Autodesk University 2007

LISPing on Purpose – Part I and Part II

Craig P. Black - Fox Valley Technical College

GD205-1L & GD211-1L Attend these classes and learn how to automate command tasks in AutoCAD using AutoLISP -- the non-programmer's language. We'll focus on functions and how they apply to the different types of data used in AutoCAD, including numbers, text strings, and points. We'll discuss the automation of drawing (Part I) and editing (Part II) in AutoCAD.

About the Speaker:

Craig both manages the Autodesk Authorized Training Center and teaches Mechanical Design at Fox Valley Technical College (FVTC) in Appleton, Wisconsin. He is a Certified Instructor in AutoCAD, and has previous certification in Mechanical Desktop (now AutoCAD Mechanical) and Architectural Desktop (now AutoCAD Architecture). He teaches at ATCs across the United States, and has been elected twice to the Autodesk Training Center Executive Committee, serving as chairperson in 2001. Craig is coauthor of the popular book "AutoCAD and its Applications – Advanced," and contributing author to "AutoCAD and its Applications – Basics," and Republic Research Training Center's "R14 AutoLISP" Autodesk Certified Courseware. Craig is no stranger to AU, having taught AutoCAD training classes at this event for several years.

Autodesk®


Autodesk
University
2007

Overview of What We Will Cover

AutoLISP is an easy-to-learn programming language for the non-programmer. It is a simple, text-based language that needs no compiler.

"Programming" is nothing more than simplifying a task or tasks by automating or reducing the steps needed to produce the desired outcome by intelligently using or reusing program supplied or user supplied data.

How's that for a mouthful of big, important sounding words? More simply, programming is intelligently obtaining and using data. AutoLISP is a very logical language, and if that fact is kept in mind, a very easy language to use. This class will show you how to recognize the types of data that you need, get that data, and then use that data.

Part 1 of this course will focus on the terminology used in AutoLISP. Once you have an understanding of those concepts, you will then learn about syntax, or the format the functions and your programs need to follow. Finally, you will create a simple program or two using math and numeric data types.

Part 2 will move into the gathering and use of other data types. You will learn how to get data from the user of your programs, and then manipulate and apply that data in various ways. You will learn how to add objects to your drawing as well as edit existing objects in your drawing.

Part 1 – Terminology and Syntax

List – anything contained within a set of parenthesis

Element – the individual items within a list

Function – AutoLISP's commands, typically the first element within a list

Argument – the elements that are "passed" to the function

Data Types – see the next 5 terms

Symbols or **Variables** – a "placeholder" that is storing data

Integers – whole numbers, no decimal point

Reals – floating point numbers, always has a decimal point

Strings – words, always contained in a set of quotation marks: "Like This"

Lists – how AutoLISP stores X, Y, and Z values: within a set of parenthesis (see the first term!)

Math Related Functions

The following simple functions help us to understand how the syntax and the terms listed previously work together. Simple, short functions can be typed in directly at the Command: prompt. This makes testing and troubleshooting (and initial learning!!) very convenient. Feel free to type some of what I am introducing, right as we go along…

(+ *num num* …)

The addition function (+) takes any number of numeric arguments and adds them together and returns the sum. The sum is returned as a real if at least one of the arguments is a real. The sum is returned as an integer if all of the arguments are integers.

(- *num num* …)

The subtraction function takes any number of numeric arguments. The first argument can be considered the base number. All additional arguments are successively subtracted from that base argument. The result is returned as a real if at least one of the arguments is a real. The result is returned as an integer if all of the arguments are integers.

(* *num num* …)

The multiply function takes any number of numeric arguments and multiplies them together and returns the product. The sum is returned as a real if at least one of the arguments is a real. The result is returned as an integer if all of the arguments are integers.

(/ num num …)

The divide function takes any number of numeric arguments. The first argument can be considered the base number. The base number is successively divided by each of the additional arguments. The result is returned as a real if at least one of the arguments is a real. The result is returned as an integer if all of the arguments are integers.

(rem num num …)

The remainder function takes any number of numeric arguments. The result is returned as an integer if all of the arguments are integers. The returned result will be the remainder of what is left after evenly dividing the base argument by the second argument. The first argument can be considered the base number. The base number is successively divided by each of the additional arguments. The result is returned as a real if at least one of the arguments is a real.

Storing Data

The (setq) function allows us to store a value, or data in a variable, or symbol.

(setq sym value)

The setq function requires an even number of arguments. The odd numbered arguments are the symbols and the even numbered arguments are the values to be stored in that symbol. For learning's and simplicity's sake, we will use (setq) with just two arguments; the symbol and the value it will hold. The symbol can be most any typed character, but not *just* a number. (A number already holds a value!) Some other characters are out-of-bounds also, such as +, -, *, /, (,)… these symbols already have meanings or value in AutoLISP. Examples of valid symbols:

A B PT PT1 PT2 MIDPT CNTRPT MIDPT CLR1 FIRST-NAME LAST-NAME DWGNUM

We can use the (setq) function to store some of the types of data we referred to earlier:

```
(setq A 1)
(setq B 2.5)
(setq C "AutoLISP")
```

The (setq) function returns the results of evaluating whatever arguments are passed to it.

Nesting Lists

The above sequence of code could be referred to as “hard coding”. The values that the symbols are holding onto are not allowed to vary. They are always going to be exactly as shown – the symbol is holding onto whatever value the programmer typed as the second argument. There is no allowance for input from the user.

AutoLISP has the functionality to allow functions to be nested within other functions. To start off, the functions that we will nest within the (setq) function will be functions that allow us to get certain data types from the users of our programs.

First, let’s look at a few data type gathering functions, then we will nest them within a (setq) to store their values.

(**getint** *str*)

Allows the user to input an integer value. Optionally allows a string argument to be used as a prompt. Returns the input value as an **integer**.

(**getreal** *str*)

Allows the user to input a real or floating point value. Optionally allows a string argument to be used as a prompt. Returns the input value as a **real** number.

(**getpoint** *pt str*)

Allows the user to input an point value. Optionally allows a string argument to be used as a prompt. Also, optionally allows a “starting” point argument. Returns the input point as a **list**.

(**getdist** *pt str*)

Allows the user to input an integer value. Optionally allows a string argument to be used as a prompt. Also, optionally allows a “starting” point argument. Returns the input value as a **real** number.

(**getstring** *flag str*)

Allows the user to input an integer value. Optionally allows a string argument to be used as a prompt. Also allows an optional argument, that if true allows spaces to be included in the input string. Returns the input string as a **string**.

Using the above functions, by themselves, will result in any data entered by the users being lost. The result will be returned to the Command: prompt, with no way to ever again retrieve that result. If the above functions are called in a format called nesting, within the (**setq**) function, the values will be stored:


```
(setq ROWS (getint "Enter number of rows: "))
(setq RAD1 (getreal "Enter circle radius: "))
(setq CPT (getpoint "Enter circle center point: "))
(setq RAD2 (getdist CPT "Enter circle radius: "))
(setq FRSTNM (getstring "Enter your first name: "))
(setq FULLNM (getstring T "Enter your full name: "))
```

Common Errors

Misspelled function:

```
; error: no function definition: SEQT
```

Missing parenthesis:

```
(_>
```

The above is one of the only really "cryptic" error messages. Most messages, such as the first one shown above, are rather self explanatory.

Using the Stored Data

The exclamation point will allow you to determine if and what value a variable may be holding. If the variable is not holding data, it will return "nil". Nil is the AutoLISP equivalent of "no value".

```
Command: (setq X 1.25)
1.25
Command: !X
1.25
Command: offset
Current settings: Erase source=No  Layer=Source  OFFSETGAPTYPE=0
Specify offset distance or [Through/Erase/Layer] <1.0000>: !X
1.25
Select object to offset or [Exit/Undo] <Exit>:  ...
```

Continuing our journey from AutoCAD to AutoLISP, the (command) function allows us to call AutoCAD commands and pass the stored data to the Command: prompt. For example, we can use some of the values we stored at the top of this page:

Command: **(command "circle" CPT RAD1)**

Command: **(command "circle" CPT RAD2)**

Put It All Together

Let's enter some lines of code at the command prompt:

```
(setq PT1 (getpoint "\nEnter first point: "))
(setq PT2 (getpoint "\nEnter next point: "))
(setq PT3 (getpoint "\nEnter next point: "))
(setq PT4 (getpoint "\nEnter next point: "))
(command "line" PT1 PT2 PT3 PT4 "c")
```

Above, we have the makings of a program! We just need to group all of these lines together as one "unit". The **(defun)** function does just that. Type the following lines in again as shown. After typing the first line, the Command: prompt will disappear and a prompt showing that there is a missing parenthesis will be shown. Just keep typing the remaining lines of code. The last line, the single ")", will satisfy the missing parenthesis error message, and the missing Command: prompt will return!

```
(defun c:BOX1 ()
(setq PT1 (getpoint "\nEnter first point: "))
(setq PT2 (getpoint "\nEnter next point: "))
(setq PT3 (getpoint "\nEnter next point: "))
(setq PT4 (getpoint "\nEnter next point: "))
(command "line" PT1 PT2 PT3 PT4 "c")
)
```

Now you have a program! Enter "Box1" at the command prompt again, and the program will run "by itself", as if the lines of code were typed separately as we did earlier!

Lastly, (well sort of "lastly"), we need to store the above lines of code in a text file. We will work on that, and much more, within the next sections…

Part 2 - The Rest of the Course

I call this the beginning of "Part 2", but in reality, I have no idea if we are at the end of the time allotment for Part 1, or at the end of our time allotment for both parts! Hopefully we are running close to on schedule.

The balance of this course will be hands on using the Visual LISP Editor within AutoCAD. We will progress through a series of functions that will enhance our knowledge of AutoLISP. We will use the HELP command within the Visual LISP editor to understand the new functions and their arguments. We will continue to work with our BOX*n* routine, building on it to make it more useful, and exploring numerous new functions along the way – in some cases we will discover numerous way to get the same result.

We now have a working program, but it will only continue to work while this drawing is open. AutoLISP programs are only active in the drawings in which they are written, or loaded. Once a new drawing is started, or the AutoCAD program is closed and then reopened – we lose our loaded lisp routines.

Storing our lisp programs as individual files is the answer to that problem. Then we won't have to keep rewriting them each time we want to use them – we just need to *load* the lisp routine.

AutoCAD supplies a built-in editor made for editing AutoLISP files, the Visual LISP Editor. To open the editor select the Tools pulldown menu, then select AutoLISP, the select Visual LISP Editor.

Pick the "New file" button, and type the following lines of code into the editor.

VERSION 1

Introduces the topic of indenting.

```
(defun c:BOX1 ()
  (setq PT1 (getpoint "\nEnter first point: "))
  (setq PT2 (getpoint "\nEnter next point: "))
  (setq PT3 (getpoint "\nEnter next point: "))
  (setq PT4 (getpoint "\nEnter next point: "))
  (command "line" PT1 PT2 PT3 PT4 "c")
)
```

As you can see, the Visual LISP Editor is indeed, visual! You will learn to appreciate the benefits of using the editor more and more as you use it.

When the lines of code are typed, use the "Save file" button to do so to the appropriate folder, calling the file "BOX*n*.lsp". Then use the "Load the active edit window" button to load the code into the current drawing. Finally, use the "Activate AutoCAD" button to go to the AutoCAD window. Type BOX1 at the Command: prompt and your program should run as it did earlier.

Another way to load a lisp program file and make it available for use is to use the (load) function directly at the Command: prompt, as such: Command: (load "box1")

The ".lsp" is not required, though you may add it if it makes you feel more comfortable. A path is not required if AutoCAD has the file's location within its search path. If the file is not in AutoCAD's search path, the syntax for inclusing the path and the file name is:

Command: (load "c:\\some-where-else\\box1")

For the remainder of the course we will successively build new versions of the program. Each version will incorporate a new concept or two and the functions required to implement those concepts. We will use the built-in HELP system to learn about the new functions. The "new" additions will typically be in bold, but occasionally we will be deleting lines of code, so it is important to read each new version in its entirety, and compare your version to the version contained in this handout. Also, notice that the name of the BOX*n* command is incremented with each version, doing this – and saving the lisp file with respectively incremented file names will allow us to keep a running record of our learning, and allow to run earlier versions of the program for comparison's sake.

VERSION 2

New functions: *(getvar) (prompt) (initget) (setvar) (princ)*

```
(defun c:BOX2 ()
  (setq OCMDECHO (getvar "cmdecho"))
  (setvar "cmdecho" 0)
  (setq OCLR (getvar "cecolor"))
  (prompt "\nYou are about to draw a four sided polygon...")
  (setq CLR (getstring "\nEnter RED, YELLOW, or BLUE: "))
  (command "color" CLR)
  (initget 1)
  (setq PT1 (getpoint "\nEnter first point: "))
  (initget 1)
  (setq PT2 (getpoint "\nEnter next point: "))
  (initget 1)
  (setq PT3 (getpoint "\nEnter next point: "))
  (initget 1)
  (setq PT4 (getpoint "\nEnter next point: "))
  (command "line" PT1 PT2 PT3 PT4 "c")
  (setvar "cecolor" OCLR)
  (setvar "cmdecho" OCMDECHO)
  (princ)
)
```

VERSION 3

New functions: *(getkword)*

```
(defun c:BOX3 ()
  (setq OCMDECHO (getvar “cmdecho”))
  (setvar “cmdecho” 0)
  (setq OCLR (getvar "cecolor"))
  (prompt "\nYou are about to draw a four sided polygon...")
  (initget 1 "Red Yellow Blue")
  (setq CLR (getkword "\nEnter color [Red, Yellow, or Blue]: "))
  (command "color" CLR)
  (initget 1)
  (setq PT1 (getpoint "\nEnter first point: "))
  (initget 1)
  (setq PT2 (getpoint "\nEnter next point: "))
  (initget 1)
  (setq PT3 (getpoint "\nEnter next point: "))
  (initget 1)
  (setq PT4 (getpoint "\nEnter next point: "))
  (command "line" PT1 PT2 PT3 PT4 "c")
  (setvar "cecolor" OCLR)
  (setvar "cmdecho" OCMDECHO)
  (princ)
)
```

VERSION 4

New functions: *(getcorner) (car) (cadr) (list) (distance)*

```
(defun c:BOX4 ()
  (setq OCMDECHO (getvar “cmdecho”))
  (setvar “cmdecho” 0)
  (setq OCLR (getvar "cecolor"))
  (prompt "\nYou are about to draw a four sided polygon...")
  (initget 1 "Red Yellow Blue")
  (setq CLR (getkword "\nEnter Red, Yellow, or Blue: "))
  (command "color" CLR)
  (initget 1)
  (setq PT1 (getpoint "\nEnter first point: "))
  (initget 1)
  (setq PT2 (getcorner PT1 "\nEnter opposite corner: "))
  (setq X1 (car PT1))
  (setq Y1 (cadr PT1))
  (setq X2 (car PT2))
  (setq Y2 (cadr PT2))
  (setq PT3 (list X1 Y2))
  (setq PT4 (list X2 Y1))
  (command "line" PT1 PT4 PT2 PT3 "c")
  (setq D (* 2 (+ (distance PT1 PT4) (distance PT4 PT2))))
  (princ D)
  (setvar "cecolor" OCLR)
  (setvar "cmdecho" OCMDECHO)
  (princ)
)
```

IMPORTANT!!!!

If you haven’t been doing so as we have gone along, **SAVE** this version as BOX4.lsp – we will be coming back to it, after we examine the following version.

Autodesk University 2007

VERSION 5

(An alternate way to accomplish something similar to what we've been doing…)

New functions: *(getdist) (getangle) (polar)*

```
(defun c:BOX5 ()
  (setq OCMDECHO (getvar "cmdecho"))
  (setvar "cmdecho" 0)
  (setq OCLR (getvar "cecolor"))
  (prompt "\nYou are about to draw a four sided polygon...")
  (initget 1 "Red Yellow Blue")
  (setq CLR (getkword "\nEnter Red, Yellow, or Blue: "))
  (command "color" CLR)
  (initget 1)
  (setq PT1 (getpoint "\nEnter lower left corner: "))
  (initget 1)
  (setq LEN (getdist PT1 "\nEnter box length: "))
  (initget 1)
  (setq HGT (getdist PT1 "\nEnter box height: "))
  (initget 1)
  (setq ANG (getangle PT1 "\nEnter vertical angle: "))
  (setq PT2 (polar PT1 0 LEN))
  (setq PT3 (polar PT2 ANG HGT))
  (setq PT4 (polar PT1 ANG HGT))
  (command "line" PT1 PT2 PT3 PT4 "c")
  (setq D (* 2 (+ (distance PT1 PT2) (distance PT1 PT4))))
  (princ D)
  (setvar "cecolor" OCLR)
  (princ)
)
```

IMPORTANT!!!!

If you haven't been doing so as we have gone along, **SAVE** this version as BOX5.lsp – we are done with this version, and will be going back to Version 4, after we examine the following sidebar.

Autodesk University 2007

SIDEBAR!! NOT A NEW VERSION

We will just examine this coded version of the box routine – we will not type this one out…

New concepts: adding remarks and white space

```
;                BOX5.lsp
;             by Craig P. Black
;         last modified 10/24/2007
;
(defun c:BOX5 ()

; The next few lines allow the user to choose a color
  (setq OCMDECHO (getvar “cmdecho”))
  (setvar “cmdecho” 0)
  (setq OCLR (getvar "cecolor"))
  (prompt "\nYou are about to draw a four sided polygon...")
  (initget 1 "Red Yellow Blue")
  (setq CLR (getkword "\nEnter Red, Yellow, or Blue: "))
  (command "color" CLR)

; gathering the data needed to draw the box
  (initget 1)
  (setq PT1 (getpoint "\nEnter lower left corner: "))
  (initget 1)
  (setq LEN (getdist PT1 "\nEnter box length: "))
  (initget 1)
  (setq HGT (getdist PT1 "\nEnter box height: "))
  (initget 1)
  (setq ANG (getangle PT1 "\nEnter vertical angle: "))
```

```
;creating the other points to draw the box
  (setq PT2 (polar PT1 0 LEN))
  (setq PT3 (polar PT2 ANG HGT)) ;ANG is in radians
  (setq PT4 (polar PT1 ANG HGT))

;finally, drawing the box
  (command "line" PT1 PT2 PT3 PT4 "c")

;displaying perimeter
  (setq D (* 2 (+ (distance PT1 PT2) (distance PT1 PT4))))
  (princ D)

;cleaning up the screen
  (setvar "cecolor" OCLR)
  (princ)
)
```

IMPORTANT!!!! - Be sure to open **BOX4**.lsp before applying the new concepts of Version 6

VERSION 6

New functions: *(foreach)*

```
(defun c:BOX6 ()
  ...
   (setq X1 (car PT1))
   (setq Y1 (cadr PT1))
   (setq X2 (car PT2))
   (setq Y2 (cadr PT2))
   (setq PT3 (list X1 Y2))
   (setq PT4 (list X2 Y1))
   (command "line" PT1 PT4 PT2 PT3 "c")
   (foreach XXX (list PT1 PT2 PT3 PT4)
     (command "circle" XXX 1)
     (command "circle" XXX 0.5)
   )
   (setq D (* 2 (+ (distance PT1 PT4) (distance PT4 PT2))))
   (princ D)
   (setvar "cecolor" OCLR)
   (setvar "cmdecho" OCMDECHO)
   (princ)
)
```

VERSION 7

New functions: *(repeat)*

```
(defun c:BOX7 ()
  (setq OCMDECHO (getvar “cmdecho”))
  (setvar “cmdecho” 0)
  (setq OCLR (getvar "cecolor"))
  (prompt "\nYou are about to draw a four sided polygon...")
  (initget 1 "Red Yellow Blue")
  (setq CLR (getkword "\nEnter Red, Yellow, or Blue: "))
  (command "color" CLR)
  (initget 7)
  (setq TIMES (getint "\nEnter number of boxes to draw: "))
  (repeat TIMES
    (initget 1)
    (setq PT1 (getpoint "\nEnter first point: "))
    (initget 1)
    (setq PT2 (getcorner PT1 "\nEnter opposite corner: ")
; notice the nesting of the (setq)s!!!!! (no need to type this line, just notice it!!!)
          X1  (car PT1)
          Y1  (cadr PT1)
          X2  (car PT2)
          Y2  (cadr PT2)
          PT3 (list X1 Y2)
          PT4 (list X2 Y1)
    )
    (command "line" PT1 PT4 PT2 PT3 "c")
  ) ; Don’t miss this closing parenthesis for the (repeat) function!!
  (setq D (* 2 (+ (distance PT1 PT4) (distance PT4 PT2))))
  (princ D)
  (setvar "cecolor" OCLR)
  (setvar "cmdecho" OCMDECHO)
  (princ)
)
```

VERSION 8

New functions: *(if) (while)*

```
...
  (command "color" CLR)
  (setq TIMES (getint "\nEnter number of boxes or <Enter> to loop: "))
  (if TIMES
    (repeat TIMES
      (initget 1)
      (setq PT1 (getpoint "\nEnter first point: "))
      (initget 1)
      (setq PT2 (getcorner PT1 "\nEnter opposite corner: "))
      (setq X1 (car PT1))
      (setq Y1 (cadr PT1))
      (setq X2 (car PT2))
      (setq Y2 (cadr PT2))
      (setq PT3 (list X1 Y2))
      (setq PT4 (list X2 Y1))
      (command "line" PT1 PT4 PT2 PT3 "c")
    )
    (while (setq PT1 (getpoint "\nEnter first point: "))
      (initget 1)
      (setq PT2 (getcorner PT1 "\nEnter opposite corner: "))
      (setq X1 (car PT1))
      (setq Y1 (cadr PT1))
      (setq X2 (car PT2))
      (setq Y2 (cadr PT2))
      (setq PT3 (list X1 Y2))
      (setq PT4 (list X2 Y1))
      (command "line" PT1 PT4 PT2 PT3 "c")
    ) ; Don't miss this closing parenthesis for the (while) function!!
  ) ; Don't miss this closing parenthesis for the (if) function!!
  (setvar "cecolor" OCLR)
...
```