Mastering the Visual LISP Integrated Development Environment

R. Robert Bell – Sparling

SD7297 How do you create and edit your AutoLISP programming language software code? Are you using a text editor such as Notepad? If so, you're missing out on all the features that the Visual LISP Integrated Development Environment (VLIDE) can provide you. The VLIDE is not only a way to write your code; it is also a way to debug it. In this session you will see why you should use the VLIDE to develop your custom add-ins, and you will discover how easy it is to write, format, and debug your code.

Learning Objectives

At the end of this class, you will be able to:

- See the benefits that the VLIDE offers over a basic text editor
- Learn how to format the source code the way you like
- Discover how to debug code using the tools in the VLIDE
- Learn how to manage and secure source code

About the Speaker

R. Robert Bell is the design technology manager for Sparling, a consulting firm that specializes in electrical engineering and technology and is headquartered in Washington state's Seattle area. He provides strategic direction, technical oversight, and high-level support for Sparling's enterprise design and production technology systems. He has been instrumental in positioning Sparling as an industry and client leader in the use of technology in virtual building and design. Robert has been in the mechanical, electrical, and plumbing (MEP) industry for over 25 years, and he has used every release of Revit MEP software (including the initial beta of Revit Systems software). Robert is a frequent speaker at the Revit Technology Conference and at Autodesk University, and he has written many articles regarding Revit MEP software and AutoCAD software. He is currently the president and chairman of the AUGI board.
rbell@sparling.com

Introduction

The Visual LISP Integrated Development Environment (VLIDE) offers many features that make it a compelling choice over a non-integrated text editor. Notepad can be used to edit LISP code but lacks an understanding of how to format the code. There are other text editors that do offer some more advanced formatting of LISP code, such as Notepad++ or TextPad. But these advanced external text editors do not offer the debugging or application creation tools that the integrated editor offers.

This class explores the features of the VLIDE and will give you plenty of reasons to ditch external editors in favor of the VLIDE.

The editor itself can be found on the Ribbon > Manage tab > Applications panel (Figure 1). It can also be launched by using the VLIDE command.

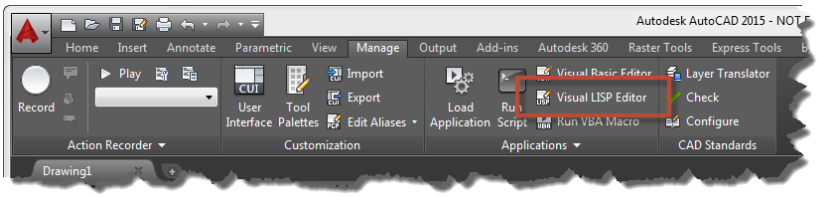

Figure 1 The Editor on the Ribbon

Benefits of the VLIDE

There are several feature of the editor that help you while writing your code. These include:

- Remembers your open files
- Color-coded text
- Find the matching parenthesis
- Console
- Inspect
- Symbol Service
- Apropos

Remembers Your Open Files

At the very least, the fact that the editor remembers the files you had open when you closed the editor and reopens them when the editor restarts is worth using the VLIDE.

Color-coded Text

Come on… how cool is that?!

Find the matching parenthesis

You can double-click on a parenthesis and the editor will locate the matching parenthesis, if there is one, and select all the intervening code.

The Console

The console allows you to perform expressions directly within the editor and see the results of those expressions. You are also able to examine the contents of variables from the console. If the console window is closed it can be opened by using the tool shown in Figure 2.

Figure 2 Console Tool

Performing Expressions

Simply type in the desired expression at the console prompt and the results of the expression will be echoed to the console (Figure 3). When the expression sets a variable, that variable will be available in the current drawing for future use while the drawing is open. This is effectively the same as typing the expression at the AutoCAD command prompt.

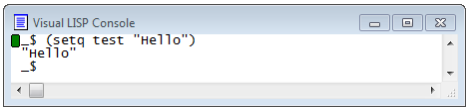

Figure 3 Performing an Expression in the Console

Examine Variables

Variables can be examined at the console by simply typing the variable name (Figure 4). Note that, unlike the AutoCAD command prompt, you do not add a leading exclamation mark (!) to the variable name. The contents of the variable will be echoed to the console.

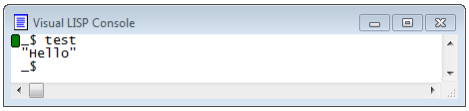

Figure 4 Examining a Variable in the Console

The Tab Key

The <Tab> key can be used in the console to cycle backwards thru the previous items entered in the console. <Shift>+<Tab> can be used to cycle forwards thru the previous items.

Inspect

Inspect (Figure 5) allows you to examine variables or the results of expressions from a dialog box rather than at the console. This is most often used while debugging code.

Figure 5 Inspect Tool

One of the great features of the Inspect tool is the ability to drill down into the exposed data. For example, Figure 6 displays a circle object where the Center property was double-clicked on, exposing the variant which was double-clicked on to see the value of the SafeArray.

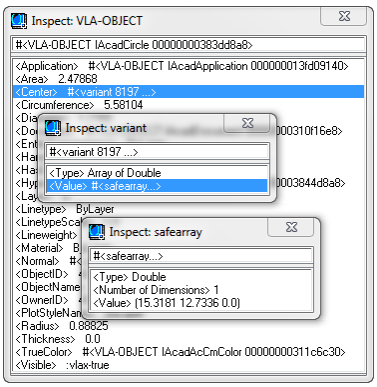

Figure 6 Drilling Into the Data with Inspect

Symbol[1] Service

This feature not only allows you see the value of a variable but to edit that value also (Figure 8). Editing the value can be done in the middle of a debugging session. You can add a variable shown in the window to the Watch window to assist with debugging.

Figure 7 Symbol Service Tool

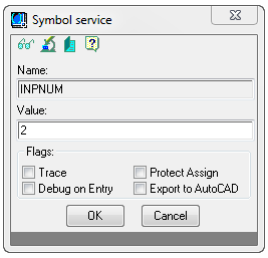

Figure 8 Symbol Service Dialog Box

If you have selected a symbol that points to a subroutine you can select the Show Definition button to see the code of the subroutine if it is loaded in the editor. This is useful when you are debugging large applications.

[1] Symbols are often thought of as variables but in LISP they can also be subroutines.

Apropos

Apropos allows you to locate symbols and quickly get to the help files on the function in the case of the built-in functions. You can filter the results using various filters and options (Figure 10).

Figure 9 Apropos Tool

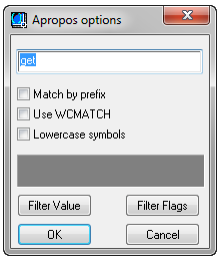

Figure 10 Apropos Options

You will need to use the filters and options at times due to the number of results (Figure 11) that you may get.

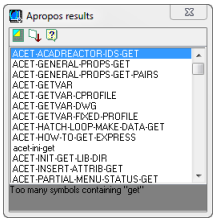

Figure 11 Too Many Results

Match by Prefix

Select this option when you know the function you are searching for starts with a specific term.

Use WCMATCH

Select this option and use the normal *wcmatch* wildcards to filter the results.

Help

The Help button on the Apropos Results dialog box will display the help text for the selected function when it is a built-in function.

Copy to Trace/Log

Even though the Apropos Results dialog may have too many results to display, you can use the Copy to Trace/Log button to see the entire list of functions. For example, search for “vla-“ using the Match by prefix option. You will still get to many results to display in the dialog box. Use the Copy to trace/Log button to get the entire list of functions that start with “vla-“.

Formatting Your Source Code

New code, and even your existing code, can be formatted according to a series of rules that determine how the code will appear. There are also formats for the parentheses, comments, and printing.

The Four Styles

There are two overall styles, single-line (Plane only) and multiple-line (Wide, Narrow, Column). The style applied to the code depends on a variety of factors.

Plane (Single-line)

The Plane style is when all the arguments are placed on the same line a separated by a single space. This style is used when:

- The position of last character of the expression does not exceed the Right Text Margin value.
- The expression's printing length[2] is less than the Approximate Line Length value.
- There are no embedded comments with the newline character.

Wide (Multiple-line)

This style places the first argument on the same line with the function but puts all subsequent arguments on new lines with the indentation the same as the first argument. This style in used when:

- The Plane style cannot be applied.
- The first item is a symbol and its length is less than the Maximum Wide Style Car Length setting.

Narrow (Multiple-line)

The Narrow style is applied for *progn* expressions and when the Plane and Wide styles cannot be applied. The first argument is place on a new line below the function at the normal indentation and all subsequent arguments line up on new lines below the first argument.

Column (Multiple-line)

This final style is seen with quoted lists and *cond* expressions. It can also be used when there are many arguments for a specific function.

Base Options

There are several options in the base Format Options dialog that are particularly useful. The tricky part is to balance the settings for margins, indenting, and other lengths with both the screen and print.

[2] The printing length is determined by the position of the last character minus the position of the starting indentation.

Experiment with the first four values until you get line formatting as you desire. Figure 1Figure 12 shows the values I'm using in AutoCAD.

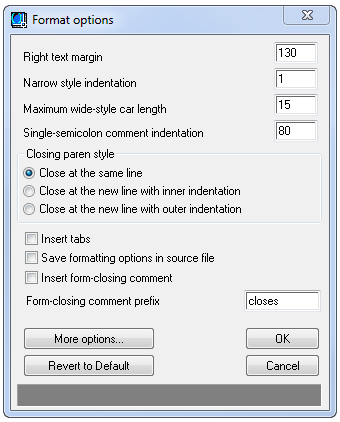

Figure 12 Base Format Options

Closing Parenthesis Styles

Many folks like to have the closing parenthesis on a line to itself and in line with the starting parenthesis. To achieve that look use the "Close at the new line with outer indentation" option (see Figure 13). However, if you want to use a format that is more compact, try the "Close at the same line" option (see Figure 14).

AttGet.lsp

```
;|
AttGet.lsp
|;

(defun i:AttGet (ent attTag / dxfVal                          ; entity type
                 done)                                        ; flag to exit loop
 (while (not done)                                            ; loop until done
  (cond ((and (= "ATTRIB" (setq dxfVal (cdr (assoc 0 ent))))  ; if sub-entity is an attdef
              (= (cdr (assoc 2 ent)) attTag)                  ; and matching attribute
         )
         (setq done T)
        )                                                     ; then exit loop
        ((= "SEQEND" dxfVal)                                  ; if at end of block's entities
         (setq done T                                         ; then exit loop
               ent nil                                        ; clear entity data
         )
        )
  )
  (if (not done)                                              ; if not done
   (setq ent (entget (entnext (cdr (assoc -1 ent)))))         ; then get next entity
  )
 )
 ent                                                          ; return entity data
)
```

Figure 13 Parentheses on a New Line

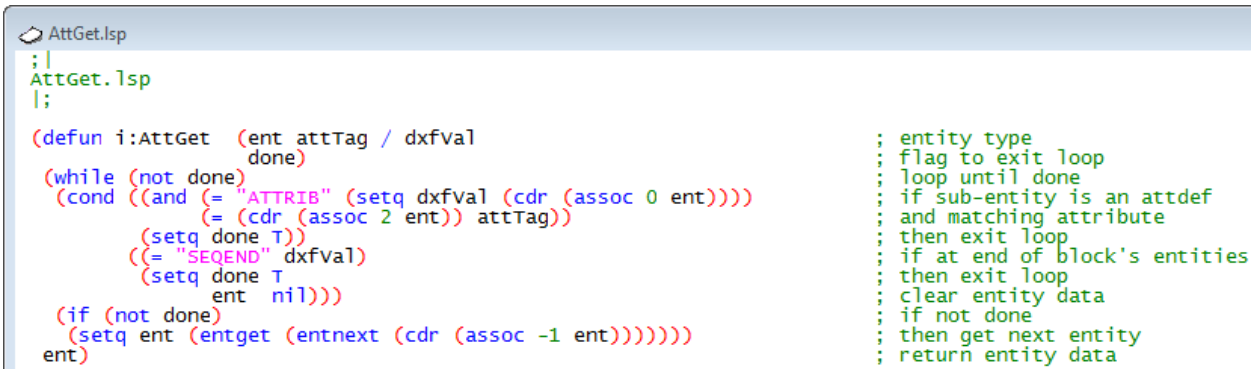

Figure 14 Parentheses on the Same Line

Insert Tabs

When this is selected, tab characters are added for indentation instead of spaces. This option is not useful if you plan on opening code in a text editor other than the VLIDE because the tabs will expand to different settings, messing up your code format.

Save Formatting Options in File

Formatting settings can be saved in the source code if you need to have different formats for specific source code files. However, it is usually best to set the formatting to something consistent and not save the formatting in individual source files.

Closing Comments

Closing comments can be automatically added by selecting that option. The interesting thing is that when you format the code, the new comment will recognize the matching parenthesis' symbol and add that symbol to the comment.

The bad thing about this feature? Editing your code and reformatting does not fix the closing comments. In other words, the feature is not useful.

Additional Options

There are additional options available when you select the More Options… button.

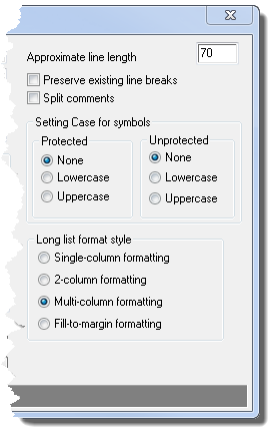

Figure 15 More Options for Formatting

Preserve Existing Line Breaks

If you have carefully formatted your source files prior to loading them in the VLIDE, you may want to select the option to preserve the existing line breaks. However, if you want to take full advantage of the formatting in the VLIDE it is usually best to turn off the option to give the editor full control over the format.

Split Comments

When this option is selected any comments that extend past the right hand margin will be split into multiple single-line comments.

Symbol Case

Given the prevalent developer practice of using camel-casing, e.g. myVariable, the option to automatically correct the case of protected or unprotected symbols to either lowercase or uppercase is of dubious value.

Long Lists

Long lists can be formatted in four different styles according to your preference. Multi-column or fill-to-margin are probably the two most useful styles if you are interested in compact code.

Page Setup Options for Printing

If you plan on printing your source code, you should take the time to set the printing options. The single most useful option is to decide what font type and size to use to get your code printed so that it is similar to the onscreen format.

Note that the margins need a leading zero for values less than 1.

Display Colors

You can change the syntax colors by using the Windows Attributes, Configure Current… option. In particular you may want to change the three comment items to have transparent white backgrounds with green text, to match what most other integrated development environments use for comments.

The top row of colors is the foreground color and the bottom row of colors is the background color.

You can see the effects of your color selections in the “Colored text” sample near the top of the dialog box (see Figure 16).

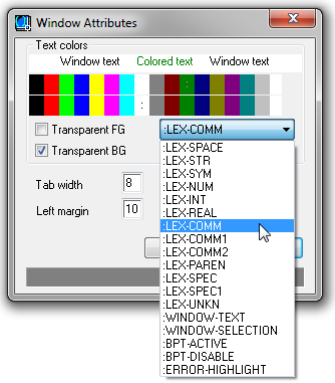

Figure 16 Editing the Color Format for Comments

Commenting Options

There are four different ways in which you can add comments, all of which are useful. Actually, that is a lie, since there is technically a fifth style when you use the option for closing comments.

```
;|

Comment.lsp
v1.0
Return the active document's name.
Used to demonstrate the 5 commenting styles.

|;

(defun C:Test  ()
;;; returns a string
 (vla-Get-Name                          ; get name
  (vla-Get-ActiveDocument
   ;; ThisDrawing
   (vlax-Get-Acad-Object)))) ;_ closes defun
```

Figure 17 All Five Commenting Styles

Single Semicolon (;)

This is the most common style of comment and places the comment to the right of the code at the single-semicolon comment indentation value.

Current Column (;;)

This style can be useful for commenting code for a specific sub-section of the code. These comments will line up with the code indentation.

Heading (;;;)

Sometimes you want a comment on the left side of the screen. The triple semicolons comment style is exactly what you need for that. However, use this style for only one or two lines of

comments. If you want multiple lines of comments aligned to the left, the next style does a better job.

In-line Comments (;| … |;)
This comment style convert the enclosed text into a comment. This works even across multiple lines of text. This makes it the perfect style to use for source code heading comments.

Closing Comments (;_ …)
This style is used when you have the option selected to add closing comments. It looks like a single-semicolon comment except that an underscore (_) is added automatically and the comment is placed one space away from the closing parenthesis.

Debugging Your Code

While the editor assists with writing and formatting the code, it really shines as a debugging tool. It is difficult to remember what it was like to manually debug AutoLISP in the days before the VLIDE was introduced.

There are two tools that really help with debugging:

- Watch
- Breakpoints/Stepping

The other two tools, Animate and Trace, are useful in only rare cases.

Watch

This tool allows you to watch the values of symbols or expressions in real-time as you step thru your code in a debugging session. This is so useful! There is even an option to add *LAST-VALUE* to the watch list, which will display the results of the last expression regardless of if the symbol or expression was added.

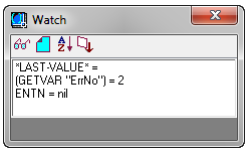

Figure 18 Watch

You can also select symbols or expressions in your source code and add them to the watch list using the context-sensitive menu (see Figure 19).

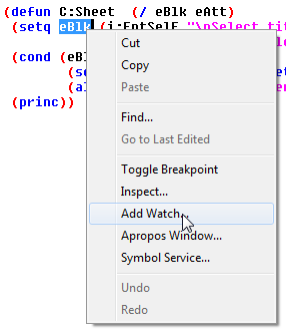

Figure 19 Add to Watch

Breakpoints/Stepping

When you discover that your code has an issue, setting breakpoints and stepping thru the code, in conjunction with the Watch window is an invaluable debugging technique.

The general procedure is to set one or more breakpoints in your code, the first breakpoint just before the code where you suspect the bug to be located, execute the code to trigger the breakpoint, and then begin stepping thru the code while watching the values of the symbols and expressions.

Breakpoints

You can easily set breakpoints by positioning the cursor where you want the break to occur and pressing the <F9> key. Once a break is triggered you then need to use one of the following options to step thru the code.

Step Into

This option steps you into the next expression. Pressing <F8> does the same thing. This will be the option that is used most often, unless you know the next expression is good.

Step Over

When you know the next expression is good, or you don't want to step into a subroutine, you can use this option to step over the expression.

Step Out

This option skips stepping thru the rest of the expressions in the current function.

Continue

This resumes normal operation of the code.

Reset

This options stops all evaluation of the code and clears all errors.

Animate

This tool steps thru the code automatically while highlighting what code is executing. This is rarely of great use unless you have no idea where an issue may be located.

Trace

Trace (Figure 20) is often used to see what is going on with loops. This is a debugging tool.

Figure 20 Trace Tool

Use the *trace* function to turn on tracing for a subroutine or function (Figure 21), run the code, then use the tool to display the Trace Stack dialog and select Copy to Trace/Log button (Figure 22) to examine the results of the trace (Figure 23).

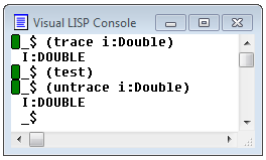

Figure 21 Turning Tracing On and Off

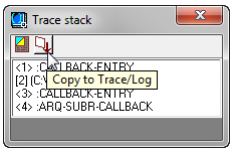

Figure 22 Trace Stack Dialog

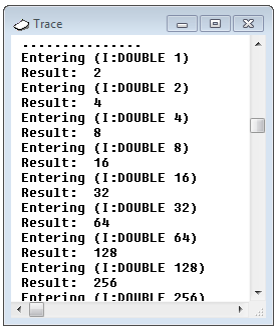

Figure 23 Trace Results

Use the *untrace* function to turn off tracing on the subroutine or function (Figure 21).

Managing and Securing Your Code

A great feature of the VLIDE is that you can create projects of multiple source code files. You are also able to use these projects to create applications by compiling the code into a single VLX file.

You can also protect your individual files by compiling them into FAS files. These individual FAS files can be loaded just like LSP files but your source code is not easily exposed. This is a good way to protect your company's developments from being taken and modified by others. (There

are approaches you can take with your code to make it more difficult to be executed away from your office, particularly when the code is protected.)

Projects

You can create a new project and then add the desired source files to it. After the project is created you can open that project and then open the individual source code files from a convenient dialog listing the files instead of browsing for the files.

However, the dialog to load a project is not quite as convenient as the common file open dialog.

Projects also have properties associated with build applications.

Application Wizard

The VLIDE File menu has a Make Application item that includes a New Application Wizard item. This makes it easy to create an application using an existing project.

Conclusion

This handout has introduced features of the VLIDE that you will find useful while writing, debugging, and maintaining your code. The VLIDE offers so much more than any external text editor due to its debugging capabilities and the ability to protect your code.

This handout did not cover all the VLIDE features but it has certainly given you information on the features that you will use most often.