An Introduction to AutoLISP

Lesson One in the CADalyst
University of Wisconsin Introduction to AutoLISP Programming course covers AutoLISP data types and program execution

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

About this course.

Introduction to AutoLISP Programming is an eight-lesson independent study course, originally published in monthly installments in CADalyst magazine. Readers may optionally register for the course in order to receive a copy of the printed course outline, a study guide containing additional AutoLISP examples, reference materials, and solutions to the practical assignments. On successfully completing an intermediate and a final mail-in project, students will receive certificates of completion and 4.0 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 - AN INTRODUCTION TO AUTOLISP

This months lesson introduces you to AutoLISP as a set of functions and other data types, and explains a variety of ways in which AutoLISP expressions and programs may be executed, or 'run' during your AutoCAD drawing session. You will see that with just two AutoLISP functions, you can begin creating programs to draw some basic geometric shapes. The concepts of local and global variables are introduced, and the lesson ends with some homework questions to test your understanding of the text.

WHAT IS AUTOLISP?

AutoLISP is a programming language that allows you to customize AutoCAD to meet your own unique requirements. It is a subset of the LISP (List Processor) programming language, which is used in applications of artificial intelligence and expert systems. Many functions have been added to the LISP program in order to interface AutoLISP directly to AutoCAD, and you will see that some AutoCAD commands have been retained as AutoLISP functions.

AutoLISP is an 'interpreted' language rather than a 'compiled' language, which means each program statement is translated and evaluated as the program is read into the computer. This differs from a compiled language for which a complete translation into machine code is made prior to a program 'execution' step.

Program statements in AutoLISP are sets of functions and other data that are used with the function contained within parentheses. An example of such a function and its other data is:

(command "line" pt1 pt2 pt3 pt4 "close")

Here, the function is 'command', which precedes the familiar AutoCAD 'line' command and its sub-command 'close'. The content of the parentheses is a self-contained statement which causes a line to be drawn between the four points pt1 through pt4, and back to pt1. You must have already defined the meaning of the four points so that AutoCAD can use them to draw the lines.

AUTOLISP DATA TYPES

Functions such as 'command', are termed 'built-in' or 'internal' functions. They are also known as subroutines, or simply 'subrs'. The subroutine terminology will be more familiar to those of you who have programmed in other languages such as FORTRAN. The 'other data' that appear with the function are called the 'arguments' of the function.

Some AutoLISP functions (subrs) require that arguments be supplied in a particular order, and others may include optional arguments, or no arguments at all. The complete set of functions is listed in the AutoLISP Programmer's Reference, published by Autodesk, Inc., and supplied to purchasers of AutoCAD.

The data types that AutoLISP recognizes are:

```
Functions (subroutines, subrs)
    Examples are +, -, *, \, command, princ, setq
    (note that the arithmetic operators are treated as functions by AutoLISP)

Arguments of the functions
    Examples are variables (also known as symbols)
        constants (real (decimal) or integer)
        character strings
        file names (descriptors)
        AutoCAD entity names
        AutoCAD selection sets
        external functions (supplied by the AutoCAD Development System)
```

AutoLISP data is supplied in the form of lists, where the items of the list can themselves be lists. For instance, in our example of the command function

```
   (command "line" pt1  pt2  pt3  pt4  "close")
```

the seven items contained in the parentheses form a list. Four of these items are the variables (symbols) pt1 through pt4 representing AutoCAD points. Now, since the points contain x, y and z coordinates, they are also lists. The other two arguments, "line" and "close" are examples of character strings.

AutoLISP Atoms

A single item of a list, which is not itself a list, is called an 'atom' in AutoLISP terminology. According to this definition, atoms are the fundamental indivisible particles of AutoLISP. Atoms are individual elements such as symbols, numbers and subrs, and a complete set of the atoms can be displayed on your screens by entering *!atomlist* at the AutoCAD command prompt. We will discuss the atomlist more fully later in this series.

EXECUTING AUTOLISP

AutoLISP input can be entered by typing in expressions in an AutoCAD session (at the command prompt), by reading in the expressions from an ASCII file, or by reading from a string variable which may represent a complete program.

Execution by typing in an AutoLISP expression

Try typing in the following at the AutoCAD command prompt:

```
   (command "line" "2,3" "4,5" " 6,3" "close")
```

Don't forget to include the parentheses, or AutoCAD won't recognize that this is an AutoLISP expression! Your AutoCAD screen should now display a triangle. In this case, the variables used for points have been replaced by character strings enclosed by double quotation marks**.** Now erase the triangle and re-type the expression with "c" replacing the "close" element. The result is the same, because the command function allows regular AutoCAD commands and subcommands to be entered.

Execution by reading an expression from an ASCII file

Now, shell out of AutoCAD and create an ASCII file (this can be done with EDLIN or the DOS 5 EDIT text editor) named PROG1.LSP that contains only the above AutoLISP expression. To execute your first AutoLISP program, erase the previous triangle, then use the following LOAD function at the command prompt:

```
   Command:  (load "prog1")
```

again, don't forget the parentheses and the double quotes, and you should see the triangle as before. Note here that the program executes and draws the triangle immediately on being loaded, and this is generally true for AutoLISP programs that contain only subrs (built-in functions).

AutoLISP execution from a file containing defined functions

In addition to subrs, it is also possible to define your own functions for AutoLISP. Such functions are called defined functions to distinguish them from the built-in functions. If a program contains a defined function, when the program is loaded, the name of the defined function must also be entered in parentheses so that it can be executed.

Try this by changing your prog1.lsp file to read the following:

```
(defun test1 ()
(command "line"  "6,3"  "8,5"  "10,3"  "c")
)
```

Don't forget to include all the parentheses. The AutoLISP subr to define functions is **(defun .....)**, and in this case the name of the defined function is *test1*. Now, if the AutoLISP file is named *prog1.lsp*, the following sequence is followed:

```
Command:  (load "prog1")
TEST1
Command:  (test1)
```

Here, the word TEST1 appears immediately when the program is loaded by the first statement, but the program is not executed until you enter '(test1)', the name of the defined function. We will see more of the **defun** function later, and the significance of the empty pair of parentheses will be made clear in the section on 'global and local variables'.

AutoLISP execution with 'C:'

If the AutoLISP defined function name begins with C:, then after loading the program, the defined function becomes a regular AutoCAD command that can be executed like other AutoCAD commands at the command prompt by entering its name without parentheses. Try editing the prog1.lsp file to read:

```
(defun C:TEST1 ()
(command "line"  "6,3"  "8,5"  "10,3"  "c")
)
```

then use the following sequence of commands:

```
Command:  (load "prog1")
C:TEST1
Command:  test1
```

The name C:TEST1 is automatically written by AutoCAD when the program is loaded. At this point, the new command 'test1' may be entered at any time. An AutoLISP program containing one or more 'defun' defined functions only needs to be loaded once in any drawing session.

AutoLISP execution from menu macros

AutoLISP expressions can be contained in menu macros to be executed when the menu label is selected. If you examine the AutoCAD release 11 menu file ACAD.MNU, you will notice that it contains many AutoLISP programs and expressions.

I use a personal menu file that I can select from a modified form of the standard ACAD.MNU. A menu item of the form **[aux-menu]^C^Cmenu;aux;** is placed at a convenient position (I use the 'File' POP menu) in the standard menu, then I place a similar command in my 'AUX.MNU' file to allow easy return to the standard menu. My preference is to leave the standard menu alone, except for this small change so that the regular AutoCAD menus are always available on demand, and any new material is then placed into the aux.mnu menu.

Here is a sample of my new POP2 menu which you can use for a basic AutoLISP development environment:

```
***POP2
[Set-ups]
   |
   |
[Lisp program development]^C^C$p2=p22  $p2=*
   |
   |

**p22
```

```
[Set-ups]
[Edit-LSP]^C^Cshell ed C:/acad11/lisp/test.lsp;graphscr
[Test-LSP]^C^C(load "/acad11/lisp/test");(test);
[Print test.lsp]^C^C(load "/acad11/lisp/printest");(printest);
[~--]
   |
```

In the above, when the POP2 menu 'Set-ups' is selected, one of the choices is 'Lisp program development', and this displays a new page that includes items to edit a file named **test.lsp** in the subdirectory **lisp** which is contained in the AutoCAD directory, **acad11**. The call to the editor in this case is ed, because that is the name of the editor I use. You may substitute edlin or edit or the name of your favorite text editor (provided that it produces ASCII output) in place of this.

This menu macro only edits the file called test.lsp, which is the name I give to the current LISP program that I am developing. To edit any other file, most text editors allow you to choose the filename after the editor has been entered, so you could simply leave out the filename until the editor is running. Edlin requires a filename to be supplied on execution, so simply give the shell command and issue the edlin command from the operating system.

The use of separate directories for lisp programs is a good management procedure that I recommend. After the lisp file has been created or edited, it can be immediately tested by the Test-LSP macro, which loads and executes test.lsp assuming a defined function named 'test' has been created.

The 'Print test.lsp' option calls a LISP program that prints the test.lsp file. The AutoLISP program to do this has been adapted from the Autodesk Inc. 'Fprint.lsp' program, which is freely available to you for modification (with the appropriate acknowledgements to Autodesk). We will examine this program in detail later in the course. In the meantime, you may use your text editor or the DOS print command to print your program.

GLOBAL AND LOCAL VARIABLES

The default mode of AutoLISP is to define all variables to be 'global', which means that the variables are generally available not only when the program containing them is executed, but also after the execution is completed. This means that other programs you write will have access to variables that were defined by previous programs. It is possible to see the values of global variables by typing their names preceded by ! at the command prompt. For instance, if pt1 is a variable that represents a point whose coordinates are 2.5,3.0,0.0, then typing !pt1 will display the three coordinate values.

Global variables appear in the ATOMLIST, and take up memory in the computer. For these reasons, and because you will probably not want variables such as PT1 to take values left over by previous programs, it is wise not to have global variables hanging around at the end of execution.

In order to avoid global variables, you may declare the program variables to be 'local' to a particular defined function by adding their names, preceded by a slash mark and a space (/) in the parentheses which are supplied for that purpose in the defun statement, thus:

(defun test2 (/ pt1 pt2 pt3 pt4)...........)

will make all of the point variables local. Note here that other variables may be defined as arguments of the defun function, appearing before the slash mark, for instance,

(defun test3 (a b / pt1 pt2 pt3 pt4).............)

means that the two arguments a and b must be supplied to the defined function test3 before it can be executed, and instead of executing with **(test3)** as before, you must supply values, such as **(test3 2.2 3.0)** so that the variables a and b can take any value you wish, in this case, 2.2 and 3.0 respectively. We will see examples of programs using global and local variables in the next lesson in this series.

Home work questions:

1. Why are the closing parentheses in the last 2 defined function examples on the same line as the opening parentheses, instead of on a separate line as in our previous defined function examples?
2. Are the arguments a and b in the last defined function example local or global variables?

3. How would you draw circles and arcs with AutoLISP expressions? Use the 'command' function to draw a circle with the '2-POINT' method, with the points at absolute coordinates 3,4 and 6,4 respectively. Test this on your computer at the AutoCAD command prompt.

4. Using only the functions 'defun' and 'command', write a program to define a function which draws a triangle and a circumscribing circle (through the vertices of the triangle). Make the 3 points of the triangle at 3,4; 6,4; and 5,6 in x,y locations respectively.

In lesson two of this series, "AutoLISP program evaluation and control", you will learn the formats of AutoLISP functions and arguments, with examples of argument types. Alternative programming methods for nesting functions are covered with examples of how to make your programs clear and easy to read. You will also learn how and when AutoLISP evaluates expressions, and how you can control the evaluation by using special 'quote' functions and other special characters. Finally, you will see how the basic assignment function allows you to create variables in your programs.

AutoLISP program evaluation and control

Lesson Two in the CADalyst
University of Wisconsin Introduction to AutoLISP Programming course covers function formats, parenthesis control and basic error checking.

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

About this course.

Introduction to AutoLISP Programming is an eight-lesson independent study course, originally published in monthly installments in CADalyst magazine. Readers may optionally register for the course in order to receive a copy of the printed course outline, a study guide containing additional AutoLISP examples, reference materials, and solutions to the practical assignments. On successfully completing an intermediate and a final mail-in project, students will receive certificates of completion and 4.0 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

article.

LESSON TWO - AUTOLISP PROGRAM EVALUATION AND CONTROL

This months lesson shows you how to write some simple AutoLISP programs. Special text characters are introduced, and the format of the AutoLISP functions and arguments is explained, with examples of argument types. Alternative programming methods for nesting functions are covered with examples of how to make your programs clear and easy to read. You will see how the basic assignment function allows you to create and use variables in your programs. You will also learn how and when AutoLISP evaluates expressions, and how you can control the evaluation by using special 'quote' functions and other special characters.

Programs are controlled by parentheses rather than by lines, and you will see examples of the correct use of parentheses and double quotes. Many errors are caused by not closing quotes and parentheses, and this lesson will show you how to check for these and some other errors. We will show some example programs to illustrate these ideas, and the lesson ends with some homework questions to test your understanding of the text.

SPECIAL CHARACTERS FOR PROGRAMMING

You already know that AutoCAD menus use special characters like the semi-colon (;) or a blank space for 'return', and the carat (^) symbol for the 'control' key. AutoLISP uses the following special characters in program statements:

```
\n     means newline
\r     means return
\"     means the character "
\t     means tab
\e     means escape
\\     means the character \
\nnn  means the character whose octal code is nnn
'      means the quote function
;      precedes comments in a program (until the end of the current line)
"      means the start and end of a literal string of characters
.      A dot preceded by a space is reserved for designating 'dotted-pairs'
```

The back-slash character (\) is used as a control character for whatever follows it in a 'literal string' of characters. A literal string of characters means that the characters are merely text, not to be confused with symbols (variables). Literal strings are contained within double quotes ("). For example, consider the 'prompt' function which is used to display messages when AutoLISP is executing:

```
(prompt "\nWelcome to AutoLISP")
```

This statement has the effect of starting a new line (because of the \n character), followed by the display of the words 'Welcome to AutoLISP'. The double quotes indicate that the individual words, **welcome**, **to**, and **AutoLISP**, are not variables that can represent values, but are simply text characters to be displayed in the command area of your AutoCAD screen.

Try the following examples at your AutoCAD command prompt, but use the spacebar instead of the 'enter' or 'return' key:

```
(prompt "This is AutoLISP")
(prompt "\nThis is AutoLISP")
```

The results should look like the following:

```
Command:  (prompt "This is AutoLISP") This is AutoLISPnil
Command:
```

and:

```
Command:  (prompt "\nThis is AutoLISP")
This is AutoLISPnil
Command:
```

In the first case, the character string is written on the same line, and in the second case a new line precedes the message. In both cases, note that the word 'nil' appears after the message. This is because the AutoLISP function 'prompt' produces (or 'returns') a value of 'nil', and we will see how to deal with this unwanted text later in the series when we consider the 'print' functions.

Note also that in these examples, the spacebar was used to enter the LISP statements. What would have happened if you had used the 'enter' key instead? Try it!

You can see that using special characters like \ and " can help in writing character strings to the screen, but what if you want to use these control characters as part of the text string itself? That is why the characters \\ and \" appear in the list of special characters. It is possible that you may want to refer to DOS pathnames, in which case the back-slash character would be part of the name. An example is

```
..."\acad\lisp\myfile.lsp"...
```

which would have to be written as

```
..."\\acad\\lisp\\myfile.lsp"...
```

We will discuss the subject of file handling in lesson six of this series, and the other special characters will be introduced in appropriate places later in this series.

AUTOLISP FUNCTION FORMATS AND ARGUMENTS

Functions, also known as subroutines, or simply 'subrs' are subject to some rules and restrictions. The subrs must be contained within parentheses, and the function name must immediately follow the open parenthesis. The 'arguments' of the function (discussed in lesson one), contain the required data for the function.

Some AutoLISP functions (subrs) require that arguments be supplied in a particular order, and others may include optional arguments, or no arguments at all. The number of arguments depend on the actual function being used.

Functions are evaluated by AutoLISP, and they return certain values, or, as we have seen in the case of the 'prompt' function, they may return 'nil'. For example, consider the following functions:

(terpri) means 'terminate printing', is equivalent to a 'newline', and requires no arguments. This function returns 'nil'.

(sin *angle*) requires one argument (an angle value in radians), and returns the sine of the angle

(* *number number number* .....) may have any number of arguments, and returns the product of all the numbers. This is the multiplication function.

(setq *var1 expr1 var2 expr2* .....) is the basic assignment function which can have any number of pairs of variables and expressions. This function assigns the content of the nth expression to the nth variable, and returns the value of the last expression in the list.

(defun *name (variable list) Function_definition_statements*) the first argument is the name of the defined function, the variable list may be empty or may be any number of global and local variables (see lesson one). An unlimited number of Function_definition_statements can be included, and defun returns the name of the function.

NESTING OF FUNCTIONS

Controlling the program sequence with parentheses allows you to 'nest' parentheses inside each other, so that LISP functions can exist inside other functions. The following example will show the many different ways in which a result can be obtained to a programming problem.

Suppose we want to evaluate the formula

```
    X = -b + ( ( ((b**2)-4ac)**(.5) )/2a )
```

for the case where

```
      a = 1.5
      b = 2.0
      c = 0.5
```

we could use the following set of functions:

```
      (setq  a  1.5)
      (setq  b  2.0)
      (setq  c  0.5)
```

Here, we use the basic AutoLISP assignment function **setq**, in which only one variable and one expression is set for each 'setq' function. We can then continue with some nested functions where the multiplication function is included in the setq function:

```
      (setq  b2  (*  b  b))
      (setq  ac4  (*  4  a  c))
      (setq  a2  (*  a  2))
```

Next, we can subtract the values under the square-root sign by setting a new variable 'd':

```
      (setq  d  (-  b2  ac4))
```

Now we introduce the square root function 'sqrt', and set another new variable 'e':

```
       (setq  e  (sqrt  d))
```

We can continue with the following, setting another new variable 'f':

```
      (setq  f  (/  e  a2))
      (setq  x  (-  f  b))
```

Our final program looks like:

```
      (setq  a  1.5)
      (setq  b  2.0)
      (setq  c  0.5)
      (setq  b2  (*  b  b))
      (setq  ac4  (*  4  a  c))
      (setq  a2  (*  a  2))
      (setq  d  (-  b2  ac4))
      (setq  e  (sqrt  d))
      (setq  f  (/  e  a2))
      (setq  x  (-  f  b))
```

Note how the multiplication, division and subtraction functions are nested inside of the assignment function. This program is simply a collection of AutoLISP functions, and will execute immediately on loading with (load "filename"), where 'filename' is the name of the file ending in the file extension .lsp, as covered in lesson one of this series.

Now, because the format for setq allows us to group together any number of pairs of variables and expressions, this program could be written as:

```
      (setq  a  1.5
                    b  2.0
                    c  0.5
                    b2  (*  b  b)
                    ac4  (*  4  a  c)
                    a2  (*  a  2)
                    d  (-  b2  ac4)
                    e  (sqrt  d)
                    f   (/  e  a2)
                    x   (-  f  b)
```

Which is clearer because fewer parentheses are used and it is obvious which variables and expressions belong to each other.

This could be written in the much less pleasing form:

```
(setq a 1.5 b 2.0 c 0.5 b2 (* b b) ac4 (* 4 a c) a2 (* a 2) d (- b2 ac4) e (sqrt
d) f (/ e a2) x (- f b))
```

Another possibility is to use a nesting approach in order to set the value of x directly as in:

```
   (setq  a  1.5
            b  2.0
            c  0.5)
   (setq x (- (/ (sqrt (- (* b b) (* 4 a c))) (* a 2)) b))
```

Note that for every open parenthesis there must exist a closing parenthesis, and the evaluation of the program is controlled by the placement of the parentheses. These examples illustrate that programs can be made simple and readable, or complex and unreadable, and it is not usually worthwhile saving lines of code by making your programs unnecessarily complicated. The last example eliminates the need to invent new variables, but is somewhat difficult to check. However, it does have a certain readability which is closer in form to the original equation than the other examples.

AN AUTOLISP PROGRAM TO DRAW A BOX

AutoCAD draws points. lines, arcs and plines, but not boxes. Here we give a very simple program to draw a square box of any size. Variables are assigned in two ways: by using the setq function, and provided as an argument to the defined function. You will also see some new subrs and their usage.

The following program is annotated by using comments. Comments begin with a semi-colon, and continue until the end of the line on which they appear.

```
 ;  BOX1.LSP, a program to draw a square with origin and length of a side.
 ;
 (defun  box1  (len)    ; The variable 'len' means the length of side
   (setq pt1  (getpoint "\nEnter the origin point: ") ; The 'getpoint' function
         pt2  (polar  pt1  0.0  len) ; Polar requires an angle in radians
         pt3  (polar  pt2  (/  pi  2.0)  len)  ; 'pi' is built in to AutoLISP
         pt4  (polar  pt3  pi  len)   ; polar, base-point, angle, distance
    )          ;  End of setting variables for points
    (command  "line"  pt1  pt2  pt3  pt4  "c")  ;  Drawing the box
 )             ;  End of the 'defun'
```

To execute this program, use:

```
      (load "box1")
      (box1  nnn)
```

where *nnn* is the length of side of the box, and will be substituted for the 'len' variable.

Note that when a variable such as 'len' is supplied as an argument to the defined function, you can not make the defined function a command by naming it with the C: prefix.

When this program executes, you will be prompted to select the lower-left corner of the box by the prompt "Enter the origin point : ". You may pick a position with the cursor, (using an osnap mode or grid and snap if desired) or enter the coordinates of the point via the keyboard in regular AutoCAD format, as though you were supplying the from or to points of a line. The box will then be drawn automatically. You can draw as many boxes as you like, after the program has been loaded, by using the (box1 nnn) format.

The new functions introduced here are 'getpoint' and 'polar'. The syntax for the getpoint function is:

(getpoint *base-point prompt-string*)

The base-point and prompt-string are optional arguments to this function. The getpoint function pauses for user input of a point, and returns the point selected. If a prompt-string is supplied, it will be displayed as though it were an AutoCAD prompt. If the optional base-point is supplied (this may be pre-defined as a variable, or may be given as a list of coordinates in the appropriate format), then AutoCAD draws a 'rubber-band' line from the base-point to the current cursor position.

The format of a point in AutoLISP is a list of the coordinates of the point. After the box program has executed, enter !pt1 at the command prompt, and you will see the list of coordinates for the point 'pt1' displayed as follows:

```
Command:  !pt1
(4.48057  7.35181  0.0)
Command:
```

Note that the value of pt1 (and any other of the point variables) can only be displayed here because they are 'global' variables, and have been added to the atomlist. If these variables were defined to be local, by writing the defined function as

(defun box1 (len / pt1 pt2 pt3 pt4)...............

then entering !pt1 at the command prompt would display 'nil'.

The polar function has the following argument list:

(polar *base-point radian-angle distance*)

There are no optional arguments, and the base-point and angle are with reference to the current user coordinate system. The polar function returns a point as a list of real numbers contained in parentheses.

AUTOLISP EVALUATION AND QUOTED LISTS

The AutoLISP evaluator takes a line of input, evaluates it, and returns a result. A line of input begins with an open parenthesis and ends with the matching closing parenthesis. Any nested lines of input are evaluated and a result is returned. Results cannot be returned until a matching closing parenthesis is found, and so the evaluation returns results for the innermost nested functions first.

Generally, variables evaluate to their current values, and constants, strings and subrs evaluate to themselves. Lists are evaluated according to the first element of the list, which is normally a subr or a defined function. The remaining elements in a list are taken as arguments of the list.

Remember that a list is contained within parentheses, and the lists that we have seen so far have been either defined functions or subrs (built-in functions). Now consider the following list:

(command "line" pt1 pt2 pt3 pt4 "c")

Here, the list has a subr as its first item, so the remaining elements are taken to be arguments of the subr. The 'command' subr which we first encountered in lesson one, provides a method for submitting AutoCAD commands (in this case 'line'), subcommands ('c'), and 2D and 3D points (pt1 through pt4). The commands and subcommands are actually strings (contained in double quotes), but let us examine the point variables. We have already seen that pt1 has the format (4.48057 7.35181 0.0) which is a list of three real numbers which represent the x, y, z coordinates of a point.

It appears that the list represented by pt1 is not compatible with the criterion for evaluation of lists given above, since the first element of the list is neither a defined function nor a subr. The same problem occurs in the 'polar' list, which also uses the list of real numbers to represent the base-point. Clearly, we need a way of expressing lists which are not functions, and which are not evaluated.

How can we set the value of a point directly in the form (setq pt1 (2.0 3.0 0.0))? Try entering

(setq pt1 (2.0 3.0 0.0))

at the command prompt. The result is an error, and your command prompt area will display (use F1 to flip to the text screen):

```
Command:  (setq  pt1  (2.0 3.0 0.0))
error:  bad function
(2.0 3.0 0.0)
(SETQ PT1 (2.0 3.0 0.0))
Command:
```

Here, AutoLISP has attempted to evaluate the (2.0 3.0 0.0) as a subr or defined function, and does not recognize it as a function.

Now enter

```
(setq pt1 (quote (2.0 3.0 0.0)))
```

Next enter

```
(setq pt2 '(2.0 3.0 0.0))
```

These last two functions both return the list (2.0 3.0 0.0), because the 'quote' function allows us to define a list which will not be evaluated. As shown in the above, there are two formats, **(quote expression)** and **'expression**.

The quote function returns the expression unevaluated, so the setq function in the example assigns the actual quoted expression to the point variable. Without the quote function, AutoLISP attempts to evaluate the list (2.0 3.0 0.0) as though the first element (2.0) were a subr or defined function, hence the failure of the first attempt at assigning a list to pt1.

In a similar manner, it would be possible to write the polar function in the form

```
(polar '(3 5 0.0) (/ pi 2.0) 3.5)
```

and the returned result would be (3 8.5 0.0).

SOME BASIC ERROR CHECKING

If you don't match your closing parentheses with your open parentheses, AutoLISP displays a prompt of the form n> where n is the number of missing closing parentheses. The way to deal with this in the simplest case is to enter the missing parentheses, such as))) if n is 3, or alternatively enter Control-C (^C) to cancel. However, a common mistake is to leave out closing double quotes, in which case AutoLISP will interpret any following parentheses as text strings and the results of entering closing parentheses will be unpredictable. If the error prompt is 3> and entering a single closing parenthesis does not reduce this to 2>, you can enter a double quote followed by another closing parenthesis. Now the number should reduce, and you can repeat the process until the program finishes execution. It is extremely unlikely that a program with such errors will execute properly, so you should edit the program and re-load and test it.

When you are writing your first programs, you can always test individual statements by entering them at the command prompt to make sure that they work before including them in your program (.lsp) file.

It is always a good idea to ensure that your variables are being assigned correctly by checking their values with the ! character at the AutoCAD command prompt after a program has run, or during the development of a program. You don't have to write a complete program in order to make some of these checks.

You can also 'comment out' any lines of your program by entering a semi-colon at the beginning of any line, but make sure that you always have matched parentheses at all times.

Home work assignments

1. Use the defun, setq, polar, getpoint, command and arithmetic functions (*, +, -, /) to write a program that will draw an "I" beam cross section as shown in figure HW1. Let the user supply the height H of the cross-section, and the origin point.

2. How can you add fillets to your I-beam cross section?

3. How would you automatically dimension the "I" beam of question1? Add this feature to your program. Hint - use the command function for the appropriate dimension types and define extra points with setq in order to specify the dimension locations.

In lesson three of this series, "Manipulating lists with AutoLISP", you will learn more programming techniques and some new functions for manipulating lists. You will see how to use individual x, y or z components of point lists and how to construct and append elements to lists. Example programs will illustrate how you can work with lists.

An Introduction to AutoLISP

Manipulating lists with AutoLISP

Lesson Three in the CADalyst/University of Wisconsin Introduction to AutoLISP Programming course shows you how to construct and work with lists.

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

About this course.

Introduction to AutoLISP Programming is an eight-lesson independent study course, originally published in monthly installments in CADalyst magazine. Readers may optionally register for the course in order to receive a copy of the printed course outline, a study guide containing additional AutoLISP examples, reference materials, and solutions to the practical assignments. On successfully completing an intermediate and a final mail-in project, students will receive certificates of completion and 4.0 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

For further information, contact, see the enrollment form attached to this article.

LESSON THREE - MANIPULATING LISTS WITH AUTOLISP

In this months lesson you will learn more programming techniques and some new functions for manipulating lists. You will see how to use individual x, y or z components of point lists and how to construct and append elements to lists. Example programs will illustrate how you can work with lists, and the lesson ends with some homework questions to test your understanding of the text.

PROGRAM STRUCTURE AND REDUCING THE NUMBER OF VARIABLES

In lesson two we wrote a program to draw a square box, and here we will use that simple program as an illustration of how to reduce the number of variables used in a program. The BOX1.LSP program, uses the point variables pt1, pt2, pt3 and pt4, which are assigned and used later in the 'command line' function.

```
;  BOX1.LSP, a program to draw a square with origin and length of a side.
;
(defun  box1  (len)  ; The variable 'len' means the length of side
   (setq  pt1  (getpoint  "\nEnter the origin point:  ")  ; The 'getpoint'
function
          pt2  (polar  pt1  0.0  len)  ;  Polar requires an angle in radians
          pt3  (polar  pt2  (/  pi  2.0)  len) ;'pi' is built in to AutoLISP
          pt4  (polar  pt3  pi  len)  ; polar, base-point, angle, distance
   )     ;  End of setting variables for points
   (command  "line"  pt1  pt2  pt3  pt4  "c") ;  Drawing the box
)  ; End of the 'defun'
```

It is possible to write this program using only one point variable as in the following listing:

```
;  BOX11.LSP, a program to draw a square with origin and length of a side.
(defun  box11  (len)
   (setq  p  (getpoint  "\nEnter the origin point:  "))
   (command  "line"  p
        (setq p (polar  p  0.0  len))
        (setq p (polar  p  (/  pi  2.0)  len))
        (setq p (polar  p  pi  len))  "close")
)  ; End of the 'defun'
```

Notice how the **command** function is used in this case. The point variable '**p**' is defined initially by the '**getpoint**' function, but during the line command, the point is re-defined with reference to itself. Each of the **setq** assignments returns the value of its expression (the new point definition), which is used as successive input to the AutoCAD line command.

The **getpoint** function cannot be used inside the **command** function, and that is why we defined the first point outside of the **command** function before starting to draw the line. Note that the **setq** function returns the value of its last expression, which means that the point assignments must be made with separate **setq** functions, rather than with a list of pairs of variables and expressions.

Reducing the number of variables in your programs is a convenience, but there are times when you will need to retain separate point variables, especially if you want to use parametric programming with automatic dimensioning. The box programs are simple forms of parametric programs, although the only parameter is the length of side of the box. If you want to dimension the box, you must have access to the points which were defined at the corners of the box. For the square box, only one side need be dimensioned, but you will need to define a point to specify the position of the dimension line.

Let's suppose that you want to use a vertical dimension on the right-hand side of the box, with the vertical dimension line 0.75 inches from the box, as shown in figure 3.1. You will need to use a program of the form of our BOX1.LSP above, with an extra point, pt5, defined to be at 0.75 inches from point 2 (pt2), so the line:

(setq pt5 (polar pt2 0.0 0.75))

should be added. The program would then include the command function for dimensioning, such as:

(command "dim1" "ver" pt2 pt3 pt5 "")

We used the form DIM1 here, so that the dimension mode would automatically terminate after placing the single dimension. Note that there is no such command as 'linear' for dimensions, and the subcommand "ver" was used directly after the "dim1" command. The rest of the sequence defines the endpoints of the side of the box and the position of the dimension line, followed by a return (2 double quotes) to accept the dimension calculated by AutoCAD. The final program, which produced the box of figure 3.1, would look like:

```
;  BOX31.LSP, a program to draw and dimension a square with origin
:             and length of a side.
;
(defun  box31  (len)  ;  The variable 'len' means the length of side
   (setq  pt1  (getpoint  "\nEnter the origin point:  ") ;  The 'getpoint'
function
          pt2  (polar  pt1  0.0  len)   ; Polar requires an angle in radians
          pt3  (polar  pt2  (/  pi  2.0)  len) ;'pi' is built in to AutoLISP
          pt4  (polar  pt3  pi  len)  ; polar, base-point, angle, distance
   ) ; End of setting variables for points
   (command  "line"  pt1  pt2  pt3  pt4  "c") ;  Drawing the box
;
;  Dimensioning the box
;
   (setq  pt5  (polar  pt2  0.0  0.75)) ;  Setting the dimension line position
   (command  "dim1"  "ver"  pt2   pt3  pt5  "")
)  ; End of the 'defun'
```

MANIPULATING LISTS

The functions for creating and manipulating lists are

```
(list expr1 expr2 expr3.....)  (cons item list)
(car list)  (cdr list)  (cadr list)  (caar  list)  (caddr list)  (cadar list)
...
(length list)  (listp item)
(last list)  (nth integer list)
(append list1 list2)  (member expr list)  (reverse list)
```

The functions **list** and **cons** are used to create lists, and the other functions manipulate and return information about the lists.

List can have any number of expressions, and it returns the list of all of the expressions in sequence. For instance, **(list 4.5 3.5 0.0)** returns the list (4.5 3.5 0.0). The statement

(setq pt1 (list 4.5 3.5 0.0))

assigns the list (4.5 3.5 0.0) to pt1, and also returns (4.5 3.5 0.0).

The function **car** returns the first item of a list, so that if pt1 is the list of the x, y, and z coordinates of a point, **(car pt1)** would return the value of the x-coordinate of the point (4.5 in this case)

The function **cdr** returns all of the items in a list except the first item, so that in our example of the list pt1, **(cdr pt1)** returns the list containing the y and z coordinates of the point pt1 (this would take the form (3.5 0.0) in our example because lists are contained within parentheses, as we discussed in lesson two of this series).

Consider the following statements:

```
    (setq  pt2  (list  2.0  3.0  0.0))
    (setq  pt2x  (car  pt2))
    (setq  pt2yz  (cdr  pt2))
    (setq  pt2y  (car  pt2yz))
```

The value of pt2 is (2.0 3.0 0.0), the value of pt2x is simply 2.0 (the first item of the list of pt2), the value of pt2yz is (3.0 0.0) because it is the list of all except the first element of pt2, and finally the value of pt2y is 3.0 (the first element of the list of pt2yz).

We could combine the last two statements as:

(setq pt2y (car (cdr pt2)))

according to the usual techniques for nesting functions in AutoLISP, and the result for pt2y will be 3.0 as before.

We may therefore always extract the second item of a list of any length simply by using the combined functions **(car (cdr *list*))**

Can you see how to extract the third item of a list which contains any number of items using the car and cdr functions in some combination?

The solution is **(car (cdr (cdr *list*)))**

AutoLISP provides functions which are combinations of car and cdr. These functions are formed by retaining the initial 'c' and the final 'r', and combine the 'a' and 'd' according to the order in which the 'a' of car and the 'd' of cdr appear in the intended combinations of car and cdr. Thus **cadr** is equivalent to **(car (cdr .....))** and **caddr** returns the same result as **(car (cdr (cdr ...)))**.

Using this terminology, **(car *list*)** returns the first element of a list, **(cadr *list*)** returns the second item of the list, and **(caddr *list*)** returns the third item of the list. Many other such combinations are possible, and AutoLISP allows nesting any combination up to four levels deep in a single function.

A rectangle example, using LIST, CAR, and CADR functions.

```
;  RECTANG1.LSP  program to draw a rectangle with
;                lower left and upper right
points.
;
(defun rectang1  ()
   (setq  pt1  (getpoint  "\nLower left point:")
          pt3  (getpoint  "\nUpper right point:")
          p2x  (car  pt3)
          p2y  (cadr  pt1)
          pt2  (list  p2x  p2y)
          pt4  (list  (car  pt1)  (cadr  pt3))
   )  ; end of point assignments
   (command  "line"  pt1  pt2  pt3  pt4  "c")
) ; end of defun rectang1
```

Here, we get the lower left and upper right points from the user. The points may be entered from the keyboard or by picking a position with the cursor and optionally snapping to a gridpoint or using an osnap mode. We name the lower left and upper right points pt1 and pt3 respectively so that we retain the same ordering of points as was used for the box example.

Notice now that we can separately define the x and y coordinates of the second point pt2 by defining p2x and p2y as shown. The **car** and **cadr** functions return the first and second items of a list, and it is convenient to regard these as the x and y components of points. This is a similar process to using the .x and .y filters in geometry construction. We then define pt2 to be the list of p2x and p2y. We can of course extend this approach to the three dimensional case by including the z-coordinate (using **caddr**) in our list, but we assume here that all points are at the same z-level.

In our definition of point 4, pt4, we have by-passed the step of assigning variable names to the x and y coordinates, and have made the car and cadr functions expressions of the list function. As you become more comfortable with LISP programming you may decide to nest functions in this way, but don't get too complicated because you may need to modify and update your programs later on, and they should be readable!

The nth item of a list.

The function **(nth *integer list*)** returns the **nth** element of *list*, where *integer* is the number of the element to return. AutoLISP numbers the elements of lists from zero upwards, so the first item of a list is obtained by (nth 0 list). If x = (3.2 4.5 0.0), (nth 2 x) returns 0.0 and (nth 0 x) returns 3.2.

If pt11 = (2.0 2.5 0.0) and pt22 = (3.0 4.5 0.0),
(setq pt33 (list (nth 0 pt11) (nth 1 pt22) 1.0)) would set pt33 = (2.0 4.5 1.0).

The cons and append functions

(cons *item list*) returns a list having item as the first element, and the elements of list as its remaining elements. In this sense, the **cons** function adds an item to the front of a list. For example, type the following at the AutoCAD command prompt:

```
   (setq  aa  '(1  2  3  4  5  6))
   (setq  bb  (cons  99  aa))
```

The first statement shows how a list can be created with the **quote** function, as we described in lesson 2 of this series. The list returned is (1 2 3 4 5 6).

The second statement returns the list (99 1 2 3 4 5 6)

If list is an atom (has only one element), then cons returns a "dotted pair", which is a special structure in AutoLISP. Dotted pairs are the way in which AutoCAD associates items together in its data structure, where a 'group code' number is associated with such items as object type, layer names etc., and is beyond the scope of this introductory course.

When you use real numbers as the arguments for functions or in lists, you cannot omit leading zeros before decimal points, otherwise AutoLISP will assume that you intend a dotted pair, and this will result in the error message 'invalid dotted pair'.

The **append** function adds an item to the end of a list, and is often used in programs to store many point positions in a single list so that the points can be used in the construction of polylines and general mesh surfaces. One example that we will see more of in later lessons is that of a cross-hatching program that is able to perform cross hatching without the need to break lines that are not end-to-end. A useful and fairly common technique to cross hatch in this situation is to create a polyline by tracing over existing geometry, and then to cross hatch the polyline. Here is a part of the program in which a series of points is being added (appended) to a pointlist.

```
; Start collecting data and set layer to 'noplot'
;
     (command "LAYER" "T" "noplot" "S" "noplot" "")
     (setq pdiff 1)
     (setq porig (getpoint "\n origin"))
     (setq ptlst nil)
;
;    Make a list of points for the 'pline'
;
  (while
     (> pdiff 0)
     (setq p (getpoint "\n next point"))
     (setq ptlst (append ptlst (list p)))
     (setq pdiff (distance p porig))
  ); end of 'while' loop
;
;    Draw a pline as a boundary for hatching
;
     (command "PLINE" porig "W" 0 0 (foreach p ptlst (command p))
     )
```

In this fragment of our program, the layer 'noplot' is first thawed and set as the current layer, then an 'origin' point is defined. The user is requested to pick a series of 'next points' in order to create a closed polyline. The variable pdiff is used to check whether a selected point is coincident with the 'origin' point. If the distance 'pdiff' is zero, the program comes out of a 'while' loop (more of that later!) and no more points are added to the point list. Finally the poly line is drawn by supplying each point in turn to the pline command, using another type of repeating 'loop' known as the foreach function (again, more of that later).

Be careful when adding points to a point list like this that you add the points as 'lists' with the list function, as we have done in the cross hatch program example. To see the significance of this, try the following at the command prompt:

First assign three points with

(setq pt1 '(2.72 1.39 0.0) pt2 '(4.43 1.39 0.0) pt3 '(4.43 2.85 0.0))

then collect those points into a point list with:

(setq ptlst nil)
(setq ptlst (append ptlst (list pt1)))

this returns ((2.72 1.39 0.0)). Note the double parentheses! Note also that we could in fact initialize our ptlst with the assignment (setq ptlst (list pt1)) instead of appending the first point to a null list.

(setq ptlst (append ptlst (list pt2)))

this returns ((2.72 1.39 0.0) (4.43 1.39 0.0)). Note that we now have a list of 2 items, each of which is itself a list of 3 items. Now try adding the third point without the 'list' function:

(setq pt3 (append ptlst pt3))

this returns ((2.72 1.39 0.0) (4.3 1.39 0.0) 4.43 2.85 0.0)

In the last case, the coordinates of point 3 have not been made into a list, and the three coordinates appear as separate items, which is not the intended format. If we now extract the third item of this list we will return the number 4.43 instead of the intended list of three numbers.

The other functions for dealing with lists are:

(member *expr list*) *expr* is one of the elements of *list*. The function returns a list with *expr* as the first element, followed by all subsequent elements of *list*.

If expr is not part of the list, nil is returned. For example:

```
(setq test1 (list "TONY"  2.5  3.4  "ABC"))

          returns the list:  ("TONY"  2.5  3.4  "ABC")

(member  2.5  test1)  returns:  (2.5  3.4  "ABC")

(member  "LUCY"  test1)  returns:  nil
```

(last *list*) returns the last element in *list*. **(last test1)** in the above example returns "ABC"

(length *list*) returns an integer which is the number of elements of *list*.

```
(length  test1)  returns 4.
```

(listp *item*) returns T if *item* is a list, and nil if *item* is not a list.

```
(listp  test1)  returns T;  (listp "ABC")  returns nil.
```

The letter T is listed in the atomlist and is used by AutoLISP in testing whether something is 'true' or 'false'. The test is part of conditional testing and is related to the use of logical functions, which is the subject of a later lesson in this series. AutoLISP actually only tests whether an item evaluates to a nil or a non-nil result. Any returned value that is not nil will be tested as 'true', and will take the value T.

It is a wise precaution not to use the letter 'T' as a variable name, although in functions such as listp and in uses related to logical functions, T is returned unevaluated, so no real harm is done.

(reverse *list*) returns *list* with its elements in reverse order.

Home work assignments

1.
Elements of lists may also be lists.

If A = (a1 a2 a3 a4), a1 = (aa bb cc), a2 = (dd ee), and a3 = (ff gg hh), what do the following functions return?

```
(caar A);  (cadar A);  (cdadr A);  (cddar A);  (cdaddr A);  (length A);
(length  (cdr A));  (last (car A));  (reverse (cddr A))
```

2.
What is the difference between **(cons *item list*)** and **(list *item list*)**?

3.
When would you use the **quote** function versus the **list** function to create a list?

4.
Write two programs using **car** and **cdr** functions to draw 'infinite' length horizontal and vertical lines respectively at any user defined position. (The user should be prompted to indicate a position.)

In lesson four of this series, "AutoLISP functions for math and geometry", we present the math functions, and some new functions for drawing geometric shapes with user interaction. Some new 'GET..' functions will be introduced to assist with geometric definitions, and example programs will illustrate the new functions. Homework assignments will test your knowledge of the text.

An Introduction to AutoLISP

AutoLISP Functions for Math and Geometry

Lesson Four in the CADalyst/University of Wisconsin Introduction to AutoLISP Programming course covers math functions and geometric shapes.

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

About this course.

Introduction to AutoLISP Programming is an eight-lesson independent study course, originally published in monthly installments in CADalyst magazine. Readers may optionally register for the course in order to receive a copy of the printed course outline, a study guide containing additional AutoLISP examples, reference materials, and solutions to the practical assignments. On successfully completing an intermediate and a final mail-in project, students will receive certificates of completion and 4.0 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 FUNCTIONS FOR MATH AND GEOMETRY

In this month's lesson, we present the math functions, and some new functions for drawing geometric shapes with user interaction. Some new 'GET..' functions are introduced to assist with geometric definitions, and example programs illustrate the new functions. Homework assignments will test your knowledge of the text.

MATH FUNCTIONS

We have already seen some of the Math functions for simple arithmetic in lesson two of this series. In that lesson, some examples of the add, subtract, multiply and divide functions were given for two numbers. Here we give some general rules for their use:

```
(+ number  number  ...)   (- number  number ...)
(* number  number  ...)   (/ number  number ...)
```

In all cases, for two numbers only, the first number is operated on by the second number, and the result is returned. For example, in the case of the division, the first number is divided by the second number and the quotient is returned, and for subtraction the first number has the second number subtracted from it.

When more than two numbers are given, the operation is repeated by each number in turn, so for division, the quotient of the first two numbers is divided by the third number, and this quotient is divided by the next number, and so on.

If all of the numbers are integers, an integer is returned, so be warned that in the case of division, some rounding off will occur.

If any of the numbers is a real (having a decimal point), a real number is returned, so if all numbers are real, or if they are any combination of integers and reals, a real number is returned.

If only one number is given, that number is returned except that in the subtraction case, the sign of the number is changed, so that (- 5) returns -5, and (+ 5), (* 5), (/ 5) all return 5 without any change of sign.

Incrementing and Decrementing

The functions **(1+ *number*)** and **(1- *number*)** are used to increment or decrement a number by 1.

(1+ *number*) returns number increased by 1, and is a shorthand version of the equivalent **(+ *number* 1)**, where number may be real or integer. **(1- *number*)** is the equivalent of **(- *number* 1)**.

For example, it is often required to use a 'counter' to count the number of times that a set of operations is to be performed. This is generally done in connection with a 'looping' function, which we will consider later in this series. We may want to set the initial value of a counter, and then use it to control the number of times a looping function executes as follows:

```
(setq  count  16)
    (......  ;entering an iteration loop
        (setq  p  (list  (+  (car  p)  delta)  (cadr  p)))
        (setq  ptlst  (append  ptlst  (list  p)))
            (setq  count  (1-   count))
    ) ; end of the iteration loop
```

This example assumes that some kind of test is performed so that the number of iterations is controlled. The variable 'count' is first set to 16, then the looping begins (the looping function is not defined here - we will leave it until later), and at the end of the loop, count is decremented by 1. In the meantime a point 'p' has its x-coordinate changed by an amount 'delta' (using the + function), and the resulting point is appended to a pointlist 'ptlst'.

Trigonometric functions

The basic trigonometry functions are:

(sin *radian_angle*) (cos *radian_angle*) (atan *number1 number2*)

Angles are measured in radians by AutoLISP functions, as indicated here. The **sin** function returns the sine of the radian angle, and the **cos** function returns the cosine of the radian angle.

The **atan** function can have one or two arguments. For the case of one argument, **atan** returns the arctangent of the number in radians. If two numbers are supplied, **atan** returns the arctangent of the first number divided by the second.

The symbol PI is known to AutoLISP, so in the following:

```
(setq  ang1  (/  pi  3.0))
(setq  res1  (sin  ang1))    would return  0.866025
(setq  res2  (cos  ang1))    would return 0.5
```

Converting angles in degrees to radians

Here is a defined function that will allow you to work in degrees instead of radians:

```
(defun  dtr (ang)
     (* pi (/ ang 180.0))
)
```

This function requires one argument, the angle in degrees, and returns the angle converted to radians.

To use the 'dtr' function, enter **(dtr *angle*)** where the angle is in degrees. With this function:

(setq x (sin (dtr 60))) sets x = 0.866025

Note the use of the defined function argument (ang) here is similar to the 'len' argument that we supplied in the box and rectangle programs of the earlier lessons. It means that every time the function is used, a value must be supplied. The supplied value then replaces the original argument (in this case 'ang'), and is operated upon accordingly, so in the dtr function, any supplied value will be divided by 180, then multiplied by PI.

Other math functions

The other math functions are the square-root **(sqrt *number*)**, the natural logarithm **(log *number*)**, and the absolute value **(abs *number*)**.

GEOMETRY WITH AUTOLISP

Functions for geometry are:

```
(polar  base_point  radian_angle  distance)
(angle  point1  point2)
(distance  point1 point2)
(inters  point1  point2  point3  point4  argnil)
(redraw)
```

The polar function was introduced in our lesson two, and is the basic method of drawing by defining points at arbitrary angles.

The **angle** function returns the angle in radians drawn between an imaginary line connecting point1 & point2, and the positive x-axis. Angles are measured positive counter-clockwise from the x-axis.

The **distance** function returns the distance between point1 and point2.

The **inters** function returns the intersection point of two imaginary lines connecting point1-point2, and point3-point4. The '***argnil***' argument is optional. If it is supplied as 'NIL', the intersection point will be returned even if the imaginary lines do not intersect. If '***argnil***' takes any other value, or is omitted, then the lines must intersect in order to return the required point.

The **redraw** function, without any arguments, behaves just as the AutoCAD redraw. Note that the redraw function can have arguments which allow it to redraw named entities, but the use of named entities is beyond the scope of this introductory course.

Let's have a look at an example to illustrate these functions. Figure 4.1 shows a square drawn at some angle 'theta', with a side length of 'len'. Two circles are drawn inscribed and circumscribed respectively. This program does not use any arguments. The length of the side of the square 'len' is supplied with the distance function, which returns a real number, (the distance between two points). Two reference points P1 and P2 are obtained from the user with the getpoint function. The second getpoint uses point P1 as a reference from which the second point is rubber-banded. The angle of orientation of the square is also defined by the points P1 and P2.

The program EX4_1.LSP is as follows:

```
;  EX4_1.LSP
;  A program to draw a square with inscribed and circumscribed circles.
;
(defun  dtr  (ang)
   (*  pi  (/  ang  180.0))
)
;
(defun ex4_1 ()
   (setq p1 (getpoint "\nPlease enter the lower left point: ")
         p2 (getpoint p1 "\nDrag the length of side and angle: ")
   )
   (setq len (distance p1  p2)  ; calculate the length 'len'
         theta (angle  p1  p2)  ; and the angle 'theta'
   )
   (setq  p3  (polar  p2  (+  theta  (dtr  90))  len)
          p4  (polar  p3  (+  theta  (dtr  180))  len)
   )
   (command "line" p1 p2 p3 p4 "c")          ; draw the square
```

```
    (setq pcenter (inters  p1  p3  p2  p4))  ; find the circle center point
    (command "circle"  pcenter  (/ len  2))  ; draw the inner circle
    (command  "circle"  pcenter  p1)             ; draw the outer circle
    (redraw)
)  ;end of the defun ex4_1.
```

When this program is loaded, the command/prompt area displays:

```
Command:  (load  "/acad11/lisp/ex4_1")
EX4_1
Command:
```

The dtr function is not displayed because the load function returns the value of the last expression in the file. In this case the last expression is the defined function ex4_1, and the **defun** function returns the name of the function being defined.

In the above example, the circles were drawn by establishing the center point of the circle with the intersection function. There are several alternatives for defining the center point, as follows:

(a) using **car** and **cadr** to calculate a distance to the circle center from point 1:

```
.
.
(setq  half_len  (/  len  2))
(setq  pcx  (+  (car  p1)  half_len))
(setq  pcy  (+  (cadr  p1)  half_len))
(setq  pc  (list  pcx  pcy))
(setq dist_to_cen  (distance  p1  pc))
(setq  pcenter  (polar  p1  (+  (dtr 45)  theta)  dist_to_cen))
.
```

(b) calculating the diagonal distance to the center with the square root using Pythagoras:

```
.
.
(setq  diagonal  (sqrt  (*  len  len)))
(setq  angle2  (angle  p1  p3))
(setq  pcenter  (polar  p1  angle2  diagonal))
.
```

(c) it is also possible to draw the circles directly without calculating a center point!

```
.
.
   (command "line" p1 p2 p3 p4 "c")           ; draw the square
   (command  "circle"  "2p"  p1  p3)           ; draw the outer circle
   (command  "circle"  "cen"  p1   (/  len  2)) ; draw the inner circle
   (redraw)
)  ; end of the defun ex4_1.
```

SOME MORE 'GET..' FUNCTIONS

Getting input from the user of your LISP programs can be done in many ways, and this gives you considerable flexibility in deciding how your defined functions will work. Consider the following program to draw an 'L' shaped beam cross-section, described in figure 4.2:

```
;  LBEAM.LSP   A program to draw an 'L' shaped beam cross-section
;  with the leg length, flange thickness, rotation angle
;  and corner radii are supplied by the user.
(defun  dtr (a)
   (* pi (/ a 180.0))
)
(defun  LBEAM ()
   (setq len (getreal "\nLeg length: ")
         flange  (getreal  "\nFlange thickness: ")
         ang  (getangle  "\nAngle of rotation: ")
         rad  (getreal  "\nCorner radius: ")
         p1   (getpoint  "\nOrigin point: ")
    )  ; end of user input
    (setq  len2  (-  len  flange))  ; calculation of inside length
    (setq  p2  (polar  p1  ang len)
           p3  (polar  p2  (+ (dtr  90)  ang)  flange)
           p4  (polar  p3  (+ (dtr  180)  ang)  len2)
           p5  (polar  p4  (+ (dtr  90)  ang)  len2)
           p6  (polar  p5  (+ (dtr  180)  ang)  flange)
    ) ;end of point definitions
    (command  "line"  p6  p1 p2 "")
    (command  "pline"  p2  "w"  0  0  p3  p4  p5  p6  "")
    (command  "fillet"  "r"  rad  "fillet"  "polyline"  "L")
    (redraw)  ; end of drawing the outline and filleting
) ; end of defun
```

The (dtr) is an 'internally defined function', used in the polar definitions of points p2 through p6. Remember, only the last defined function of a program is returned, so if you use internally defined functions, be sure to define them ahead of the main function.

The 'GET' functions for this example are 'getreal', 'getangle', and 'getpoint'.

(getreal). The **getreal** function pauses for user input of a real number, and returns that number. In this case, the 'L' shaped leg length 'len', the flange thickness 'flange', and the corner radius 'rad' are obtained with the **getreal** function, which returns a real number, even if you enter an integer.

(getangle). This function returns the angle in radians, even though you enter the angle in the AutoCAD current units, which may be degrees or grads etc. The angle returned depends upon the orientation defined by the ANGBASE system variable in the current UCS. This system variable can be set with the UNITS command to be 0,90,180, or 270, and sets the zero angle direction.

(getorient). A similar function, **(getorient *pt prompt*)** returns an angle that is not affected by the setting of ANGBASE. Use **(getangle)** for relative angles, and **(getorient)** for absolute measure. (Review your AutoLISP Programmer's Reference for the effect of the system variable ANGDIR, which controls clockwise/counterclockwise directions).

(getint). The **getint** function pauses for user input of an integer, which can be positive or negative in sign. The following example uses **(getint)** for setting and restoring the drawing units precision system variable LUPREC.

```
        (setq dp (getint "\nNumber of decimal places for dimensioning: "))
        (setq oldluprec (getvar "LUPREC"))
        (setvar "LUPREC" dp)
    .
    .   ;  dimensioning commands
    .
        (setvar "LUPREC" oldluprec)
```

Home work questions

1.
Surface machining with a 'ball-nose' cutter requires that the cutting tool has a 'step-over' distance, which produces the scalloped shape shown in figure 4.3. Write a program to calculate the 'cusp' height as shown, assuming the tool diameter DIA and the step-over distance STEP are supplied as user input. Store the cusp height in a global variable CUSP so it can be displayed using the !CUSP command at the command prompt after the program has executed.

2.
Rearrange your program to calculate the step-over distance STEP, given DIA and CUSP as user input.

3.
Write a program to draw and dimension the tool set-up of figure 4.3. Use the cutter diameter DIA and the cusp height CUSP as user input to your program. Set the dimension precision to 3 decimal places for the step-over dimension STEP, and 2 decimal places for the other dimensions.

In lesson five of this series, "User Interaction in AutoLISP", the 'GET..' functions are explored more fully, along with some initial conditions for using them. You will also learn about printing functions and prompting for user input. Example programs will illustrate the new functions, and homework assignments will test your knowledge of the text.

An Introduction to AutoLISP

User Interaction in AutoLISP

Lesson Five in the CADalyst/University of Wisconsin Introduction to AutoLISP Programming course covers GETing, prompting and printing.

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

About this course.

Introduction to AutoLISP Programming is an eight-lesson independent study course, originally published in monthly installments in CADalyst magazine. Readers may optionally register for the course in order to receive a copy of the printed course outline, a study guide containing additional AutoLISP examples, reference materials, and solutions to the practical assignments. On successfully completing an intermediate and a final mail-in project, students will receive certificates of completion and 4.0 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 - USER INTERACTION IN AUTOLISP

In this lesson, the 'GET..' functions are explored more fully, along with some initial conditions for using them. You will also learn about printing functions and prompting for user input. Example programs will illustrate the new functions, and homework assignments will test your knowledge of the text.

THE GET FUNCTIONS

In lessons two and three of this series, we introduced and used the **(getpoint)** function. Then last month in lesson four, we looked at the functions **(getreal)**, **(getint)**, **(getangle)**, **(getorient)**, and **(getvar)**. We will now explore the functions **(getcorner)**, **(getdist)**, **(getstring)**, **(getkword)**, and **(initget)**.

(getcorner) returns a point in the current UCS, and in this respect it is similar to **(getpoint)** except that **(getcorner)** always requires a reference (base) point so that it can 'rubber-band' a rectangular box from the base point as you move your cursor to the new point position. The following program RECTANG1.LSP shows its use:

```
;;  RECTANG1.LSP  A program to draw a rectangle
;;  using the (getcorner) function
;;
(defun  rectang1  ()
   (setq  pnt1  (getpoint  "First corner  "))  (terpri)
   (setq  pnt3  (getcorner  pnt1  "Opposite corner..."))  (terpri)
   (setq  pnt2  (list  (car  pnt1)  (cadr  pnt3)))
   (setq  pnt4  (list  (car  pnt3)  (cadr  pnt1)))
   (command  "line"  pnt1  pnt2  pnt3  pnt4  "c")
   (redraw)
) ; end of the defun
```

The **(getcorner)** function has two arguments as shown, the required base point and the optional prompt. We have introduced the **(terpri)** function, which prints a newline on the screen and is an alternative to the newline character '\n'. Use of this function makes the preceding prompt easier to read.

(getdist) pauses for user input of a distance, and returns a real number. The function may have two optional arguments, a point and a prompt. The distance may be supplied as two points, or, if the optional point is supplied as an argument, a single point may be used as input.

In our earlier programs to draw boxes, we supplied the box length as an argument to the defined function. Instead of this, you could use the **(getdist)** function as follows:

```
(defun  BOX  ()
     (setq  pt1  (getpoint  "First point:  ")) (terpri)
     (setq  len  (getdist  pt1  "Drag the length of side")) (terpri)
     (setq  pt2  (polar  pt1  0.0  len))
.
.    etc...
```

The **(getdist)** function forces the cursor to be connected to the point argument pt1 with a 'rubber band'. If no base point is given as an argument, you can simply enter a real number in the usual AutoCAD way, remembering that AutoCAD also accepts using two points connected by a rubber band for input of real numbers.

(getstring) lets you input a string, and returns up to 132 characters of that string. This can be used for file names, but if the filename contains backslashes, they are converted to double backslashes so that the returned string can be used by other functions which use filenames. We will cover file handling in lesson six next month.

Now, because AutoCAD input is normally terminated by a space as well as by the 'return' or 'enter' keys, we need some way to handle character strings that contain separate words with spaces. This is allowed by using an optional argument to handle spaces. For example

(setq name (getstring T "\nEnter your full name: "))

uses the variable T as a non-nil flag to indicate that spaces will be accepted as part of the input, and here, the variable name will be assigned the entered name including any spaces. The variable T is part of the atomlist of AutoLISP, and is often used to indicate a non-nil argument. For this reason it is not a good idea to redefine T, particularly if there is any possibility of T taking a nil value!

Sometimes, the entered response should not contain any spaces, such as when filenames are required, so the optional non-nil argument is omitted, as in:

(setq oldname (getstring "\nEnter the old .LSP file name: "))

In either case, the entered string is returned as a string contained within double quotes.

(getkword) pauses for user input of a predefined keyword, and is a little different from the other GET... functions, because it requires the **(initget)** function to precede it. The **(initget)** function has a an optional string argument that defines a list of the allowed keywords for the **(getkword)** function. For example, the following extract from a cross-hatching program requires a yes/no answer:

```
(initget  1  "Yes  No")
(setq  ans  (getkword  "\nDo the layers NOPLOT and HATCH exist? (Yes/No):"))
```

Here, the allowable keywords are defined in the string argument of the **(initget)** function. The use of capitals specifies the 'required' portion of the keyword to be entered, so in this example, you may respond to the prompt with Y or y or N or n. You may also enter the entire keyword or any part of it that contains the required portion (shown capitalized). The keywords are repeated in the **(getkword)** function to show the user the format expected. If the entered keyword is not recognized, AutoCAD asks the user to try again.

The number 1 in the **(initget)** function is a 'control bit' argument, which means that a null input is not allowed. There are many other control bits that may be used with other **(get..)** functions, but the number 1 is the only option for **(getkword)**. If this option is left out, a null input would be allowed, and you may use this to define a default condition, in the same way that AutoCAD does for most of its subcommand options. For example, the above example could be written:

```
(initget "Yes  No")
(setq ans (getkword "\nDo the layers NOPLOT and HATCH exist? Yes/<No>:"))
(if (null ans) (setq ans "No") ) ......
```

Here, we allow a null response, because there is no control bit argument in the **(initget)** function, and we imply that 'No' is the default keyword by using the standard AutoCAD display of <> brackets. In order to make the variable 'ans' take this default value, we need some more sophisticated AutoLISP code in the form of the logical 'if' statement, which is covered in lesson seven of this series. The 'if' statement shown tests for a null response, and sets 'ans' to 'No' if the test is true.

Note that the use of the <> brackets is simply a convention and does not actually do anything to set default conditions - you need to write code to make that happen, as in our example.

(initget) provides options only for the next **(get...)** function to follow, but not for any subsequent **(get...)** functions. The syntax for the **(initget)** function is:

(initget control_bits string_argument)

as we saw in the example of the **(getkword)** function.

The optional control bit values are:

Bit Value	Meaning
1	Do not allow null (the ENTER key) input
2	Do not allow zero input
4	Do not allow negative values
8	Do not check the drawing limits, even if LIMCHECK is on
16	Not currently used - (earlier versions returned 3D points)
32	Rubber band lines and boxes have dashed line type
64	Return a 2D distance for the (getdist) function, instead of 3D

These control bits can be added together if you want to include more than one item. For instance, if you are using the **(getint)** function and you do not want to allow null, zero, or negative values, you may provide 7 as the control bit. Some programmers use **(+ 1 2 4)** as in **(initget (+ 1 2 4))** instead of **(initget 7)** to make it clear that it is intended to include three options.

The **(initget)** function is only active until the next time a **(get...)** function appears in a program, so you don't need to reset the arguments of **(initget)** when it is not needed. **(getstring)** and **(getvar)** are not affected by the control bits of **(initget)**.

The number of control bits entered depends on the **(get...)** function which follows the **(initget)**, and the following table shows which control bits can be used with the **(get...)** functions:

Function	INITGET control bits
GETINT	1, 2, 4
GETREAL	1, 2, 4
GETDIST	1, 2, 4, 32, 64
GETANGLE	1, 2, 32
GETORIENT	1, 2, 32
GETPOINT	1, 8, 32
GETCORNER	1, 8, 32
GETKWORD	1
GETSTRING	
GETVAR	

Here is an example of the use of **(initget)** in our program for drawing a box:

```
(defun  BOX  ()
   (initget 1)  ;  The 3D point cannot be null
   (setq pt1  (getpoint  "First point:  ")) (terpri)
   (initget 7)  ;  The length cannot be null, zero, or negative
   (setq len  (getdist  pt1  "Drag the length of side")) (terpri)
   (setq pt2  (polar  pt1  0.0  len))
.
.    etc...
```

PRINTING FUNCTIONS

The printing functions and their arguments are:

```
(prompt  string)
(princ  expression  filename)
(prin1  expression  filename)
(print  expression  filename)
```

We have already seen the **(prompt)** function in lesson two of this series. The function displays the *string* on the command/prompt area of your AutoCAD screen, and it returns nil.

The **(prompt)** function must have a string as the argument, but the other printing functions can have any argument type. The other printing functions can optionally have a filename (a file descriptor) as an argument so they can print to open files or to the AutoCAD screen. We will leave this aspect until the next lesson which deals with files.

(princ), **(prin1)** and **(print)** return their expression data in different formats, as we show in the following.

Try setting your screens to text mode (with F1), then enter the following the command prompt:

```
(prompt  "\\Hello AutoLISP")
(princ  "\\Hello AutoLISP")
(prin1  "\\Hello AutoLISP")
(print  "\\Hello AutoLISP")
```

Your screens will display:

```
Command:  (prompt "\\Hello AutoLISP")
\Hello AutoLISPnil

Command:  (princ "\\Hello AutoLISP")
\Hello AutoLISP"\\Hello AutoLISP"

Command:  (prin1 "\\Hello AutoLISP")
"\\Hello AutoLISP""\\Hello AutoLISP"

Command:  (print "\\Hello AutoLISP")

"\\Hello AutoLISP"  "\\Hello AutoLISP"

Command:
```

Let's see what has happened:

(prompt) interprets special characters, prints strings without double quotes, and returns nil.

(princ) interprets special characters, prints strings without double quotes, and returns the expression unevaluated.

(prin1) does not interpret special characters, prints strings with double quotes, and returns the expression.

(print) is similar to prin1, but inserts a blank line before, and a space after the printed expression.

Now try the following:

```
(setq  x  4.0)
(print  x)
(prompt  x)
```

and you will notice that **(print)** actually prints 4.0 and returns 4.0, whereas **(prompt)** gives an error message **(error: bad argument type)** because the variable to be printed is not a string.

Note that **(prompt (setq b "Goodbye"))** is fine, because setq returns a string in this case.

A common usage of **(prompt)** is at the front of a program, before the first **(defun)**, when you may want to verify that your program is being loaded, thus:

```
;;  IBEAM.LSP
;;  A program to design and dimension I-beams
;;
(prompt  "Please wait:   Loading IBEAM command .......")
(defun  C:IBEAM  ()
   (initget  7)
   (setq  height  (getreal  "\nHeight:  "))
   ....
```

Remember that if a program consists of AutoLISP subrs (defined functions), these functions will execute one by one when the program is loaded. Therefore the **(prompt)** function will execute immediately, but the **(defun)** function is not executed until its final closing parenthesis is entered and interpreted, and this may take a while if the **(defun)** is lengthy.

Printing functions can be used to verify user input or to print the results of calculations. In our IBEAM example, for instance, the entered height of the beam can be verified as follows:

```
;;  IBEAM.LSP
;;  A program to design and dimension I-beams
;;
(prompt  "Please wait:   Loading IBEAM command .......")
(defun  C:IBEAM  ()
   (initget  7)
   (setq  ht  (getreal  "\nHeight:  "))
   (princ  "Beam height is ")
   (princ  ht)
   (princ  ", is this O.K?  Yes/No") (terpri)
   (initget 1 "Yes  No")
   (setq  ans  (getkword))
   ....
```

The **(getkword)** has no arguments because the prompting has already been given by the **(princ)** statements. The example emphasizes that **(princ)** can use both string and real arguments, but we can reduce the number of **(princ)** statements by 'concatenating' all three prompts together with the **(strcat)** function which, as its name implies, can concatenate only character strings together. The real number for the height can be converted to a string with the **(rtos)** (real-to-string) function, so the **(princ)** could look like:

(princ (strcat "Beam height is " (rtos ht) ", is this O.K.? Yes/No"))

The difference between the these approaches is that the first one prints the real number exactly as it was input by the user. In the **(rtos)** case, the real number is returned as a string that looks like a number in the current 'units' settings, which by default is decimal, with 4 decimal places. **(rtos)** has two optional arguments which control the units mode and the precision respectively, thus **(rtos ht 2 3)** returns the ht variable in decimal units with 3 decimal places for the precision. The **(rtos)** mode numbers are 1, scientific; 2, decimal; 3, engineering; 4, architectural; and 5, arbitrary fractional units.

A similar conversion function to handle integers is the **(itoa)** (integer to alpha) function. I think it would be better to name this function **(itos)** (integer to string) for consistency with the **(rtos)** function, so I am inclined to define the **(itos)** function as follows:

(defun itos (a) (itoa a))

Even simpler is the more elegant:

(setq itos itoa)

which assigns the subr **(itoa)** to **(itos)**. In either case the original function (subr) is unchanged.

If you use **(setq itos itoa)**, then entering !itos and !itoa at the command prompt will both list the same thing: **<Subr: #2a664>**, which is the original subr definition for itoa. Note that (itoa) does not have mode or precision arguments, as they are irrelevant for integers.

Home work questions

1. Why did we suggest that you set your screens to 'text' mode before entering the prompt and printing functions?

2. Write a program to draw a square box with options to draw the outline of the box either as separate lines or as a polyline. Let the user enter L or P (upper or lower case) in order to choose the line or polyline option. Also, force the user to enter a non-nil, non-zero and non-negative length for the side of the square. **Hint:** the **(command)** function can use string variables as its arguments, so you can assign a variable to be the appropriate command with the **(getkword)** function.

3. Write a program that can insert any of six blocks, such as nut-and-bolt type fasteners, where the user selects the block name by entering the minimum number of characters from a keyword list. Have the program display a slide of the six items with labels for identification, so that the user can select the appropriate keyword. Redraw the screen to eliminate the slide before the selected block is inserted. Also, make the user pre-select an insertion point (for instance with the **(getpoint)** function) so that the 'insert' command works completely automatically with all of the subcommands provided. For those not familiar with slides, the command to make a slide is MSLIDE, and to view the slide, use VSLIDE with the slide filename as a subcommand.

In lesson six of this series, "Input/output and file handling with AutoLISP", you will learn more about printing functions for files and how to make 'clean' endings to your programs. The lesson will also cover debugging techniques, error handling and some hints on reducing the number of variables in your programs. Example programs will illustrate the new functions, and homework assignments will test your knowledge of the text.

An Introduction to AutoLISP

Input/Output and File Handling with AutoLISP
Lesson Six in the CADalyst/University of Wisconsin Introduction to AutoLISP Programming course covers printing functions, testing and debugging techniques.

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

About this course.

Introduction to AutoLISP Programming is an eight-lesson independent study course, originally published in monthly installments in CADalyst magazine. Readers may optionally register for the course in order to receive a copy of the printed course outline, a study guide containing additional AutoLISP examples, reference materials, and solutions to the practical assignments. On successfully completing an intermediate and a final mail-in project, students will receive certificates of completion and 4.0 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 - INPUT/OUTPUT AND FILE HANDLING WITH AUTOLISP

In this lesson you will learn more on printing, and how to use the print functions to get 'clean' endings to your programs. You will learn how to deal with external files for reading and writing data, and some example programs will illustrate techniques for file handling. You will also learn about error handling and testing and debugging your programs. Other techniques for writing modular programs will be introduced. As usual, homework assignments will test your knowledge of the text.

EXTERNAL FILES

AutoLISP has many functions that let you create and communicate with external files. You can use files to store data created during your programs, or you can read data from an external file to be used in your programs. An example of reading data is in parametric design, where the design parameters are stored in a text data file, which can be created with your favorite ASCII text editor.

The simplest way to store the design data in a file is a line-by-line sequence of values. For example, suppose we want to create an I-beam with the beam height, width, and web thickness, these values could be entered into a file as follows:

```
4.5
3.2
0.625
```

where each number represents the height, width, and thickness respectively. Now, suppose you call this file BEAMDATA.DAT, conforming to the naming conventions for DOS files, then the procedure to read the data into your IBEAM.LSP program would be:

1. Open the file called BEAMDATA.DAT in 'read-only' mode.
2. Read each line of the file in turn, and assign its value to an appropriate variable name.
3. Close the file.
4. Calculate any other data, and get an origin point to position the I-beam in your drawing.
5. Define any other required points, then draw and dimension the beam.

When you work with files like this, it is necessary to open and close them, as we indicate here. The AutoLISP function to open a file is

(open *filename mode*)

The filename and mode are strings, given in double quotes. The mode can be read "r", write "w", or append "a". Read mode means read-only, so that any data in the file is protected from being over-written, and you will use this mode when you want to extract data from the external file. Write mode allows you to write (or print) data to the file. If the file already contains data, the data will be over-written, because anything you write to the file gets written at the start, or beginning of the file.

If you want to add data to an existing file, then use the 'append' mode, which opens a file and positions itself at the end of the file, so that any data that is written to the file is placed after the existing data, instead of on top of it as is the case with 'write' mode. The mode string should be a single lower-case letter "r", "w", or "a".

The **(open)** function returns a file code number, which must be used for any subsequent communication (reading, writing, or printing) to and from the file. Therefore, it is usual to use **(setq)** with **(open)** as in:

(setq datf (open *filename mode*))

where the variable 'datf' is used to store the file code number, or file 'descriptor'.

To close a file that has been opened, you must use the file descriptor in the **(close)** function, as follows:

(close *filedescriptor*)

which, in the example given above, would be **(close datf)**.

READING DATA FROM EXTERNAL FILES

For the I-beam example, your program could start with the following extract from IBEAM5.LSP:

```
;IBEAM5.LSP  A program to take parametric data from an
;      external file to draw the I-beam. Uses (atof).
;
(defun  ibeam5  ()
;              Get the user input from the file BEAMDATA.DAT
        (setq  datf   (open   "beamdata.dat"   "r"))
        (setq  h  (atof  (read-line  datf))
               width  (atof  (read-line  datf))
               thickness  (atof  (read-line  datf))
               inner_height  (-  h  (*  2  thickness))
               inner_width  (/  (-  width  thickness)  2)
        )
        (close  datf)
        (setq   pt1  (getpoint  "\nLower left corner point: "))
        .
        .
        .
```

Two new functions are introduced here, **(read-line)** and **(atof)**. After the file has been opened in read-mode and the file descriptor has been assigned to the variable 'datf', the **(read-line)** function reads the first line of the file. **(read-line)** has one argument, the file descriptor, and it returns the 'next line' in the file as a string. This means that the file is automatically advanced to the next line so the sequence of **(read-line)** functions reads successive lines of the data file. If the actual data is a number, it is still returned as a string, which must be converted to a number, and there are some options as to how to do this. In the example shown, we have used the **(atof)** function, which converts 'alpha-to-floating point' (in other words 'string-to-real'). The usage of **(atof)** is simply **(atof *string*)**, where *string* is the string to be converted. In our example the string is returned from **(read-line)**.

The IBEAM5.LSP program reads a total of three lines altogether, converting each one to a real (decimal) number. There is also a function for converting strings to integers, the **(atoi)** function, which you can use in similar circumstances when you need to read in an integer from your external data file.

In our example, the other variables inner_height and inner_width are calculated for convenience so that they can be used later on in the point definitions for the outline of the I-beam cross-section.

There is a **(read)** function that has a string as its argument, and it returns the first item in the string converted to the appropriate data type, so if the first item in the string is a real, integer, list or actual string of characters, **(read)** will return it properly converted to its data type real, integer, etc. This is an alternative to the use of **(atof)** or **(atoi)**, and the above example would work just as well if we had programmed it as follows:

```
;IBEAM5.LSP  A program to take parametric data from an
;      external file to draw the I-beam. Uses (read).
;
(defun  ibeam5  ()
;               Get the user input from the file BEAMDATA.DAT
        (setq  datf   (open   "beamdata.dat"   "r"))
        (setq  h  (read  (read-line  datf))
               width  (read  (read-line  datf))
               thickness  (read  (read-line  datf))
               inner_height  (-  h  (*  2  thickness))
               inner_width  (/  (-  width  thickness)  2)
        )
        (close  datf)
        (setq   pt1  (getpoint  "\nLower left corner point: "))
        .
        .
```

Note that if you use **(read)** like this instead of **(atof)**, it returns whole numbers as integers, so you must be careful to create your parametric data in the appropriate form, using decimal format for whole numbers as necessary, otherwise you would need to specify the '2' in

```
               inner_height  (-  h  (*  2  thickness))
               inner_width  (/  (-  width  thickness)  2)
```

as 2.0:

```
            inner_height  (-  h  (*  2.0  thickness))
            inner_width  (/  (-  width  thickness)  2.0)
```

to guarantee that the variables are assigned as real numbers.

The BEAMDATA.DAT file can contain descriptive notes, such as:

```
4.5         I-Beam height
3.2         Width
0.625       Web and Flange Thickness
```

(read) only takes the first item of the string, so the descriptive text in your data file will be ignored, and **(atof)** also ignores the text part of the string if the numbers come first as we show here. You can test these concepts at your AutoCAD command prompt, as in the following sequence:

```
Command:  (setq  f  (open  "beamdata.dat"  "r"))
<File:  #2c4c4>


Command:  (setq  h1 (read-line  f))
"4.5\tI-Beam height  "


Command:  (atof  h1)
4.5


Command:  (read  h1)
4.5


Command:  (close  f)
nil


Command:
```

The file descriptor format is <File: #nnn>, and both **(atof)** and **(read)** return the real number 4.5, and not the following text. I used a tab key to create the BEAMDATA.DAT file, so the string that is returned from **(read-line)** shows the **\t** special character for the tab key. Don't forget to close the file after you have finished to avoid possible problems with exceeding the allowed number of open files specified in your config.sys file.

WRITING DATA TO EXTERNAL FILES

Other related functions for use with files are the print functions that we covered last month in lesson five of this series, **(princ)**, **(prin1)**, and **(print)**. In lesson five, we gave the syntax of these functions as

```
    (princ  expression  filename)
    (prin1  expression  filename)
    (print  expression  filename)
```

The filename in each of these cases is actually the 'file descriptor' that we defined above, so the usage of **(princ)** would be for example:

```
    .....
    (setq  f  (open  "outdata.txt"  "w"))
    (setq  x  (*  PI  0.25))
    (princ  x   f)
    (close  f)
    .......
```

The **(write-line)** function is the counterpart of **(read-line)**, and it only writes strings to a file, but does not write the double quotes associated with character strings. Note that strings can be string variables, so that

(write-line "Here is a string" f)

is equivalent to

```
(setq  strx  "Here is a string")
(write-line  strx   f)
```

(write-line) and the print functions allows you to save the results of your computational programs in separate files.

The filenames that are used for printing can be regarded as 'devices', such as your printer, so it is possible to write information directly to your printer by assigning the printer port LPT1 (or possibly PRN) for parallel ports, or COM1 for serial ports, assuming your computer is configured in this way. For example, if you wanted to list the height and cross-sectional area of a closed polyline (possibly a simple I-beam cross-section), the area command stores the area in the system variable AREA, so you could write:

```
    ....
    (command  "pline"  p1 p2 p3 p4 p5 p6 p7 p8 p9
                                           p10 p11 p12 "c")
    (command  "area"  "e"  "L")
    (setq  csa  (getvar  "AREA"))
    (setq  pf  (open  "lpt1"  "w"))  ; opens the LPT1 port
    (write-line  "BEAM DATA"  pf)
    (princ  "Beam height = "  pf)  (princ  h  pf)  (princ  "\n"  pf)
    (write-line (strcat  "Cross-section area = "  (rtos  csa))  pf)
    (close  pf)
    ...
```

Here, we used **(write-line)** for the first string, then **(princ)** was used three times to write a string, followed by a real number variable, followed by a new-line. An alternative is to use **(write-line)** with **(strcat)** and **(rtos)** to convert numbers to strings, to join strings together, and to place a new-line automatically.

USING THE PRINT FUNCTIONS TO GET CLEAN ENDINGS TO YOUR PROGRAMS

The print functions return absolutely nothing (not even 'nil'), and this can be used to take care of having 'nil' written every time you run a program. Simply use **(print)** or **(princ)** with no arguments at the end of your programs (just before the final close of the defun), as in this example of the rectangle program:

```
    ;;  RECTANG1.LSP  A program to draw a rectangle
    ;;  using the (getcorner) function
    ;;
    (defun  rectang1  ()
         (setq  pnt1  (getpoint  "First corner  "))  (terpri)
         (setq  pnt3  (getcorner  pnt1  "Opposite corner..."))  (terpri)
         (setq  pnt2  (list  (car  pnt1)  (cadr  pnt3)))
         (setq  pnt4  (list  (car  pnt3)  (cadr  pnt1)))
         (command  "line"  pnt1  pnt2  pnt3  pnt4  "c")
```

```
      (redraw)
      (princ)    ; the way to get a clean ending to the program
   ) ;  end of the defun
```

DEBUGGING TECHNIQUES

Most of the problems you will encounter in your early days of programming will be associated with mismatched parentheses, wrong spelling of variable names, and trying to use the wrong type of argument in a function. Unfortunately, the error messages that AutoLISP produces may not indicate a direct problem. For instance, if you open some double quotes at the start of a string prompt, but you fail to close the double quotes, the next double quotes in your program will be taken to be the missing 'closing' quotes. This can result in the message 'exceeded maximum string length', which is a straight forward problem to resolve, but it may also result in a 'malformed string' error if the next double quotes do not cause a very long string. Consider the following program, LSHAP.LSP, which draws an 'L' shaped cross-section:

```
   (defun LSHAP ()
        (setq P0 (GETPOINT "\nEnter Start Point: "))
        (setq P1 (GETPOINT P0 "\nEnter Second Point: "))
        (setq ang1 (angle p0 p1))
        (setq d1 (distance p0 p1))
        (setq a90 (* pi 0.5))
        (setq ang2 (+ ang1 a90))
        (setq ang3 (+ ang2 a90))
        (setq th (* d1 0.25))
        (setq d2 (- d1 th) )
        (setq p2 (polar p1 ang2 th) )
        (setq p3 (polar p2 ang3 d2))
        (setq p4 (polar p3 ang2 d2))
        (setq p5 (polar p4 ang3 th) )
         (command "line" p0 p1 p2 p3 p4 p5 "close")
   ;
   ;**************** adding fillet radii ***********
   ;

      (setq pf31 (polar p2 ang3 (* d2 0.6))
            pf32 (polar p3 ang2 (* d2 0.4))
            pf21 (polar p1 ang2 (*  th 0.6))
            pf22 (polar p2 ang3 (* d2 0.4))
            pf41 (polar p3 ang2 (* d2 0.6))
            pf42 (polar p4 ang3 (*  th 0.4))
      );  set fillet points
         (setq frad (* th 0.75))
      (command "fillet" "r" frad
               "fillet" pf21 pf22
               "fillet" pf31 pf32
               "fillet" pf41 pf42 "redraw")
   )
```

In this example, if a set of double quotes is missing from the **(GETPOINT)** functions, the 'exceeded maximum string length' results, but leaving off any of the double quotes at the command fillet section results in a 'malformed string'.

SPECIAL CASE ERRORS

The LSHAP.LSP program draws the shape satisfactorily if it is displayed as a large size on your screen, but fails for very small displays, as shown in figure 6-1.

The command prompt area from figure 6-1 shows on the text screen as:

```
   Command:  (lshap)

   Enter Start Point:
   Enter Second Point:  line  From  point:
   To point:
   To point:
   To point:
   To point:
```

```
   To point:
   To point:  close
   Command:  fillet Polyline/Radius/<Select first object>:  r  Enter fillet
radius <0.
   6343>:  0.066200060300832
   Command:  fillet  Polyline/Radius/<Select first object>:
   Select second object:
   Lines are parallel
   *Invalid*
   error:  Function cancelled
   (COMMAND "fillet" "r" FRAD "fillet" PF21 PF22 "fillet" PF31 PF32 "fillet"
PF41 P
   F42  "redraw")
   (LSHAP)

   Command:
```

The error here is the invalidity of filleting parallel lines. However, the program clearly places points at fillet pick-positions that are obviously on separate lines (see the code that defines the new points for filleting in the LSHAP.LSP program). The problem here is that even though points PF21 and PF22 are on separate lines, AutoCAD uses the 'pickbox' method of selecting the objects, and in this case, the drawing of the part is too small for the pickbox to distinguish between the lines.

This shows that it is possible for programs to fail simply due to unusual cases in which they are applied. It is often the case that your programs work perfectly well, until you let someone else use them, and software development relies very much on third party testing to ensure that the software is as fool-proof as possible.

TEN DEBUGGING TECHNIQUES

1. Use semicolons to comment out sections of your code in order to track down errors.

2. Read the AutoLISP returned error messages in the text-screen (use the F1 key).

3. Write **(prompt)** or **(princ)** statements to check that the execution has progressed to certain key places. For example, you might place the statement **(prompt "I made it to here!...")**

4. You can pause your programs by inserting the **(getint)** statement as in **(getint "\nHit return to continue...")**, because instead of actually supplying an integer, you may also return the 'enter' key for a null input. The program will pause until you enter, so you will know that the program executed properly until that point, and your debugging can continue in the later statements.

5. Write modular programs to separate sections of defined functions (see the next section)

6. Use template, or 'prototype' programs for common sections of code.

7. Use meaningful variable names.

8. Write 'readable' code with sufficient annotation.

9. Test for limiting values to find 'special case' errors.

10. Use the **(trace)** and **(untrace)** functions to help in debugging. The **(trace function1 ...)** function is used for debugging one or more functions that you supply as the arguments of **(trace)**. After this statement, each time the function(s) called by trace is/are evaluated, a display of the function appears, and the result of the function is printed. **(untrace)** takes the **(trace)** off.

WRITING MODULAR PROGRAMS

You can split your programs up into modules that can be executed and debugged separately (or sometimes in sequence), in order to make the debugging process more tractable. In the case of the LSHAP.LSP program, a modular approach might look like:

```
   (defun inputdata ()
       (setq   P0 (GETPOINT "\nEnter Start Point: ")
                    P1 (GETPOINT P0 "\nEnter Second Point: ")
              ang1 (angle p0 p1)
```

```
                    d1 (distance p0 p1)
                    a90 (* pi 0.5)
                    ang2 (+ ang1 a90)
                    ang3 (+ ang2 a90)
                    th (* d1 0.25)
                    d2 (- d1 th)
                    p2 (polar p1 ang2 th)
                    p3 (polar p2 ang3 d2)
               p4 (polar p3 ang2 d2)
               p5 (polar p4 ang3 th)
          ) ; end of setq
     ) ; end of defun inputdata


     ;**************** adding fillet radii ***********
     (defun  fillt  (filrad)
          (setq   pf31 (polar p2 ang3 (* d2 0.6))
                  pf32 (polar p3 ang2 (* d2 0.4))
                  pf21 (polar p1 ang2 (*  th 0.6))
                  pf22 (polar p2 ang3 (* d2 0.4))
                  pf41 (polar p3 ang2 (* d2 0.6))
                  pf42 (polar p4 ang3 (*  th 0.4))
          );  set fillet points
        (command "fillet" "r" filrad
                  "fillet" pf21 pf22
                  "fillet" pf31 pf32
                  "fillet" pf41 pf42 "redraw")
     ) ; end of defun fillt




     (defun  draw1  ()
           (command "line" p0 p1 p2 p3 p4 p5 "close")
     ) ; end of defun draw1

     (defun  drawfils  ()
     ;  set the fillet radii
          (setq  frad  (*  th  0.75))
          (fillt  frad)
     ) ; end of defun drawfils

     (defun  LSHAP  ()
          (inputdata)
          (draw1)
          (drawfils)
          (princ)
     ) ; end of defun lshap
```

Here, the program has been split into a number of defined functions. When all of the defined functions are written in sequence in a program, the name of the last defined function of the sequence is returned on loading the program.

After each individual part has been run and debugged as a separate program, you can assemble the parts in the order shown so that the defun LSHAP is last as shown, and its name is echoed at the command prompt when the program is loaded. The defun LSHAP executes three other defined functions, **(inputdata)**, **(draw1)**, and **(drawfils)**, then the program ends with a **(princ)** to get a clean ending.

(inputdata) collects the two required input points, then calculates all of the other corner points of the L shape. This function can be written and debugged completely separately from any of the other functions.

The next defined function **(draw1)** requires that the points p0 through p5 are defined, which means that **(inputdata)** must correctly acquire and calculate these points before **(draw1)** can be executed. **(draw1)** then completes the simple task of drawing the basic outline of the L shape.

(drawfils) executes the defined function **(fillt)**, which must therefore be programmed and running properly before **(drawfils)** can be run. **(fillt)** has an argument (the fillet radius), which is normally supplied by **(drawfils)**. However, **(fillt)** can still be executed independently, but following the execution of **(inputdata)** and **(draw1)**, by supplying a fillet radius such as 0.5 in the following:

```
Command:  (load  "/acad11/lisp/fillt")
(FILLT)
Command:  (fillt  0.5)
```

After you have run and debugged **(inputdata)**, **(draw1)** and **(fillt)**, you can run **(drawfils)**. The way to run **(drawfils)** independently is to first load and execute **(inputdata)** and **(draw1)**, and then load, but do not execute **(fillt)**. Then load and execute **(drawfils)**, which will automatically execute **(fillt)** as it is intended to do.

This type of load and execute lets you run each defined function as a separate program for the purpose of debugging. After each of the individual programs is running properly, you can combine all of the functions into a single file, using your text editor.

Alternatively, you may write all of the sections in turn in one long program, debugging each section as it is written in turn. Remember that only the last defined function is returned at the command prompt, so if you want to execute the preceding defined functions you must remember their names and execute them in sequence as required.

Home work assignments

This month's assignment includes reading parametric data from an external file, writing data to another external file, and writing modular programs:

Write a program to draw two views of a base-plate as shown in figure 6-2. Your program should take the design parameters, height, width, thickness, corner radius, diameter and location of all of the mounting holes shown from an external file called BASEPLAT.DAT. Your program should calculate the area of the top view (Hint, draw the outside outline as a polyline rectangle, then fillet the "Last" polyline object, then use the AREA command as we have shown in this lesson). You can calculate the areas of the holes separately and subtract them from the plate area to arrive at the net area. Next, calculate the volume of the plate by multiplying the net area by the plate thickness. Place center marks as shown on all of the circles, but any other dimensioning is entirely optional.

Write the area and volume out to an external file called BASEPROP.DAT.

Write separate program modules (defined functions) to: 1) Take input data, including the plate center position as an origin from the user, and the parametric data from the external file; 2) Draw the top view; 3) Draw the side view; 4) Calculate the net area and volume; and 5) run the whole program.

You might also use colors for your programs. To do this use **(command "color" 1)**, where 1 is red, 2 is yellow, 3 is green etc.

ERRATA: Mr. Lee Moore of Facility Management, BIA in South Dakota spotted an error in Lesson Three, in the July issue of CADalyst. Near the end of the article, I invited you to type in some statements aimed at using the **(append)** function as follows:

(setq ptlst (append ptlst (list pt2)))

this returns ((2.72 1.39 0.0) (4.43 1.39 0.0)). Note that we now have a list of 2 items, each of which is itself a list of 3 items. Now try adding the third point without the 'list' function:

(setq pt3 (append ptlst pt3))

this returns ((2.72 1.39 0.0) (4.3 1.39 0.0) 4.43 2.85 0.0)

Lee correctly observed that the last **(setq)** should be **(setq ptlst (append ptlst pt3))**. Well spotted Lee! The intention here was to update the point list variable ptlst, and not the point pt3. The danger here is that the statement as written does in fact return the same result as the correct version, but of course, the intended variables will not contain the correct information. I will be even more diligent in future, as this was most definitely not a 'deliberate' mistake.

In lesson seven of this series, "Logic and branching out with AutoLISP", you will learn some of the more sophisticated programming capabilities of AutoLISP, including relational and logical functions. You will also learn how to impose conditions with the branching functions (if) and (cond).

An Introduction to AutoLISP

Logic and Branching out with AutoLISP

Lesson Seven in the CADalyst
University of Wisconsin Introduction to AutoLISP Programming course covers relational, logical, conditional and branching functions.

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

About this course.

Introduction to AutoLISP Programming is an eight-lesson independent study course, originally published in monthly installments in CADalyst magazine. Readers may optionally register for the course in order to receive a copy of the printed course outline, a study guide containing additional AutoLISP examples, reference materials, and solutions to the practical assignments. On successfully completing an intermediate and a final mail-in project, students will receive certificates of completion and 4.0 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 - LOGIC AND BRANCHING OUT WITH AUTOLISP

In this lesson you will learn some of the more sophisticated programming capabilities of AutoLISP, including relational and logical functions. You will also learn how to impose conditions with the branching functions **(if)** and **(cond)**, and you will see many examples of the use of these intelligent decision-making functions in example programs and parts of programs. The lesson ends with some suitable homework questions to test your knowledge of the text.

DECISION MAKING IN AUTOLISP

In lesson five, you saw that it is possible to make choices with suitable prompting from **(getkword)**. This month you will learn that your programs can make certain choices automatically, with intelligent decisions based on your own design rules.

Let's start with the IBEAM that we have used quite a lot in this series of lessons. We can impose some rules on the design of the I-beam, for instance by specifying that the thickness of the web should not exceed 25% of the width of the cross-section. Figure 7-1 shows the I-beam with the dimensions of Height HT, flange thickness FT, web thickness WT, width WI, and radius RAD.

To implement our first design rule, part of your program could read as follows:

```
(defun  ibeam  ()
.
.
   (setq WI  (getreal  "\nBeam width: "))
   (setq WT (getreal  "\nWeb thickness: "))
   (if  (>  WT  (*  WI  0.25))  ; test the design rule
       (setq  WT  (*  WI  0.25))  ; impose the restriction
   ) ; end of if
.
.
) ; end of defun
```

Here, we allow the designer to choose the beam width, **WI**, and the web thickness, **WT**, independently, but we check to see if the web thickness is greater than 0.25 x WI. If this is so, then we force WT to be equal to 0.25 x WI. To accomplish this, we have used two new functions, if **(if)** and greater than **(>)**. The **(if ..)** function works by using a test expression, followed by an expression which is evaluated if the test expression is valid, in other words if it is not null. If the test expression is null, the following expression is ignored, and an optional 'else' expression is evaluated. We will return to the **(if)** function later in this lesson, where you will learn more of the 'else' expression.

RELATIONAL FUNCTIONS

The test expression uses a relational function, the greater than **(>)** function. Relational functions return **T** if the relationship is true, and nil if it is not. In this case, the greater than test expression has the format **(> atom1 atom2).** If **atom1** is numerically greater than **atom2**, this returns **T**. Note that lists cannot be compared, only expressions that have a single value (atoms), but these can include strings.

The atoms in our example are **WT** and **(* WI 0.25)** respectively, so as long as **WT**, the web thickness is not greater than **0.25 x WI**, the test will return nil. The next expression, **(setq WT (* WI 0.25))** will then be ignored, and the **(if)** statement will end as shown. The designer therefore may choose any value of **WT** less than or equal to **WI/4**, but the program will prevent greater values from being used.

Relational functions have the format

(*relation atom1 atom2 atom3 ......*)

where the relation applies to each successive atom in turn, so that in the case of greater than, the function returns **T** if the atom1 is greater than atom2, which is in turn greater than atom3 and so on for all of the succeeding atoms. The relational functions are valid for numbers and strings.

Other relational functions are:

(= atom1 atom2 ...) The equal to function returns **T** if the atoms are equal, and nil otherwise.
(/= atom1 atom2 ...) The 'not equal to' function returns **T** if the atoms are not equal, and nil otherwise.
(< atom1 atom2 ...) The 'less than' function returns **T** if the first atom is less than the second.
(<= atom1 atom2 ...) The 'less than or equal to' function returns **T** if the first atom is less than or equal to the second.
(>= atom1 atom2 ...) The 'greater than or equal to' function returns **T** if the first atom is greater than or equal to the second.

Here are some other functions that may be used as tests with the **(if)** function:

(eq expression1 expression2) returns **T** if the expressions are identically equal. The expressions can be lists, but they must be exactly the same list.
(equal expression1 expression2 tolerance) returns **T** if the expressions evaluate to the same thing. The expressions can be lists. If the expressions evaluate to be numbers, the optional tolerance can be specified to determine how closely the numbers should match. This is useful in cases where some numerical round-off errors may otherwise prevent an exactly equal comparison.
(listp item) returns **T** if the item is a list, and nil otherwise.
(minusp number) returns **T** if the argument is a negative number.
(not item), **(null item)** both of these functions return **T** only if the item is nil.

Consider the case where you need to constrain both the minimum and maximum web thickness. Suppose the minimum value of **WT** is to be one tenth of the beam width, and its maximum value is as before, one quarter of the beam width. There are now two conditions to be satisfied, so following the previous example, we could simply write:

```
(defun  ibeam  ()
.
.
   (setq WI  (getreal  "\nBeam width: "))
   (setq WT (getreal  "\nWeb thickness: "))
   (if  (>  WT  (*  WI  0.25))  ; test the upper bound
       (setq  WT  (*  WI  0.25))  ; impose the restriction
   ) ; end of if
   (if  (<  WT  (*  WI  0.1))  ; test the lower bound
       (setq  WT  (*  WI  0.1))  ; impose the restriction
   ) ; end of if
.
.
) ; end of defun
```

This example is a perfectly valid way to test for each of the conditions for the minimum and maximum web thicknesses, but here is a more appropriate function for testing multiple conditions, the **(cond)** function:

BRANCHING FUNCTIONS FOR DECISION MAKING

There are two branching functions:

```
(cond  (test1  then1)  (test2  then2)  ....)
(if  test  then  else)
```

CONDITIONAL EVALUATION WITH (COND)

The **(cond)** function evaluates each test expression in turn until a test expression returns a value other than nil. The 'then' expression which is associated with the non-nil test is then evaluated, and the rest of the pairs of expressions are ignored. Note that there can be no 'else' expression in any of the argument lists. We could write the last example as:

```
(defun  ibeam  ()
.
.
   (setq WI  (getreal  "\nBeam width: "))
   (setq WT (getreal  "\nWeb thickness: "))
   (cond
      ((>  WT  (*  WI  0.25))  (setq  WT  (*  WI  0.25)))
      ((<  WT  (*  WI  0.1))     (setq  WT  (*  WI  0.1)))
   ) ; end of cond
.
.
) ; end of defun
```

Here we have included both the maximum and minimum tests for WT in a single conditional statement. Now, if the web thickness exceeds its maximum value, the **(cond)** function ignores all following tests, and the program continues after the 'end of cond' parenthesis.

Compare this with the two **(if)** statement approach, in which both cases must be tested even if the first one is satisfied. The **(cond)** approach is more efficient than having several sequential **(if)** functions, especially if you place the most likely conditions first, because as soon as a non-nil test is encountered, **(cond)** does not evaluate any further tests. Note the double open parentheses which are used to group the 'test' and 'then' expressions into pairs.

Now consider the case where the designer has chosen a valid web thickness within the design constraints. In both the double **(if)** program and the **(cond)** program, the tests would both fail (return nil), and the 'then' expressions would both be ignored, and the program continues normally. However, in the case of the **(cond)** function, there is an opportunity to note the fact that none of the conditions were met, by placing a dummy condition that will always be 'true' (non-nil) at the end of all of the test-then pairs of conditions.

This type of 'default' case is important when the 'then' result is the only way that a variable obtains its value. In the case of the web thickness, the value of WT already exists before the **(cond)** function is entered, so that value serves as the default. However, it is possible that a default value does not exist. Let's suppose that we want to use a 'standard' corner radius RAD whose value depends upon ranges of beam heights according to the following table:

Beam height HT	Corner radius RAD
under 2"	0.125"
2" to 4"	0.25"
over 4" to 6"	0.5"
over 6"	0.625"

The **(cond)** function could be programmed as follows to meet these design criteria:

```
(defun  ibeam  ()
.
.
   (setq HT  (getreal  "\nBeam height: "))
   (cond
      ((<  HT  2.0)  (setq  RAD 0.125))
      ((and  (>=  HT  2.0)    (<=  HT  4.0))  (setq  RAD 0.25))
```

```
        ((and  (>  HT  4.0)   (<=  HT  6.0))   (setq  RAD  0.5))
        (T   (setq  RAD  0.625))
     ) ; end of cond
  .
  .
  ) ; end of defun
```

We have introduced two new functions, **(and)** and **(T)**.

In fact **T** is not a function, but is an atom in the standard atomlist, whose value is **T**, and therefore not nil. Remember that we used this atom in the **(getstring)** function in lesson five of this series to indicate a non-nil flag. Well, here it is again, serving as an expression which is again non-nil, thus guaranteeing that if all else fails, **(cond)** will always find this condition, and will evaluate the corresponding 'then' expression that follows. Here we are setting the radius equal to 0.625 if all other tests fail.

The **(and)** function returns **T** only if all of its succeeding expressions return non-nil. The format is:

(and *expression1 expression2* ....)

where any number of expressions can be used as successive tests. The tests in this case cover a range of values, so we are really testing that RAD lies within a certain range of values. The first **(and)** in our example is

(and (>= HT 2.0) (<= HT 4.0))

which uses two expressions. The first expression is the relational function greater than or equal to, and it returns **T** if HT is greater than or equal to 2.0. Similarly, the second expression returns **T** if **HT** is less than or equal to 4.0. If either one of the expressions does not return **T**, in other words if either of the expressions returns nil, the entire **(and)** function will return nil, and that means **HT** is outside the range of 2.0 and 4.0.

In the example, the **(and)** functions are the 'tests' of the **(cond)** function, so the parentheses group them together with their corresponding 'then' expressions.

Another related function for decision making is **(or *expression1 expression2* ...)**. This function returns **T** if any one of its expressions returns **T**. The list of expressions is examined in turn, and as soon as a **T** is returned, the evaluation is stopped, and the **(or)** function returns **T**. This function is the opposite of **(and)**, because in order to return nil, all of the expressions of **(or)** must return nil.

CONDITIONAL EVALUATION WITH (IF)

We have seen examples of **(if)** functions that used the 'test' and 'then' expressions, but so far we have not used the 'else' expression which is evaluated if the **(if)** test expression is nil. The full syntax of **(if)** is

(if *test then else*)

This function evaluates the 'test' expression, and if this is not nil, it evaluates the 'then' expression, in which case the 'else' expression is ignored. If the 'test' expression returns nil, the 'then' expression is ignored, and the 'else' expression is evaluated. The 'else' expression is optional, in which case the 'then' expression is evaluated only if the 'test' expression returns a non-nil value.

For example, in a cross-hatching program that we will publish in full in our last lesson next month, the hatch boundary is supposed to be created on a layer called NOPLOT, and the hatch pattern is to be created on the layer HATCH. If these layers don't exist, we give the user the chance to have the layers created automatically, or else the program can be terminated with a 'quit' option. We might therefore have:

```
   (setq ans2 (getstring
      "\n Shall I create 'noplot' & 'hatch' or quit?
      enter Y (Create layers) or Q (Quit):  ")) (terpri)
   (if
      (or (=  ans2 "Y")  (=  ans2 "y"))  ; the test expression
      (command "LAYER" "N" "hatch,noplot" ; the 'then'
         "C" "6" "hatch" "C" "2" "noplot" "") ; expression
      (quit)  ; the 'else' expression
   );   end of if "create layers or quit"
```

Here we use **(or)** to accept answers in upper or lower case, and if the answer to the **(getstring)** prompt is either Y or y, the layers are created and assigned colors, or else the function **(quit)** is used to terminate the program. Two functions,

(quit) and **(exit)** don't appear to be documented in the AutoLISP Programmer's Reference, but they are in the atomlist. Either of these functions will abort the current lisp program.

As an alternative to **(or)**, you can use the function **(strcase *string*)**, which returns the string argument in upper case as in:

```
(if
   (=  (strcase  ans2)  "Y")  ; the test expression
   (command "LAYER" "N" "hatch,noplot" ; the 'then'
      "C" "6" "hatch" "C" "2" "noplot" "") ; expression
   (quit)  ; the 'else' expression
);   end of if "create layers or quit"
```

Even better is to guarantee an upper case **"Y"** or **"Q"** by using **(initget)** with **(getkword)** as in:

```
(initget  1  "Y   Q")
(setq ans2 (getkword
   "\n Shall I create 'noplot' & 'hatch' or quit?
   enter Y (Create layers) or Q (Quit):  ")) (terpri)
(if
   (=  ans2 "Y") ; the test expression
   (command "LAYER" "N" "hatch,noplot" ; the 'then'
      "C" "6" "hatch" "C" "2" "noplot" "") ; expression
   (quit)  ; the 'else' expression
);   end of if "create layers or quit"
```

We have covered the **(if)** function for a single 'then' expression and a single 'else' expression. What happens if you want to do several actions if a test expression is not nil It appears from the above that if there were two 'then' statements, the second of them would be assumed to be the 'else' expression. The **(progn)** function gives us a way to handle such situations:

(progn *expression1 expression2* ...)

(progn) evaluates each of its expressions in turn, but is itself a 'single' expression. You can use **(progn)** as the 'then' and/or the 'else' expressions of the **(if)** function, and in any other circumstances when a single expression is expected.

The following example is another part of the cross-hatching program that we will list in full next month. The extract shows that you can use logical branching to set default values in AutoCAD prompts, and that you can 'nest' **(if)** statements inside each other for more complex options and decision-making.

This part of the hatch program asks the user if a HATCH pattern has already been selected, and the input of a Yes or Select is enforced. The first **(if)** tests that the response is "Select", and if this is the case, a series of expressions that follow in the **(progn)** function are evaluated. These expressions initialize the name of the hatch pattern m:hp to nil, then the command HATCH is given with the ? option to list the names of the hatch patterns on the screen, then **(getstring)** is used to obtain the name of the hatch pattern.

```
   (initget 1 "Yes  Select")
   (setq YS (getkword "\n\n Have you pre-selected a hatch pattern?
           enter Y to continue or S to select from a list: "))
   (if  (=  YS "Select")
           (progn              ;progn for selecting the hatch pattern
                 (setq m:hp nil)
                 (command "HATCH" "?" "")
                 (setq m:hp (getstring "\n\nEnter hatch pattern name <ANSI31>:
"))
                 (if   (=  m:hp "")
                        (setq m:hp "ANSI31") ;hatch pattern default
                 )              ;end of if m:hp is equal to ""
           )                   ;end of progn for auto-hatch pattern
   )                           ;end of if for hatch pattern selection
```

The 'default' name ANSI31 is given here, so that if the user simply enters a 'return', the following **(if)** tests that m:hp is equal to **""**, and if this is so, the name m:hp is set to "ANSI31". Neither of the **(if)** statements in this example uses the 'else' option, as it is not needed in either case.

The hatch pattern variable m:hp used by AutoCAD. The AutoCAD release 11 menu ACAD.MNU makes reference to this variable in the HATCH menus. Therefore, when this variable is set, it becomes the default for future hatching just as though it had been pre-selected in AutoCAD. Note that the test for a 'return' response for **(getstring)** is **(= m:hp "")**, and not **(= m:hp nil)** or **(null m:hp)**, because **(getstring)** takes input in the form of a string, and the double quotes must be used. This is different from the **(getkword)**, which returns nil if you enter a return only.

Note also that AutoCAD internal variable names usually have the colon (**:**) as the second character in order to be somewhat distinct from variable names that you are likely to use. Be warned that use of the colon in the names of variables may change the values of AutoCAD variables, and you should be careful when you assign values to the internal variables such as m:hp.

Home work assignment.

1. Write a program for the I-Beam of figure 7-1. Let the user input all of the values shown in the figure, but automatically enforce all of the following design rules.

(i) The beam width WI should never exceed 0.8xHT, and should never be less than 0.5xHT.
(ii) The flange thickness FT must be in the range 0.05xHT to 0.15xHT
(iii) The web thickness WT must be in the range 0.1xWI to 0.2xWI
(iv) The corner radius RAD should have either of two standard values of 0.125" and 0.5", corresponding to beam heights HT less than or equal to 4", and HT greater than 4" respectively, except where the flange thickness is less than 0.15". If FT is less than 0.15", RAD should be 0.8x FT.
(v) The beam height takes precedence over all other values, and should be used as it is provided.

In lesson eight, the last of this present series, "Intelligent Programming with AutoLISP", you will learn more of the sophisticated programming capabilities of AutoLISP, including the looping functions which can be used to perform repetitive operations. An example of an intelligent cross-hatching program will illustrate the concepts of looping. You will also learn how to program your menus for an efficient AutoLISP development environment.

Intelligent Programming with AutoLISP

Lesson Eight in the CADalyst/University of Wisconsin Introduction to AutoLISP Programming course covers looping functions and some useful menus for AutoLISP development.

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

About this course.

Introduction to AutoLISP Programming is an eight-lesson independent study course, originally published in monthly installments in CADalyst magazine. Readers may optionally register for the course in order to receive a copy of the printed course outline, a study guide containing additional AutoLISP examples, reference materials, and solutions to the practical assignments. On successfully completing an intermediate and a final mail-in project, students will receive certificates of completion and 4.0 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 - INTELLIGENT PROGRAMMING WITH AUTOLISP

In this, the final lesson of this introductory course, you will learn about the looping functions of AutoLISP, with an example of a cross-hatching program which uses many of the functions that have appeared throughout the course. You will also see how to make your own menus for an AutoLISP development environment. The lesson ends with some suitable homework questions to test your knowledge of the text.

LOOPING FUNCTIONS

Looping means that you want to continue to execute a set of activities for as long as, or until, some criterion is met. One possible criterion is to continue executing the set of activities until you cancel with a control-C. This approach is not normally highly recommended, but we mention it because the most frequent problem that new programmers get when they first write programs with looping functions is that the program gets 'stuck' in an infinite loop, and the only way out is to cancel.

We will cover three looping functions here:

(repeat *integer expression1 expression2* ....)

(foreach *item list expression1 expression2* ...)

(while *test expression1 expression2* ...)

All of the looping functions can have any number of expressions which are evaluated each time the loop is run. The simplest looping function is **(repeat)**, which repeats execution by evaluating each expression in turn a set number of times, determined by the 'integer' argument.

The following program CONCIRCS.LSP is an example of a program that will draw any number of concentric circles, each one increasing in diameter by an amount 'delta', supplied by the user. The program also takes input from the user for the number of concentric circles, the center point of the set of circles, and the initial diameter.

```
(defun  concircs  ()
   (setq  num  (getint  "\nNumber of circles: "))
   (setq  p  (getpoint  "\nCircle center point: "))
   (setq  dia  (getreal  "\nInitial diameter:  "))
   (setq  delta  (getreal  "\nDiameter increment: "))
```

```
    (repeat  num
        (command  "CIRCLE"  p  "D"  dia)
        (setq  dia  (+  dia  delta))
    ) ; end of repeat
) ; end of defun
```

The repeat loop in this example has the user supplied 'num' as its integer, for the number of circles to be drawn. The expressions evaluated in the repeat loop are the **(command "CIRCLE"...)** and the reassignment of the diameter of the circle.

The next looping function, **(foreach)**, is commonly used in connection with lists of points. In lesson three of this series, you saw that it is possible to append lists of point coordinates to a 'point' list. Now, each item of that point list can be operated upon in sequence, as in **(foreach pt ptlst (command pt))** which assigns the name 'pt' to each item of the list 'ptlst' in turn, and then evaluates the expression **(command pt)** for each item in the list. The **(command)** function is used here simply to 'dump out' the item pt, just as though it was entered from the keyboard. Note that **(command)** does exactly the same thing with AutoCAD commands and sub-commands - it dumps them out as though they were keyed-in. This is a useful way to put point values into line and/or pline commands as in:

(command "pline" (foreach pt ptlst (command pt)))

GRAPH2.LSP is a program which use **(foreach)** to plot a graph after reading the y-ordinate values from an external file.

```
;; GRAPH2.LSP  A program to plot a graph on the graphic screen, taking
;; any number of y-ordinate values from an external file (.txt)
(defun graph2 ()
;                    Initialization and read data from the file
     (setq ptlst nil  px  (-  1))
     (setq fname (getstring "\nEnter the txt data file name: "))
     (setq fname (strcat fname ".txt"))
     (setq datafile (open fname "r"))   ; opens file in read mode

     (if datafile
         (progn
            (setq numpoints (read (read-line datafile)))
               (repeat numpoints
                  (setq px (1+  px))
                  (setq py (read (read-line  datafile)))
                  (setq pp (list px py))
                  (setq ptlst (append ptlst (list pp)))
                ) ; end of the repeat loop
                (command "pline" (foreach pt ptlst (command pt)))
                (close datafile)
         ) ; end of progn
         (princ (strcat "Cannot open file " fname))
     ) ; end of 'if'
     (redraw)  (princ)
) ; end of defun
```

GRAPH2.LSP assumes that the y-ordinates are stored in a file with the file extension .TXT, and this is concatenated to the filename that is input by the user. Notice how the 'if' statement uses the resulting file name as the test. This ensures that the file descriptor actually exists, otherwise the message "Cannot open file...." is displayed. If the file is successfully opened, the first line of the file is read in and is taken as the number of y-values to follow. The file contains the number of y-values, followed by each y-value in turn on a separate line as in this data file:

```
12
0.0
1.567
1.874
4.682
3.675
...etc
```

In this case, the variable numpoints would be set to 12, and the repeat loop will thus execute or 'loop' 12 times. The repeat loop sets the x-ordinate px by adding an increment of 1 to its previous value. The y-ordinate is then read from the external file and a point is defined by 'listing' the x and y values together before appending the points to a pointlist 'ptlst'.

The pline command uses the foreach loop to dump out the points from the point list, the external file is closed and the 'if' statement is completed. Finally, the graph is redrawn to clean up the picture. This program may be extended to include the x and y axes with suitable lines and text.

The third looping function **(while)** continues to evaluate expressions while, or for as long as, some test condition is true. In our I-BEAM example of lesson 7, you used branching statements to force the web thickness of the beam to be no greater than the width of the beam. However, it is possible that the user simply entered the wrong number, in which case it would be nice to have the opportunity to correct it while still imposing the design rules.

Consider the case where you need to constrain the maximum web thickness WT to be one quarter of the beam width WI. The conditional statement of the example of lesson 7 can be replaced by a loop as shown in this partial program for the IBEAM.

```
(defun  ibeam  ()
.
.
   (setq WI  (getreal  "\nBeam width: "))
   (setq WT (getreal  "\nWeb thickness: "))
   (while  (>  WT  (*  WI  0.25))  ; test the upper bound
       (princ  "\nEnter a value no greater than  ")
       (princ  (*  WI  0.25))  (princ  ": ")
       (setq  WT  (getreal))
   ) ; end of while
.
.
) ; end of defun
```

Here, instead of simply setting the value of the web thickness WT to a quarter of the beam width, the user is forced by the while loop to enter an appropriate value to meet the design rule.

The following program, HATCH2.LSP, has some other examples of the use of branching and looping functions. HATCH2.LSP is a way to cross-hatch an area without the need to break the hatch boundary. The program creates a polyline which is used as the hatch boundary. The user indicates intersection points by tracing each point in turn around some existing geometry, and the program automatically stops collecting boundary points when any point is coincident with the first point selected.

This program was first written in 1987, and was last revised in March 1991, so it is getting quite elderly, but it still serves as a useful training tool to show the looping and branching functions.

```
;  HATCH2.LSP VER 4, MARCH 1991
;
;  A Hatch program by Tony Hotchkiss
;
;
;  HATCH2 is designed to allow cross hatching when the hatch boundaries
;  are not necessarily contiguous. This feature eliminates the need
;  for breaking lines before hatching.
;
;  The following intelligent features are included in version 4:
;
;    1)  The program uses layers 'hatch' and 'noplot'.  The user is
;        asked whether or not these layers exist.  If the layers do
;        not exist, the user has the options (a) that the layers will
;        be created automatically, or (b) the user can quit the
;        program.
;
;    2)  The hatch scale may be chosen by the user, or if required, it
;        is set automatically for any size of drawing.
;
;    3)  The hatch pattern may be pre-selected, or selected from a list
;        by the user, or a default (ANSI31) may be used.
;
;    4)  Object snap (OSNAP) is set automatically to INTersection
;        for the duration of the program, because most points on a
;        hatch boundary occur at intersections.  If a curved boundary
;        is required to be hatched, then points along the boundary can
;        be selected provided that the aperture does not include an
;        intersection.
;
;    5)  OSNAP is reset to its original setting at the end of the program.
;
;    6)  After execution, the original current layer is restored as the
;        current layer.
;
;    7)  Any entities (objects) which are created for construction purposes
;        are deleted automatically at the end of execution. In this way
;        there is no accumulation of unwanted entities even if HATCH2 is run
;        many times.
;
;*********************************************************************
```

After this lengthy set of comments, the program gets under way with some initialization and layer setting:

```
(defun C:HATCH2 (/ osnp ocmd ans ans2 curlayer porig pdiff)
;                          Initialization
        (setq ocmd (getvar "CMDECHO"))
        (setvar "CMDECHO" 0)
        (setq osnp (getvar "OSMODE"))
        (setvar "OSMODE" 32)
        (setq curlayer (getvar "CLAYER"))
;
;    Check with user to see if the layers 'hatch' and 'noplot' exist.
;          If not, then offer the option to create or quit.
;
     (initget "Y N")
     (setq ans (getkword "\nDo the layers noplot & hatch exist?
             enter Y or N: ")) (terpri)
    (if  (= ans "N")
      (progn              ;progn for no layers exist
          (initget  "Y Q")
          (setq ans2 (getkword "\nCreate 'noplot' & 'hatch' or quit?
             enter Y (Create layers) or Q (Quit):  ")) (terpri)
          (if (= ans2 "Y")
               (command "LAYER" "N" "hatch,noplot"
                               "C" "6" "hatch" "C" "2" "noplot" "")
               (quit)
          )         ;   end of if "create layers or quit"
      )             ;   end of progn 'no layers'
   )                 ;   end of if no layers exist
```

The program now continues by collecting the polyline points for the hatch boundary:

```
;                     Start collecting data and set layer to 'noplot'
;
     (command "LAYER" "T" "noplot" "S" "noplot" "")
     (setq pdiff 1)
     (setq porig (getpoint "\n origin"))
     (setq ptlst nil)
;
;                              Make a list of points for the 'pline'
;
  (while
     (> pdiff 0)
     (setq p (getpoint "\n next point"))
     (setq ptlst (append ptlst (list p)))
     (setq pdiff (distance p porig))
  )                            ;end of 'while' loop
;                                 Draw a pline as a boundary for hatching
;
     (command "PLINE" porig "W" 0 0 (foreach p ptlst
                              (command p)))
```

The next step is to set the hatch scaling and select the hatch pattern. The program then does the hatching and ends by resetting the system variables.

```
   ;                          Start of layer setting and hatching.
   ;                          Hatch scale is optionally entered or
   ;                          automatically calculated to limits/20
   ;                          before the hatch command is executed.
   ; ******************************************************
      (initget "Y N")
      (setq ans (getkword "\n\n Do you want automatic scaling?
              enter Y or N:  ")) (terpri)
      (if  (= ans "N")
             (progn              ;progn for no auto scaling
                 (setq skale (getreal "\nEnter the hatch scale ")) (terpri)
             )                  ;end of progn 'no auto-scaling'
             (progn              ;progn for else - auto-scaling
                 (setq distskale (distance (getvar "LIMMIN")
                                                          (getvar
"LIMMAX")))
                 (setq skale (/ distskale 20))
             )                   ;end of progn for auto-scaling
       )                          ;end of if for scaling
   ; ******************************************************
      (initget "Y S")
      (setq ans (getkword "\n Have you pre-selected a hatch  pattern?
                enter Y to continue or S to select from a list: ")) (terpri)
      (if  (= ans "S")
             (progn              ;progn for selecting the hatch pattern
                 (setq m:hp nil)
                 (command "HATCH" "?" "")
                 (setq m:hp (getstring
                                       "\nEnter hatch pattern name <ANSI31>:
"))
                 (terpri)
                 (if (= m:hp "")
                       (setq m:hp "ANSI31") ;hatch pattern default
                 )              ;end of if m:hp is not a string
             )                   ;end of progn for auto-hatch pattern
       )                          ;end of if for hatch pattern selection
   ; ******************************************************
     (command "LAYER" "T" "hatch" "S" "hatch" "")
     (command "HATCH" m:hp skale "" "L" "")
   ;                                Erase pline and reset layers
     (command "LAYER" "S" "noplot" "F" "hatch" "")
     (command "ERASE" "L" "")
     (command "LAYER" "T" "hatch" "S" curlayer "" "REDRAW")
     (setvar "OSMODE" osnp)
     (setvar "CMDECHO" ocmd)
     (princ)
   ) ;  end of the program
```

Note that you may need to adjust the number of spaces in some of the string prompts if you use this program directly, because there is a chance that the maximum string length may be exceeded due to the indentation used when the string prompts are split for ease of reading. Figures 8-1 through 8-7 show a typical execution of this hatch routine.

AUTOLISP DEVELOPMENT MENUS

In lesson one of this series, I showed you some menu commands that would simplify the creation and testing of your AutoLISP programs. Here we will add to the menus so that you can create & edit programs, test them, print the current program, list your LISP directory to both the screen and to the printer, change the name of the current 'test.lsp' program for permanent storage, and finally load any AutoLISP program from your LISP directory.

I put these commands into an auxiliary menu file called AUX1.MNU, which I store with my other menus in a MNUS directory under the executable AutoCAD directory, which I call ACAD11 at the time of writing. (This will change to ACAD12 by the time this lesson appears in print!)

I am usually reluctant to make major changes to the standard ACAD.MNU file, so after copying that file to ACD11.MNU, I simply added the single menu macro command as follows:

[AUX1.MNU]^C^Cmenu /acad11/mnus/aux1;

This line appears in the POP menu Files which contains the Save, End and Quit choices in Release 11. Then, in my AUX1.MNU file I added a POP2 Menu as follows:

```
.
.
***POP2
[Set-ups]
[Initialize Layers]^C^C^P(progn  (setvar  "cmdecho" 0)+
(command "linetype" "s" "hidden" "s" "phantom" "s"+
"center" "s" "bylayer" "") (command "layer" "n"+
"1,DIM,CENTER,HIDDEN,NOPLOT,TEXT,TBLOCK,HATCH"  "")(princ));+
(command "layer" "c" "1" "1,TEXT,TBLOCK" "c" "3" "DIM"+
"c" "4" "CENTER" "c" "6" "HATCH,HIDDEN"+
"c" "2" "NOPLOT" "")(princ);(command "layer" "l" "hidden" "hidden"+
"l" "center" "center" "s" "1" "")(setvar "cmdecho" 1)(princ);^P
[Lisp program development]^C^C^C$p2=p22 $p2=*
[ADS and C development]^C^C^C$p2=p24 $p2=*
[Set decimal places]^C^C^C$p2=p23 $p2=*

**p22
[Set-ups]
[Edit-LSP]^C^C^Cshell ed C:/acad11/lisp/test.lsp;graphscr
[Test-LSP]^C^C^C(load "/acad11/lisp/test");(test);
[Print test.lsp]^C^C^Cscript /acad11/scripts/cpy2prn
[List lisp directory to screen]^C^Cscript /acad11/scripts/lst2scr
[List lisp directory to printer]^C^C^Cscript /acad11/scripts/lstlsp
[Change .lsp filename]^C^C^P(load "/acad11/lisp/savtest");(savtest);^P
[Load a lisp program]^C^C^P(defun lode ()+
(setq fnam (getstring "Filename: "))+
(load (setq fullname (strcat "/acad11/lisp/" fnam ".lsp"))))+
(lode);^P
[~--]
[previous]^c^c$p2= $p2=*
...... etc.
```

The POP2 page **p22 contains the commands, scripts, and LISP routines that make my life easier:

[Edit-LSP]^C^C^Cshell ed C:/acad11/lisp/test.lsp;graphscr

[Edit-LSP] is a simple macro to shell out and edit the current 'test.lsp' file, which I keep along with my other AutoLISP programs in the lisp subdirectory. You can substitute your favorite editor in place of my 'ed'.

[Test-LSP]^C^C^C(load "/acad11/lisp/test");(test);

After editing, this macro tests the current program by loading and executing 'test.lsp'.

[Print test.lsp]^C^C^Cscript /acad11/scripts/cpy2prn

To get a hard copy of the program during debugging, this macro uses a script file to print the 'test.lsp' file. The script file **CPY2PRN.SCR** is simply:

```
shell
copy \acad11\lisp\test.lsp prn
graphscr
```

`[List lisp directory to screen]^C^Cscript /acad11/scripts/lst2scr`

Here, the script file LST2SCR.SCR lists the LISP directory to the AutoCAD text screen:

```
   shell
   dir \acad11\lisp\*.lsp | sort | more

[List lisp directory to printer]^C^C^Cscript /acad11/scripts/lstlsp
```

LSTLSP.SCR lists my LISP directory to the printer as follows:

```
   shell
   dir \acad11\lisp\*.lsp > prn

[Change .lsp filename]^C^C^P(load "/acad11/lisp/savtest");(savtest);^P
```

I save my working LISP program files by executing this SAVTEST.LSP program:

```
   ;;; SAVTEST.LSP   A program to save "test.lsp" under a new name
   ;;;    Program by Tony Hotchkiss, adapted from the Autodesk FCOPY.LSP
   (defun savtest (/ oldname old newname new)
     (setq oldname nil newname nil)
     (setq oldname (getstring "\nEnter the old .lsp file name <Test>: "))
            (if (= oldname "") (setq oldname "test"))
     (setq oldname (strcat "/acad11/lisp/" oldname ".lsp"))
     (setq newname (getstring "\nEnter the new file name (1-8 chars): "))
     (setq newname (strcat "/acad11/lisp/" newname ".lsp"))
     (cond ((null (setq old (open oldname "r"))) ;test open oldname "read"
         (princ (strcat "I can't open " oldname " for reading"));result if null
          )
          (if old    ; test old exists
           (progn
                  (setq new (open newname "w"))
                  (while (setq lin (read-line old))
                      (write-line lin new)
                  );end of read-write while file lines exist
                  (close old)
                  (close new)
           ); progn
        ); if old exists
     ); conditional function
    (princ)
   ); defun
```

This program is total overkill for the job, but it illustrates some interesting loops, branches and tests. The job could be done alternatively by shelling out of AutoCAD and entering the DOS copy command to change the filename, but you would do more typing.

```
   [Load a lisp program]^C^C^P(defun lode ()+
   (setq fnam (getstring "Filename: "))+
   (load (setq fullname (strcat "/acad11/lisp/" fnam ".lsp"))))+
   (lode);^P
```

Finally, here is a program that is written into the menu macro to load any lisp program from the LISP directory. Note the use of the ^P to toggle menu echoing on and off. It actually affects messages and prompts, and in this case, ^P prevents the entire defined function 'lode' from being written at the command prompt.

Well, I hope this has been as much fun for you as it has been for me. All that remains is to give you the last homework assignment of this introductory course, and to wish you good programming.

Home work assignment.

1. Write a program to draw graphs from y-ordinate data supplied by an external file, similar to the GRAPH2.LSP program of this lesson. Start your graphs at a user supplied origin (you will then need to add the y-value of the start point to each of the y-values in the external file before defining the points pp). Give your graphs axes and titles with numbers on the axes as shown in figure 8-8. For axis numbering and length, you will need to find the maximum value of the y-values, and this can be done with the following code:

```
           (setq index 0 ymax 0.0)
```

```
(repeat numpoints
        (if (< ymax (cadr (nth index ptlst)))
                (setq ymax (cadr (nth index ptlst)))
        )
        (setq index (1+ index))
)
```

Your program should be able to handle any positive numbers in the y-direction, plotted against equal increments in the x-direction. The number of points along the x-axis will be the same as the number of values in the external file, which will typically be in the range 5 to 25.