November 30 – December 3, 2004 ◊ Las Vegas, Nevada

Fundamentals of AutoLISP

dave espinosa-aguilar – Toxic Frog Multimedia
Volker Cocco (Assistant)

CP11-4 **Course Description:**

AutoLISP® has been around for a long time and has always separated the AutoCAD® green thumbs from the gurus. This course begins by debunking some popular rumors that AutoLISP is going away and that AutoCAD VBA is better than AutoLISP. Both languages have their strengths and weaknesses. The amount of AutoLISP code used in CAD-dependent industries today remains tremendous, and AutoLISP is more powerful today than ever. It's free, it's gentle and forgiving, and it provides users with the ability to create new AutoCAD commands in minutes. This class helps seasoned AutoCAD users enter the world of customization and programming using AutoCAD's native graphical language. The class provides over 30 examples of code to help you understand the syntax of this language. The material is designed for intermediate-level AutoCAD users who have never programmed in AutoLISP before.

About the Speaker:

As a consultant in CAD and multimedia for 18 years, dave trains professionals in architectural and engineering firms on the general use, customization, and advanced programming of Autodesk® design and visualization software. He has authored facilities management applications for several Fortune 500 companies using ObjectARX®, VBA, and AutoLISP® technologies. dave also creates graphics applications and animations for Toxic Frog Multimedia and has coauthored several books including NRP's Inside 3D Studio MAX® series. dave served on the Board of Directors for AUGI® for 6 years, including serving as president in 1996.

Hi, my name is dave espinosa-aguilar, and I've been using and training folks on AutoCAD since the early 1980's. I work with about 125 companies on average a year doing onsite training and writing custom applications for them, and I get to see how a lot of people throughout the country use the program in new and creative ways. I've also been programming in AutoCAD since it was first possible. AutoLISP has been around for some time now, and programming capability really separates the AutoCAD greenthumbs from the gurus. It's free, and it provides users with the ability to create their own AutoCAD functions in minutes. It's very forgiving too (it won't destroy your operating system unless you're really trying hard), and it's a graphical language which works natively with AutoCAD entities. In other words, the more you know about basic AutoCAD usage, the more powerful your AutoLISP programming becomes!

AutoCAD Release 2006 is Already Here!!!

Well, not really. But it can seem like it when you have as much control over new features as AutoLISP can give you. Autodesk has historically packed a lot of new and useful features in their new releases, but they can't develop every function a user can think of, especially when it is specifically geared to the way your office (and perhaps your office alone) does work. So the company does the next best thing: they give their users a tool powerful enough to develop their own new commands. Once mastered, AutoLISP can give you the means to create so many new features of your own that after you've developed your own utilities it will seem like you've got a new release of AutoCAD on your desktop.

Yes, it's going to take some practice to master AutoLISP, but then.... so did the PLINE command once upon a time, right? AutoLISP isn't a hard language to master. It just takes working with it for a while. Once you get the language under your belt, WATCH OUT WORLD! Some of the most successful third-party applications ever developed for the AutoCAD world began as collections of powerful AutoLISP routines, and it's amazing how 15 minutes of typing can provide you with commands you've dreamed of having available for years… if you don't believe me, check out POLYTEXT at the end of this handout!!

The documentation for this class is merely the code which will be reviewed, line by line, during the class. Each of these exercises provides expressions and algorithms to grasp progressively more complex concepts. If you have any questions about the purpose of any function used here, feel free to write the author about it at dave.espinosa-aguilar@autodesk.com

"LISP" stands for **LIS**t **P**rocessing: AutoLISP is a subset of the entire LISP language.

You can evaluate AutoLISP code at the command prompt just like a regular command.

AutoLISP uses open and closed parenthesis to organize what code should be performed and in what order. You need the same amount of open parenthesis as you do closed parenthesis to make a complete AutoLISP state or function or program work properly. But the world won't come to an end if you don't do this correctly.

AutoLISP evaluates expressions without using an '='. Unless you tell AutoLISP not to, it **always** evaluates the expressions you give it. Examine these basic math functions. Note verb/noun pattern, and associative (order doesn't matter) , non-associate (order does matter) behavior:

```
(+ 1 2) → 3                      (/ 5 2) → 2 what happened? integers used!
(+ 1 2 3 4 5) → 15 (associative) (/ 5 2.0) → 2.5 one real involved fixes this
(* 1 2 3 4) → 24 (associative)   (/ 25 4 3) → 2
(- 5 2) → 3 (non-associative)    (/ 25.0 4.0 3.0 → 2.08333
(- 2 5) → -3 (non-associative)   (/ 25.0 4 3) → 2.08333
```

AutoLISP is a "language" and languages have vocabularies. Some new words:

"Syntax" is the set of words (and their required usage structure) that make up the AutoLISP language. These words do specific things when they are used in certain ways on values and even other words.

A **"variable"** (or "symbol") is like a named box which can hold any kind of value inside of it. Once you put a value inside a variable, the variable remembers that value and you can get that value back anytime by simply remembering the box's name. You can clean out this box, fill it with other things, or even get rid of the box. **Boxes may not be named after any word in the syntax!** A safe strategy for box naming is to use 6 characters or less and to use letter-number combinations as box names (ex: radius2). By using letter-number (in that order) combinations for names, you're almost guarenteed of not accidentally using an AutoLISP command for a name.

We combine syntax & variables to form **"expressions"** (or sentences). Variables can hold many types of values:

"Integers" (or "fixed numbers") are whole numbers without decimal values in them. How many kids do you have? Hopefully not 2.5. They're great for counting things up, making decisions and doing quick math.

"Real" numbers (which are also called "floated numbers") are numbers that do have decimal values in them. They're very accurate (to 16 decimal places—take note, the **LIST** command only reports up to 8 decimal places bu AutoCVAD's accuracy is actually a lot more than this) but because they're of that they're also more anal to work with and generally slower to work with (they use up more memory than integers).

"Strings" are like ordinary words or sentences that we use to write sentences and they always have quotes ("") around them to distinguish them from numbers (integers or real/floated numbers). "dave" is a string. "Hi there. How are you today?" is a string. "22" is a string. "22" does **NOT** have a numeric value to it and there are all sorts of reasons why you would want a number to look and act like a word instead of like a number (ex: creating prompts—more later on this).

"Booleans" (it's not soup, it's named after a Mr. Boole who invented the thing) are truth conditions (a value which says if something is true or false) that make it possible for us to make decisions in our programming (if something is true, do this. if not, do that). When something is true, it evaluates to a value of **T**. if it is false, it evaluates to a value of **nil**. Nil does not mean zero!! Nil means oblivion. It means nothingness. If something is equal to zero, we at least have a numeric concept of it. But nil means we know nothing about it.

"Lists" (some call them "arrays") are like trains of values. They have an engine and a bunch of box cars following it. You can make the engine and the boxcars out of any kind of value anytime you want (try that with Visual Basic) Lists have an order to them so that you can easily get the value of any particular boxcar value (or even what's in the engine). Lists always have parenthesis around them just as strings have quotes (ex: 1 2 3).

<u>Our first AutoLISP command (other than math functions):</u> (**setq)** which stands for "set quotient". The (**setq)** function does three things: it creates a box, it names the box, and it puts a value in that named box. Below at the command prompt we can type:

```
Command: (setq value1 1)
1
```

The **online help** is invaluable in learning and remembering AutoLISP syntax. In AutoCAD 2005, we find AutoLISP syntax under Help|Developer Help|AutoLISP Reference|AutoLISP Functions. If you go to the "s" section, you'll find setq. It tells us the syntax for the setq function is (setq sym expr [sym expr]…). "sym" stands for the variable name, "expr" stands for the value assigned to the variable. "…" means that you can several values in several boxes in one setq statement! (setq val1 1 val2 2 val3 3) would set values 1, 2, and 3 to three boxes.

Notice that, like math expressions, the (setq) expression has evaluated what it should put in the new box named value1. If we want to see what value is inside any named box, we can **"bang"** the box (exclamation point):

```
Command: !value1
1
```

The order in which AutoLISP expressions are evaluated depends on how they are organized with parenthesis. Expressions "on the inside" go first. Expressions "on the outside" go last:

<u>the expression (+ 3 (* 4 (/ 4 (- 10 8)) 3)) reduces to:</u>

```
(+ 3 (* 4 (/ 4 2) 3)) → (+ 3 (* 4 2 3)) → (+ 3 24) → 27
```

<u>there are other math functions like **sqrt** (square root):</u>

```
(sqrt (+ 4 (* 6 2))) → (sqrt (+ 4 12)) → (sqrt 16) → 4
```

<u>if we fill variables with values, we can then perform functions on those variables:</u>

```
(setq val1 1) → 1
(setq val2 2) → 2
(+ val1 val2) → 3
(/ (+ val1 val2) val2) → (/ (+ 1 2) 2) → (/ 3 2) → 1 ack! That's not right!
```

<u>Introducing functions like (fix)/(float), (atoi)/(atof), and (rtos)/(itoa) which change variable value types:</u>

```
(fix 10.5) → 10        (atoi "22") → 22        (rtos 10.5) → "10.5"
(float 10) → 10.0      (atof "22") → 22.0      (itoa 10) → "10"
```

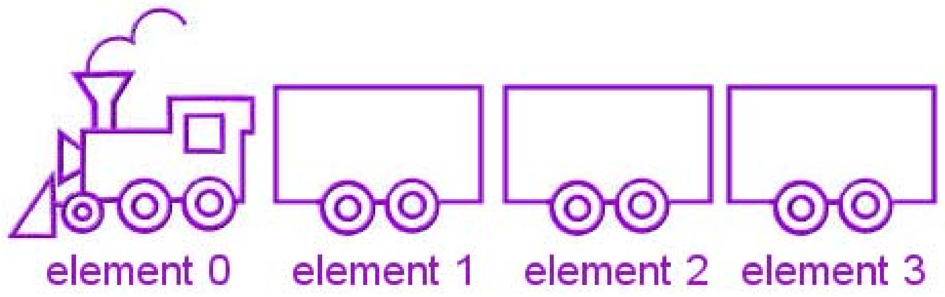
<u>so to go back to our broken equation:</u>

```
(float val1) → 1.0 by floating any number involved, we fix the problem
(/ (+ val1 val2) val2)) → (/ (+ 1.0 2) 2) → (/ 3.0 2) → 1.5 there we go
```

By using (setq) you can overwrite values in variables with other values. You can replace integers with reals or strings or booleans. Be careful though because once you overwrite a value in a variable, the previous value is gone.

```
(setq val1 10.5) → 10.5 this overwrites any value previously in variable val1
(setq val2 (- val1 (fix val1))) → ? what is variable val2 being overwritten by?
(setq val2 (- 10.5 (fix 10.5))) → (setq val2 (- 10.5 10)) → (setq val2 0.5)
!val2 → 0.5 previous value of variable val2 is overwritten by a value of 0.5
```

Working with lists is like working with trains. Lists always have parenthesis around them and the items or elements or values (people use these words interchangeably) are in a specific order starting with item 0 (not 1!!). Think of the first item in the train as the engine and all the other values as boxcars. An engine pulling 3 boxcars is a list with 4 items. To see what the engine is holding, look at item 0. To see what the "caboose" is holding, look at item 3 (not 4!!).



functions like (list), (car), (cdr) and (nth) create and evaluate lists (think trains):

```
(list 1 2 3 4 5) → (1 2 3 4 5)  the list function creates the train
(setq train1 (list 1 2 3 4 5)) → (1 2 3 4 5)
!train → (1 2 3 4 5)  variable train1 now holds this entire list of numbers
(setq engine1 (car train1)) → 1
!engine → 1 the car function evaluates the engine (element 0) only
(setq boxcars1 (cdr (train1)) → (2 3 4 5)
!boxcars1 → (2 3 4 5) the cdr function evaluates boxcars as a list
(nth 0 train1) → 1 the nth function evaluates the item you specify in the list
(nth 1 train1) → 2
(nth 4 train1) → 5
(car (cdr train1)) → (car (2 3 4 5)) → 2
(cdr (cdr train1)) → (cdr (2 3 4 5)) → (3 4 5)
(car (cdr (cdr train1))) → (car (cdr (2 3 4 5))) → (car (3 4 5)) → 3
```

the (reverse) function evaluates (does not replace) a list backwards:

```
(reverse train1) → (5 4 3 2 1) reverse simply evaluates, it does not overwrite!!
!train → (1 2 3 4 5)
```

<u>the (cons) function adds a new item to the **front** of the train (a new engine)</u>

```
(cons 0 train1) → (0 1 2 3 4 5) again this just evaluates, it does not overwrite
!train1 → (1 2 3 4 5)  train still holds its original list
(setq train1 (cons 0 train1)) → (0 1 2 3 4 5) a new engine has been added
```

How do we add a new caboose? Let's add a value of 6 to the "end" of the train:

```
!train1 → (0 1 2 3 4 5)
(reverse train1) → (5 4 3 2 1 0)
(cons 6 (reverse train1)) → (6 5 4 3 2 1 0)  so 6 is evaluated as a new engine
(reverse (cons 6 (reverse train1))) → (0 1 2 3 4 5 6)
(setq train1 (reverse (cons 6 (reverse train1)))) → (0 1 2 3 4 5 6) new train1
```

We can determine how long the train is with the **(length)** function:

```
!train1 → (0 1 2 3 4 5 6)
(length train1) → 7
```

<u>AutoCAD uses a lot of 3D coordinates which are just a list of three real numbers:</u>

```
(setq point1 (list 1.0 1.0 0.0)) → (1.0 1.0 0.0)
```

The more you know about AutoCAD, the more you know about AutoLISP. Cose your eyes and think about the sequence you would normally use to TYPE the commands to create a new layer named WALL that is yellow. The (command) function behaves identically. Try this:

```
(command "layer" "m" "wall" "c" "2" "wall" "") → nil (nothing to evaluate)
```

A new layer is created. The (command) function uses strings (values in quotes) as if you had typed them at the command prompt yourself. You can spot AutoLISP expressions right in the heart of this sequence of strings too:

```
Command: (setq point1 (list 1.0 1.0 0.0) point2 (list 2.0 2.0 0.0)) → (2 2 0)
Command: (command “line” point1 point2 “”) → nil (a LINE was just drawn!!)
```

What is the "" for at the end of the statement? It's a <return>!

The (getpoint) function allows the user to pick a point and it evaluates a list of 3 real numbers:

```
Command: (getpoint)   the cursor waits for you to pick a point → (x y z)
```

How do we pull the x, y and z values from a (getpoint) function?

```
(setq point1 (getpoint)) → (1.0 2.0 0.0)  this is an example point (1.0 2.0 0.0)
(setq x (car point1)) → 1.0 OR (setq x (nth 0 point1)) → 1.0
(setq y (car (cdr point1))) → 2.0 OR (setq y (nth 1 point1)) → 2.0
(setq z (car (reverse point1))) → 0.0 OR (setq z (nth 2 point1)) → 0.0
```

According to the online help the full syntax of (getpoint) function let's us use a prompt and even rubberband from a previous point. The "\n" string prefix tells the prompting to use a "new line". Also note the aesthetics used such as a space after the colon to imitate AutoCAD's prompt styling:

```
(setq point1 (getpoint "\nPick your first point: "))
(setq point2 (getpoint point1  "\nPick second point: "))
(command "line" point1 point2 "")
```

AutoLISP doesn't care how these lines of code are organized so long as they are in the proper sequence. For example, the above three lines could actually be written as:

```
(setq point1 (getpoint "\nPick your first point: "))(setq point2 (getpoint point1
"\nPick second point: "))(command "line" point1 point2 "")
```

To create these lines of code as a **program**, we use the (defun) function. The syntax for defining a function allows us to define a key WORD to be used at the command prompt and everything that falls inside of this definition runs when the word gets typed:

```
(defun C:DOIT ()
(setq point1 (getpoint "\nPick your first point: "))
(setq point2 (getpoint point1  "\nPick second point: "))
(command "line" point1 point2 "")
)
```

The key WORD in this case is DOIT. Notice that the first line doesn't have all the parenthesis closed, but the last line actually closes the original (defun) function. We can copy and paste this entire program to the Command: prompt and then type DOIT and it will behave just like a regular AutoCAD command!! Pretty slick eh?

To save this kind of thing to a file that you can share with others, we open **Notepad** (or any ASCII editor--- warning, Microsoft Word DOC files do not save in ASCII format by default), type these lines of code in Notepad, save the file as an ASCII text file with a **.LSP** file extension (even though it is a .TXT file really), and then we can give that file to others to use. To load a program, use the (load) function at the Command: prompt. Notice the weird UNIX drive/subdirectory notation:

```
Command: (load "c:/whatever/myroutine.lsp")
```

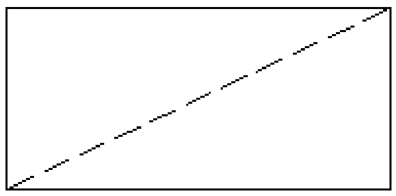
```
C:DOIT
Command: DOIT  → and it does it!
```

More aesthetics: in the code version below we use the (setvar) function to set a system variable (CMDECHO set to a value of 0) to hide the line-by-line execution of a running program instead of reporting everything while it runs. This makes things run more elegantly from a user perspective. We also use the (princ) function so that nothing gets reported as an evaluation when the program runs which also makes things look nice from a user perspective. We also introduce remarks which are really handy for reminding yourself what lines of code do:

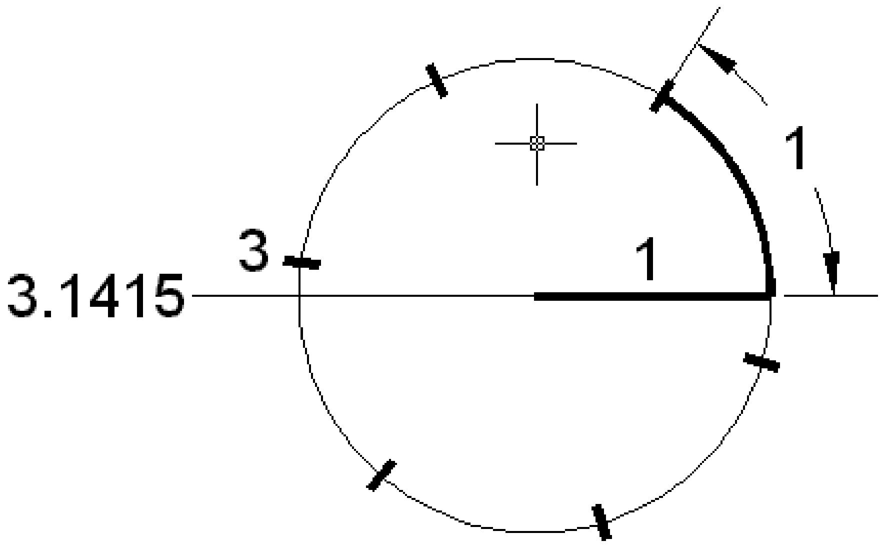
```
; this is called a remark. It doesn't do anything but remind us what we're doing
; here's another remark. This program is our first program with remarks.
(defun C:DOIT () ; the () thingie after C:DOIT is used to globalize/localize vars
  (setvar "CMDECHO" 0) ; the (setvar) function sets a system variable!
  (setq point1 (getpoint "\nPick your first point: ")) ; notice we're indenting
  (setq point2 (getpoint point1  "\nPick second point: "))
  (command "line" point1 point2 "") ; you can put remarks after AutoLISP statements
  (princ) ; this statement tells AutoCAD NOT to evaluate anything unless told to.
) ; this finished off the (defun) command. All parenthesis have to be balanced out.
```

Write a program which gets two diagonally opposed points and draws a box from them:

```
(defun C:LINEBOX ()
  (setvar "cmdecho" 0)
  (setq p1 (getpoint "\nPick first point: ")
        p2 (getpoint p1 "\nPick second point: ")
        p3 (list (nth 0 p1) (nth 1 p2) 0.0)
        p4 (list (nth 0 p2) (nth 1 p1) 0.0)
  ) ; notice only one (setq) statement is used for all 4 points!
  (command "line" p1 p3 p2 p4 p1 "")
  (princ)
)
```



There is really no such thing as an "inch" or a "mile" or a "degree" or an "angle" in AutoCAD. All we really have to work with are numbers. We can use numbers to describe magnitudes (distances), but how would we use a number to describe a direction (angle)?



Introducing the radian. In ancient times, someone realized that if you draped a string the length of a circle's radius around the circumference of that circle, that you got a little over 3 lengths of string (3.1415926).

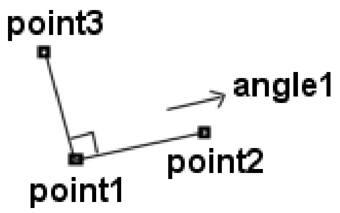
If one assumes that the direction to the exact right is an angle of 0.0, then you could also assume that the direction exactly to the left is "3.1415926 string lengths from the exact right" (180 degrees). So a length of string from exact right can actually tell you a direction to go from the circle's centerpoint, and AutoCAD uses radians in this way to understand angles.

We think of "pi" (or 3.1415926 radians) as being equal to 180.0 "degrees." To calculate how many degrees(d) to radians(r) there are, we can use the basic relationship:

```
180.0     d
-----  = ---  so:  (setq r (/ (* d pi) 180.0))
  pi      r
```

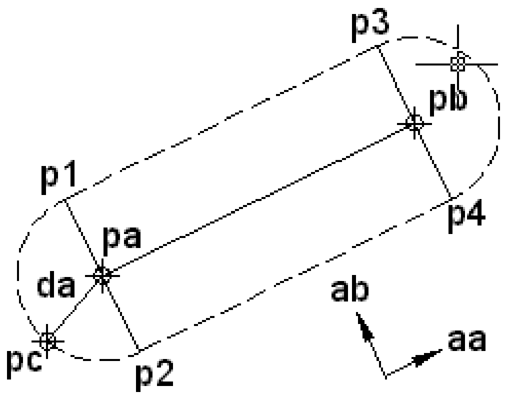
The **(distance)** and **(angle)** functions calculate distances and angles (in radians) between two points (2 lists of 3 real numbers each). The **(pi)** function is a built-in constant with a value of 3.14159265358979323. The **(polar)** function enables us to determine a new point from an existing point using an angle (in radians) and distance. The routine below calculates the distance and angle between two user-provided points, calculates an orthogonal angle from the angle between these two points, and the determines a new third point (from point1):

```
(setq point1 (getpoint "\nPick your first point: ")
      point2 (getpoint point1 "\nPick your second point: ")
      distance1 (distance point1 point2)
      angle1 (angle point1 point2)
      point3 (polar point1 (+ angle1 (* pi 0.5)) distance1)
) ; draw a line from !point3 to verify it works
```



<u>Write a program which uses three point picks to create an island tag:</u>

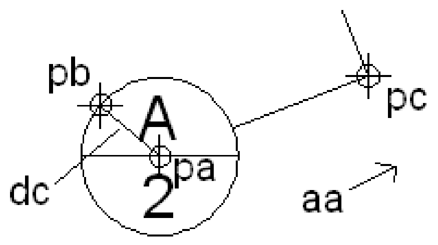
```
(defun C:ISLE ()
   (setvar "cmdecho" 0)
   (setq pa (getpoint "\nBeginning point: ")
         pb (getpoint pa "\nOrientation point: ")
   )
   (grdraw pa pb 1 1) ; this is a redraw function
   (setq pc (getpoint pa "\nRadius point: "))
   (setq aa (angle pa pb)
         ab (+ aa (* pi 0.5)) da (distance pa pc)
         p1 (polar pa ab da) p2 (polar pa (+ ab
pi) da)
         p3 (polar pb ab da) p4 (polar pb (+ ab pi) da)
```



```
   )
   (command "pline" p2 p4 "a" p3 "l" p1 "a" p2 "cl")
   (princ)
)
```

Create a tag with superscript, subscript and cross-sectional tag:

```
(defun C:TAG ()
   (setq pa (getpoint "\nPick tag centerpoint: ")
         pb (getpoint pa "\nPick tag cross-section/orientation point: ")
         pc (getpoint pa "\nPick tag radial point: ")
         aa (angle pa pb)
         dc (distance pa pc)
         suptxt (getstring T "\nEnter superscript: ")
         subtxt (getstring T "\nEnter subscript: ")
   )
   (command "circle" pa pc)
   (command "line" (polar pa pi dc) (polar pa 0.0 dc) "")
   (setq pd (polar pa aa dc) pe (polar pb (+ aa (* pi 0.5)) dc))
   (command "line" pd pb pe "")
   (command "text" "j" "m" (polar pa (* pi 0.5)(* dc 0.5))(* dc 0.5) "0.0" suptxt)
   (command "text" "j" "m" (polar pa (* pi 1.5) (* dc 0.5))(* dc 0.5) "0.0" subtxt)
   (princ)
)
```



Working with strings: string manipulation can be done with the **(strcat)**, **(getstring)** and **(prompt)** functions. (strcat) "glues" strings together into one string. **(getstring)**, like (getpoint), also has some parameters to it. If a 'T' value is provided, the user can enter spaces in the text being entered. (getstring) also provides a prompt similar to (getpoint). The (prompt) function spits any **string value** to the Command: prompt.

```
(setq name1 "dave") → "dave"
(setq name2 "volker") → "volker"
(setq name3 (strcat name1 name2))
!name3 → "davevolker"
(setq name1 (getstring "\nEnter a name: "))  → user can enter a string value
(setq sentence1 (getstring T "\nEnter a sentence: "))  → user can use spaces
(prompt name1) → the name the user entered is shown on the Command: prompt.
```

Working with prompts: suppose you have a variable named total1 with a numeric (integer) value of 5 stored in it, and suppose you want to report to the user how many boxes of stuff have been found in a drawing. In order to make the integer number 5 part of the reporting prompt (which requires a string value only, no numbers are allowed to be passed to the prompt function), we have to convert the number 5 to a string equivalent "5" so that it can be "glued" with the other two strings of the prompt before it is reported at the Command: prompt. Now do you see why functions like (itoa) are important?

```
(setq total1 5) → 5   which is an integer value, not a string
(setq text1 (strcat "\nYou have " (itoa total) "boxes of stuff in your drawing."))
(prompt text1) → sends the combined string to the Command: prompt
!text1 → "You have 5 boxes of stuff in your drawing."
```

Working with files: AutoLISP can detect, create, open, append, overwrite and delete ASCII texts files. We use the **(findfile)** function to see if a file already exists. We use the **(open)** function to create a new file or to open an existing file. You can open a file using the "w" parameter to write/overwrite content to the file, or the "a" parameter to open an existing file and append it with content, or the "r" parameter to open the file to read its content. We use the **(close)** function to close an open file **and you always want to close files you have opened**. We use the **(read-line)** function to read a line of text from an open file, and we use the **(write-line)** function to write a line of text to an open file. Note again the weird UNIX notation on filenames.

```
(setq f (findfile "c:/autoexec.bat")) → "C:\\autoexec.bat" if the file exists
(setq f (open "c:\\dave.txt" "w")) → #<file "c:\\dave.txt"> ready to write content
(close f) → nil   which closes the open file c:\\dave.txt
(defun C:TEXTWRITE ()
  (setvar "CMDECHO" 0)
  (setq txt (getstring T "\nEnter a sentence: "))
  (setq f (open "c:\\test.txt" "w")) ; 'write' mode
  (write-line txt f)
  (close f)
  (princ)
) ; go open the file c:\\test.txt with notepad… the string will be there.

(defun C:TEXTAPPEND ()
  (setvar "CMDECHO" 0)
  (setq txt (getstring T "\nEnter a sentence: "))
  (setq f (open "c:\\test.txt" "a")) ; 'append' mode
  (write-line txt f)
  (close f)
  (princ)
) ; go open the file and notice that the string has been appended to the original
```

```
(defun C:TEXTREAD ()
  (setvar "CMDECHO" 0)
  (setq f (open "c:\\test.txt" "r"))
  (setq txt (read-line f))
  (prompt txt)
  (princ)
) ; the command prompt reports the first string in the file
```

But what if we want to read all of the strings in a file? Time to learn all about **loops** and see how Booleans get used to make decisions.

Working with truth conditions: something true is valued **T** and something false is valued **nil**:

```
(= 1 1) → T         (= 1 0) → nil            (= 1 1.0) → T             (= 1 1.01) → nil
(/= 1 1) → nil      (/= 1 0) → T             (/= 1 1.0) → nil          (/= 1 1.01) → T
(> 2 1) → T         (>= 2 1) → T             (>= 1 1) → T              (<= 1 1) → T
(= “w” “w”) → T     (= “w” “x”) → nil        (= “Hi” “hi”) → nil       (= “Hi” “Hi”) → T
(and (= 1 1) (= 2 2)) → T                    (and (= 1 1) (= 1 2)) → nil
(or (= 1 1) (= 1 2)) → T                     (or (= “hi” “hi”) (= 1 1)) → nil
```

The **(if)** function lets us make a decision to do one thing or another based on whether a condition is true or false. If the condition is true, it does the first thing. If it is false it does the second thing:

```
(if T (prompt “It is true.”) (prompt “It is false.”)) → It is true.
(if nil (prompt “It is true.”) (prompt “It is false.”)) → It is false.
```

The **(while)** function lets us loop through a series of statements while a condition is true:

```
(setq counter1 0)
(while (< counter1 10)
  (prompt (itoa counter1))
  (setq counter1 (+ counter1 1))
  (if (> counter1 5) (prompt “B”) (prompt “A”))
)
```

What happens and why? → 0A1A2A3A4A5B6B7B8B9Bnil … cycle thru the routine to see. Let’s re-examine the TEXTREAD function and get it to report all of the lines of text in the file it opens:

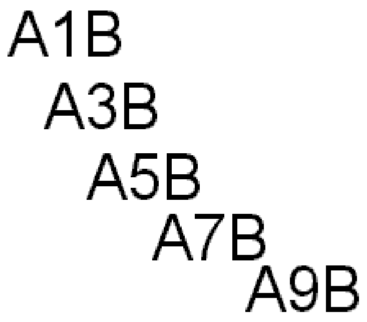
```
(defun C:TEXTREAD ()
  (setvar "CMDECHO" 0)
  (setq f (open "c:\\test.txt" "r"))
  (setq txt (read-line f))
  (while (/= nil txt) ; the end of a file is nothingness. Test for nothingness.
    (prompt txt)
    (setq txt (read-line f)) ; get the next line in the file. Is it nothingness?
  )
  (princ)
)
```

Working with keywords: we can force the user to only provide certain keywords as responses using the **(initget)** and **(getkword)** functions. The (initget) function establishes the values that are acceptible by the (getkword) function when the user is prompted. In the example below, only a value of A, B or C will be accepted.

```
(initget “A B C”)
(setq answer1 (getkword “\nEnter A, B, or C because nothing else will work: “))
```

The program below not only shows how you can use a (while) loop to repeat an action until the user right-clicks to exit the loop, but how to do some very fancy text labeling. Note use of new **(getreal)** and **(getint)** functions.

```
(defun C:ITEXT ()
   (setvar "cmdecho" 0)
   (setq ta (getreal "\nEnter text height: ")
         na (getint "\nEnter beginning number: ")
         nb (getint "\nEnter increment step: ")
         pref (getstring "\nEnter prefix: ")
         suff (getstring "\nEnter suffix: ")
   )
   (setq pa (getpoint "\nPick point: "))
   (while (/= nil pa) ; if the user right-clicks, that returns a nil value
      (setq tstr (strcat pref (itoa na) suff))
      (command "text" "j" "m" pa (rtos ta 2 2) "0" tstr)
      (setq na (+ na nb))
      (setq pa (getpoint "\nPick point: "))
   )
   (princ)
)
```



The **(cond)** function is like a multiple **(if)** function: if a truth condition is met, the adjacent statement is performed. A default condition is set up in this example: if the user uses the ENTER key, this acts like a nil response which is acceptible and a nil response sets the answer1 variable to C which is a legal response. AutoCAD's default choice styling with <> is used in the prompt too.

```
(defun C:CHOICE ()
  (setvar "cmdecho" 0) (initget "A B C")
  (setq answer1 (getkword "\nEnter your choice A/B/<C>: "))
  (if (= nil answer1) (setq answer1 "C"))
  (cond
    ((= answer1 "A") (prompt "\nYou picked A."))
    ((= answer1 "B") (prompt "\nYou picked B."))
    ((= answer1 "C") (prompt "\nYou Picked C."))
  )
  (princ)
)
```

Working with the entity database: now the real magic begins. The **(entsel)** function allows the user to select any entity and report back (as a list) the entity's internal name and the point by which it was selected. It can include a prompt like (getpoint) or (getkword).

```
Command: (entsel "\nPick something: ")
Select Object: (<Entity name: 7ef58e98> (25.233 20.7702 0.0))
```

The **(entget)** function takes the entity name you provide it with and reports back the entire database record for that entity. **(entget (car (entsel)))** is a lazy way to report any record.

```
(defun C:GETRECORD ()
  (setq thing1 (entsel "\nPick an entity to display the entire record for: "))
  (setq name1 (car thing1))
  (setq record1 (entget name1))
)
```

When you run the above program, you get the Select Objects: prompt. Pick an entity (try a LINE for starters) and you will get an "entity list" (a list within a list) of group codes and their values. We've seen these "dotted pairs" before with the (cons) function. Each pair has an integer representing a group code and the group code's value. Some group codes are explained below. They look an awful lot like **DXF** codes don't they? <wink>

```
(
  (-1 . <Entity name: 7ef58e98>)  ← the same hexadecimal name (entsel) provides
  (0 . "LINE")  ← entity type
  (330 . <Entity name: 7ef58cf8>) ← the same hexadecimal name (entsel) provides
  (5 . "8B") ← handle (LIST command reports this)
  (100 . "AcDbEntity")
  (67 . 0) ← color? linetype?
  (410 . "Model") ← tilemode?
  (8 . "0") ← layer
  (100 . "AcDbLine")
  (10 18.2842 14.873 0.0) ← starting point
  (11 26.3622 21.8109 0.0)  ← ending point
  (210 0.0 0.0 1.0) ← UCS
)
```

Using a "group code/value" format, the **(cons)** function can create dotted pairs:

`(cons 8 "WALL") → (8 . "WALL")`

The **(assoc)** function evaluates an item of an entity list **as a list** based on a "dotted pair" group code you provides it. If the above entity list for a LINE entity is refered to, then:

`(assoc 8 record1) → (8 . "0") this evaluates the layer information of the list`

`(cdr (assoc 8 ea)) → "0"  note: (cdr) on dotted pairs` **`does not evaluate a list!!`**

MAGIC: Write a program to export selected TEXT entities to an ASCII file!!

```
(defun C:EXPTEXT ()
  (setvar "CMDECHO" 0)
  (setq fname1 (getstring "\nEnter file name: "))
  (setq ent1 (entsel "\nPick a text entity to export: "))
  (while ent1
    (setq entlist1 (entget (car ent1)))
    (setq text1 (cdr (assoc 1 entlist1)))
    (setq file1 (open fname1 "a")) ; note append mode
    (write-line text1 file1)
    (close file1)
    (setq ent1 (entsel "\nPick next text entity or right-click to exit: "))
  )
  (princ)
)
```

Working with tables: AutoCAD has tables of LAYERs, BLOCKs, and other categories of information which can be listed and edited. The **(tblnext)** function, once provided a table category such as "LAYER" or "BLOCK", evaluates subsequent table entries as it is used. If a **T** parameter is provided, (tblnext) evaluates the first entry in the table. Each successive use of (tblnext) report the next entry in the table. Each entry is like an entity list (a list of lists) including all the properties of that category. When the last entry has been reported, the next use of (tblnext) evaluates to nil.

```
(tblnext “LAYER” T) → ((0 . "LAYER") (2 . "0") (70 . 0) (62 . 7) (6 .
"Continuous")) using the T parameter reports the first layer in the table
(tblnext “LAYER”) → ((0 . "LAYER") (2 . "1") (70 . 0) (62 . 1) (6 . "Continuous"))
(tblnext “LAYER”) → nil  so in this case the drawing only has 2 layers
(tblnext “BLOCK” T) → nil so in this case the drawing has no blocks defined yet
(cdr (assoc 2 (tblnext “LAYER” T))) → “0”  evaluates the name of the first LAYER
(cdr (assoc 2 (tblnext “LAYER”))) → “1” evaluates the name of the next layer
(cdr (assoc 2 (tblnext “LAYER”))) → nil  no more layers found in the table
```

<u>MAGIC: Write a program to report all LAYER and BLOCK names in a drawing!!!</u>

```
(defun C:REPORT ()
  (setvar "CMDECHO" 0)
  (prompt "\nLayers:")
  (setq lname1 (cdr (assoc 2 (tblnext "LAYER" T))))
  (while lname1
    (prompt (strcat "\n" lname1))
    (setq lname1 (cdr (assoc 2 (tblnext "LAYER"))))
  )
  (prompt "\nBlocks:")
  (setq bname1 (cdr (assoc 2 (tblnext "BLOCK" T))))
  (while bname1
    (prompt (strcat "\n" bname1))
    (setq bname1 (cdr (assoc 2 (tblnext "BLOCK"))))
  )
  (princ)
)
```

The **(subst)** function evaluates (but does not set) what an entity list would look like if you substituted one dotted pair for another in a specified entity list. **(entmod)** forces that substitution to actually take place. Note the new use of **(getvar)** function to get the value of a system variable.

<u>MAGIC: Write a program to change a picked entity's layer to the current layer:</u>

```
(defun C:CLAY ()
   (setvar "CMDECHO" 0)
   (setq layer1 (getvar "CLAYER"))
   (setq ename1 (car (entsel "\nPick entity: ")))
   (while ename1
   (setq elist1 (entget ename1))
     (setq elist1 (subst (cons 8 layer1) (assoc 8 elist1) elist1))
```

```
      (entmod elist1)
      (setq ename1 (car (entsel "\nPick next entity: ")))
    )
    (princ)
)
```

As we have seen earlier, the **(if)** function allows you to execute **one** statement, whether its test condition is true or nil. If you need multiple statements executed, the **(progn)** function "glues" together multiple statements so they get treated as one statement:

```
(if T (prompt “True”) (prompt “False”)) → True
(if T (progn (prompt “This is ”)(prompt “true.”)) (prompt “False”)) → This is true.
(if nil (progn (prompt “This is ”)(prompt “true.”)) (prompt “False”)) → False.
```

Working with selection sets: the **(ssget)** function lets the user select many objects at once (using standard Select Objects: modes such as fence, crossing, window, etc) and save them to a selection set variable. The **(ssname)** function acts on a selection set like the **(nth)** function acts on a list once you specify the item integer value to treat (item 0 is the first item in both cases). The **(sslength)** function acts on a selection set the way (length) acts on a list. The **(entdel)** function deletes an entity if the entity's name is specified.

MAGIC: Write a program to delete all selected entities on a specified layer!!

```
(defun C:LAYDEL ()
  (setvar "CMDECHO" 0)
  (setq ent1 (car (entsel "\nPick entity to specify deletion layer: ")))
  (if ent1
    (progn
      (setq layer1 (cdr (assoc 8 (entget ent1))))
      (prompt "\nPick entities to evaluate: ")
      (setq ss1 (ssget))
      (if ss1
        (progn
          (setq count1 0 total1 (sslength ss1))
          (while (< count1 total1)
            (setq ent1 (ssname ss1 count1) layer2 (cdr (assoc 8 (entget ent1))))
            (if (= layer1 layer2) (entdel ent1))
            (setq count1 (+ count1 1)) ; increment the counter
          ) ; end while
        ) ; end second progn
      ) ; end second if
    ) ; end first progn
  ) ; end first if
  (princ)
)
```

The **(ssget)** function has several powerful parameters that build selection sets in different ways:

```
(setq point1 (list 1.0 1.0 0.0) point2 (list 2.0 2.0 0.0))
(setq ss1 (ssget “C” point1 point2)) → selects entities using Crossing mode
(setq ss1 (ssget “W” point1 point2)) → selects entities using Window mode
(setq ss1 (ssget “x”)) → selects everything in the database
```

If a “filter list” is provided to the (ssget) function when it uses the “x” mode, then (ssget) builds a selection set from the filter criteria. The filter list is just a list of dotted pairs. So, if you wanted to collect all entities which are LINEs and on layer “WALL”, your dotted pairs would be (0 . “LINE”) and (8 . “WALL”). These can be constructed with (cons 0 “LINE”) and (cons 8 “WALL”) statements. The filter list then would be built with (list (cons 0 “LINE”) (cons 8 “WALL”)) and this list would be passed to the (ssget “x” <filter list>) statement.

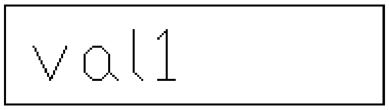
<u>MAGIC: Write a program to delete all entities on the layers of selected entities (Scrub WILHOME)!!!</u>

```
(defun C:LAYDEL ()
  (setvar "CMDECHO" 0)
  (setq ent1 (car (entsel "\nPick entity to establish deletion layer: ")))
  (while ent1
    (setq elist1 (entget ent1) layer1 (cdr (assoc 8 elist1)))
    (setq sset1 (ssget "x" (list (cons 8 layer1))))
    (command "erase" sset1 "")
    (setq ent1 (car (entsel "\nPick entity to establish next deletion layer: ")))
  )
  (princ)
)
```

AutoLISP doesn’t care how lines of code are organized spatially so long as they’re properly sequential. The program below actually works but is very hard to understand with all the code sandwiched together. Sometimes it is worthwhile to take a program like this and break each line down, indenting as you go, to understand what is really happening. You may not be able to paste this much sandwiched code to the Command: prompt either.

<u>What does this program actually do? Bust it out into separate lines to figure this out:</u>

```
(defun C:LWID () (setvar "cmdecho" 0)(setq sa (ssget "x" (list (cons 0 "LINE")(cons
8 "0"))))(if sa (progn (setq ca 0 ta (sslength sa)) (while (< ca ta)  (setq enta
(ssname sa ca) ea (entget enta)  pa (cdr (assoc 10 ea))) (command "pedit" (list
enta pa) "y" "w" "0.5" "x") (setq ca (+ ca 1)))))(princ))
```



Working with nested entities: if a BLOCK (0 . “INSERT”) has attributes (0 . “ATTRIB”), the attribute entities follow right behind the INSERT entity in order. The **(entnext)** function, once an entity name has been specified, finds the name of the next entity in the database. To verify this, use **RECTANG** to draw a rectangle and inside of it create an **ATTDEF** tag value VAL1. Create a **BLOCK** from these two entities called “THING” and **INSERT** the BLOCK.

```
(setq entname1 (car (entsel “\nPick a BLOCK: “)))
(setq elist1 (entget entname1))
(setq entname2 (entnext entname1))
(setq elist2 (entget entname2)) →
((-1 . <Entity name: 7efaff78>) (0 . "ATTRIB") (330 . <Entity name: 7efaff70>)
(5 . "A7") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "0") (100 .
"AcDbText") (10 25.7225 20.4924 0.0) (40 . 1.29853) (1 . "value1") (50 . 0.0)
(41 . 1.0) (51 . 0.0) (7 . "Standard") (71 . 0) (72 . 0) (11 0.0 0.0 0.0) (210
0.0 0.0 1.0) (100 . "AcDbAttribute") (2 . "VAL1") (70 . 0) (73 . 0) (74 . 0))
```

The **(ssadd)** function enables you to add entities to a new selection set by providing entity names.

<u>MAGIC: Create a selection set of BLOCKs by a specified attribute value!!</u> Note: insert the above block several times with values of 100 and 200 to distinguish them from each other to test:

```
(defun C:SSATT ()
  (setvar "cmdecho" 0)
  (setq text1 (getstring "\nEnter attribute value: "))
  (setq sset1 (ssget "x" (list (cons 0 "INSERT")(cons 2 "THING"))))
  (setq sset2 nil sset2 (ssadd)) ; nil clears out prior values if they exist
  (if sset1 (progn
    (setq count1 0 total1 (sslength sset1))
    (while (< count1 total1)
      (setq ent1 (ssname sset1 count1) ent2 (entnext ent1))
      (setq elist2 (entget ent2) text2 (cdr (assoc 1 elist2)))
      (if (= text2 text1) (ssadd ent1 sset2))
      (setq count1 (+ count1 1))
    )
  ))
  (command "move" sset2 "" "0,0" "0,0") ; this is a slick trick!
  (prompt "\nBlocks are waiting for you in the Previous selection set.")
  (princ)
)
```

The Thinking Process Behind the Development of a Typical AutoLISP Routine:

In past years, students of this class have requested a portion of it be dedicated to the code development process. It’s not enough to get the syntax under one’s belt. One also needs an idea how the code evolves from idea to application. Part of the reason this process has not been examined during the class is because admittedly a lot of it is intuitive and much of it requires a solid understanding of how AutoCAD works internally.

This year the tutorial instructors were given double the amount of pages they could include in their handouts, so I chose to dedicate a significant portion of this handout to the development of an application from inception to implementation. It is told in the manner of a CAD manager talking himself thru the development process from beginning to end. As with 95% of all AutoLISP applications and utilities developed, it begins with a need that is not met by AutoCAD software out of the box...

POLYTEXT: text along a polyline path

The Problem: our office needs a way to drape textual content along any given polyline. TEXT and MTEXT entities do not accomodate this need, and it is unlikely that AutoCAD will include it anytime soon as it has never been on the top 10 Wishlist items. Not long ago there was an Express Tool which enabled a special type of arc text entity to be draped along an arc or circle entity. However, our office has the latest release of AutoCAD installed and this arc text tool not only doesn’t install with it or seem to behave itself under it, but again it always fell short of what we really needed. Developing an AutoLISP routine to do this is warranted.

The Pre-Politics: if there was a way that I could track how much time it takes our users to simulate text draped along a polyline, I could feed that cost figure to the management to explain why we need to dedicate some funding to developing a utility that automates it. I need their financial justification for the development of this utility. But we simply don’t drape text along polylines because it would be incredibly time-consuming using standard TEXT and MTEXT entities and editing methods. Financial justification will come down to management declaring we need this ability badly enough to warrant its in-house development and me giving them a reasonably accurate estimate of the time it will take me to create a tool that does this. I must also be certain of a method for doing it which I can present to them. If they accept my idea, I’m clear to work on this project. I must be sure to establish who owns the code... most likely the company that pays me to develop it will own it, but at least I want credit for its development.

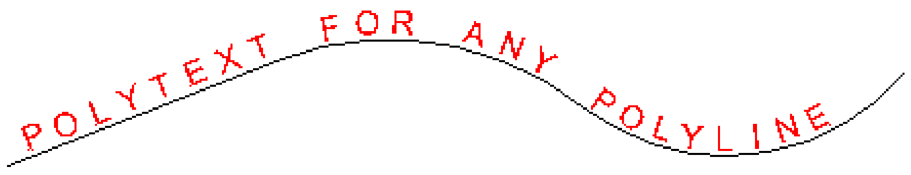
The Angle: POINT entities and BLOCKs can be draped along polylines using the MEASURE and DIVIDE commands. However, POINT entities are not aligned with the polylines like BLOCKs are. The idea behind my application is to have the user enter a string of text (including spaces and punctuation) which would be busted up into individual characters (letters, spaces and punctuation marks) and counted out. The character count would then be used to drape an equal number of aligned dummy BLOCKs along the desired pre-existing polyline. The total number of characters to drape along the polyline must physically fit along the polyline, so there would have to be some spacing method set up to control this, based on DIVIDE or MEASURE’s own internal workings. The dummy BLOCKs, once draped correctly along the polyline, could then be swapped out for the characters. To make the entire string of text ‘selectable’ if i need it deleted, I could create a special layer for these BLOCKs/characters to be inserted on. That way, I can also make the utility delete everything found in the drawing on that dedicated layer by examining the layer of any single selected character in the draped text.

User Input: I need to ask the user for the string of text, obviously. I also need the user to select a pre-existing Lightweight Polyline, specify a distance between the characters, and provide a name for the dedicated layer to be used for placing all these BLOCKs on (which can also be used for the dummy block). Text height also needs to be asked. This routine could later be modified to offset text to one side or the other of the selected polyline. That's for another day's wishlist item.

Some Initial Dangers: I could just develop this routine to use the current style, but I'd have to make some assumptions about that style when I use the (command) syntax since command prompt sequences change based on a Style's definition. The string of text has to physically fit along the line or else I need some way to ensure that the user is warned that it won't fit. If the text is too big, letters might overlap each other. If the text is too small, the text might look like points on top of the polyline.

The Code Breakdown: The entire program POLYTEXT including its co-utility DELPOLYTEXT is provided below. Remarks are included in the code throughout to explain the thinking behind each sequence of code lines.

```
(defun C:POLYTEXT ()
;don't report the utility functions as it runs - this is aesthetically pleasing
  (setvar "cmdecho" 0)
;get the text height as a real number
  (setq txt (getstring T "\nEnter the string to drape along the polyline: "))
  (setq pa (getpoint “\nPick 2 points to establish text height: “))
  (setq pb (getpoint pa) txth (distance pa pb))
;establish number of characters in string
  (setq txtlen (strlen txt))
;get the dedicated layer name, make it with color 1 and make it current
  (setq laynam (getstring "\nEnter dedicated layer name: "))
  (command "layer" "m" laynam "c" "1" laynam "")
;get the polyline entity list information
  (setq enta (car (entsel "\nPick the lightweight polyline: ")))
  (setq ea (entget enta))
;count how many items are in the polyline entity list and initialize a counter
  (setq ca 0 ta (length ea))
;cycle thru all items in the polyline list until you find the first 10 group
;which is the starting vertex of the polyline. create a lightweight polyline and
;(entget (car (entsel))) it to see a typical list for reference
  (setq test (car (nth ca ea)))
  (while (/= test 10)
    (setq ca (+ ca 1))
    (setq test (car (nth ca ea)))
  )
;create a 3D point from the first vertex of the polyline
;remember that a lightweight only has 2D coordinates -- add a 0.0 for z
  (setq xa (cadr (nth ca ea)))
  (setq ya (caddr (nth ca ea)))
  (setq pa (list xa ya 0.0))
;establish the character separation distance from the first vertex
  (setq pb (getpoint pa "\nPick character separation distance point: "))
  (setq da (distance pa pb))
;check to see if this will physically work.
;
;     (distance between characters) x (number of characters) = ?
;                         polyline length = ?
;
;if polyline length is less than (char)x(dist) figure, shut down application
;get polyline length by LISTing it and getting PERIMETER system variable
;use (graphscr) to return to graphics mode after LIST command used
  (setq test1 (* da txtlen))
  (command "list" enta "")(graphscr)
  (setq test2 (getvar "perimeter"))
  (if (< test1 test2) (progn
; create a dummy block with the same name as dedicated layer at first vertex
    (command "text" "j" "bc" pa txth "0" "I")
    (setq entb (entlast))
    (command "block" laynam pa entb "")
;remember last entity in database before draping new blocks so that you can
;go back to it later on and find insertion points and rotations of all entities
;that follow it (recently created block insertions)
    (setq entb (entlast))
;drape block along the polyline using MEASURE
    (command "measure" enta "b" laynam "y" da)
;drap each block for a character using intersion point and rotation info
    (setq ca 0)
    (while (< ca txtlen)
      (setq entb (entnext entb) eb (entget entb))
      (setq pa (cdr (assoc 10 eb)) ra (cdr (assoc 50 eb)))
      (command "text" "j" "bc" pa txth (angtos ra 2 2) (substr txt (+ ca 1) 1))
      (setq ca (+ ca 1))
    )
;delete all dummy blocks
    (setq sa (ssget "x" (list (cons 0 "INSERT")(cons 2 laynam))))
    (command "erase" sa "")
    ) ; finish text-fits condition
    (prompt "\nThis text won't fit using this spacing.") ; warn text doesn’t fit
  ) ; finish entire if statement
  (princ)
)
```



```
(defun C:DELPOLYTEXT ()
  (setvar "cmdecho" 0)
;get the layer name of the selected character
  (setq enta (car (entsel "\nPick polytext character to delete: ")))
  (setq ea (entget enta) laynam (cdr (assoc 8 ea)))
;erase all entities on that layer
  (setq sa (ssget "x" (list (cons 8 laynam))))
  (command "erase" sa "")
;purge the block
  (command "purge" "b" laynam "n")
;purge the layer after setting current layer to "0" incase
  (command "layer" "s" "0" "")
  (command "purge" "la" laynam "n")
  (princ)
)
```