AUTODESK UNIVERSITY

IT18197-L

Managing What Your LISP Routines Manage

Craig P. Black
Fox Valley Technical College

Learning Objectives

- Write shortcut routines that put things back to how they were
- Use lists to store system variables that the program will modify
- Create your own custom error routines
- Understand more of the subtle nuances of how LISP works

Description

Writing AutoLISP code is one thing, troubleshooting it, and restoring the system “back to normal” when crashes occur while troubleshooting can be a headache. Take this class and take away your headaches. Learn how to quickly get things back to normal. Learn how to use lists to your advantage – I mean, LISP does stand for List Processing! Learn how to create your own error routine to set things back the way they were before your fancy new program crashed and burned. Crashing and burning is part of the process of getting a new time saving routine off the ground – this class will make those crashes less painful. This session features AutoCAD. AIA Approved

Your AU Expert

Craig Black has been working with Autodesk, Inc., products since 1985. He has been teaching in the Mechanical Design Technologies programs of Fox Valley Technical College (FVTC) since 1990. At FVTC, he has developed 2 programs within his department, the CAD Management Advanced Certificate and the Mechanical CAD Drafting Technical Diploma. He has been presenting at Autodesk University for many years, offering classes on diverse topics ranging from programming and customization to solid modeling and CAD standards. He is a co-author of the widely acclaimed AutoCAD and Its Applications textbook, published by Goodheart-Wilcox, and he has been contracted to do training all over the United States. When not teaching, he works as a programmer/consultant for a large furniture manufacturer based in Wisconsin. Time and weather permitting, Black likes to squeeze in a few rounds of golf each week, as well as cheer for his beloved 13-time NFL World Champion Green Bay Packers.

Learning Objective: Shortcut to Fix Stuff Even Before Its Broke

As programmers and developers we write programs to make ourselves, our departments, and our companies more productive. Often, while developing a program, our program changes the user's working environment (on purpose) – but then the program crashes, and we have to put things back "manually" before fixing the errors.

Our first step, even before troubleshooting our programs, is to write a "safety net" routine to change the environment back to the way it was before the program crashed. Here is an example of one that I created. You can delete or add lines as you deem necessary.

FavSet.LSP

```
(defun c:FAVSET ()
  (setvar "CMDECHO" 0)              ; turns off (command) prompting
  (setvar "CLAYER" "OBJECT")        ; sets current layer to OBJECT
  (setvar "OSMODE" 39)              ; sets object snaps to END, MID, CEN, INT
  (setvar "POLARANG" (/ pi 12))     ; sets polar tracking angle to 15 degrees
  (setvar "PICKBOX" 4)              ; sets pickbox size to 4 pixels
  (setvar "AUNITS" 0)               ; sets angular unit type to decimal degrees
  (setvar "AUPREC" 2)               ; sets angular unit precision to 2 places
  (setvar "LUNITS" 2)               ; sets linear unit type to decimal
  (setvar "LUPREC" 4)               ; sets linear unit precision to 4 places
  (setvar "FILLETRAD" 0)            ; sets fillet radius to 0
  (setvar "LTSCALE" 0.5)            ; sets linetype scaling to .5
  (setvar "AUTOSNAP" 63)            ; turns on all AutoSnap features
  (setvar "TEXTSTYLE" "Regular")    ; sets the current textstyle to Regular
  (command "dimstyle" "r" "DEC-2PL") ; sets the current dimstyle to DEC-2PL
  (princ)
)
```

FIGURE 1. FAVSET.LSP

Now, whenever a program crashes during testing, typing FAVSET at the command line will return the environment back to "normal".

Rather than writing a program and forcing it to crash after adjusting some of the above settings, we will test it "manually" with the following exercise.

Exercise No.1 – FAVSET.lsp

Watch the instructor for a quick-tip on copying and pasting!

1. Open the Test Settings.dwg file
2. Open the Visual LISP Editor
3. Start a new program file
4. Type the above lines of code shown in Figure 1 (in the interest of saving time ignore the gray comments after the semi-colons)
5. Save the file as FAVSET.lsp
6. Press the “Load active edit window” button
7. Switch to AutoCAD
8. Change the current layer to something other than Object
9. Change the running object snaps to QUAdrant, PERpendicular, and NODe
10. Type PICKBOX and set it to 25
11. Turn off Polar Tracking and Object Snap Tracking
12. Scan the screen and mentally note what you have just changed
13. Type FAVSET at the Command: prompt
14. Scan the screen and note that things have been set back to “normal”

Learning Objective: Use Lists to Store System Variables that the Program will Modify

When a program is going to modify system variables so that it can do the things it needs to do it is always a good practice to first “get” the current settings of those variables and store them so that they can be used to set them back to normal just before completion of the program.

The Old Way

We have all probably seen or written programs that are written as follows:

```
(defun C:LISTSTORE (/             |CMDECHO|    |CLAYER|     |OSMODE|
                    |POLARANG|    |PICKBOX|    |AUNITS|     |AUPREC|
                    |LUNITS|      |LUPREC|     |FILLETRAD|  |LTSCALE|
                    |AUTOSNAP|    |TEXTSTYLE|  |DIMSTYLE|   PT1
                    PT2           PT3
                   )

  (setq |CMDECHO| (getvar "CMDECHO"))
  (setq |CLAYER| (getvar "CLAYER"))
  (setq |OSMODE| (getvar "OSMODE"))
  (setq |POLARANG| (getvar "POLARANG"))
  (setq |PICKBOX| (getvar "PICKBOX"))
  (setq |AUNITS| (getvar "AUNITS"))
  (setq |AUPREC| (getvar "AUPREC"))
  (setq |LUNITS| (getvar "LUNITS"))
  (setq |LUPREC| (getvar "LUPREC"))
  (setq |FILLETRAD| (getvar "FILLETRAD"))
  (setq |LTSCALE| (getvar "LTSCALE"))
  (setq |AUTOSNAP| (getvar "AUTOSNAP"))
  (setq |TEXTSTYLE| (getvar "TEXTSTYLE"))
  (setq |DIMSTYLE| (getvar "DIMSTYLE"))

  (setvar "CLAYER" "HIDDEN")
  (prompt (strcat "\nCurrent layer was changed to "
                  (getvar "CLAYER")
                  "."
          )
  )
  (setvar "OSMODE" 0)
  (prompt "\nRunning object snaps were turned off.")
  (setvar "AUTOSNAP" 0)
  (prompt "\nPolar and object snap tracking were turned off.")
  (setq PT1 (getpoint "\nPick a first point: "))
  (setq PT2 (getpoint "\nPick a second point: "))
  (setq PT3 (getpoint "\nPick a third point: "))
  (command "line" PT1 PT2 PT3 "Close")

  (setvar "CMDECHO" |CMDECHO|)
  (setvar "CLAYER" |CLAYER|)
  (setvar "OSMODE" |OSMODE|)
  (setvar "POLARANG" |POLARANG|)
  (setvar "PICKBOX" |PICKBOX|)
  (setvar "AUNITS" |AUNITS|)
  (setvar "AUPREC" |AUPREC|)
  (setvar "LUNITS" |LUNITS|)
  (setvar "LUPREC" |LUPREC|)
  (setvar "FILLETRAD" |FILLETRAD|)
  (setvar "LTSCALE" |LTSCALE|)
  (setvar "AUTOSNAP" |AUTOSNAP|)
  (setvar "TEXTSTYLE" |TEXTSTYLE|)
  (command "dimstyle" "r" |DIMSTYLE|)
  (prompt
    (strcat "\nCurrent layer is now " (getvar "CLAYER") ".")
  )
  (prompt "\nRunning object snaps are now back to normal.")
  (prompt
    "\nPolar and object snap tracking are turned back on."
  )
  (princ)
)
```

FIGURE 2. OLD STYLE OF STORING AND RESETTING VARIABLES

The New Way

Storing all of the variable settings in a single list, and then "reading" that list one item at a time is a much cleaner way to handle storing and resetting variables.

The (mapcar) function can be used to do this. This function performs a function on each item in the list, in order. It takes the form of:

(mapcar *function list1 … listn*)

For example:

(setq FIRSTNAMES (list "Jimi " "Eric " "Duane "))

(mapcar 'prompt FIRSTNAMES)
Returns: Jimi Eric Duane (nil nil nil)

(setq LASTNAMES (list "Hendrix" "Clapton" "Allman"))

(mapcar 'strcat FIRSTNAMES LASTNAMES)
Returns: ("Jimi Hendrix" "Eric Clapton" "Duane Allman")

Note the new code below and the application of the (mapcar) function:

```
(defun C:LISTSTORE (/ VARLIST VARVALS PT1 PT2 PT3)
  (setq VARLIST (list "CMDECHO"     "CLAYER"      "OSMODE"      "PICKBOX"
                      "AUNITS"      "AUPREC"      "LUNITS"      "LUPREC"
                      "FILLETRAD"   "LTSCALE"     "POLARANG"    "AUTOSNAP"
                      "TEXTSTYLE"
                     )
  )
  (setq VARVALS (mapcar 'getvar VARLIST))

  (setvar "CLAYER" "HIDDEN")
  (prompt
    (strcat "\nCurrent layer was changed to "
            (getvar "CLAYER")
            "."
    )
  )
  (setvar "OSMODE" 0)
  (prompt "\nRunning object snaps were turned off.")
  (setvar "AUTOSNAP" 0)
  (prompt "\nPolar and object snap tracking were turned off.")
  (setq PT1 (getpoint "\nPick a first point: "))
  (setq PT2 (getpoint "\nPick a second point: "))
  (setq PT3 (getpoint "\nPick a third point: "))
  (command "line" PT1 PT2 PT3 "Close")
  (mapcar 'setvar VARLIST VARVALS)
  (prompt
    (strcat "\nCurrent layer is now " (getvar "CLAYER") ".")
  )
  (prompt "\nRunning object snaps are now back to normal.")
  (prompt
    "\nPolar and object snap tracking are turned back on."
  )
  (princ)
)
```

FIGURE 3. NEW STYLE OF STORING AND RESETTING VARIABLES

Exercise No.2 – LISTSTORE.lsp

1. If not already open, open the Test Settings.dwg file
2. If not already open, open the Visual LISP Editor
3. Start a new program file
4. Type the above lines of code shown in Figure 3 (in the interest of saving time just put “CLAYER”, “OSMODE”, and “AUTOSNAP” in the variables list to be saved)
5. Save the file as LISTSTORE.lsp
6. Press the “Load active edit window” button
7. Switch to AutoCAD
8. Set any layer other than “Hidden” as the current layer
9. Turn polar tracking and object snap tracking on
10. Type LISTSTORE into the Command: prompt
11. As it is asking you to “Pick the first point” note that the hidden layer is current and polar tracking and object snap tracking is now off.
12. Finish following the prompts of the program
13. Note that the previous layer is now current again, and polar tracking and object snap tracking are now on again.

Learning Objectives: Create Your Own Custom Error Routines

Even before we discuss creating our own error routine, we will examine how to keep errors from happening in the first place.

When you need to get data of a certain type from a user, use the appropriate function:

(getint)	Returns an integer. Returns nil if user hits <enter>.
(getreal)	Returns a real. Returns nil if user hits <enter>.
(getdist)	Returns a real. Returns nil if user hits <enter>.
(getstring)	Returns a string. Returns an empty string if user hits <enter>.
(getpoint)	Returns a point list. Returns nil if user hits <enter>.
(entsel)	Returns a list of an ename and a point list. (Beyond the scope of this course)
(ssget)	Returns a pickset. (Beyond the scope of this course)

Notice that (getint), (getreal), and (getpoint) can all return nil if the user hits <enter>

Keeping the User from Hitting the <enter> Key

The (getxxx) functions actually have their own little error trap routine built into them. If you are using the (getint) function and the user enters something other than an integer, a message appears explaining the error and gives the user a chance to try again.

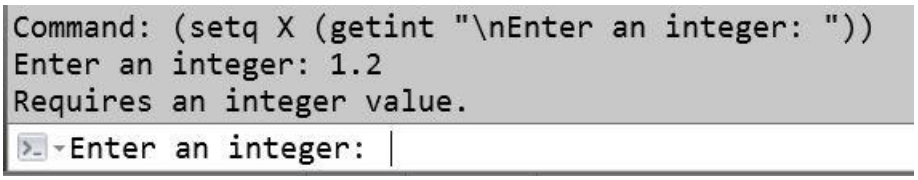

FIGURE 4. THE (GETXXX) FUNCTIONS HAVE A BUILT IN ERROR TRAP

While the provided error traps are nice, they are not fool proof. The user can still mess up the program by hitting the <enter> key, or as “real” programmers like to say, “providing null input”.

The (initget) function can be used to keep this from happening and provide another level of error trapping. A number of bit coded arguments can be passed to (initget) to provide different functionality. The form (initget) takes on is as follows:

(initget x)

Where x is one of the following:

1 disallows null input (no hitting the <enter> key!)
2 disallows input of zero
3 disallows input of a negative value

(There are other bit codes that can be used, but they are beyond the scoped of this course)

The (initget) function should almost always be used before calling the (getxxx) functions.

The (initget) function applies to the next (getxxx) function only.

The (initget 1) function does not work with the (getstring) function, as the (getstring) function does not return nil when <enter> is typed.

Examine the following code:

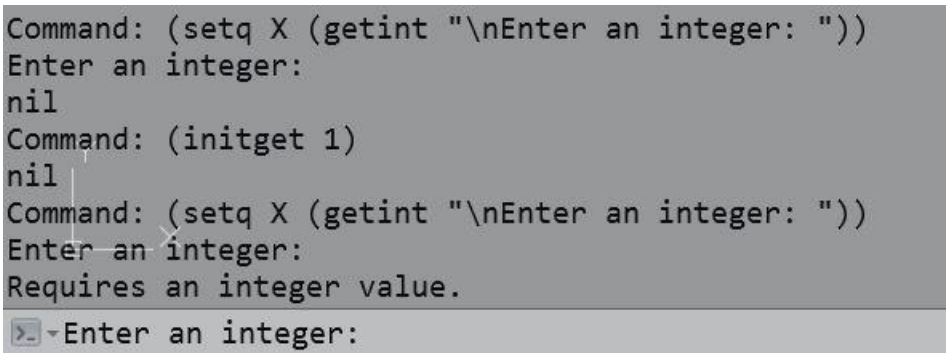

FIGURE 5. THE (INITGET) FUNCTION PROVIDES ANOTHER LEVEL OF ERROR TRAPPING

Follow the instructor as he demonstrates using and not using (initget) with a few (getxxx) functions.

Exercise No.3 – Add (initget) Expressions

```
(setvar "OSMODE" 0)
(prompt "\nRunning object snaps were turned off.")
(setvar "AUTOSNAP" 0)
(prompt "\nPolar and object snap tracking were turned off.")
(initget 1)
(setq PT1 (getpoint "\nPick a first point: "))
(initget 1)
(setq PT2 (getpoint "\nPick a second point: "))
(initget 1)
(setq PT3 (getpoint "\nPick a third point: "))
(command "line" PT1 PT2 PT3 "Close")
```

FIGURE 6. ADDING THE (INITGET) FUNCTION TO THE PROGRAM

1. Activate the Visual LISP editor
2. If not already open, open the LISTSTORE.lsp file
3. Add (initget 1) on the line previous to each of the (getpoint) expressions as shown in Figure 6.
4. Save the file
5. Press the “Load active edit window” button
6. Switch to AutoCAD
7. Run the command – trying hitting the <enter> key when prompted to pick a point.

AutoLISP’s *error* function

There is a built in “error messenger” in AutoLISP. It basically tells you what went wrong when something goes wrong. It can be redefined to handle additional tasks as well.

Examine the sequence on the following page:

```
(setq X 1)
1

(prompt X)
; error: bad argument type: stringp 1

(defun NEWERROR (msg) (prompt "\nSomething went wrong..."))
NEWERROR

(setq OLDERROR *error*)
#<SUBR @00000000337fe8e0 *ERROR*>

(setq *error* NEWERROR)
#<SUBR @000000003b1c9390 NEWERROR>

(prompt X)
Something went wrong...

(setq *error* OLDERROR)
#<SUBR @00000000337fe8e0 *ERROR*>

(prompt X)
; error: bad argument type: stringp 1
```

Making a More Meaningful Custom Error Routine

The built in error message provided by AutoLISP is nice – but it really doesn't do anything else besides tell you what went wrong. Any changes your program may have made to the environment are left "as is". You are left to having to execute your FAVSET command that we wrote earlier to put things back to how they were before your program crashed.

We can make it so the error routine not only tells us what went wrong, but also puts things back to how they were before the program crashed.

An overview of what the new code does:

At the beginning, a NEWERROR function is created that will reset the variables (per our previously written code), let the user know that the variables have been reset, and set the error routine back to the built in error routine.

The main program then stores the old error routine and sets the error routine to the newly defined version. At the end of the program, just before completion, the old error routine is restored.

Examine the code on the following page:

```
(defun NEWERROR (MSG)
  (mapcar 'setvar VARLIST VARVALS)
  (prompt "\nVariables reset to previous values...")
  (setq *ERROR* OLDERROR)
  (princ)
)

(defun C:LISTSTORE (/ VARLIST VARVALS PT1 PT2 PT3)
  (setq OLDERROR *ERROR*)
  (setq *ERROR* NEWERROR)
  (setq VARLIST (list "CMDECHO"     "CLAYER"      "OSMODE"      "PICKBOX"
                      "AUNITS"      "AUPREC"      "LUNITS"      "LUPREC"
                      "FILLETRAD"   "LTSCALE"     "POLARANG"    "AUTOSNAP"
                      "TEXTSTYLE"
                    )
  )
  (setq VARVALS (mapcar 'getvar VARLIST))

  (setvar "CLAYER" "HIDDEN")
  (prompt
    (strcat "\nCurrent layer was changed to "
            (getvar "CLAYER")
            "."
    )
  )
  (setvar "OSMODE" 0)
  (prompt "\nRunning object snaps were turned off.")
  (setvar "AUTOSNAP" 0)
  (prompt "\nPolar and object snap tracking were turned off.")
  (initget 1)
  (setq PT1 (getpoint "\nPick a first point: "))
  (initget 1)
  (setq PT2 (getpoint "\nPick a second point: "))
  (initget 1)
  (setq PT3 (getpoint "\nPick a third point: "))
  (command "line" PT1 PT2 PT3 "Close")

  (mapcar 'setvar VARLIST VARVALS)
  (prompt
    (strcat "\nCurrent layer is now " (getvar "CLAYER") ".")
  )
  (prompt "\nRunning object snaps are now back to normal.")
  (prompt
    "\nPolar and object snap tracking are turned back on."
  )
  (setq *ERROR* OLDERROR)
  (princ)
)
```

FIGURE 7. A MORE USEFUL CUSTOM ERROR ROUTINE

Exercise No.4 – Adding your own Custom Error Routine

1. Activate the Visual LISP editor
2. If not already open, open the LISTSTORE.lsp file
3. Add the code to define the NEWEROR routine as shown in Figure 7
4. Add the two lines after the (defun c:LISTSTORE… line to redefine and store the error routine
5. Add the line near the end of the program, on the line just before the (princ) to reset the error routine
6. Save the file
7. Press the “Load active edit window” button
8. Switch to AutoCAD
9. To be on the safe side, run the FAVSET command to set everything back to “normal”.
10. Run the LISTSTORE command
11. Before picking the first point, note the layer has changed, and that polar tracking, object snap tracking, and running object snaps have been turned off.
12. Hit the <esc> key when prompted to pick a point. (This will crash the program, and would normally leave our variables all messed up – but they have been set back to “normal”!)

Learning Objective: Understand More of the Subtle Nuance of How AutoLISP Works

AutoLISP is a very logical language. We use functions to obtain results. We have to pass data to functions. The data needs to be passed in a logical order. We get data from users, and that data must be obtained logically. Functions then return data in a very logical format.

Troubleshooting your LISP routines is almost always related to data types. Functions can only take arguments of a certain data type. Conveniently, there are only a few common data types:

- Integers – whole numbers, no decimal point, negative numbers preceded with a - sign
- Reals – includes a decimal point, decimal place must be "covered" (0.25), negative numbers preceded with -
- Strings – text values, always enclosed in quotation marks
- Lists – multiple (or individual) data types enclosed in parenthesis
- Enames – AutoCAD "objects"
- Picksets – a group of AutoCAD "objects"

Again, functions typically require arguments of specific data types, and they return values of specific data types.

Watch the instructor demo what is meant by various returned error messages, including:

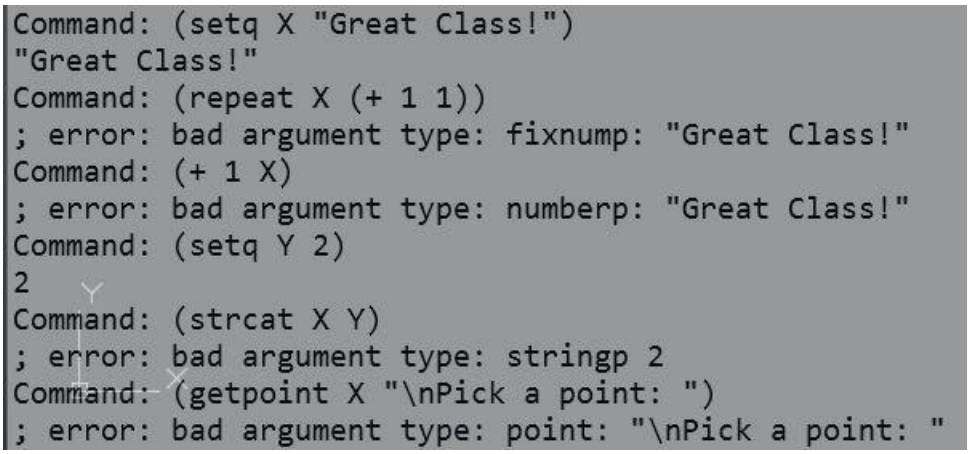

FIGURE 7. PREDICATE TESTING ERROR MESSAGES