

Introduction to AutoLISP Programming Correspondence Course

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

Study Guide, part 1 of 8

Lesson One - An introduction to AutoLISP

Solutions to the homework questions:

1. Question: 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?

Answer: In the text of lesson one, the following four defined function examples were given (we number them here 1.1 to 1.4 for clarity):

```
(defun test1 ()                                1.1
(command "line" "6,3" "8,5" "10,3" "c")
)
```

.....The AutoLISP subr to define functions is (defun...), and in this case the name of the defined function is test1.....

```
(defun C:TEST1 ()                              1.2
(command "line" "6,3" "8,5" "10,3" "c")
)
```

```
(defun test2 (/ pt1 pt2 pt3 pt4).....)        1.3
```

```
(defun test3 (a b / pt1 pt2 pt3 pt4).....)    1.4
```

Parentheses must be in pairs of open and closed parentheses. Notice that after the first occurrence of a defined function example (numbered 1.1 in the above), it is stated that

The AutoLISP subr to define functions is (defun

This gives the clue that the function requires the final closing parenthesis and that it may be on the same line. In fact, as we discover in lesson two, the entire program is controlled by parentheses rather than by lines. Placing the closing parenthesis on a separate line, under the corresponding open parenthesis, is a matter of programming style. The program would still run even if it was written on a single line.

2. Question: Are the arguments a and b in the last defined function example local or global variables?

Answer: All of the defun examples have nested pairs of parentheses, where the inside pair controls the program variable types. The last example is contained in the following text:

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 that 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
defined as arguments of the (defun) function should appear 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 (progl) before it can be executed, and instead of executing with

Introduction to AutoLISP Programming

(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.

It states here that to make the variables local, they should be preceded by the slash mark and a space (/). However, the variables that appear before the slash are arguments that only exist for that particular defined function. This means that they are always local, even though their values are assigned by the calling statement from another function, for example (test3 2.2 3.0) as shown. Note that the line of dots preceding the last closed parenthesis in the defun statement implies that there is at least one AutoLISP expression to be executed as the defined function. Generally, there will be many other AutoLISP expressions which are executed as though they were a single expression.

3. Question: 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.

Answer: The answer to the first part of this question is to use the 'command' function, as suggested in the second part. The general form of the function to draw circles is:

(command "circle" method parameters)

The circle methods are:

1. Cen, Dia
2. Cen, Rad
3. 2-Point
4. 3-Point
5. TTR

The equivalent command function for each of the above are the same methods and parameters that you would key in to make AutoCAD draw circles. For instance, to draw a circle with the Cen, Dia (center and diameter) method, you first enter 'circle', then key in the center point coordinates, then enter 'd' for diameter, then key in the diameter value. The sequence would resemble:

circle 2.5,3.5 d 1.25

For the purposes of the command function, these entries should all appear as character strings so that AutoCAD accepts the entries as though they were entered from the keyboard. The command function to draw this circle with center at $x=2.5, y=3.5$, and a diameter of 1.25, is:

(command "circle" "2.5,3.5" "d" "1.25")

Your AutoCAD screen will look like the following figure 1.1:

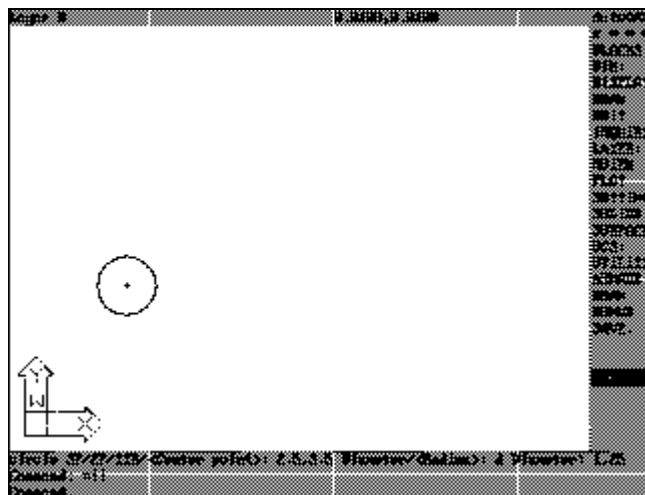


Figure 1.1 AutoCAD screen after entering the circle command function

Figure 1.1 shows the prompt and subcommand sequence for the center point position 2.5,3.5 followed by the choice 'd' for diameter and the final entry 1.25 for the diameter value. Notice that on the next line, after the command prompt, AutoLISP has generated the word 'nil' after executing the command function. This is the value that is 'returned' by the AutoLISP command function, and we will comment on this later in the series.

Introduction to AutoLISP Programming

The parameters to use in the command function depend on the method used to draw the circles, as shown in the following examples:

1. Circle with center at x=2, y=3, Radius=0.75
(command "circle" "2,3" "r" "0.75")
2. Circle with 2-Point definition, diameter end points at x=3, y=4, and x=6, y=4
(command "circle" "2p" "3,4" "6,4")

Note: This is the solution to the homework circle example.

3. Circle through three points on the circumference at x,y coordinates of 4,4 ; 5,4.5 ; 6,4
(command "circle" "3p" "4,4" "5,4.5" "6,4")

To draw arcs, a similar method is used where the available choices for arcs are shown in the following table:

Arc Method	Command function
3-Point	(command "arc" "2,2" "2.5,3" "4,2")
Start, Cen, End	(command "arc" "4,2" "c" "5,1" "6.2")
Start, Cen, Angle	(command "arc" "4,3" "c" "5,1" "a" "60")
Start, Cen, Length	(command "arc" "4,3.5" "c" "5,1" "l" ".75")
Start, End, Angle	(command "arc" "4,2" "e" "5,3" "a" "60")
.....etc.	

When each of these commands is entered in turn, the results will show on your AutoCAD screens as in figure 1.2:

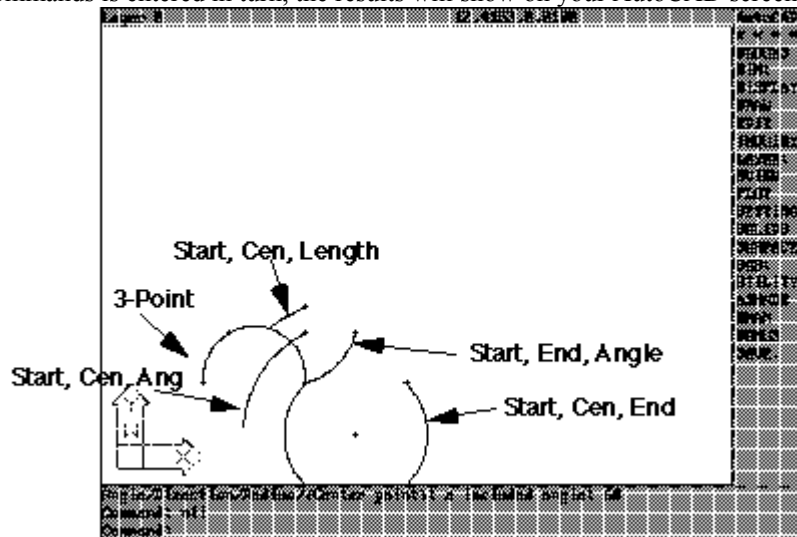


Figure 1.2 Result of entering arc command functions

Note that you should use the arc options "c", "a", "l", etc. only when they are not the default. For example, all 'Start points' are the default option, so no "s" choice is made, and the arc start point coordinates are supplied directly. The same is true for the 'end' designation of the 'Start, Cen, End' method. The only arc method that uses default values for all three items is the '3-Point' option. Note the value of 'nil', shown in figure 1.2, that is returned by the AutoLISP command function as before.

4. Question: 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.

Answer: The name of the .lsp file is your choice, here, I called it 'test4', and the name of the defined function I also named 'test4' (giving the defined function the same name as the AutoLISP program file is a common practice that I use for programs that execute from a single defined function). Following the format of the lesson one text, here is a listing of the AutoLISP file TEST.LSP:

```
(defun test4 ()
  (command "line" "3,4" "6,4" "5,6" "c")
  (command "circle" "3p" "3,4" "6,4" "5,6")
)
```

The solution here is to use the 3-Point method of drawing the circumscribed circle.

Introduction to AutoLISP Programming

To create (edit) and execute this program, the menu macros suggested in the text of lesson one makes things easier and quicker than entering the shell command followed by the DOS command for editing the file.

Note that you must include every double quote and parenthesis exactly in the order required, otherwise AutoLISP will complain.

An alternative form of the program is as follows, in which the circle command immediately follows the close of the line command:

```
(defun test4 ()
  (command "line" "3,4" "6,4" "5,6" "c" "circle" "3p" "3,4" "6,4" "5,6")
)
```

Another form which is easier to read is:

```
(defun test4 ()
  (command "line" "3,4" "6,4" "5,6" "c"
           "circle" "3p" "3,4" "6,4" "5,6")
)
```

Note here that a new-line has been inserted between the line and circle commands, even though the close parenthesis has not been completed. This shows how the program flow is controlled by the parentheses rather than by new-lines. Note also the indentation used in order to align the open and final close parentheses of the defun statement, to make it clear that they are part of the same pair of parentheses.

It is also possible to draw the circle first, followed by the line, as follows:

```
(defun test4 ()
  (command "circle" "3p" "3,4" "6,4" "5,6"
           "line" "3,4" "6,4" "5,6" "c")
)
```

It should be clear by now that there are many ways to achieve the same end when you write LISP programs. The bottom line is that the program should produce the required result. Later in this course you will learn programming styles for legibility and organized development.

So far, the above examples could have been programmed simply by writing the appropriate AutoCAD commands in a menu macro. In the next lesson, you learn about using variables for storing the locations of points, and you will begin to see that LISP is a more powerful tool than using AutoCAD commands in menu macros for customizing.

Reference literature for AutoLISP

My favorite books for AutoLISP programming are:

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

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

2. "AutoLISP in Plain English, 3rd Edition" George O. Head, Ventana Press, P.O. Box 2468, Chapel Hill, NC 27515, 1990 Tel: 919-942-0220 FAX: 1-800-877-7955

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

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

3. "The Complete AutoLISP Desk Reference" Tandy G. Willeby, Ariel Books, Inc. P.O. Box 203550, Austin, TX 78720, 1990 Tel: 512-250-1700

This is a cross-referenced guide to all of the AutoLISP functions and their syntax. It lists the functions by chapter in various categories and also gives all of the functions in alphabetic order in a single chapter. In

Introduction to AutoLISP Programming

addition, there are many tables for ASCII codes, Octal codes, DXF group codes, and system variables. At the time of writing the book does not include release 11 functions.

Other sources for AutoLISP are the magazines CADalyst and CADENCE.

Introduction to AutoLISP Programming Correspondence Course

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

Study Guide, part 2 of 8

Lesson Two - AutoLISP Program Evaluation and Control

Solutions to the homework questions:

1. **Question:** 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 2-HW1. Let the user supply the height H of the cross-section, and the origin point.

Answer: Here is an example program to draw the I beam:

```
;; IBEAM1.LSP A program by Tony Hotchkiss to draw
;; an "I" cross-section beam with height H, flange and
;; web thickness H/8 and width 0.8 x H.
;;
(defun IBEAM1 (h) ; user supplies the height 'h'
  ; user input
  (setq pt1 (getpoint "\nLower left corner point: "))
  (setq width (* h 0.8) ; calculations
    thickness (/ h 8.0)
    inner_height (- h (* 2 thickness))
    inner_width (/ (- width thickness) 2)
  )
  ; calculate new points
  ; counter -clockwise
  (setq pt2 (polar pt1 0.0 width)
    pt3 (polar pt2 (* PI 0.5) thickness)
    pt4 (polar pt3 PI inner_width)
    pt5 (polar pt4 (* PI 0.5) inner_height)
    pt6 (polar pt5 0.0 inner_width)
    pt7 (polar pt6 (* PI 0.5) thickness)
    pt8 (polar pt7 PI width)
    pt9 (polar pt8 (* PI 1.5) thickness)
    pt10 (polar pt9 0.0 inner_width)
    pt11 (polar pt10 (* PI 1.5) inner_height)
    pt12 (polar pt11 PI inner_width)
  )
  ; draw the outline
  (command "line" pt1 pt2 pt3 pt4 pt5 pt6 pt7
    pt8 pt9 pt10 pt11 pt12 "close")
) ; end of the defun
```

This program starts with some comments giving the name of the program and its purpose. The comments have 2 semi-colons to start them off. There is no particular significance to this, it is merely a style that some AutoLISP programmers adopt to make more of a 'banner' heading. If you look at some of the programs that come with your AutoCAD from Autodesk, you will see that they use 3 semi-colons in their headers.

The program creates twelve points for the corners of the I-beam shown in figure 2.1.

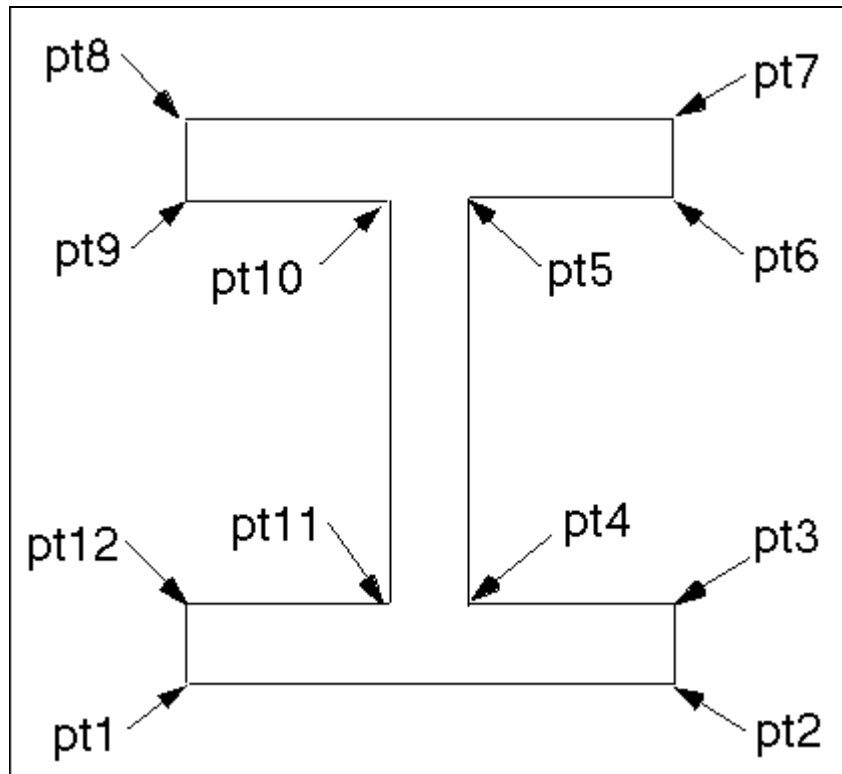


Figure 2.1 Point locations on the I-beam

The defun statement has a single argument, 'h', the height of the I-beam, so to load and use the program the following sequence will be entered at the command prompt (assuming a beam height of 6.5):

```
Command: (load "/acad11/lisp/ibeam1")
IBEAM1
Command: (ibeam1 6.5)
```

After the prompt for the lower left corner point, you can key-in a point or indicate a point with the cursor, and the I-beam will be drawn.

If you want **h** to be an integer, you must allow for this in your program, because **h** is used in the calculation of the **thickness** and the **inner_height** variables as shown in our program. Note that in the calculations we have used $(* h 0.8)$ and $(/ h 8.0)$ in the definitions of **width** and **thickness** respectively. The arithmetic functions return integers only if all of the arguments are integers, so even if **h** is an integer, these calculations will return real values. Subsequent uses of the real (decimal numbers) variables **width** and **thickness** will automatically return real values for **inner_height** and **inner_width**. From this, it is clear that to specify the **thickness** to be $(/ h 8)$ will cause an erroneous value if **h** is an integer.

The distances **inner_height** and **inner_width** are the distances between pt4 & pt5, and pt3 & pt4 respectively, and are used in the polar definitions of points as shown.

The program defines the points pt2 through pt12 with the polar function, then uses the command function to draw the outline of the I-beam.

You may refine the program to make the points pt1 through pt12 local, but only do this when the program is running properly. To do this, the defun statement would look like:

```
(defun IBEAM (h / pt1 pt2 pt3 pt4 pt5 pt6
               pt7 pt8 pt9 pt10 pt11 pt12)
```

You could also include the variables **width**, **thickness**, **inner_height**, and **inner_width** in your list of local variables. Another refinement would be to add a **(redraw)** function after drawing the lines to eliminate the blipmarks.

Introduction to AutoLISP Programming

A variation of your program is to use the points pt2 through pt12 as they are defined within the "line" command as in the IBEAM2.LSP program listing. This method of using the point definition within the **command** function works because the **setq** function returns the value of its last expression. Each point must be assigned with its own **setq** function, rather than the multiple 'variable/expression' form of IBEAM1.LSP.

```
;; IBEAM2.LSP A program by Tony Hotchkiss to draw
;; an "I" cross-section beam with height H, flange and
;; web thickness H/8 and width 0.8 x H.
;;
(defun IBEAM2 (h) ; user supplies the height 'h'
  ; user input
  (setq pt1 (getpoint "\nLower left corner point: "))
  (setq width (* h 0.8) ; calculations
    thickness (/ h 8.0)
    inner_height (- h (* 2 thickness))
    inner_width (/ (- width thickness) 2)
  )
  ; draw the outline
  (command "line" pt1
    (setq pt2 (polar pt1 0.0 width))
    (setq pt3 (polar pt2 (* PI 0.5) thickness))
    (setq pt4 (polar pt3 PI inner_width))
    (setq pt5 (polar pt4 (* PI 0.5) inner_height))
    (setq pt6 (polar pt5 0.0 inner_width))
    (setq pt7 (polar pt6 (* PI 0.5) thickness))
    (setq pt8 (polar pt7 PI width))
    (setq pt9 (polar pt8 (* PI 1.5) thickness))
    (setq pt10 (polar pt9 0.0 inner_width))
    (setq pt11 (polar pt10 (* PI 1.5) inner_height))
    (setq pt12 (polar pt11 PI inner_width))
    "close"
  ) ; end of the command
  (redraw)
) ; end of the defun
```

Note that it is not possible to include the first point 'pt1' inside the **command** function because the **get...** functions can not be used in **command** function calls.

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

Answer: The crudest way to add fillets is to define some points on the appropriate lines and then use the fillet command as follows:

Consider the corner at pt3 of figure 2.1. If this were filleted in the usual way with AutoCAD commands, you would be asked to select two objects either by pointing or by entering coordinates from the keyboard. Since you would want your program to fillet automatically with minimum intervention by the user, the point coordinates option is preferred.

You will therefore need to create two extra points for each filleted corner, and in the case of the corner at pt3, we show two suitable points, ptf31 and ptf32, in figure 2.2.

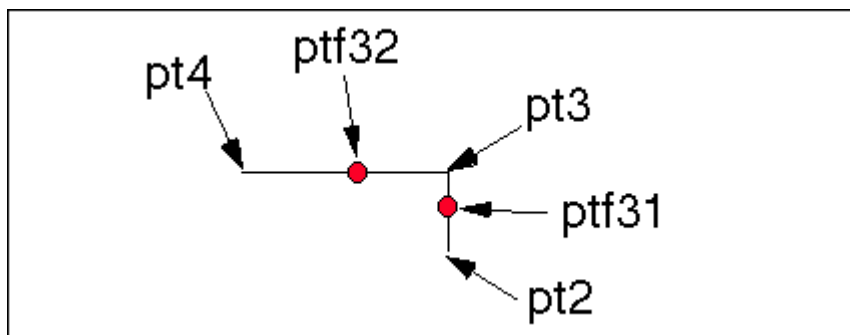


Figure 2.2 Fillet point positions ptf31 and ptf32

Points ptf31 and ptf32 may be defined as follows:

Introduction to AutoLISP Programming

```
(setq ptf31 (polar pt3 (* PI 1.5) (* thickness 0.4))
      ptf32 (polar pt3 PI (* inner_width 0.4))
)
```

We use the multiplier 0.4 here to establish that the point coordinates are nearer to the corner being filleted, although, of course, this would make no difference to the effect of the fillet. Similar points should be defined at the corners pt4, pt5, pt6, pt9, pt10, pt11 and pt12.

To create the fillets, if the fillet radius is 0.75 of the thickness, the following code could be added to your program after drawing the outline of the I-beam:

```
(setq frad (* thickness 0.75))
(command "fillet" "r" frad
        "fillet" pf31 pf32
        "fillet" pf41 pf42
        "fillet" pf51 pf52
        "fillet" pf61 pf62
        "fillet" pf91 pf92
        "fillet" pf101 pf102
        "fillet" pf111 pf112
        "fillet" pf121 pf122
); end of fillets
```

The above requires rather a lot of definition of points and fillets, and a better procedure is to define certain sections of your outline as polylines for filleting, as follows:

```
;; IBEAM3.LSP A program by Tony Hotchkiss to draw
;; and fillet an "I" cross-section beam with height H, flange and
;; web thickness H/8 and width 0.8 x H.
;;
(defun IBEAM3 (h) ; user supplies the height 'h'
                ; user input
  (setq pt1 (getpoint "\nLower left corner point: "))
  (setq width (* h 0.8) ; calculations
        thickness (/ h 8.0)
        inner_height (- h (* 2 thickness))
        inner_width (/ (- width thickness) 2)
  )
                ; calculate new points
                ; counter -clockwise
  (setq pt2 (polar pt1 0.0 width)
        pt3 (polar pt2 (* PI 0.5) thickness)
        pt4 (polar pt3 PI inner_width)
        pt5 (polar pt4 (* PI 0.5) inner_height)
        pt6 (polar pt5 0.0 inner_width)
        pt7 (polar pt6 (* PI 0.5) thickness)
        pt8 (polar pt7 PI width)
        pt9 (polar pt8 (* PI 1.5) thickness)
        pt10 (polar pt9 0.0 inner_width)
        pt11 (polar pt10 (* PI 1.5) inner_height)
        pt12 (polar pt11 PI inner_width)
  )
                ; draw the outline
  (setq frad (* thickness 0.75))
  (command "fillet" "r" frad)
  (command "pline" pt2 pt3 pt4 pt5 pt6 pt7 "")
  (command "fillet" "polyline" "L")
  (command "pline" pt8 pt9 pt10 pt11 pt12 pt1 "")
  (command "fillet" "polyline" "L")
  (command "line" pt1 pt2 "" "line" pt7 pt8 "")
  (redraw)
); end of the defun
```

Introduction to AutoLISP Programming

Here, two sections of polyline are drawn and filleted. The fillet command references the 'last' object created (the polyline) by using the "L" sub-command. It would be equally possible to select the polyline by referencing one of its points.

3. Question: 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.

Answer: The program IBEAM4.LSP will draw, fillet and dimension an I-beam. The program assumes that any necessary system and dimension variables have already been set before the program executes. Later in this series we will discuss some useful system variables.

```
;; IBEAM4.LSP A program by Tony Hotchkiss to draw, fillet, and
;; dimension an "I" cross-section beam with height H, flange and
;; web thickness H/8 and width 0.8 x H.
;;
(defun IBEAM4 (h) ; user supplies the height 'h'
  ; user input
  (setq pt1 (getpoint "\nLower left corner point: "))
  (setq width (* h 0.8) ; calculations
    thickness (/ h 8.0)
    inner_height (- h (* 2 thickness))
    inner_width (/ (- width thickness) 2)
  )
  ; calculate new points
  ; counter -clockwise
  (setq pt2 (polar pt1 0.0 width)
    pt3 (polar pt2 (* PI 0.5) thickness)
    pt4 (polar pt3 PI inner_width)
    pt5 (polar pt4 (* PI 0.5) inner_height)
    pt6 (polar pt5 0.0 inner_width)
    pt7 (polar pt6 (* PI 0.5) thickness)
    pt8 (polar pt7 PI width)
    pt9 (polar pt8 (* PI 1.5) thickness)
    pt10 (polar pt9 0.0 inner_width)
    pt11 (polar pt10 (* PI 1.5) inner_height)
    pt12 (polar pt11 PI inner_width)
  )
  ; draw the outline
  (setq frad (* thickness 0.75))
  (command "fillet" "r" frad)
  (command "pline" pt2 pt3 pt4 pt5 pt6 pt7 "")
  (command "fillet" "polyline" "L")
  (command "pline" pt8 pt9 pt10 pt11 pt12 pt1 "")
  (command "fillet" "polyline" "L")
  (command "line" pt1 pt2 "" "line" pt7 pt8 "")
  ;***** Dimension the I-beam *****
  (setq dimdist (* thickness 2.0)) ; Distance from the part
  ; to the dimension.
  ; Now define new points.
  (setq pt1hor (polar pt1 (* PI 1.5) dimdist)
    pt2ver (polar pt2 0.0 dimdist)
    pt8ver (polar pt8 PI dimdist)
    pt12rad1 (polar pt12 (* PI 1.75) (* frad 0.414))
    pt12rad2 (polar pt12 (* PI 0.25) dimdist)
  )
  (command "dim1" "hor" pt1 pt2 pt1hor ""
    "dim1" "ver" pt2 pt7 pt2ver ""
    "dim1" "ver" pt9 pt8 pt8ver ""
    "dim1" "rad" pt12rad1 "<> (TYP, 8 PLCS)" pt12rad2
  )
  ; end of dimensions
  (redraw)
```

Introduction to AutoLISP Programming

) ; end of the defun

The program requires the single argument for the height of the I-beam. Figure 2.3 shows what your screen will look like after executing IBEAM4.LSP with a beam height of 5.5.

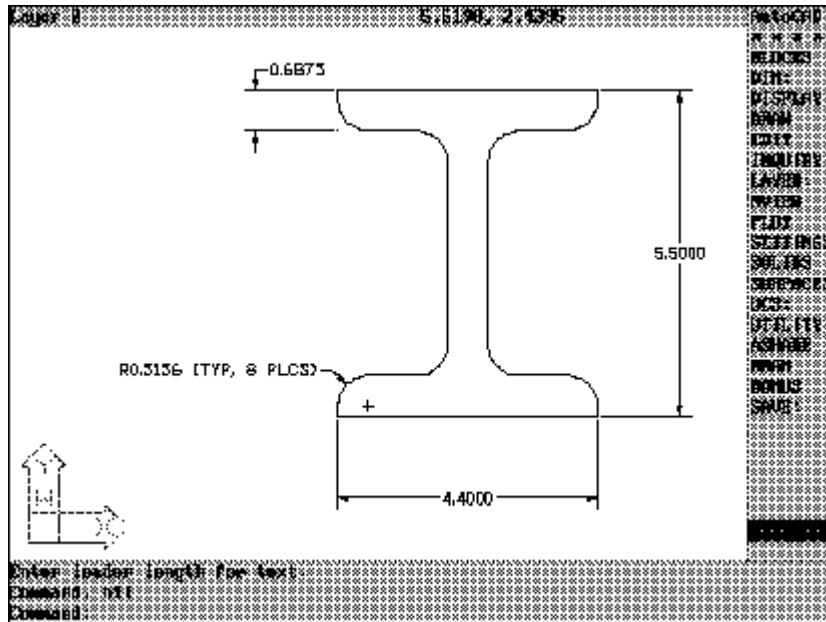


Figure 2.3 Execution of the IBEAM4.LSP program with height 5.5

For dimensioning, I have set a distance 'dimdist' to be twice the thickness of the web of the beam section. This distance is arbitrary, and you may choose to use a constant number to conform to your office standards. The new points which are defined for placing the dimensions are pt1hor, pt2ver, pt8ver, pt12rad1 and pt12rad2. These points are shown in figure 2.4. Note that for the horizontal and vertical dimensions, no attempt has been made to position the point at the actual dimension text position, because AutoCAD will automatically place the dimensions centrally if there is room available.

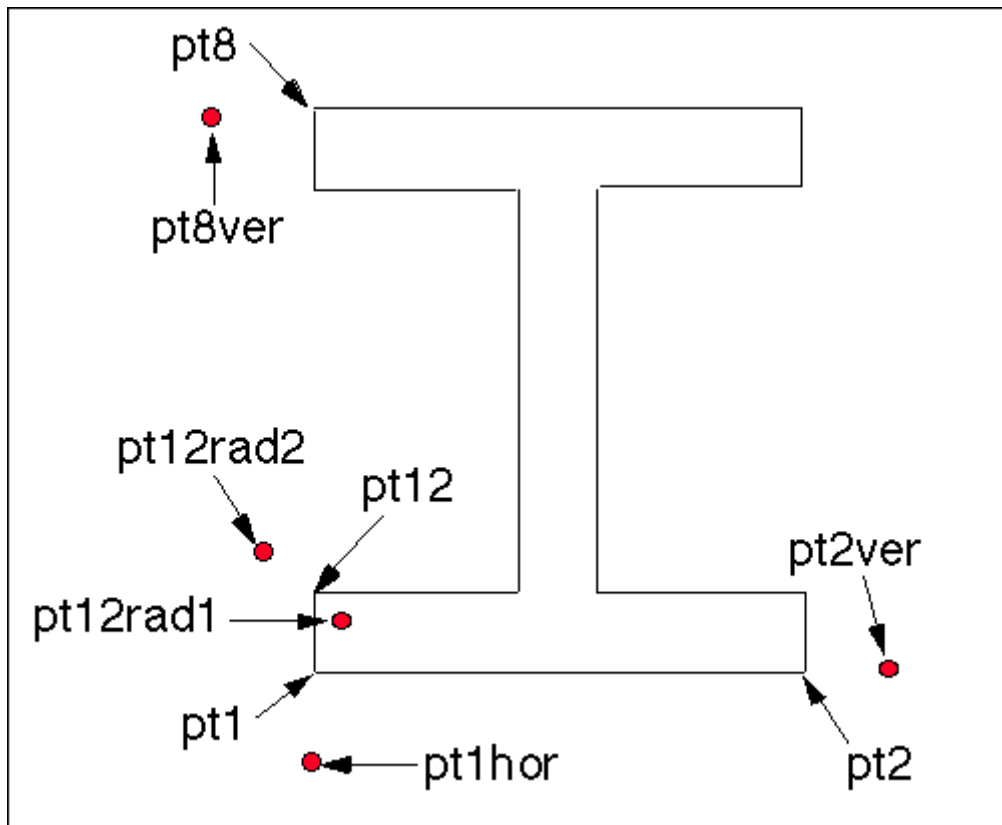


Figure 2.4 Point definitions for dimensioning

Introduction to AutoLISP Programming

The radial dimension needs a point to select a position on the radius to be dimensioned. We have selected the corner at pt12 to be the typical radius dimension, and the selected point is therefore at a distance of $0.414 \times$ the fillet radius, frad , in a direction of $+315^\circ$ or -45° from the horizontal, as shown in figure 2.5

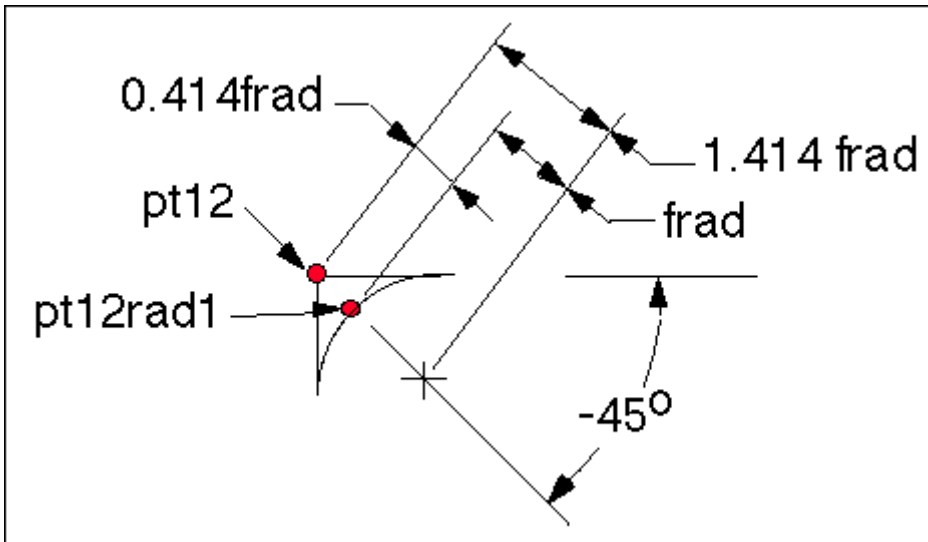


Figure 2.5 Definition of point pt12rad1

The single form of the dimension command, "**dim1**" is used in the program to avoid the necessity to issue a Ctrl-C to end the dimensioning session. The DIM1 form of the command always returns to the command prompt after each dimension.

Introduction to AutoLISP Programming Correspondence Course

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

Study Guide, part 3 of 8

Lesson Four - AutoLISP Functions for Math and Geometry

Solutions to the homework questions:

1. **Question:** 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.

Introduction to AutoLISP Programming Correspondence Course

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

Study Guide, part 4 of 8

Lesson Four - AutoLISP Functions for Math and Geometry

Solutions to the homework questions:

1. **Question:** 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.

Answer: How good is your coordinate geometry? Figure SG4.1 shows the geometry of the CUSP (C), the tool radius (R), and the step-over distance STEP (S).

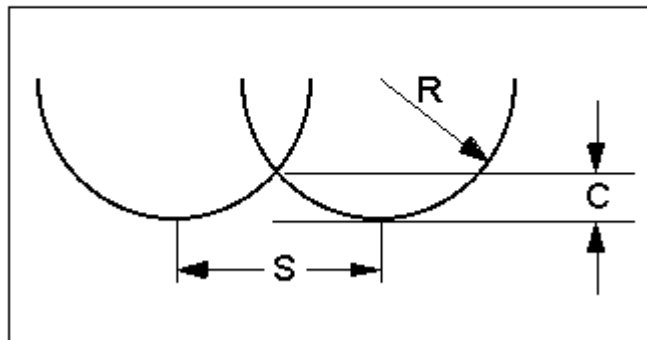


Figure SG4-1 The ball-nose cutter problem

Figure SG4-2 shows the relationship in triangular form between the variables.

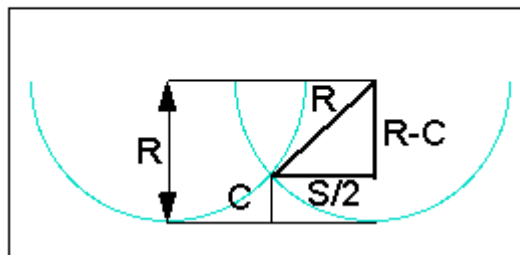


Figure SG4-2 The triangle relationships between R, S, and C

From figure SG4-2, using the Pythagoras' theorem, the square on the hypotenuse is equal to the sum of the squares of the other two sides, thus:

$$\left(\frac{S}{2}\right)^2 + (R - C)^2 = R^2$$

Equation 1

Multiplying out the squared items gives:

Introduction to AutoLISP Programming

$$\frac{S^2}{4} + (R^2 + C^2 - 2RC) = R^2$$

Equation 2

Removing the brackets and collecting terms, we have equation 3:

$$\frac{S^2}{4} + C^2 = 2RC$$

Equation 3

Rearranging in the standard form of the quadratic equation:

$$C^2 - 2RC + \frac{S^2}{4} = 0$$

Equation 4

The classical solution to the quadratic equation is:

$$C = \frac{2R \pm \sqrt{(2R)^2 - S^2}}{2}$$

Equation 5

We will divide the right-hand side by its denominator as in equation 6, which is more suitable for AutoLISP programming. Note that taking the minus sign is the relevant solution here:

$$C = R - \sqrt{R^2 - \frac{S^2}{4}}$$

Equation 6

The program CUSP1.LSP calculates and returns the required cusp (scallop) height:

```
;CUSP1.LSP A program to calculate the cusp height given a
; step-over distance and diameter of a ball-nose cutter.
(defun CUSP1 ()
  (setq dia (getreal "\nEnter tool diameter: "))
  (setq S (getreal "\nEnter the step-over distance: "))
  (setq R (/ dia 2))
  (setq cusp (- R (sqrt (- (* R R) (/ (* S S) 4)))))
)
```

Here, we ask the user to supply the tool diameter, which is used to define the radius in order to conform to the formula of equation 6. We solve the equation with the single **(setq)** statement as shown, but you may also break this down into more steps with simpler expressions, for instance:

Introduction to AutoLISP Programming

```
(defun CUSP1 ()
  (setq dia (getreal "\nEnter tool diameter: "))
  (setq S (getreal "\nEnter the step-over distance: "))
  (setq R (/ dia 2)
        R2 (* R R)
        S2 (* S S)
        S24 (/ S2 4)
        R2MINUSS24 (- R2 S24)
  ) ; end of preliminary calculations
  (setq cusp (- R (sqrt R2MINUSS24)))
) ; end of the defun
```

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

Answer: In this case, you can rearrange the equation (3) from the above to give:

$$S^2 = 4(2RC - C^2)$$

Equation 7

which leads to the formula for S:

$$S = 2\sqrt{C(2R - C)}$$

Equation 8

CUSP2.LSP is a program to solve for the step over distance S, given the cusp (scallop) height and the tool diameter:

```
;CUSP2.LSP A program to calculate the step-over distance
; given a cusp height and diameter of a ball-nose cutter.
(defun CUSP2 ()
  (setq dia (getreal "\nEnter tool diameter: "))
  (setq cusp (getreal "\nEnter the cusp height: "))
  (setq R (/ dia 2)
        C cusp
  )
  (setq step (* 2 (sqrt (* C (- (* 2 R) C)))))
)
```

You can test your answers with the following table relating the tool diameter, cusp height and step-over distance:

DIAMETER	STEP-OVER	CUSP-HEIGHT
0.5000	0.4000	0.1000
0.5000	0.3000	0.0500
0.5000	0.2000	0.02087
0.5000	0.1000	0.00505
0.3750	0.2000	0.0288928
0.3750	0.1000	0.0067896
0.3750	0.0500	0.00167414
0.3750	0.0200	0.0002668
0.2500	0.2000	0.0500
0.2500	0.1000	0.0104356
0.2500	0.0500	0.0025255
0.2500	0.0400	0.00161037
0.2500	0.0100	0.000100

Introduction to AutoLISP Programming

0.2500	0.0050	2.500E-05
0.2500	0.0020	4.000E-06

3. **Question:** 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.

Answer: Here is a program CUSP3.LSP, that will draw and dimension the tool set-up.

```
;CUSP3.LSP A program to draw and dimension the ball-nosed
; cutting tool given the tool diameter and the cusp
; height as user input.
;
(defun CUSP3 ()
    ; Get the user input
    (setq dia (getreal "\nEnter tool diameter: "))
    (setq cusp (getreal "\nEnter the cusp height: "))
    (setq pt1 (getpoint "\nTool-end position: "))
    (setq R (/ dia 2) ; define R and C
          C cusp
    )
    ; next, calculate the 'step'
    (setq step (* 2 (sqrt (* C (- (* 2 R) C)))))
; Now define the points to draw the outline of a tool:
    (setq pt1x (car pt1)
          pt1y (cadr pt1)
          pt2 (list (+ pt1x R) (+ pt1y R)) ; arc-end
          pt3 (polar pt2 (/ PI 2.0) dia);polar distance = 'dia'
          pt4 (polar pt3 PI dia)
          pt5 (polar pt4 (* PI 1.5) dia)
    ) ; end of definition points
    (command "pline" pt3 pt4 pt5 "A" pt1 pt2 "L" "C")
    (command "copy" "L" "" pt1
             (setq pt6 (polar pt1 0.0 step))
    ) ; end of the copy command
;
; Start dimensioning:
; Store the old value of the LUPREC system variable:
;
    (setq oldluprec (getvar "LUPREC"))
;
; The 'step-over' dimension:
; Set the number of decimal places to 3 for the 'step' dimension.
;
    (setvar "LUPREC" 3)
;
; First define a dimension point, then use it for dimensioning:
    (setq pt7 (polar pt6 (* PI 1.5) R))
    (command "dim1" "hor" pt1 pt6 pt7 "")
;
; The 'cusp' dimension:
;
    (setvar "LUPREC" 2)
    (setq pt8 (polar pt6 0.0 dia))
    (setq pt9 (list (+ (car pt1) (/ step 2))
                   (+ (cadr pt1) C)))
    (command "dim1" "ver" pt6 pt9 pt8 "")
;
; The radius dimension:
;
    (setq pt10 (polar pt1 (/ PI 2.0) R)
          pt11 (polar pt10 (* PI 1.25) R)
```

Introduction to AutoLISP Programming

```
) ; definition of pt11 on the tool radius at 225 degrees
(setq pt12 (polar pt11 (* PI 1.25) R))
(command "dim1" "radius" pt11 "" pt12)
)
```

The approach taken here is that one complete tool outline is created as a polyline, which is copied at a distance of 'step' in the x-direction. The program begins by getting user input for the tool diameter 'dia', the cusp height 'cusp', and the 'tool-end' position 'pt1' (shown in figure SG4-3) for drawing the first tool outline. The point pt1 is an arbitrary position which establishes where the drawing will appear on your screen and also provides a reference point for the definition of the tool corner points pt2 through pt5 as shown in figure SG4-3. The functions (getreal) and (getpoint) are used for the initial user input.

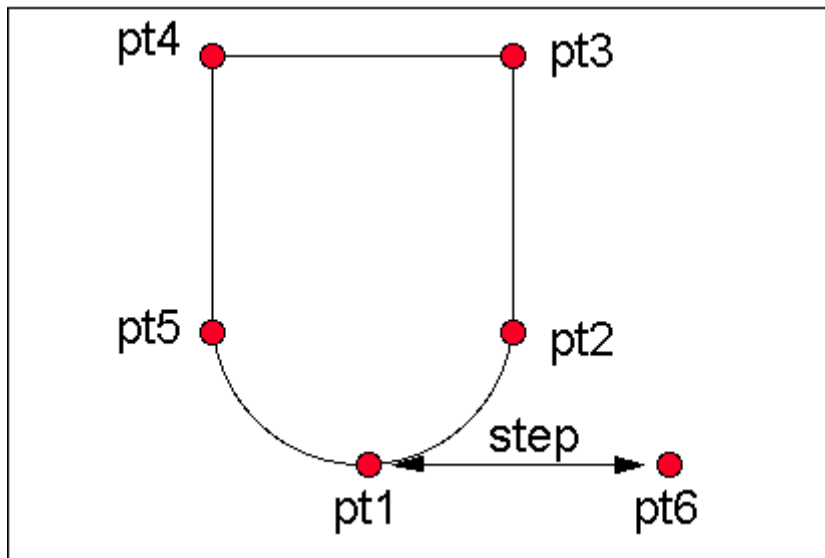


Figure SG4-3 Point definitions

The program then defines the symbols R and C to be the tool radius and the cusp height in a form more suited to the formula which defines the step-over distance 'step', which is given in equation 8 of the solution to question 2 above. The points pt2 through pt5 are then defined using the functions (car), (cadr), and (polar).

The (command) function is then used to create the 'pline'. Note how I used the "A" and the "L" options to set the polyarc mode and to return to the polyline mode:

```
(command "pline" pt3 pt4 pt5 "A" pt1 pt2 "L" "C")
```

I started the pline at point pt3, in order to start with a straight line segment, because arc segments are automatically set to match the tangent direction of preceding line segments.

The copy command, used to create the second tool outline uses a (setq) function to define point pt6, shown in figure SG4-3:

```
(command "copy" "L" "" pt1
  (setq pt6 (polar pt1 0.0 step))
) ; end of the copy command
```

The copy command prompts for an entity selection, and the "L" signifies the 'last object created' for the selected entities. In this case, the last object to be created is the polyline tool outline. The two double quotes following the "L" are used to indicate a null response ('enter') to show that all object selection is completed. The copy command then needs a base point and a 'second point' for positioning the copied object. The 'second point' is defined with the (setq) function, which returns the value of its last expression - in this case the polar point definition. The copy command would have worked equally well without the (setq) function, thus:

```
(command "copy" "L" "" pt1
  (polar pt1 0.0 step)
) ; end of the copy command
```

Introduction to AutoLISP Programming

However, it is convenient to define the point pt6, as it is used later for dimensioning.

The rest of the program dimensions the part, using a technique of defining extra points for dimension positions, followed by the use of the 'dim1' command to create each dimension. In the case of the radius dimension, the position selected on the arc (pt11) is defined by first defining the arc center position, pt10:

```
(setq pt10 (polar pt1 (/ PI 2.0) R)
      pt11 (polar pt10 (* PI 1.25) R)
) ; definition of pt11 on the tool radius at 225 degrees
(setq pt12 (polar pt11 (* PI 1.25) R))
(command "dim1" "radius" pt11 "" pt12)
)
```

Introduction to AutoLISP Programming Correspondence Course

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

Study Guide, part 5 of 8

Lesson Five - User Interaction in AutoLISP

Solutions to the homework questions:

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

Answer: This is simply a question of seeing clearly what the respective print functions produce. The line spacing can be different if you use the command/prompt area of your screens with graphic screen mode instead of using the text screen. Also, you can see all of the suggested printing options together only on the text screen.

2. **Question:** 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 **(getkeyword)** function.

Answer: The program LORP.LSP shows the use of a variable name which is substituted for the command "Line" or "Pline", depending on the choice made by the user of the program.

```
;;; LORP.LSP  Draws a square box with options to use
;;;          regular lines or polylines.
(defun lorp ()
  (setq p0 (getpoint "\nLower left point: "))
  (initget 7)
  (setq len (getreal "\nLength of side: "))
  (initget 1 "Line Pline")
  (setq lpname (getkeyword "\nLine/Pline: "))
  (command lpname p0
    (setq p (polar p0 0.0 len))
    (setq p (polar p (/ pi 2.0) len))
    (setq p (polar p pi len)) "close")
  ) ; end of defun
```

The program starts by asking for a lower-left point of the square box. Next, we want the input of the length of side of the square to be constrained so the null, negative or zero responses are not allowed. This feature is provided by the **(initget 7)** function. The next line, **(initget 1 "Line Pline")** supplies the allowed keywords that can be accepted by the **(getkeyword)** function. The '1' specifies that a null input will not be allowed, otherwise the variable *lpname* will not be assigned an appropriate string for the **(command lpname....)** function.

Figures SG5-1 and SG5-2 show the command/prompt sequence when the program is loaded and executed:

Introduction to AutoLISP Programming

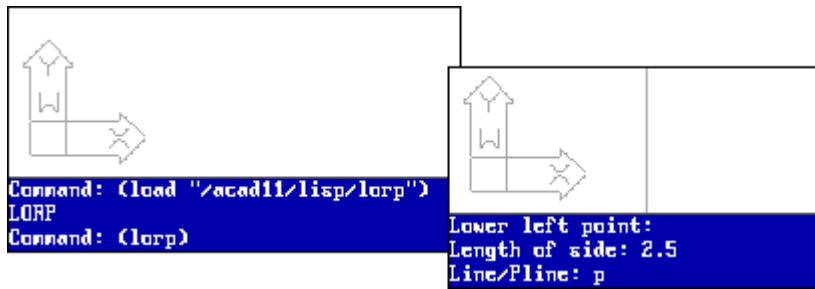


Figure SG5-1 Command prompts for problem 2, drawing with either Line or Pline

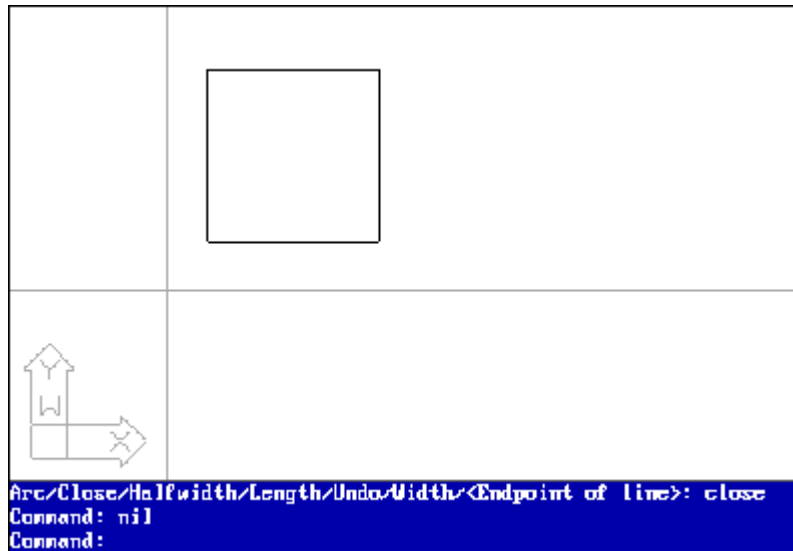


Figure SG5-2 Execution of problem 2, drawing with a choice of Line or Pline

The (load) function uses the full path name to load the program file from the \ACAD11\LISP directory, where I keep my LISP programs. In AutoCAD, you use forward slashes as directory delimiters, so the statement is actually:

(load "acad11/lisp/lorp")

The program is executed with (**lorp**), after which the prompts are given for the lower left point, the length of side, and the choice of Line/Pline. The only required input at the Line/Pline prompt is the capitalized portion of the text, but any number of the following characters are also acceptable, thus for the Pline choice, you may enter p, pl, pli, plin, or pline, and these may be in either upper or lower case..

Note that (**getkeyword...**) with its (**initget...**) function forces the result to be either "Line" or "Pline", exactly as required for the (**command..**) function. The variable *lpname* is assigned the character string with double quotes so that it is ready to be substituted into the (**command**) function. In the case shown, you can see from figure SG5-1 that 'p' was entered in response to the prompt "Line/Pline". The (**initget**) recognizes this as a match for the Pline key word, because only the letter 'P' was given in capital letters.

The technique for drawing the outline of the box is to re-use the point **p** in successive (**polar..**) functions, as we described in lesson three of this series.

After the program has executed, you can enter **!lpname** at the command prompt to see that its value is either "Line" or "Pline", even though you entered L or P respectively in upper or lower case.

Once your program is running satisfactorily, you might make all of your variables 'local' by changing the (defun ..) statement to:

(defun lorp (/ p0 p lpname len) ...

This example shows that it is possible to make intelligent decisions based on user input simply by using the (**getkeyword**) with a list of options supplied by (**initget**), instead of using a more elaborate set of branching and logical functions, which we will cover in the later lessons of this course.

Introduction to AutoLISP Programming

3. **Question:** 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.

Answer: At first sight, this seems to be a complicated problem, but the LISP program required can be written in very few lines, as is shown by the program **INS1.LSP**. Here we have another use of **(getkeyword)** with multiple choices provided in the keyword list of the preceding **(initget)**.

The program begins by displaying a slide which contains pictures of the blocks to be inserted. It is therefore necessary to create the slide as well as the blocks. Use **MSLIDE** to make the slide, and **WBLOCK** to make external blocks. In this example the slide is called **'XSLIDE'**, and is stored in a directory **\ACAD11\SLDS**. The blocks are named **ONE**, **TWO**, **THREE**, **FOUR**, **FIVE**, and **SIX**, and they are stored in the directory **\ACAD11\SYMBOLS**.

```
;INS1.LSP  A program to insert any of six blocks
;
(defun C:INS1 ()
;
  (command "VSLIDE"  "/acad11/slds/XSLIDE")
  (initget 1 "One Two THree Four FIVE Six")
  (setq b11 (getkeyword "\nBlock name: One/Two/THree/Four/FIVE/Six: "))
  (setq b11 (strcat "/acad11/symbols/" b11))
  (redraw)
  (setq pt1 (getpoint "\nInsertion point: "))
  (command "insert" b11 pt1 "" "" "")
)
```

Note the use of **(strcat)** to convert the block name into the full pathname so that the blocks can be retrieved from a separate 'symbols' directory.

The program uses **(getpoint)** for the block insertion point **pt1**, and the name of the block is stored in the variable **b11**, so the insert command continues with no further user intervention.

The use of a slide and a multiple choice set of keywords is a popular alternative to the icon menus in some applications, and for complicated parts, it may execute faster than the icon menus. The approach is certainly relatively easy to implement, with just a few lines of code as shown. Remember to include a **(redraw)** before the insert command to clear the slide from the screen.

The execution of **INS1.LSP** is shown in figure SG5-3.

Introduction to AutoLISP Programming

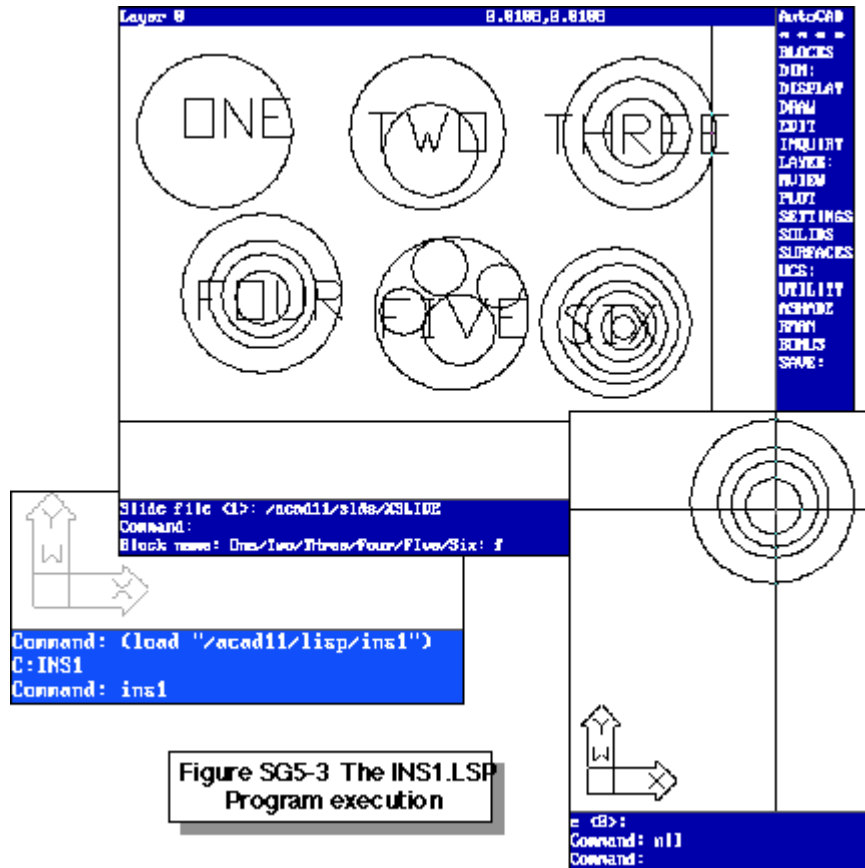


Figure SG5-3 The INS1.LSP Program execution

This program uses the **C:** form of the defined function name so that it can be entered as an AutoCAD command **INS1** instead of including parentheses. The program starts by displaying the slide showing all six blocks with their names (One through Six), together with the prompt to enter the block name, with choices that look like regular AutoCAD choices. It is necessary to distinguish between Two & Three, and Four & Five, so the first two letters have been used for **THree** and **FIve** respectively in the **(initget)** list. Remember that the **(initget)** list is quite independent of the prompt string, so it is your responsibility to offer the appropriate choices with the correct syntax in the **(getkword)** function, so that the choices offered match those given by **(initget)**.

The example of figure SG5-3 shows that the lower case 'f' was entered, and this produces the 'Four' option because that is the only choice with a single letter 'F' option. If the FIVE block is required, you would need to enter a minimum of 'fi' in either upper or lower case. The slide is cleared from the screen by the **REDRAW** command before the block is inserted. Note that the inserted block does not have the name FOUR attached to it, as it appears on the slide. It is quite normal for the blocks to be different from the slide pictures because the slides may need names or titles, and should be simpler images than the actual blocks to be inserted so that the slide can be displayed as quickly as possible.

Introduction to AutoLISP Programming Correspondence Course

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

Study Guide, part 6 of 8

Lesson Six - Input/Output and File Handling with AutoLISP

Solutions to the homework questions:

1. **Question:** 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.

Answer: The program BASEPLAT.LSP contains various modules to draw the baseplate:

First, some introductory comments:

```
;BASEPLAT.LSP A program to draw and dimension
; a base-plate, reading data from an external
; file called BASEPLAT.DAT
; The program calculates the volume of the
; plate, and writes this to an external file
; called BASEPROP.DAT
;
```

I always like to write comments that tell the user what the program is supposed to be doing. In this case, we can alert the user to the fact that an input file BASEPLAT.DAT is to be expected.

The first defined function I called 'inputdat', and this module takes in all of the necessary design data, such as an origin, which is the baseplate center point, and then the parameters are read in from the external data file.

The inputdat function is:

```
(defun inputdat ()
  (setq p0 (getpoint "Plate center point: "))
  (setq datf (open "baseplat.dat" "r"))
  (setq height (atof (read-line datf))
        width (atof (read-line datf))
        th (atof (read-line datf))
        rad (atof (read-line datf))
        adia (atof (read-line datf))
        adim (atof (read-line datf))
        bdim (atof (read-line datf)))
```


Introduction to AutoLISP Programming

```

cdim (atof (read-line datf))
bdia (atof (read-line datf))
cdia (atof (read-line datf))
ddim (atof (read-line datf))
) ; end of reading in data
(close datf)
) ; end of defun inputdat
;

```

The variables defined in the above are shown in figure SG6-1:

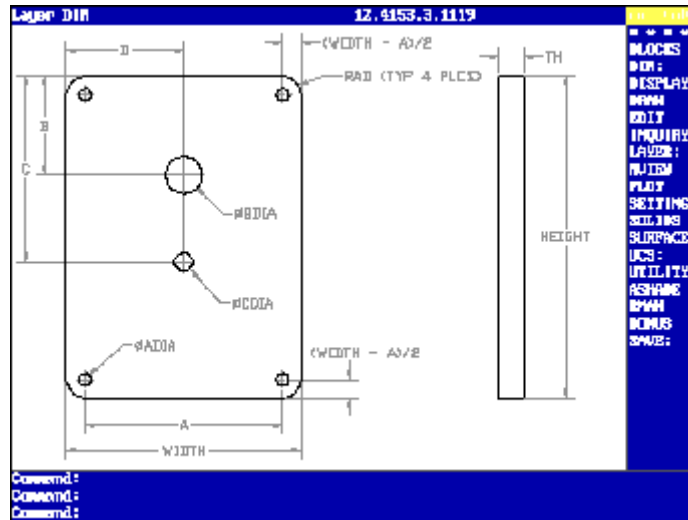


Figure SG6-1 The Baseplate

The next module to draw the top view I have called topview. The module starts by defining points relative to the start point, p0, which is obtained by the previous module 'inputdat':

```

(defun topview ()
  (setq p1 (list (- (car p0) (/ width 2))
                (- (cadr p0) (/ height 2))))
  p2 (polar p1 0.0 width)
  p3 (polar p2 (/ PI 2) height)
  p4 (polar p3 PI width)
) ; end of outline points
(command "color" 2)
(command "pline" p1 p2 p3 p4 "close")
(command "fillet" "r" rad "fillet" "polyline" "L")
(command "area" "e" "L")
(setq csa (getvar "AREA"))

```

The outline is drawn as a polyline and is filleted as a polyline before the area is found, as we hinted at in the assignment instructions.

The topview function continues by defining a point at the center of the lower left hole, using the variable adim which was read in from the external file:

```

;
; define a point for the lower left hole
;
  (setq inset (/ (- width adim) 2.0))
  (setq pta (list (+ (car p1) inset)
                 (+ (cadr p1) inset)))

```

Next, I made an array of four holes, by drawing the first hole with (command "circle"...), then (command "array"...). The points ptb and ptc are then defined for the centers of the 'b' and 'c' holes, which are then drawn:

```

; draw and array four holes, then two holes
;
  (command "circle" pta "d" adia)
  (setq hdim (- height (* inset 2)))
  (command "array" "L" "" "R" 2 2 hdim adim)

```

Introduction to AutoLISP Programming

```
(setq ptb (list (+ (car p4) ddim)
                (- (cadr p4) bdim)))
(setq ptc (list (car ptb)
                (- (cadr p4) cdim)))
(command "circle" ptb "d" bdia
        "circle" ptc "d" cdia)
```

Now that the 'a', 'b', and 'c' holes are drawn, topview proceeds to add the center marks. The 'dimcen' variable is set to cenval, then is adjusted for center marks that extend beyond the individual circles. Note how I defined points that lie on each of the circumferences of the circles:

```
; Draw the circle centers
(setq ped1 (polar pta 0.0 (/ adia 2)))
(setq ped2 (polar ped1 0.0 adim)
      ped3 (polar ped2 (/ PI 2) hdim)
      ped4 (polar ped3 PI adim)
      ped5 (polar ptb 0.0 (/ bdia 2))
      ped6 (polar ptc 0.0 (/ cdia 2))
)
(command "color" 1)
(setq cenval (- (/ rad 10)))
(setvar "dimcen" cenval)
(command "dim1" "center" ped1
        "dim1" "center" ped2
        "dim1" "center" ped3
        "dim1" "center" ped4)
(setq cenval (- (/ bdia 20)))
(setvar "dimcen" cenval)
(command "dim1" "center" ped5)
(setq cenval (- (/ cdia 20)))
(setvar "dimcen" cenval)
(command "dim1" "center" ped6)
); end of defun topview
```

The side view function, sideview, is much simpler than the top view:

```
(defun sideview ()
  (setq ps1 (polar p2 0.0 adim)
        ps2 (polar ps1 0.0 th)
        ps3 (polar ps2 (/ PI 2) height)
        ps4 (polar ps3 PI th)
  ); end of side view points
  (command "color" 2)
  (command "pline" ps1 ps2 ps3 ps4 "close")
) ; end of defun sideview
;
```

The fourth module calculates the net area and the volume of the baseplate, and writes the results to an external file BASEPROP.DAT:

```
(defun results ()
  (setq netarea (- csa
                  (/ (* PI (* cdia cdia)) 4)
                  (/ (* PI (* bdia bdia)) 4)
                  (* PI (* adia adia)))
  )
  (setq vol (* netarea th))
  (setq outf (open "BASEPROP.DAT" "w")); opens the file
  (write-line "BASE PLATE DATA" outf)
  (write-line (strcat "Baseplate area = " (rtos netarea)) outf)
  (write-line (strcat "Baseplate volume = " (rtos vol)) outf)
  (close outf)
) ; end of defun results
;
```

Finally, all of these modules are executed by the last module C:BASEPLAT. This 'executing' module should appear last in your program in order to have its name displayed on the command prompt area when the program is loaded.

Introduction to AutoLISP Programming

```
(defun C:BASEPLAT ()
  (inputdat)
  (topview)
  (sideview)
  (results)
;
  (redraw)
  (princ)
); end of the defun baseplat
```

The file BASEPLAT.DAT that I used in order to test the program has the following format:

```
6.5   Height
4.75  Width
0.5   Thickness
0.375 Corner radius
0.25  A-diameter
4.0   A-dimension
2.0   B-dimension
3.75  C-dimension
0.75  B-diameter
0.375 C-diameter
2.375 D-dimension
```

The order of the data in this file must match the order of reading lines in the 'inputdat' function. At this level of AutoLISP programming there is no other way for you to check that the assigned variables take the correct values. In more advanced LISP programming it is possible to match the variable in some way with the name given in the data file. In such a case, the input data file could have a random order. However, this leads to needlessly complicated programming.

When you are developing a modular program such as this, it is a good idea to decide on the modules, and then write the 'executing' module with semi-colons to 'comment out' the functions, so that you can test the modules one at a time, thus the first module could be written with the C:BASEPLAT function appended to it as in:

```
(defun inputdat ()
|
) ; end of defun inputdat
;
(defun C:BASEPLAT ()
  (inputdat)
;   (topview)
;   (sideview)
;   (results)
;
  (redraw)
  (princ)
); end of the defun baseplat
```

Here, I have replaced the 'inputdat' function statements with the '|' symbol instead of displaying the whole defined function just to illustrate that the C:BASEPLAT defined function can be appended like this. This way, you can always retain the execution name C:BASEPLAT and simply add each module in turn. To execute and debug the first module, the semicolons are placed at the beginning of the lines of the other functions 'topview', 'sideview' and 'results'.

After the first module has successfully executed, you can add the second module. The way to check that the first module works properly is to dump out the values of the variables at the command prompt with the **!variable_name** format, after executing the program, as in:

```
Command: !height
6.5
Command: !width
4.75
Command: !th
0.5
.....etc
```

Introduction to AutoLISP Programming

Usually, in modular programming, some modules will depend on other modules for their variables, so add the modules one at a time. The second module program would look like:

```
(defun inputdat  ()
|
) ; end of defun inputdat
;
(defun topview  ()
|
); end of defun topview
;
(defun C:BASEPLAT  ()
      (inputdat)
      (topview)
;      (sideview)
;      (results)
;
      (redraw)
      (princ)
); end of the defun baseplat
```

Here, both 'inputdat' and 'topview' are executed because the last function, C:BASEPLAT has semicolons only on the 'sideview' and 'results' functions. You can thus add each module in turn and remove the corresponding semicolons from the executing function, so debugging is confined to only one module at a time.

When all of the modules are correctly working, with my input data the results appear in the file BASEPROP.DAT as:

```
BASE PLATE DATA
Baseplate area = 30.0057
Baseplate volume = 15.0029
```

Introduction to AutoLISP Programming Correspondence Course

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

Lesson Seven - Logic and Branching out with AutoLISP

Solutions to the homework questions:

1. **Question:** Write a program for the I-Beam of figure SG7-1. 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 $0.8 \times HT$, and should never be less than $0.5 \times HT$.
- (ii) The flange thickness FT must be in the range $0.05 \times HT$ to $0.15 \times HT$
- (iii) The web thickness WT must be in the range $0.1 \times WI$ to $0.2 \times WI$
- (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 $0.8 \times FT$.
- (v) The beam height takes precedence over all other values, and should be used as it is provided.

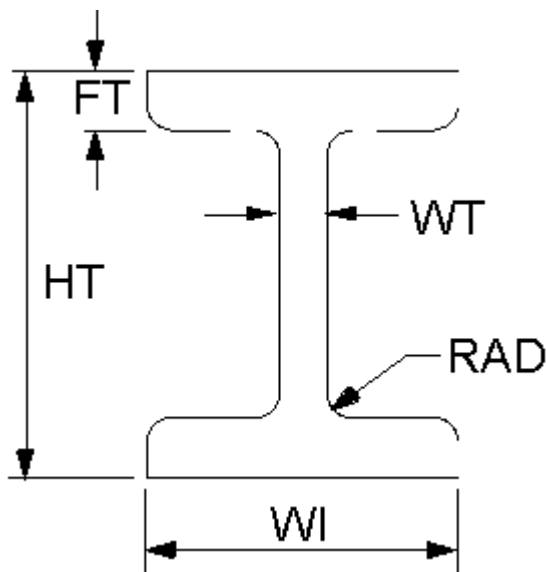


Figure 7-1 The I-BEAM

Answer: The program IBEAM7.LSP is a possible solution. The design allows for any value of the beam height HT , so it can be set with no conditions. The next value to be set is the beam width WI , and we can impose its constraints with a **(cond)** function:

```
; IBEAM7.LSP A program to draw an I-beam
; incorporating design rules
(defun ibeam7 ()
; get user input and impose restrictions (design rules)
  (setq ht (getreal "\nBeam Height: "))
  (setq wi (getreal "\nBeam Width: "))
  (cond
    ((< wi (* 0.5 ht)) (setq wi (* 0.5 ht)))
    (> wi (* 0.8 ht)) (setq wi (* 0.8 ht)))
  )
)
```

The first design rule is "The beam width WI should never exceed $0.8 \times HT$, and should never be less than $0.5 \times HT$." The order of the **(cond)** tests should be the 'most likely' condition first. In this case I assumed that the width is more likely to

Introduction to AutoLISP Programming

be too small than too large, so I have used the 'less than' test first. These assumptions as to what is most likely depend entirely on your particular design philosophy, so as an example it doesn't really matter which order you have the conditional tests.

The next design rule is "*The flange thickness FT must be in the range 0.05xHT to 0.15xHT*", and this can be imposed by a similar use of the **(cond)** function as follows:

```
(setq ft (getreal "\nFlange Thickness: "))
(cond
  ((< ft (* 0.05 ht)) (setq ft (* 0.05 ht)))
  ((> ft (* 0.15 ht)) (setq ft (* 0.15 ht)))
)
```

So far, the design rules have imposed restrictions that depend on the relationship with the design variable and the height of the beam only. The next restriction, on the web thickness, is related to the value of the beam width, so the beam width must have already been established. Note that the design rules will not necessarily be in the order in which the parameter can be programmed, although in this example they are. Next, we have the rule "*The web thickness WT must be in the range 0.1xWI to 0.2xWI*", and that is again handled with the **(cond)** function:

```
(setq wt (getreal "\nWeb Thickness: "))
(cond
  ((< wt (* 0.1 wi)) (setq wt (* 0.1 wi)))
  ((> wt (* 0.2 wi)) (setq wt (* 0.2 wi)))
)
```

The conditions for the radius, RAD, can be imposed by considering the general rule of having two standard values determined by **(cond)**, as follows, but placing the over-riding consideration that "*if FT is less than 0.15, then RAD is set to 0.8 x FT*" following the **(cond)** conditions. Note that you do not have to test explicitly for HT to be greater than 4.0, because this can be made to be the 'default' condition by using the **T** atom as shown.

```
(cond
  ((<= ht 4.0) (setq rad 0.125))
  (T (setq rad 0.5))
)
(if (< ft 0.15) (setq rad (* 0.8 ft)))
```

This completes the implementation of the design rules, and we are now ready to perform calculations of the corner points for constructing the beam geometry. Here we calculate the corner points relative to a "Lower left point" that is obtained with **(getpoint)**:

```
; calculate corner points of the beam
;
(setq p1 (getpoint "\nLower left point: "))
(setq L1 (/ (- wi wt) 2))
(setq L2 (- ht (* ft 2.0)))
(setq p2 (polar p1 0 wi))
(setq p3 (polar p2 (/ pi 2.0) ft))
(setq p4 (polar p3 pi L1))
(setq p5 (polar p4 (/ pi 2.0) L2))
(setq p6 (polar p5 0 L1))
(setq p7 (polar p6 (/ pi 2.0) ft))
(setq p8 (polar p7 pi wi))
(setq p9 (polar p8 (+ pi (/ pi 2.0)) ft))
(setq p10 (polar p9 0 L1))
(setq p11 (polar p10 (+ pi (/ pi 2.0)) L2))
(setq p12 (polar p11 pi L1))
;
```

The points p1 through p12, and the new lengths L1 & L2 are indicated in figure SG7-2

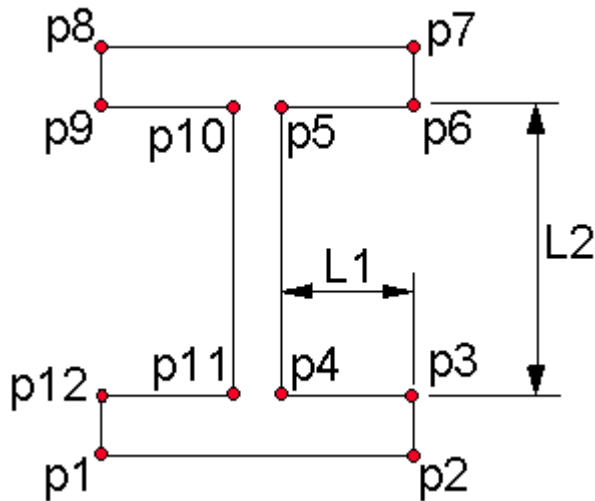


Figure SG7-2 Point and L1 & L2 definitions

All that remains now is to draw the outline of the beam cross-section:

```

; draw the outline of the beam
;
  (command "pline" p2 p3 p4 p5 p6 p7 "")
  (command "fillet" "r" rad "fillet" "polyline" "L")
  (command "pline" p8 p9 p10 p11 p12 p1 "")
  (command "fillet" "polyline" "L")
  (command "line" p1 p2 "")
  (command "line" p7 p8 "")
  (redraw) (princ)
) ; end of the defun

```

The strategy for drawing the outline is to draw polylines through the side points so that the fillets can be easily applied, selecting the "Last" entity with the fillet "polyline" option. This is done for each of the two polylines in turn, and then the "line" command is used to draw the lower and upper edges respectively as shown.

The alternative to using polylines is to draw ordinary lines, and then define points on those lines for selection to place fillets. This is a cumbersome method, and if separate lines and arcs are required, the polylines can easily be exploded after the fillets have been created. The polyline can be selected by using one of its end points as shown in the following sequence:

```

(command "pline" p8 p9 p10 p11 p12 p1 "")
(command "fillet" "polyline" "L")
(command "explode" p8)

```

This program could also be written in modules as follows:

```

; IBEAM7.LSP A program to draw an I-beam
; incorporating design rules
(defun input ()
; get user input and impose restrictions (design rules)
  (setq ht (getreal "\nBeam Height: "))
  (setq wi (getreal "\nBeam Width: "))
  (setq ft (getreal "\nFlange Thickness: "))
  (setq wt (getreal "\nWeb Thickness: "))
  (setq p1 (getpoint "\nLower left point: "))
) ; end of input
;
(defun rules ()
  (cond
    ((< wi (* 0.5 ht)) (setq wi (* 0.5 ht)))
    (> wi (* 0.8 ht)) (setq wi (* 0.8 ht)))
  )
  (cond
    ((< ft (* 0.05 ht)) (setq ft (* 0.05 ht)))
  )
)

```

Introduction to AutoLISP Programming

```

    ((> ft (* 0.15 ht)) (setq ft (* 0.15 ht)))
  )
  (cond
    ((< wt (* 0.1 wi)) (setq wt (* 0.1 wi)))
    ((> wt (* 0.2 wi)) (setq wt (* 0.2 wi)))
  )
  (cond
    ((<= ht 4.0) (setq rad 0.125))
    (T (setq rad 0.5))
  )
  (if (< ft 0.15) (setq rad (* 0.8 ft)))
) ; end of rules
(defun pointset ()
; calculate corner points of the beam
;
  (setq L1 (/ (- wi wt) 2))
  (setq L2 (- ht (* ft 2.0)))
  (setq p2 (polar p1 0 wi))
  (setq p3 (polar p2 (/ pi 2.0) ft))
  (setq p4 (polar p3 pi L1))
  (setq p5 (polar p4 (/ pi 2.0) L2))
  (setq p6 (polar p5 0 L1))
  (setq p7 (polar p6 (/ pi 2.0) ft))
  (setq p8 (polar p7 pi wi))
  (setq p9 (polar p8 (+ pi (/ pi 2.0)) ft))
  (setq p10 (polar p9 0 L1))
  (setq p11 (polar p10 (+ pi (/ pi 2.0)) L2))
  (setq p12 (polar p11 pi L1))
) ; end of pointset
(defun outline ()
;
; draw the outline of the beam
;
  (command "pline" p2 p3 p4 p5 p6 p7 "")
  (command "fillet" "r" rad "fillet" "polyline" "L")
  (command "pline" p8 p9 p10 p11 p12 p1 "")
  (command "fillet" "polyline" "L")
  (command "line" p1 p2 "")
  (command "line" p7 p8 "")
) ; end of outline
(defun ibeam7 ()
  (input)
  (rules)
  (pointset)
  (outline)
;
  (redraw)
  (princ)
) ; end of ibeam7

```

In this modular example, all input is placed into the **(input)** function, and this makes it easier to check that all of the relevant parameters are supplied.

Another thing that you may notice about the **(rules)** function is that a similar set of commands is repeated with different parameters several times:

```

(defun rules ()
  (cond
    ((< wi (* 0.5 ht)) (setq wi (* 0.5 ht)))
    ((> wi (* 0.8 ht)) (setq wi (* 0.8 ht)))
  )
  (cond
    ((< ft (* 0.05 ht)) (setq ft (* 0.05 ht)))
    ((> ft (* 0.15 ht)) (setq ft (* 0.15 ht)))
  )
)

```


Introduction to AutoLISP Programming

```
)  
(cond  
  ((< wt (* 0.1 wi)) (setq wt (* 0.1 wi)))  
  ((> wt (* 0.2 wi)) (setq wt (* 0.2 wi)))  
)  
(cond  
  ((=< ht 4.0) (setq rad 0.125))  
  (T (setq rad 0.5))  
)  
(if (< ft 0.15) (setq rad (* 0.8 ft)))  
) ; end of rules
```

The first three (**cond**) functions are identical in format to each other, and in a case like this it would be convenient to simply call a function with parameters for each of them. In the first (**cond**) function, we have two variables WI and HT, and two constants 0.5 and 0.8, so we could write a function as follows:

```
(defun xcond (V1 V2 C1 C2)  
  (cond  
    ((< V1 (* C1 V2)) (setq V1 (* C1 V2)))  
    ((> V1 (* C2 V2)) (setq V1 (* C2 V2)))  
  )  
) ; end of xcond
```

Here, the variables are called V1 and V2, and the constants are represented by C1 and C2. The (**cond**) function returns the value of the last item that it evaluates, so if one of the tests is true, the corresponding (**setq**) function will return a value for V1. If neither of the tests is true, then (**cond**) will return a value of nil. Now, nil is alright as a returned value in the previous examples because V1, or its equivalent will not be changed, and any subsequent use of its value will give the appropriate results.

However, the defined function (**xcond**) returns the value of the last evaluated item, which may be assigned as in (**setq a (xcond a b c d)**), where a, b, c, and d are the substituted arguments of (**xcond**).

Now, if WI was originally within the design rules, then neither of the conditional tests will be true (non nil), and therefore the (**cond**) function and (**xcond**) will return nil. For this reason it is essential to have a 'default' value returned such as in:

```
(defun xcond (V1 V2 C1 C2)  
  (cond  
    ((< V1 (* C1 V2)) (setq V1 (* C1 V2)))  
    ((> V1 (* C2 V2)) (setq V1 (* C2 V2)))  
    (T V1)  
  )  
) ; end of xcond
```

The 'default' test is simply T, which is always true (non nil), and therefore the expression V1 is returned if all other tests fail.

To use the defined function to set the conditions relating WI and HT, which has WI and HT as the variables, and 0.5 and 0.8 for the constants, you can simply write:

```
(setq wi (xcond wi ht 0.5 0.8))
```

NOTE!! The function (**xcond**) returns the new value of WI, but does not automatically reassign the new value to the variable, so you must add the (**setq wi (...)**) to accomplish this.

The (**rules**) defined function will now be:

```
(defun rules ()  
  (setq wi (xcond wi ht 0.5 0.8))  
  (setq ft (xcond ft ht 0.05 0.15))  
  (setq wt (xcond wt wi 0.1 0.2))  
  (cond  
    ((=< ht 4.0) (setq rad 0.125))  
    (T (setq rad 0.5))  
  )  
  (if (< ft 0.15) (setq rad (* 0.8 ft)))  
)
```

Introduction to AutoLISP Programming

) ; end of rules

The complete program now looks like the updated version IBEAM8.LSP:

```
; IBEAM8.LSP A program to draw an I-beam
; incorporating design rules
(defun input ()
  ; get user input and impose restrictions (design rules)
  (setq ht (getreal "\nBeam Height: "))
  (setq wi (getreal "\nBeam Width: "))
  (setq ft (getreal "\nFlange Thickness: "))
  (setq wt (getreal "\nWeb Thickness: "))
  (setq p1 (getpoint "\nLower left point: "))
) ; end of input
(defun xcond (V1 V2 C1 C2)
  (cond
    ((< V1 (* C1 V2)) (setq V1 (* C1 V2)))
    ((> V1 (* C2 V2)) (setq V1 (* C2 V2)))
    (T V1)
  )
) ; end of xcond
(defun rules ()
  (setq wi (xcond wi ht 0.5 0.8))
  (setq ft (xcond ft ht 0.05 0.15))
  (setq wt (xcond wt wi 0.1 0.2))
  (cond
    ((<= ht 4.0) (setq rad 0.125))
    (T (setq rad 0.5))
  )
  (if (< ft 0.15) (setq rad (* 0.8 ft)))
) ; end of rules
(defun pointset ()
; calculate corner points of the beam
  (setq L1 (/ (- wi wt) 2))
  (setq L2 (- ht (* ft 2.0)))
  (setq p2 (polar p1 0 wi))
  (setq p3 (polar p2 (/ pi 2.0) ft))
  (setq p4 (polar p3 pi L1))
  (setq p5 (polar p4 (/ pi 2.0) L2))
  (setq p6 (polar p5 0 L1))
  (setq p7 (polar p6 (/ pi 2.0) ft))
  (setq p8 (polar p7 pi wi))
  (setq p9 (polar p8 (+ pi (/ pi 2.0)) ft))
  (setq p10 (polar p9 0 L1))
  (setq p11 (polar p10 (+ pi (/ pi 2.0)) L2))
  (setq p12 (polar p11 pi L1))
) ; end of pointset
(defun outline ()
; draw the outline of the beam
  (command "pline" p2 p3 p4 p5 p6 p7 "")
  (command "fillet" "r" rad "fillet" "polyline" "L")
  (command "pline" p8 p9 p10 p11 p12 p1 "")
  (command "fillet" "polyline" "L")
  (command "line" p1 p2 "")
  (command "line" p7 p8 "")
) ; end of outline
(defun ibeam8 ()
  (input)
  (rules)
  (pointset)
  (outline)
;
  (redraw)
  (princ)
)
```

Introduction to AutoLISP Programming

```
) ; end of ibeam8
```

A major advantage in splitting things up into modules is in organization. You can plan your 'algorithm', or method of solution, so that the executing program can be written first, as in:

```
(defun ibeam8 ()  
  (input)  
  (rules)  
  (pointset)  
  (outline)  
  ;  
  (redraw)  
  (princ)  
) ; end of ibeam8
```

Of course, this program has no way of executing as is without the supporting modules, but it can be used as a starting point to enable you to organize your work, since the 'flow' of operations is already defined. Now the job is to write each module in turn, remembering that in each module you may want to define new functions such as (**xcond**) that can be used as required.

Introduction to AutoLISP Programming Correspondence Course

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

Study Guide, part 8 of 8

Lesson Eight - Intelligent Programming with AutoLISP

Solutions to the homework questions:

1. **Question:** 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.

Answer: A typical output from your program will look like figure SG8-1

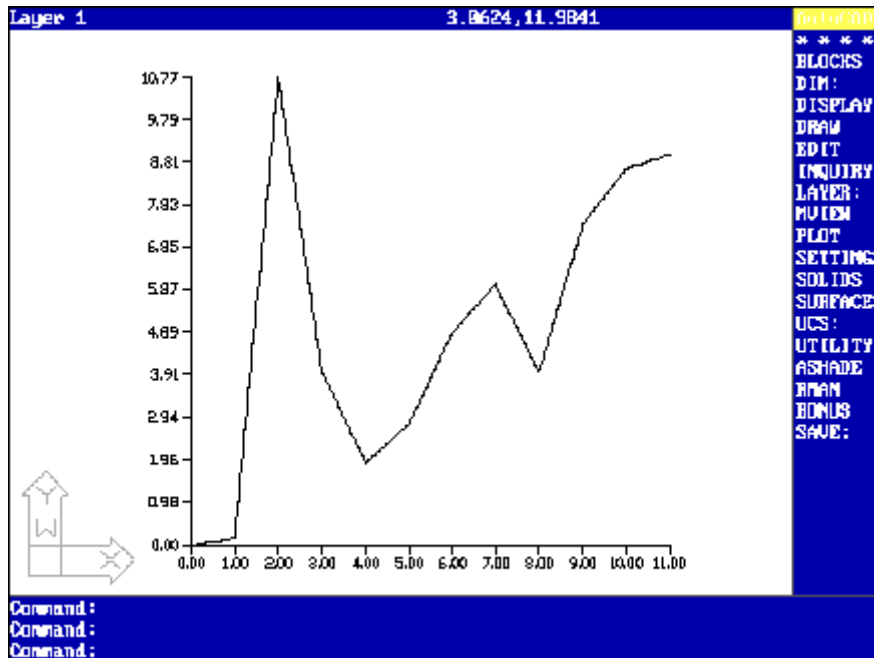


Figure SG8-1, The result of a graph plotting program

The program GRAPH.LSP is a possible solution to plotting the graph, and was used to draw the figure SG8-1.

```
;;GRAPH.LSP A program to plot a graph on the graphic screen, taking
;; any number of y-ordinate values from an external file (.txt)
(defun graph ()
; Initialization and read data from the file
  (setq p0 (getpoint "\nPlot origin: "))
```

Introduction to AutoLISP Programming

```
(setq ptlst nil px (- (car p0) 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
```

The program starts by getting a graph plot origin point, then initializes a point-list and an x coordinate of a point, px. The text data file name is then input by the user and concatenated with the file extension '.txt', before the file is opened in read mode.

The program continues with a test to determine that the data file exists, then reads in the first line in the file which contains the number of y-coordinates of the data points. The x-coordinates are spaced at an interval of 1 unit apart, starting with the x-coordinate of the plot origin point. The point x-y coordinates are then assembled into points before being added to the point list 'ptlst':

```
(if datafile
  (progn
    (setq numpoints (read (read-line datafile)))
    (repeat numpoints
      (setq px (1+ px))
      (setq py (read (read-line datafile)))
      (setq py (+ (cadr p0) py))
      (setq pp (list px py))
      (setq ptlst (append ptlst (list pp)))
    ) ; end of the repeat loop
  ) ;
```

The next section of the program finds the maximum y-ordinate value, using the code supplied with the question:

```
; find the maximum value of the y-ordinates
;
(setq index 0 ymax 0.0)
(repeat numpoints
  (if (>= ymax (cadr (nth index ptlst)))
    (setq ymax ymax)
    (setq ymax (cadr (nth index ptlst)))
  ); end of if
  (setq index (1+ index))
); end of repeat
```

The next section of the program draws the graph axes, using the origin point and the maximum x and y coordinate values, px and ymax respectively. Note that px has been incremented previously so that it is now the maximum value:

```
; draw the axes
(command "line" p0 (setq p1 (list (car p0) ymax)) "")
(command "line" p0 (list px (cadr p0)) "")
;
```

Next, the axis numbering is added by first calculating the range of x and y values. The x-range is simply the number of points minus 1, as we arbitrarily decided to space out the x-ordinates by a unit. A more sophisticated approach would be to relate the x range to the y range so that a 'square' or perhaps a better rectangular shaped graph would result. Here, we define a value of 'xinc' to be 1.0, so that the later code can refer to 'xinc', and there will be an opportunity to experiment with different values later.

```
; add the axis labels and numbers
(setq xinc 1.0
  xrange (* (- numpoints 1) xinc)
  yrange (- ymax (cadr p0))
  th (/ xrange 50.0) ; text height is set to xrange/50
  yinc (/ yrange (- numpoints 1))
  ptx1 p0
  pty1 p0
  ptx2 (polar p0 (* pi 1.5) th) ; a set of points is defined to
  pty2 (polar p0 pi th) ; be 'th' distance from the
axes
)
```

Introduction to AutoLISP Programming

```

(repeat numpoints
  (command "line" ptx1 ptx2 "") ; draw the increment lines
  (command "line" pty1 pty2 "")
  (command "text" "TC" (polar ptx2 (* pi 1.5) (/ th 2)) th 0
    (rtos (- (car ptx1) (car p0)) 2 2) "") ; write x-
ordinates
  (command "text" "MR" (polar pty2 pi (/ th 2)) th 0
    (rtos (- (cadr pty1) (cadr p0)) 2 2) ""); write y-ordinates
  (setq ptx1 (list (+ (car ptx1) xinc) (cadr ptx1))); redefine pts
  (setq ptx2 (list (+ (car ptx2) xinc) (cadr ptx2)))
  (setq pty1 (list (car pty1) (+ (cadr pty1) yinc)))
  (setq pty2 (list (car pty2) (+ (cadr pty2) yinc)))
); end of repeat

```

The 'text' command prompts for Justify/Style/<Start point>;, and it would be possible to enter 'J', followed by the alignment options. However, it is also permissible to enter the alignment option immediately, as in (**command "text" "TC"**), as we did here for the x-ordinate text positions. The next text prompt is for the text start point, for which we use (**polar ptx2 (* pi 1.5) (/ th 2)**). The text height is then specified as 'th', followed by a zero rotation angle. Note how the text string values are calculated for the x ordinates, using (**rtos (- (car ptx1) (car p0)) 2 2**) to give 2 decimal places on the conversion from a real to a string value.

After the ordinate text is written, the ordinate line end points are redefined to the next increment position, and the whole process is repeated in the repeat loop.

Finally, the program draws the actual graph curve using a polyline, before closing the data file. The if statement continues with the 'else' option which is simply to print out the message that the file could not be opened, and the program ends with a redraw and a princ:

```

      (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

```

Now let's have a look at some alternatives to drawing such graphs. The external data file has as its first record the number of y-coordinates to follow, but what if we want to supply an arbitrary number of ordinates without specifying the number explicitly? Our program will need to be able to count the number of data values, and this can be programmed as follows:

```

;;GRAPH3.LSP A program to plot a graph on the graphic screen, taking
;; any number of y-ordinate values from an external file (.txt)
;; The file only contains y-values (not the number of values)
(defun graph3 ()
; Initialization and read data from the file
  (setq p0 (getpoint "\nPlot origin: "))
  (setq ptlst nil px (- (car p0) 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 0)
      (while
        (setq py (read (read-line datafile)))
        (setq py (+ (cadr p0) py))
        (setq px (1+ px))
        (setq pp (list px py))
        (setq ptlst (append ptlst (list pp)))
        (setq numpoints (1+ numpoints))
      ) ; end of the while loop
    )
  )
;
....etc

```

Introduction to AutoLISP Programming

In this case, the first part of the program uses a while loop, which tests that there is data assigned to 'py', and if so, the variable numpoints is incremented by one. Numpoints is initialized to zero before the while loop starts. The rest of the program is unchanged.

Another alternative is to set the aspect ratio of the graph, because the x ordinate has been set to a unit spacing interval, which may lead to some odd aspect ratios of the respective axes for very small or very large y-values. We could, for instance, ask the user to supply the size of the plot from a selection of possible values, such as 6 x 4, or 11 x 8, then use those values to set the size by scaling the picture to suit. GRAPH4.LSP is such a program:

```
;;GRAPH4.LSP A program to plot a scaled graph on the graphic
;; screen, taking any number of y-values from an external file (.txt)
;; The file only contains y-values (not the number of values)
(defun graph4 ()
;      Initialization and read data from the file
  (setq oblp (getvar "BLIPMODE"))
  (setq ocmd (getvar "CMDECHO"))
  (setq orth (getvar "ORTHOMODE"))
  (setvar "BLIPMODE" 0)
  (setvar "CMDECHO" 0)
  (setvar "ORTHOMODE" 0)
  (setq p0 (getpoint "\nPlot origin: "))
  (prompt "\nEnter the first number 6, 8, 11, or enter 6x4 format, ")
  (prompt "\nto choose the size of plot: 6x4/8x6/<11x8>: ")
  (setq size (getstring))
  (cond
    ((or (= size "6") (= size "6x4")) (setq xsize 6.0 ysize 4.0))
    ((or (= size "8") (= size "8x6")) (setq xsize 8.0 ysize 6.0))
    ((or (= size "") (= size "11") (= size "11x8"))
      (setq xsize 11.0 ysize 8.0))
    (T (progn
      (prompt "\Bad entry, - default size used.")
      (setq xsize 11.0 ysize 8.0)
    ); progn
  ); last condition
); cond
```

Here, the input section includes options for the graph size, and the (cond) function provides different ways of specifying the xsize and ysize variables. Note the default case, if the entered data doesn't match any of the options, the default <11x8> is assumed.

The next section reads in the y-values and appends them to a list 'yvals' without reference to the x-coordinate.

```
(setq yvals nil)
(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 0)
    (while (setq str1 (read-line datafile))
      (setq py (read str1))
      (setq yvals (append yvals (list py)))
      (setq numpoints (1+ numpoints))
    ); end of the while loop
    (close datafile)
```

In the next section the maximum value of y is found from the list 'yvals':

```
; find the maximum value of the y-ordinates
;
  (setq index 0 ymax 0.0)
  (repeat numpoints
    (if (>= ymax (nth index yvals))
      (setq ymax (nth index yvals))
      (setq ymax (nth index yvals))
```

Introduction to AutoLISP Programming

```
); if
  (setq index (1+ index))
); repeat
```

The xsize and ysize variables are now used to draw the axes of the graph at the intended size:

```
; draw the axes

(command "line" p0 (setq pymax
  (list (car p0) (+ (cadr p0) ysize))) "")
(command "line" p0 (setq pxmax
  (list (+ (car p0) xsize) (cadr p0))) "")
```

The next section adds the axis values based on an assumed number of increments, which is arbitrarily set to 10 in this case. This will produce a total of 11 numbers per axis. The actual interval between the axis numbers is given by xinc and yinc respectively as shown in the first four statements:

```
;          add the axis labels and numbers
(setq incs 10
  xinc (/ xsize incs)
  th (/ xsize 50.0)
  yinc (/ ysize incs)
  ptx1 p0
  pty1 p0
  ptx2 (polar p0 (* pi 1.5) th)
  pty2 (polar p0 pi th)
)
(setq valxx (- 1))
(setq valyy (- 1))
(repeat (1+ incs)
  (command "line" ptx1 ptx2 "")
  (command "line" pty1 pty2 "")
  (command "text" "TC" (polar ptx2 (* pi 1.5) (/ th 2)) th 0
    (rtos (setq valxx (1+ valxx)) 2 2) "")
  (command "text" "MR" (polar pty2 pi (/ th 2)) th 0
    (rtos (* (setq valyy (1+ valyy)) (/ ymax incs)) 2 2) "")
  (setq ptx1 (list (+ (car ptx1) xinc) (cadr ptx1)))
  (setq ptx2 (list (+ (car ptx2) xinc) (cadr ptx2)))
  (setq pty1 (list (car pty1) (+ (cadr pty1) yinc)))
  (setq pty2 (list (car pty2) (+ (cadr pty2) yinc)))
);repeat
```

In the above, the (command "text"...) will print out actual values of the y-ordinate at the appropriate intervals, taking the maximum value of y and dividing by the number of increments as the actual value of an increment based on the data supplied in the file. This number, expressed as (/ ymax incs) is multiplied by 0, 1, 2, 3....etc. at each increment along the axis. The sequence 0, 1, 2, 3... is produced by repeating the loop and changing the value of the multiplier in (setq valyy (1+ valyy)), where the value of valyy is initialized to -1 as shown by (setq valyy (- 1)). Note here that the (- ...) function can be used with a single argument, in which case it returns the argument with its sign reversed.

The calculated y value is then converted to a string to be written as the 'text', so the total 'text' argument looks like:

```
(rtos (* (setq valyy (1+ valyy)) (/ ymax incs)) 2 2)
```

It is assumed that the x-values are given by (setq valxx (1+ valxx)), where valxx is also initialized to -1, in other words the x-values simply consist of the series 0, 1, 2, 3.. etc. up to a maximum value of the number of increments (in this case, 10).

As before, the axis marker lines are incremented and drawn inside the repeat loop.

The next and final section draws the polyline, using a scaling factor for the y-values so that the plotted curve matches the printed ordinate values:

```
(setq num (- 1.0))
(command "pline" (foreach y yvals
  (setq p (list (+ (car p0)
```


Introduction to AutoLISP Programming

```
(* (/ xsize (- numpoints 1)) (setq num (+ num 1.0))))  
  (+ (cadr p0) (* y (/ ysize ymax)))  
  ); list  
  ); setq  
  (command p)); foreach  
); command  
) ; end of progn  
  (princ (strcat "Cannot open file " fname))  
) ; end of 'if'  
(setvar "BLIPMODE" oblp)  
(setvar "CMDECHO" ocmd)  
(setvar "ORTHOMODE" orth)  
(princ)  
) ; end of defun
```

A complication here is that the number of points in the data file does not match the number of axis increments, and the size of the graph is no longer simply the difference between the graph origin point and the maximum y-value. Instead we use a suitable 'scaling' as shown. The scaling consists of multiplying each of the y-values in turn by the factor (`/ ysize ymax`). The (`command "pline"...`) function is rather complicated, but it is really just another use of the standard (`foreach point_list (command point)`) function that we use to supply the points to a polyline. In this case, we do not actually have a list of points, but we have a list of the y-values of the required points. We need to take each y-value from the list and combine it with an x-ordinate with the (`list`) function before using the (`command point`) function to 'dump' out the value of the point.

The loop starts off with (`foreach y yvals (...)`) as usual. In fact, what we have is actually (`foreach y yvals (setq p.....) (command p)`), so we are not dumping out the y-values, but rather the points 'p'. Note that we are assigning the points to 'p' with (`setq p ...`), and although (`setq p ..`) returns the value of the points 'p' at every iteration of the loop, the points are not considered to be 'dumped out' to the (`command "pline"...`) until we use the format (`command p`) as shown.

Now, the item (`setq p..`) is of the standard form (`setq p (list x_value y_value)`), where the `x_value` is calculated as:

```
(+ (car p0) (* (/ xsize (- numpoints 1)) (setq num (+ num 1.0))))
```

Here, we add the x-ordinate of the origin point (`car p0`) to each increment of x. the increment of x is simply (`/ xsize (- numpoints 1)`), or the size of x divided by the total number of points minus one in the data file (to be plotted). The final value of x will therefore be `xsize` plus the x-ordinate of the origin point. We have already decided that the `xsize` will be either 6, 8, or 11, so if we are to plot 'numpoints' points, including `x = 0`, then the number of intervals between plotted points is `(- numpoints 1)`, hence its appearance in the calculation.

The next job is to multiply each increment by the series 0, 1, 2, 3...etc as before, and we use the same counting device as before, in the form of (`setq num (+ num 1.0)`) where 'num' is initialized to -1.0 as shown.

The y-values are given as (`+ (cadr p0) (* y (/ ysize ymax))`), which is a much simpler expression, as each successive y value comes from the list `yvals`, and must simply be scaled by the scale factor (`/ ysize ymax`) before being added to the y-ordinate of the origin point (`cadr p0`) as shown.

Perhaps the most surprising feature of this approach is that the points `p` that are assigned by (`setq ..`) are not taken as points in the (`command "pline" ...`) even though they are returned by the (`setq p ..`) function. The answer is that (`foreach ..`) can have any number of expressions after its first two arguments, which are the 'item_name' and the 'list_of_items' respectively. The result of the last expression evaluated is returned by (`foreach..`), and we have made that expression (`command p`), and not (`setq p..`).

In fact, we cannot rely on (`setq p ..`) to provide each point, because only the last point generated in the loop would be available for the (`command "pline"...`), so we must use (`command p`) to do the job.