the diameter and the length of the bolt respectively. We will show in the next lesson that you can precede each of these numbers with strings associated with group code 1000 for more clarity and organization.

Now, add the xdata to the entity list and modify the AutoCAD data base as in the next sequence:

```
  Command:  (setq exdata (append elist (list xdat)))
  Command:  (entmod exdata)
```

Your screen will look like that of figure 6-2, and the xdata will be attached to the circle.

In order to show the extended entity data list, you need to add the registered application name as an argument to (entget), for instance as in:

(setq elist2 (assoc -3 (entget (entlast) '("fasteners"))))

This will return the list associated with code -3 (the extended entity data), as follows.

(-3 ("FASTENERS" (1000 . "socket-head") (1000 . "16-UNF") (1002 . "{")
(1041 . 0.75) (1041 . 1.5) (1002 . "}")))

Note how the APPID argument of (entget) is shown as the the quoted list of a string.

Now here's the best part! Copy the circle and scale the new entity with the xdata by a factor of 2, then get the dxf list of the new entity with (entget) as we did before, and you will see:

(-3 ("FASTENERS" (1000 . "socket-head") (1000 . "16-UNF") (1002 . "{")
(1041 . 1.5) (1041 . 3.0) (1002 . "}")))

Notice how the xdata is transferred to the new entity, and the numbers in group code 1041 representing the diameter and length have doubled, because they are scaled with the parent entity. This means that you can have much better representation of attributes with these intelligent extended entity data than you can with AutoCAD's regular attributes.

The question of how to handle data when the group codes are repeated is an organizational matter that needs to be discussed further. In our next lesson (seven), we will use programs to organize and create the xdata with an example that attaches intelligent attributes to groups of entities.

Homework:

1. 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?

2. 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?

3. 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?

4. 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).

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

In lesson seven of this series, "Extended Entity Data", you will learn more about extended entity data. You will learn how to attach intelligent attributes to your geometry, using the XDATA DXF group codes. Programming examples will be used to show you how to create and append XDATA, and homework assignments will test your understanding of the lesson.

Extended Entity Data

Lesson Seven in the CADalyst/University of Wisconsin Advanced AutoLISP Programming course shows you how to attach extended entity data as intelligent attributes.

by
Anthony Hotchkiss, Ph.D, P.Eng.

About this course.

Advanced AutoLISP Programming is an eight-lesson independent study course, originally published in monthly installments in CADalyst magazine. Before you begin this course, it is assumed that you have reached a basic level of AutoLISP programming, equivalent to completing the preceding 'Introduction to AutoLISP Programming' course which was originally published in CADalyst, and is now available as an independent study course from the University of Wisconsin-Madison. On successfully completing an intermediate and a final mail-in project, registered students will receive certificates of completion and 5.0 CEU's (continuing education units).

For further information, contact:

Steve Bialek
Engineering Professional Development
University of Wisconsin-Madison
432 North Lake Street
Madison, Wisconsin 53706
(608) 262-1735

LESSON SEVEN -- EXTENDED ENTITY DATA

In this lesson you will learn more about extended entity data. You will learn how to attach intelligent attributes to your geometry, using the XDATA DXF group codes. You will use programs to organize and create the xdata with an example that attaches intelligent attributes to groups of entities. Homework assignments will test your understanding of the lesson.

ATTACHING EXTENDED ENTITY DATA

In lesson six of this series you attached extended entity data to an object by creating a list of the form

(-3 ("application-name" (1000 . "...")))

and appended the list to the dxf list of the object, using (append) and (entmod). Here we show an example of a LISP defined function that will create the xdata list based on any type of extended entity data that you require. The program can be modified to include any of the dxf group codes, for any application and data structure that you might need.

Our defined function is called ADDXDAT, and it performs a subset of the tasks executed by the Autodesk program XDATA.LSP, which is a sample program of

AutoCAD used to create any extended entity data. ADDXDAT is intended for adding attribute information to objects, but is also easier to read because it is simpler than XDATA.LSP. The program creates an xdata list assigned to the symbol 'xdat,' and begins by getting the application name from the user. The application name must be registered before you can modify any object by adding the xdata to it. The list 'xdat' groups all of the xdata together inside the "{" and "}" parentheses, and our first assignment is sets the opening brace of this pair with (setq xdat (list (cons 1002 "{"))) as shown

```
  ;;; ADDXDAT --  A function to create an xdata list
  (defun addxdat (/ appname xdat more xdname xdtype xdval item)
     (setq appname (getstring "\nApplication ID name: "))
     (setq xdat (list (cons 1002 "{")))
  ;;;  The major segment of the function is a while loop, which tests
  ;;;  that you want to add 'more' xdata, so we set 'more' to 't':
     (setq more t)
     (while more
  ;;;  The xdata is defined as an item name, an item type, and value:
        (setq xdname (getstring "\nItem name <exit>: "))
  ;;;  If the default 'exit' is used, the function bypasses the 'if'
statement
        (if (/= xdname "")
           (progn
  ;;;  The 'item name' is added to the 'xdat' list with the correct
  ;;;  dxf code '1000'
              (setq xdname (cons 1000 xdname))
              (setq xdat (append xdat (list xdname)))
  ;;;  The 'item type' is forced to be String, Distance, Integer, or
Real:
              (initget "String Distance Integer Real")
              (setq xdtype (getkword
                 "\nItem type String/Distance/Integer/Real: "))
  ;;;  The user now enters the 'item value.'
              (setq xdval (getstring "\nItem value: "))
  ;;;  The dotted pair of the dxf code and the item value is formed
  ;;;  depending on the item type using the (cond) function
              (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
  ;;;  The dotted pair is now added to the list of 'xdat'
              (setq xdat (append xdat (list item)))
  ;;;  If an item name was given, we set 'more' to 't'
              (setq more t)
           ); progn
  ;;;  Or else, we set 'more' to nil, and the 'if' statement is
completed
           (setq more nil)
        ); if
  ;;;  That also ends the 'while' loop:
     ); while
  ;;;  Finally, we add the closing "}" to the end of  'xdat,'
     (setq xdat (append xdat (list (cons 1002 "}"))))
  ;;;  and we add the (-3  ("application-name"..  to the front of
'xdat':
     (setq xdat (list -3 (cons appname xdat)))
  ); end of defun
```

The function contains a (cond) section, which constructs the appropriate dxf dotted pair depending on the intended type of the data. Note that the item value 'xdval' is supplied as a string with the statement **(setq xdval (getstring "\nItem value: ")).**

Now, depending on the value of 'xdtype,' the 'xdval' is either left as a string, converted to a real (floating point) number, or converted to an integer at the time the dotted pairs are constructed. For example, one of the (cond) statements is **((= xdtype "Distance") (cons 1041 (atof xdval)))**, which use (atof) for the type conversion.

To use ADDXDAT, we have an example of a rectangular end plate for a heat-exchanger. Figures 7.1 and 7.2 show different views the end plate that is intended to support a number of tubes. The extended entity data has attributes for the plate-type, number of tubes, material, length, height, and hole-diameter.

The xdata is added by executing the function ADDXDAT with the following command/prompt sequence:

```
  Command: (addxdat)
  Application ID name:  Heatext
  Item name <exit>:  Plate-type
  Item type String/Distance/Integer/Real:  s
  Item value:  Rectangular
  Item name <exit>:  Number-of-tubes
  Item type String/Distance/Integer/Real:  i
  Item value:  18
  Item name <exit>:  Material
  Item type String/Distance/Integer/Real:  s
  Item value:  Steel
  Item name <exit>:  Length
  Item type String/Distance/Integer/Real:  r
  Item value:  8
  Item name <exit>:  Height
  Item type String/Distance/Integer/Real:  r
  Item value:  4
  Item name <exit>:  Hole-diameter
  Item type String/Distance/Integer/Real:  r
  Item value:  .75
  Item name <exit>:
  (-3 ("Heatext" (1002 . "{") (1000 . "Plate-type")...
```

The program ABLOCK.LSP from lesson six can be used with the (addxdat) function to make an anonymous block and add xdata to it. First the LISP programs are loaded, and an application name is registered with

```
  Command:  (load "/acad12/lisp/addxdat")
  (ADDXDAT)
  Command:  (load "/acad12/lisp/ablock")
  C:ABLOCK
  .
  .
  Command:  (regapp "Heatext")
  .
```

Next, the plate of figure 7.1 is used to create an anonymous block with ABLOCK, and the insert entity list is assigned to 'elist.' The extended entity data is assigned to 'xdat', before being appended to 'elist', and finally the anonymous block is modified and updated with the following command/prompt sequence:

```
Command:  ablock
.
.
Command:  (setq elist (entget (entlast)))
Command:  (setq xdat (addxdat))
.
.
Command:  (setq newlist (append elist (list xdat)))
Command:  (entmod newlist)
Command:  (entupd (entlast))
```

Remember to register the application name, otherwise you will get the error message 'Invalid application name in 1001 group' when you try to (entmod newlist).

The distances (length, height, and hole-diameter) were associated with the group code 1041, which can be scaled with the parent entity, so if you copy the end-plate and scale it, these values will also be scaled automatically. This makes a much more powerful attribute system than the regular AutoCAD attributes, which are not scaled with the object to which they are attached.

RETRIEVING EXTENDED ENTITY DATA

You need to use the application name to get the xdata list, as in

(setq xlist (entget (car (entsel)) '("Heatext")))

Then you can get a list of the xdata with

(setq xd (assoc -3 xlist))

followed by

(foreach x (cdadr xd) (print x))

which produces the list:

```
(1002 . "{")
(1000 . "Plate-type")
(1000 . "Rectangular")
(1000 . "Number-of-tubes")
(1070 . 18)
(1000 . "Material")
(1000 . "Steel")
(1000 . "Length")
(1041 .  8.0)
(1000 . "Height")
(1041 . 4.0)
(1000 . "Hole-diameter")
(1041 . 0.75)
(1002 . "}")
```

The (member) function is useful for retrieving individual values from an xdata list that has its values preceded by a label as in the above example. To obtain the material of the plate and the hole diameter, you can use

(cdadr (member '(1000 . "Material") (cdadr (assoc -3 xlist))))

(cdadr (member '(1000 . "Hole-diameter") (cdadr (assoc -3 xlist))))

which return "Steel" and 0.75 respectively.

Figure 7.3 shows the text screen with the lists produced by !xlist, !xd, (cdadr (member '(1000 . "Hole-diameter")(cdadr (assoc -3 xlist)))), and (cdadr xd).

RESTRICTIONS ON THE EXTENDED ENTITY DATA

The amount of extended entity data is currently limited to about 16 kilobytes (16,383 bytes), so your programs that attach xdata to objects should check that there is enough room for the xdata, particularly if you are adding xdata to objects that already contain xdata.

There are two AutoLISP functions (xdsize) and (xdroom) that return the amount of xdata in a list of extended entity data and the amount of remaining space available for more xdata in a given entity. The format of these commands is:

(xdsize *list*) and **(xdroom *ename*)**

where 'list' is a list that could be returned by (assoc -3 elist), and 'ename' is the name of an entity containing xdata.

If you are adding extended entity data to objects that already have xdata, then you should check the size of the new xdata with (xdsize), and compare this to the available room to add xdata to the old xdata, using (xdroom). For instance, in our above example, if you want to add more xdata to an object that has an application ID of "HEATEXT", you might get the entity name and dxf list with (setq ename (car (entsel))), and (setq elist (entget ename '("Heatext")), then create some new xdata by making a list of the form ((1002 . "{")(1000 . "String-name")(1042 . "Item-value")(1000 . "}")), where the "String-name" and "Item-value" are your new attributes. If this new data is given the symbol 'newxdat', it would be appropriate to include statements such as:

```
(if (> (xdroom ename) (xdsize newxdat))
(progn
...statements to attach the newxdat to the old xdata..
); progn
(prompt "\nNot enough room for the new xdata...")
); if
```

Note that you will need to add the new xdata inside the (-3 ("application-id"...)) section of the xdata. This lesson's homework assignment will deal with adding this type of new xdata.

Homework:

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

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

In lesson eight, the final one of this series, "More Programming Techniques", you will learn some of the methods and techniques that can make your programs more user friendly and more fool-proof. A method of using the (set) function for simplifying the job of setting and resetting system variables is shown, as well as a technique for testing for open files in your error handler. You will also learn how to make your functions 'recursive', and when recursion can help and hinder. Homework assignments will test your understanding of the lesson..

More Programming Techniques for AutoLISP

Lesson Eight in the CADalyst/University of Wisconsin Advanced AutoLISP Programming course shows you how to make your programs more user friendly.

Anthony Hotchkiss, Ph.D, P.Eng.

About this course.

Advanced AutoLISP Programming is an eight-lesson independent study course, originally published in monthly installments in CADalyst magazine. Before you begin this course, it is assumed that you have reached a basic level of AutoLISP programming, equivalent to completing the preceding 'Introduction to AutoLISP Programming' course which was originally published in CADalyst, and is now available as an independent study course from the University of Wisconsin-Madison. On successfully completing an intermediate and a final mail-in project, registered students will receive certificates of completion and 5.0 CEU's (continuing education units).

For further information, contact:

Steve Bialek
Engineering Professional Development
University of Wisconsin-Madison
432 North Lake Street
Madison, Wisconsin 53706
(608) 262-1735

LESSON EIGHT -- MORE PROGRAMMING TECHNIQUES

In this, the final lesson of the current series, you will learn some programming ideas that contribute to user friendliness and to programming elegance. A method of using the 'set' function for simplifying the job of setting and resetting system variables is shown, as well as a technique for testing for open files in your error handler. You will also learn how to make your functions 'recursive', and when recursion can help and hinder. Homework assignments will test your understanding of the lesson.

SETTING SYSTEM VARIABLES

In one of the study guides that accompany this course, I suggested a method of setting system variables by using a defined function (setv) that takes the system variable and the new value as arguments, as in:

(setv "CMDECHO" 0)

The defined function (setv) creates a symbol (variable name) to store the old value of the system variable, and assigns the new value to the system variable. The conventional way to do this is to use:

```
(setq oldcmd (getvar "CMDECHO"))
(setvar "CMDECHO" 0)
```

for each system variable to be set. Using (setv) eliminates the need for the first of these statements, and is quicker, more convenient, and quite an elegant use of the (set) function.

The (set) function assigns an expression to a quoted symbol, and returns the value of the expression. The format is (set 'x 3.0) where 3.0 is the expression that is returned, and to which the quoted value of x is set. Now consider the expression (read "symbolname"), which returns SYMBOLNAME, not as a string, but as a symbol (in other words, there are no double quotes in the returned value of the expression.)

Try entering such an expression in the (set) function at the command/prompt, as in the following sequence:

```
Command:  (set 'x (read "mysymbol"))
MYSYMBOL
Command:  !x
MYSYMBOL
```

Here, it is clear that x has the value of MYSYMBOL. If you now use (set) to set a value for x without quoting the x, the symbol MYSYMBOL will be set to the value you tried to assign to x, as follows:

```
Command:  (set x 2.0)
2.0
Command:  !x
MYSYMBOL
Command  !mysymbol
2.0
```

You see that x still has the assigned value of the symbol MYSMBOL, but MYSYMBOL now has the value of 2.0. This is called assigning a value indirectly, and is a unique feature of the function (set). We can use this property to indirectly capture the old value of any system variable, as shown in the defined function (setv):

```
;;;SETV   A function to define system variables
(defun setv (systvar newval)
   (set 'x (read (strcat systvar "old")))
   (set x (getvar systvar))
   (setvar systvar newval)
); setv
```

SETV takes the two arguments 'systvar' and 'newval', where systvar is the name of a system variable and newval is the value to be assigned to that system variable. Note the use of the (set 'x...), in which x takes the value of the symbol with a name that is the concatenation of the system variable name and 'old'. Next the current value of the system variable is assigned indirectly to the new symbol. For instance, suppose the initial value of "CMDECHO" is 1, then the symbol CMDECHOOLD would be assigned the value 1 by this process.

SETV is called with the 'setting' defined function:

```
;;; SETTING  A function that sets system variables
;;;                     by calling (setv)
(defun setting ()
   (setq oerr *error*)
   (setq *error* err)
   (setv "CMDECHO" 0)
   (setv "BLIPMODE" 0)
); end of setting
```

Each time setv is called, the global variable 'x' is set to a new symbol name. The symbol names are assigned values indirectly, and they are also global variables. Later on in the program, when the system variables are to be restored to their initial values, the symbol names must be regenerated by defining and calling a defined function (rsetv), which takes a single argument, the system variable string, as in

(rsetv "CMDECHO")

The function (rsetv) is simply:

```
;;; RSETV  A function to reset system variables
(defun rsetv (systvar)
   (set 'x (read (strcat systvar "old")))
   (setvar systvar (eval x))
); rsetv
```

Here, because we have only one 'x' symbol, it must be re-assigned for each system variable, so (set 'x...) is used again in (rsetv), followed by an assignment of the system variable systvar to the evaluated 'x'

The (resetting) function calls (rsetv):

```
;;; RESETTING  A function to reset system
;;;                       variables by calling (rsetv)
(defun resetting ()
   (rsetv "CMDECHO")
   (rsetv "BLIPMODE")
   (setq *error* oerr)
); end of resetting
```

If you use all of the above functions in your 'prototype' or 'template' LISP program, the job of setting and resetting system variables is simplified.

TESTING FOR OPEN FILES

During any AutoCAD session, a number of files are automatically opened and remain open until the drawing session ends. If you open files in your programs, they contribute to the number of currently opened files, and this number is limited by the FILES statement in your CONFIG.SYS system file for DOS. Increasing this number enlarges the memory space required to maintain a table of open file information, so you should plan to use some kind of 'optimum' number, such as 40, which allows AutoCAD to have about 20 files open simultaneously.

There is no direct way to check for open files, because the file handle that is created by the (open) function remains in existence even when the file is closed. For this reason, statements such as (if f1 (close f1)) will not be effective if the file is already closed. This type of statement can be effective with good organization, as we see in the following examples.

One possibility for testing for open files is to create a symbol as a flag whenever you open a file, so you might write:

(setq f1 (open "MYFILE.TXT" "w") ftest1 "is_open")

Then you could end your programs with a series of statements of the form:

(if (= ftest1 "is_open") (close f1))

If you close a file elsewhere in your program, you might add the statement:

(setq ftest1 "is_closed")

Another approach is to have a file numbering scheme that always uses 'fnnn', where 'nnn' is a number attached to the letter 'f'. You could then close all files by calling a defined function such as:

```
(defun closeall (/ n)
   (setq n 0)
   (repeat numfiles
      (set 'y (read (strcat "f" (itoa (setq n (1+ n))))))
      (if (eval y) (close (eval y)) )
   ); repeat
); closeall
```

Here, the number of iterations of the repeat loop is controlled by the symbol 'numfiles', which could be set either as a counter each time a file is opened, or as a constant maximum number. The (set) function has been used to attach the number portion of the file name to the 'f' prefix, and provided that the numbering scheme is adhered to, this should do the job of ensuring that all files are closed at the end of your program. If a file has been created and closed, this approach will not work because the error handler will be called with a message 'file not open' and the repeat loop will terminate.

RECURSION

When a defined function calls itself from within itself, it is said to be 'recursive'. A very simple example of recursion is when a function calls itself so that it executes as an infinite loop, as with

```
;;; MYFUN  A function that calls itself
(defun myfun ()
   statement 1...
   statement 2...
   .
   (myfun)
); myfun
```

Recursion in a purely mathematical sense is when each item of a list (or polynomial series) is defined by applying a formula to preceding terms in order to generate successive terms. I had an occasion to apply a fibonacci series in the generation of a pattern. The fibonacci series is one in which the next term is produced by adding together the two previous terms in the series, where the first two terms are 1 and 1, so the series looks like

1 1 2 3 5 8 13 21 34 55 89... etc.

I wrote a function to generate the nth term in the series using two approaches, (a) without using recursion, and (b) using recursion. The results are shown in FIB1 and FIB2:

```
; FIB1  A function to return the nth term in the series
;   1, 1, 2, 3, 5, 8, 13, 21...... (fibonacci series)
(defun fib1 (n)
   (setq x 1)
   (setq x1 0 x2 0)
```

```
      (repeat n
         (setq x2 x)
         (setq x (+ x1 x2))
         (setq x1 x2)
      )
   ); end
```

FIB1 starts the series with zero in order to make the nth term correct, because the 2nd term is the sum of zero and one. The function is somewhat crude but effective, and the results are intuitively correct, adding x1 and x2 together to produce x at every evaluation of the repeat loop.

FIB2 is a more mathematically elegant solution because it acknowledges the fact that the first two terms are 1, so if n is equal to either 1 or 2, the result for x is 1. The recursion is in the statement where the function calls itself using n-1 and n-2 as its arguments instead of n. The computation stores each successive result until the value of n-1 is less than or equal to 2, in which case the solution to (fib 2) or (fib 1) is simply 1, and the function does not continue to call itself. At that point, all of the stored values are added together to give the final answer. In the case of the fibonacci series, each of the successive solutions is 1, so the final result is the addition of a set of 1's.

```
   ; FIB2  A function to return the nth term in the series
   ;   1, 1, 2, 3, 5, 8, 13, 21...... (fibonacci series)
   (defun fib2 (n)
      (if (or (= n 1) (= n 2))
         (setq x 1)
         (setq x (+ (fib2 (- n 1)) (fib2 (- n 2))))
      ); if
   ) ; end
```

Figure 8-1 shows the progress of FIB2 for the fifth term. In the figure, the function is called FIB, so if n is greater than 2, as in (FIB 5), then x is assigned to be equal to (FIB 4) + (FIB 3) etc. The levels of computation in which the individual results are stored are called a 'stack', and the words for adding to and taking from the stack are 'pushing' and 'popping' respectively. Early versions of AutoCAD allocated memory for LISP as a LISPheap and a LISPstack, until the extended memory versions were released. The stack is really just an area of memory which must be large enough to store all of the intermediate operations and results until the recursion ends, at which point the summation process is completed.

If very high numbers are used, the recursive version of the FIB function takes longer to execute than the non-recursive version because recursion needs a stack of the intermediate values being processed, which means using up a lot of memory.

I used a fibonacci series to create a pattern of a pine cone, shown in figure 8-2. The number of 'leaves' in each successive circular array of the pine cone grows according to the fibonacci series, starting with 3 leaves as shown. In addition, the lengths of the leaves are also in the ratio of the fibonacci series. For smaller numbers as required by this pattern, the recursive technique is well suited.

RECURSION AND AUTOLISP

We mentioned stacks in connection with recursive functions, but the use of stacks is not just for functions that call themselves from within. Any time that AutoLISP needs to store intermediate values in order to evaluate nested functions, those values are stored in a stack (the LISPstack), and the way that AutoLISP works is by a process of recursion in the sense of evaluating nested functions. Whenever you create a defined function, all of the statements in the function are nested inside the (defun), and that implies a recursive operation in which a stack space is used.

All of this reinforces the idea that you should write your programs in short modules for efficiency in memory usage as well as in testing and debugging.

A LISP DEVELOPMENT MENU

I have revised my LISP development menu macros to include the file dialogue box, called with (getfiled) that contributes greatly to user friendliness in developing AutoLISP programs. My LISP DEVELOPMENT MENU is:

```
  .
  .
  [->Lisp development]
    [Edit file]^C^C^P+
  (setq fn (getfiled "File to edit" "/ACAD12/LISP/TEST" "lsp" 8))+
  (command "shell" (strcat "edit " fn))(princ)^P
    [Load & Run TEST.LSP]^C^C(load "/acad12/lisp/test");(test);
    [Load a lisp program]^C^C^P+
  (setq fn (getfiled "Select File" "/ACAD12/LISP/" "lsp" 8))(load
fn);^P
    [Copy lisp file]^C^C^P+
  (setq ofn (getfiled "File to Copy" "/ACAD12/LISP/TEST" "lsp" 8))+
  (setq nfn (getfiled "New File Name" "/ACAD12/LISP/TEST" "lsp" 1))+
  (command "shell" (strcat "copy " ofn " " nfn))(princ);^P
    [->List and print]
      [Print file]^C^C^P+
  (setq fn (getfiled "File to print" "/ACAD12/LISP/TEST" "lsp" 8))+
  (command "shell" (strcat "prnt " fn))(princ)^P
      [List lisp directory to screen]^C^Cscript
/acad12/scripts/lst2scr
      [<-<-List lisp directory to printer]^C^Cscript
/acad12/scripts/lstlsp
  [->ADS and C development]
  .
```

The menu is shown cascaded in figure 8-3, and the result of choosing the item 'Edit file' is the file dialog box shown in figure 8-4. The function (getfiled) has 4 arguments for the file prompt, the file prefix, the file extension, and a bit-code 8 (flag) that means 'perform a library search for the filename entered. If you select a file with the cursor from the list of files, the filename without any extension is displayed in the file edit box. Other possible flags are:

1, used when you want to create a new file that may overwrite an existing file. If you select a file from the displayed list a warning message is displayed about overwriting an existing file.

2, The 'Type it' button on the file dialogue box is disabled and cannot be selected. For the other flag settings, selecting the 'Type it' button makes the dialogue box disappear, and (getfiled) returns 1. In all other cases (getfiled) returns the full path name of the selected file.

4, Lets you enter any file extension or no file extension at all. A selected file is displayed as the filename and the extension in the file edit box, but the path name is not displayed.

In all cases, if a file is selected from the displayed list, (getfiled) returns the full path name of the file in the format that uses 2 backslashes (\\) for the directory delimiters, and is therefore suitable for any AutoCAD command. For editing AutoLISP programs, I use the regular DOS 5 editor 'edit', which takes the filename as its argument, so the command function is (command "shell" (strcat "edit " fn)), where fn is the name of the file returned by (getfiled). You can replace the call to 'edit' with your own editor if it accepts the filename as an argument.

When I develop a new LISP program, or if I work on updating an old program, I normally use the temporary name TEST.LSP, then if I name the main defun (test), I can load and run the program with a single pick of the next menu item [Load & Run TEST.LSP]. For loading programs other than TEST.LSP, I use (getfiled) with (load fn) as shown in the [Load a lisp program] macro.

When the test program is working satisfactorily it can be copied to another name with the next menu macro [Copy lisp file] which makes 2 calls to (getfiled), one for the file to be copied, and the other for the new file. In both cases the default file name is TEST, because the main reason for copying files is to continue development on them using the automatic load and run feature with the name TEST. Note that the flag for the new file is 1, which is necessary if you want to create new file names.

The section of the menu that deals with printing and listing files uses a special program, written in the C language and compiled and linked as PRNT.EXE. The call to the printer is in the command function (command "shell" (strcat "prnt " fn)). The executable file PRNT.EXE converts any ASCII text file into Postscript format, ready to print on my Postscript printer. I use many Windows applications in my course development, and Postscript is the standard format for those applications, so to save the time taken to reset my printer for DOS printing, the PRNT.EXE program was written by Troy Jacobson, an undergraduate at UW-Madison. An alternative to the command for non-Postscript printers is (command "shell" (strcat "copy " fn " " "LPT1")) if your printer is connected to the first parallel port of your computer.

In all of the above "shell" macros, remember to add a space to the concatenated arguments of the DOS commands, otherwise there will be no delimiters and DOS will not recognize the commands. For the final pair of macros I call script files, but you might also invent your own directory listing macros in LISP. The straight forward approach of using "shell" followed by the DOS command is not reliable because the DOS switch uses a forward slash, and that is interpreted as a pause for input even though it is supplied as an operating system command. The use of DOS commands in script files eliminates such problems, as do the use of LISP statements as shown in the other menu macros.

An update to lesson 6 "AutoCAD Anonymous Blocks"

I would like to thank Mr. Jon Fleming of Massachusetts, who pointed out that the use of a function to find the name of the next anonymous block is not necessary because the (entmake) function returns the name of the block that it makes at the "ENDBLK" statement. I had previously used the defined function (uname) to keep track of the anonymous block names under a variety of conditions such as redefining blocks, and undoing part or all of a drawing session where anonymous blocks are created.

As Jon suggested, the ABLOCK.LSP works just as well with:

```
(defun C:ABLOCK (/ ins_pt name ss1 i n)
   (setq ins_pt (getpoint "n\Insertion point: "))
   (entmake (list '(0 . "BLOCK") '(2 . "*U") '(70 . 1) (cons 10
ins_pt)))
   (prompt "n\Select entities: ")
   (setq ss1 (ssget) i (sslength ss1) n -1);  Note -1 instead of (-
1)
   (repeat i
      (entmake (cdr (entget (ssname ss1 (setq n (1+ n))))))
   )
   (setq name (entmake '((0 . "ENDBLK"))))
   (entmake (setq t1 (list '(0 . "INSERT") (cons 2 name) (cons 10
ins_pt))))
   (command "MOVE" "L" "" ins_pt pause "REDRAW")
)
```

This is the last lesson in the series of this correspondence course. I hope it has been as much fun for you as it has for me. If you have any questions or comments, please call Tony Hotchkiss at 608-262-8219, or fax 608-263-3160, or email to *hotchkis@macc.wisc.edu*

Homework:

The final homework questions are quite trivial, but need to be considered for successful implementation of the ideas in the lesson.

1. Why do you need to include the (eval) function in the resetting function to restore the system variables to their initial values. Why not use (setvar systvar x) instead of (setvar systvar (eval x))?

2. Why does the filename returned by (getfiled) in the form C:\\ACAD12\\LISP\\TEST.LSP work with the shell command (command "shell" (strcat "edit " fn)) as shown in the menu macro? If you typed this format at the DOS prompt after shelling out of AutoCAD, it would not be accepted.

Finally, if any of you have different ways of dealing with the 'open' files problem, I would like to hear from you. That is, as an alternative to using CHKDSK /F after ending the AutoCAD session!