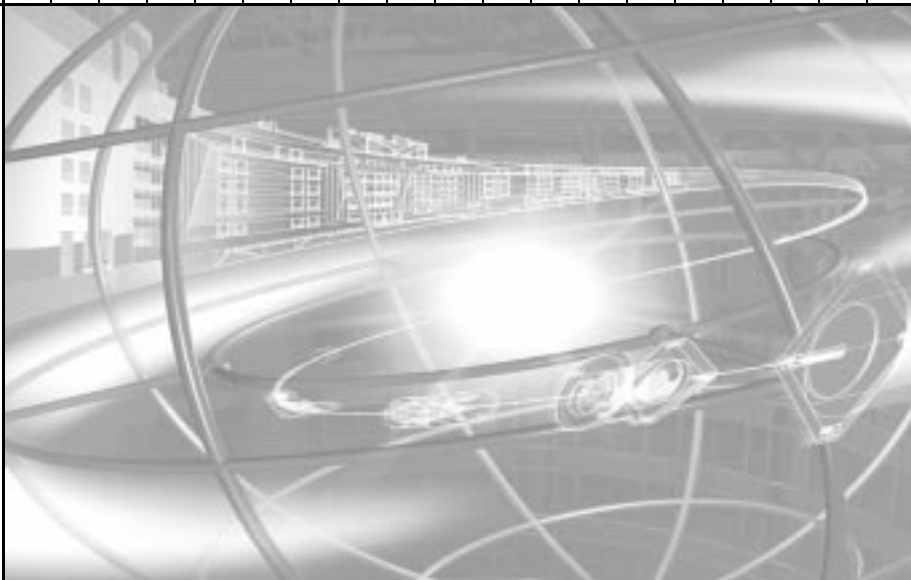


AutoCAD[®] 2000



VISUAL LISP[™] TUTORIAL

Copyright © 1999 Autodesk, Inc.

All Rights Reserved

AUTODESK, INC. MAKES NO WARRANTY, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, REGARDING THESE MATERIALS AND MAKES SUCH MATERIALS AVAILABLE SOLELY ON AN "AS-IS" BASIS.

IN NO EVENT SHALL AUTODESK, INC. BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH OR ARISING OUT OF PURCHASE OR USE OF THESE MATERIALS. THE SOLE AND EXCLUSIVE LIABILITY TO AUTODESK, INC., REGARDLESS OF THE FORM OF ACTION, SHALL NOT EXCEED THE PURCHASE PRICE OF THE MATERIALS DESCRIBED HEREIN.

Autodesk, Inc. reserves the right to revise and improve its products as it sees fit. This publication describes the state of this product at the time of its publication, and may not reflect the product at all times in the future.

Autodesk Trademarks

The following are registered trademarks of Autodesk, Inc., in the USA and/or other countries: 3D Plan, 3D Props, 3D Studio, 3D Studio MAX, 3D Studio VIZ, 3D Surfer, ADE, ADI, Advanced Modeling Extension, AEC Authority (logo), AEC-X, AME, Animator Pro, Animator Studio, ATC, AUGI, AutoCAD, AutoCAD Data Extension, AutoCAD Development System, AutoCAD LT, AutoCAD Map, Autodesk, Autodesk Animator, Autodesk (logo), Autodesk MapGuide, Autodesk University, Autodesk View, Autodesk WalkThrough, Autodesk World, AutoLISP, AutoShade, AutoSketch, AutoSolid, AutoSurf, AutoVision, Biped, bringing information down to earth, CAD Overlay, Character Studio, Design Companion, Drafrix, Education by Design, Generic, Generic 3D Drafting, Generic CADD, Generic Software, Geodysey, Heidi, HOOPS, Hyperwire, InsideTrack, Kinetix, MaterialSpec, Mechanical Desktop, Multimedia Explorer, NAAUG, Office Series, Opus, PeopleTracker, Physique, Planix, Rastation, Softdesk, Softdesk (logo), Solution 3000, Tech Talk, Texture Universe, The AEC Authority, The Auto Architect, TinkerTech, WHIP!, WHIP! (logo), Woodbourne, WorkCenter, and World-Creating Toolkit.

The following are trademarks of Autodesk, Inc., in the USA and/or other countries: 3D on the PC, ACAD, ActiveShapes, Actrix, Advanced User Interface, AEC Office, AME Link, Animation Partner, Animation Player, Animation Pro Player, A Studio in Every Computer, ATLAST, Auto-Architect, AutoCAD Architectural Desktop, AutoCAD Architectural Desktop Learning Assistance, AutoCAD DesignCenter, Learning Assistance, AutoCAD LT Learning Assistance, AutoCAD Simulator, AutoCAD SQL Extension, AutoCAD SQL Interface, AutoCDM, Autodesk Animator Clips, Autodesk Animator Theatre, Autodesk Device Interface, Autodesk PhotoEDIT, Autodesk Software Developer's Kit, Autodesk View DwgX, AutoEDM, AutoFlix, AutoLathe, AutoSnap, AutoTrack, Built with ObjectARX (logo), ClearScale, Concept Studio, Content Explorer, cornerStone Toolkit, Dancing Baby (image), Design Your World, Design Your World (logo), Designer's Toolkit, DWG Linking, DWG Unplugged, DXF, Exegis, FLI, FLIC, GDX Driver, Generic 3D, Heads-up Design, Home Series, Kinetix (logo), MAX DWG, ObjectARX, ObjectDBX, Ooga-Chaka, Photo Landscape, Photoscape, Plugs and Sockets, PolarSnap, Powered with Autodesk Technology, Powered with Autodesk Technology (logo), ProConnect, ProjectPoint, Pro Landscape, QuickCAD, RadioRay, SchoolBox, SketchTools, Suddenly Everything Clicks, Supportdesk, The Dancing Baby, Transforms Ideas Into Reality, Visual LISP, and Volo.

Third Party Trademarks

Élan License Manager is a trademark of Élan Computer Group, Inc.

Microsoft, Visual Basic, Visual C++, and Windows are registered trademarks and Visual FoxPro and the Microsoft Visual Basic Technology logo are trademarks of Microsoft Corporation in the United States and other countries.

All other brand names, product names or trademarks belong to their respective holders.

Third Party Software Program Credits

ACIS® Copyright © 1994, 1997, 1999 Spatial Technology, Inc., Three-Space Ltd., and Applied Geometry Corp. All rights reserved.

Copyright © 1997 Microsoft Corporation. All rights reserved.

International CorrectSpell™ Spelling Correction System © 1995 by Lernout & Hauspie Speech Products, N.V. All rights reserved.

InstallShield™ 3.0. Copyright © 1997 InstallShield Software Corporation. All rights reserved.

Portions Copyright © 1991-1996 Arthur D. Applegate. All rights reserved.

Portions of this software are based on the work of the Independent JPEG Group.

Typefaces from the Bitstream® typeface library copyright 1992.

Typefaces from Payne Loving Trust © 1996. All rights reserved.

The license management portion of this product is based on Élan License Manager © 1989, 1990, 1998 Élan Computer Group, Inc. All rights reserved.

Autodesk would like to acknowledge and thank Perceptual Multimedia, Inc., for the creative and technical design and the development of the Visual LISP Garden Path tutorial.

GOVERNMENT USE

Use, duplication, or disclosure by the U. S. Government is subject to restrictions as set forth in FAR 12.212 (Commercial Computer Software-Restricted Rights) and DFAR 227.7202 (Rights in Technical Data and Computer Software), as applicable.

Contents

	Introduction	1
	The Garden Path Revisited: Working in Visual LISP	2
	Tutorial Overview	3
Lesson 1	Designing and Beginning the Program	5
	Defining Overall Program Goals	6
	Getting Started with Visual LISP	7
	Looking at Visual LISP Code Formatting	8
	Analyzing the Code	9
	Filling the Gaps in the Program	9
	Letting Visual LISP Check Your Code	11
	Running the Program with Visual LISP	12
	Wrapping Up Lesson 1	12
Lesson 2	Using Visual LISP Debugging Tools	13
	Differentiating between Local and Global Variables	14
	Using Local Variables in the Program	15
	Examining the gp:getPointInput Function	16
	Using Association Lists to Bundle Data	17
	Putting Association Lists to Use	18
	Storing the Return Value of gp:getPointInput in a Variable	19
	Examining Program Variables	19
	Revising the Program Code	21
	Commenting Program Code	24
	Setting a Breakpoint and Using More Watches	24
	Using the Debug Toolbar	25
	Stepping through Code	27

Watching Variables As You Step through a Program	29
Stepping Out of the gp:getPointInput Function and into C:Gpmain30	
Wrapping Up Lesson 2	32

Lesson 3 Drawing the Path Boundary 33

Planning Reusable Utility Functions	34
Converting Degrees to Radians	34
Converting 3D Points to 2D Points	35
Drawing AutoCAD Entities	37
Creating Entities Using ActiveX Functions	37
Using entmake to Build Entities	37
Using the AutoCAD Command Line	37
Enabling the Boundary Outline Drawing Function	38
Passing Parameters to Functions	38
Working with an Association List	39
Using Angles and Setting Up Points	40
Understanding the ActiveX Code in gp:drawOutline	42
Ensuring That ActiveX Is Loaded	43
Obtaining a Pointer to Model Space	43
Constructing an Array of Polyline Points	44
Constructing a Variant from a List of Points	45
Putting It All Together	46
Wrapping Up Lesson 3	49

Lesson 4 Creating a Project and Adding the Interface 51

Modularizing Your Code	52
Using Visual LISP Projects	53
Adding the Dialog Box Interface	55
Defining the Dialog Box with DCL	55
Saving a DCL File	58
Previewing a Dialog Box	58
Interacting with the Dialog Box from AutoLISP Code	59
Setting Up Dialog Values	59
Loading the Dialog File	60
Loading a Specific Dialog into Memory	60
Initializing the Default Dialog Values	61
Assigning Actions to Tiles	61
Starting the Dialog	63
Unloading the Dialog	63
Determining What to Do Next	64
Putting the Code Together	64
Updating a Stubbed-Out Function	65
Providing a Choice of Boundary Line Type	66

	Cleaning Up	67
	Running the Application.	68
	Wrapping Up Lesson 4	68
Lesson 5	Drawing the Tiles	69
	Introducing More Visual LISP Editing Tools.	70
	Matching Parentheses	70
	Completing a Word Automatically	71
	Completing a Word by Apropos	72
	Getting Help with a Function	72
	Adding Tiles to the Garden Path.	73
	Applying Some Logic	73
	Applying Some Geometry	74
	Drawing the Rows	74
	Drawing the Tiles in a Row	77
	Looking at the Code.	78
	Testing the Code	81
	Wrapping Up Lesson 5	81
Lesson 6	Acting with Reactors.	83
	Reactor Basics	84
	Reactor Types	84
	Designing Reactors for the Garden Path.	85
	Selecting Reactor Events for the Garden Path	85
	Planning the Callback Functions.	85
	Planning for Multiple Reactors	86
	Attaching the Reactors.	87
	Storing Data with a Reactor	88
	Updating the C:GPath Function	88
	Adding Reactor Callback Functions.	91
	Cleaning Up After Your Reactors.	92
	Test Driving Your Reactors	92
	Examining Reactor Behavior in Detail.	93
	Wrapping Up Lesson 6	94
Lesson 7	Putting It All Together	95
	Planning the Overall Reactor Process	96
	Reacting to More User-Invoked Commands	97
	Storing Information within the Reactor Objects.	99
	Adding the New Reactor Functionality.	102
	Adding Activity to the Object Reactor Callback Functions.	103
	Designing the gp:command-ended Callback Function.	104

Handling Multiple Entity Types	104
Using ActiveX Methods in Reactor Callback Functions	105
Handling Nonlinear Reactor Sequences	105
Coding the command-ended Function	107
Updating gp:Calculate-and-Draw-Tiles	110
Modifying Other Calls to gp:Calculate-and-Draw-Tiles	112
Redefining the Polyline Boundary	114
Looking at the Functions in <i>gppoly.lsp</i>	114
Understanding the gp:RedefinePolyBorder Function	115
Understanding the gp:FindMovedPoint Function	116
Understanding the gp:FindPointInList Function	116
Understanding the gp:recalcPolyCorners Function	118
Understanding the gp:pointEqual, gp:rto2, and gp:zeroSmallNum Functions	118
Wrapping Up the Code	119
Building an Application	120
Starting the Make Application Wizard	120
Wrapping Up the Tutorial	121
LISP and AutoLISP Books	122
AutoLISP Books	122
General LISP Books	122
Index	123

Introduction

This tutorial is designed to demonstrate several powerful capabilities of the Visual LISP™ programming environment for AutoCAD® and introduce features of the AutoLISP® language that may be new to you.

The purpose of the tutorial is to draw a garden path using an automated drawing tool that minimizes drafting time and shows the power of parametric programming. You will learn how to create a drawing routine that automates the generation of a complex shape—the kind of drafting operation you do not want to draw over and over again, if it means starting from scratch each time.



In This Chapter

- The Garden Path Revisited: Working in Visual LISP
- Tutorial Overview



The Garden Path Revisited: Working in Visual LISP

This tutorial is intended for experienced AutoCAD users and assumes you have some familiarity with either LISP or AutoLISP. It also assumes you understand basic Windows® file management tasks such as creating directories, copying files, and navigating through the file system on your hard disk or network.

If you are familiar with AutoLISP and have used earlier versions of the Garden Path tutorial, you will notice several differences:

- The Visual LISP (VLISP™) environment is introduced. This environment provides you with editing, debugging, and other tools specific to the creation of AutoLISP applications. Previous versions of the Garden Path tutorial taught AutoLISP language concepts—not VLISP development tools.
- New ActiveX™ and Reactor functions of AutoLISP are demonstrated, as well as several other extensions to the AutoLISP language provided with VLISP.
- The tutorial has been thoroughly redesigned. Even if you are familiar with the previous version, you will encounter entirely different source code and a much more extensive tutorial.
- There are two possible execution contexts for the Garden Path tutorial. The application may be run as interpreted LISP in piecemeal files and/or functions that are loaded into a single document. Alternately, the program code can be compiled into a VLX application, denoted by a *.vlx executable. A VLX operates from a self-contained namespace that can interact with the application-loading document.

Tutorial Overview

Your goal in this tutorial is to develop a new command for AutoCAD that draws a garden path and fills it with circular tiles. The tutorial is divided into seven lessons. As you progress from lesson to lesson, you receive progressively less detailed instructions on how to perform individual tasks. Help is available in the VLISP documentation if you have any questions.

Lessons 4 and 5 are at an intermediate level and go beyond basic AutoLISP concepts. Lessons 6 and 7 contain advanced and fairly complex programming tasks and are designed for experienced AutoLISP developers.

All the source code for drawing the garden path at each stage of development is available on the AutoCAD installation CD, but the tutorial files are only included in your installation if you chose Full install, or if you chose Custom install and selected the Samples item. If you previously installed AutoCAD and did not install the samples, rerun the install, choose Custom, and select only the Samples item.

The directory structure for the source code files follows the tutorial lesson plan:

```
<AutoCAD directory>\Tutorial\VisualLISP\Lesson1
```

```
<AutoCAD directory>\Tutorial\VisualLISP\Lesson2
```

and so on.

It is recommended you do not modify the sample source code files supplied with AutoCAD. If something is not working correctly within your program, you may want to copy the supplied source code into your working directory. Throughout the tutorial, the working directory is referred to as:

```
<AutoCAD directory>\Tutorial\VisualLISP\MyPath
```

If you choose a different path for your working directory, substitute your directory name at the appropriate times.

Finally, read the Getting Started section of the *Visual LISP Developer's Guide*. It has a brief introduction to many concepts you need to complete this tutorial.

Designing and Beginning the Program

In this first lesson, you'll begin by defining what the application will do. Using the Visual LISP (VLISP) development environment, you'll create a LISP file and begin writing AutoLISP code to support your application. In the process, you'll begin to discover how VLISP facilitates application development.

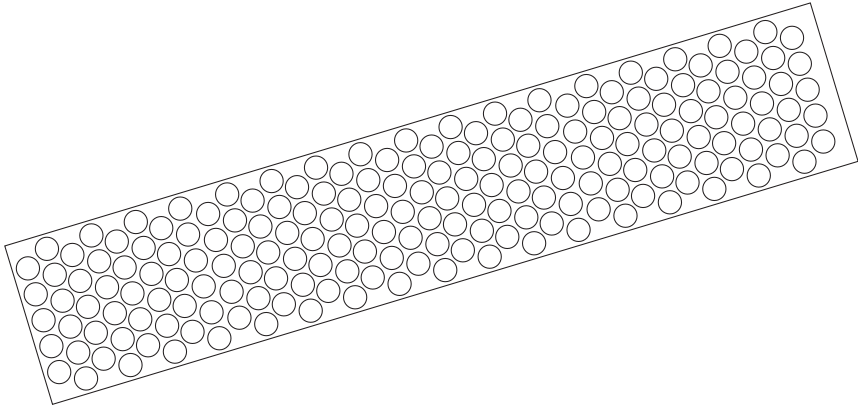
In This Chapter

1

- Defining Overall Program Goals
- Getting Started with Visual LISP
- Looking at Visual LISP Code Formatting
- Analyzing the Code
- Filling the Gaps in the Program
- Letting Visual LISP Check Your Code
- Running the Program with Visual LISP
- Wrapping Up Lesson 1

Defining Overall Program Goals

Developing an AutoLISP program begins with an idea for automating some aspect of AutoCAD. It may be a need to speed up a repetitive drafting function, or to simplify a complex series of operations. For the tutorial, the garden path you want your program to draw is a complex shape with a variable number of components, based on initial input from the user. Here's what it will look like:



Your program must do the following to draw the garden path:

- Given a start point, an endpoint, and a width, draw a rectilinear boundary. The boundary can be at any 2D orientation. There should be no limit on how large or small it can be.
- Prompt the user for tile size and tile spacing values. The tiles are simple circles and will fill the boundary but must not overlap or cross the boundary.
- Place the tiles in alternating rows.

To see how things should work, you can run a completed version of the application that is supplied with AutoCAD.

To run the supplied example

- 1 From the AutoCAD Tools menu, choose Load Application.
- 2 Select *gardenpath.vlx* from the *Tutorial\VisualLISP* directory, and choose Load.
- 3 Press Close.
- 4 At the Command prompt, enter **gpath**.
- 5 Respond to the first two prompts by picking a start point and an endpoint in the AutoCAD graphics window.
- 6 Enter **2** at the Half Width of Path prompt.
- 7 Choose OK when prompted by the Garden Path Tile Specifications dialog box.

Getting Started with Visual LISP

Now that you've seen how the application is supposed to work, you can begin developing it with VLISP. But first, it helps to demonstrate what can happen when VLISP is waiting for control to return from AutoCAD. You may have already encountered this.

To see Visual LISP wait for control to return from AutoCAD

- 1 At the AutoCAD Command prompt, enter **vlisp** to start Visual LISP.
- 2 Switch back to the AutoCAD window (either select AutoCAD from the taskbar or press ALT+TAB and choose AutoCAD), and enter **gpath** at the AutoCAD Command prompt.
- 3 Before responding to the prompts from **gpath**, switch back to the VLISP window.



In the VLISP window, the mouse pointer appears as a VLISP symbol and you cannot choose any commands or enter text anywhere in the VLISP window. The pointer symbol is a reminder that there is an activity you must complete in AutoCAD before resuming work with VLISP. Remember this whenever you see the VLISP pointer.

- 4 Return to the AutoCAD window and respond to all the prompts from **gpath**.

Now you are ready to begin building the garden path application.

To begin application development with Visual LISP



- 1 From the VLISP File menu, choose New File.
- 2 Enter the following code in the text editor window (it is the window titled "<Untitled-0>"); you can omit the comments, if you wish:

```
;;; Function C:GPath is the main program function and defines the
;;; AutoCAD GPATH command.
(defun C:GPath ()
  ;; Ask the user for input: first for path location and
  ;; direction, then for path parameters. Continue only if you have
  ;; valid input.
  (if (gp:getPointInput) ;
      (if (gp:getDialogInput)
          (progn
            ;; At this point, you have valid input from the user.
            ;; Draw the outline, storing the resulting polyline
            ;; "pointer" in the variable called PolylineName.
            (setq PolylineName (gp:drawOutline))
            (princ "\nThe gp:drawOutline function returned <")
            (princ PolylineName)
            (princ ">")
            (Alert "Congratulations - your program is complete!")
          )
        (princ "\nFunction cancelled.")
      )
    (princ "\nIncomplete information to draw a boundary.")
  )
  (princ) ; exit quietly
)
;;; Display a message to let the user know the command name.
(princ "\nType gpath to draw a garden path.")
(princ)
```

- 3 Choose File ► Save As from the menu, and save the code in the new file as <AutoCAD directory>\Tutorial\VisualLISP\MyPath\gpmain.lsp.
- 4 Review your work.

Looking at Visual LISP Code Formatting

VLISP recognizes the various types of characters and words that make up an AutoLISP program file and highlights the characters in different colors. This makes it easier for you to spot something incorrect quickly. For example, if you miss a closing quotation mark following a text string, everything you type continues to display in magenta, the color denoting strings. When you enter the closing quotation mark, VLISP correctly colors the text following the string, according to the language element it represents.



As you enter text, VLISP also formats it by adding spacing and indentation. To get VLISP to format code you copy into its text editor from another file, choose Tools ► Format Code in Editor from the VLISP menu.

Analyzing the Code

The `defun` function statement defines the new function. Notice the main function is named `C:GPath`. The `C:` prefix establishes that this function is callable from the AutoCAD Command line. `GPath` is the name users enter to launch the application from the AutoCAD Command prompt. The functions that obtain input from users are named `gp:getPointInput` and `gp:getDialogInput`. The function that draws the garden path outline is `gp:drawOutline`. These names are prefixed with `gp:` to indicate they are specific to the garden path application. This is not a requirement, but it is a good naming convention to use to distinguish application-specific functions from general utility functions you frequently use.

In the main function, `princ` expressions display the results of the program if it runs successfully, or a warning message if the program encounters an unexpected event. For example, as will be seen in Lesson 2, if the user presses ENTER instead of picking a point on the screen, the call to `gp:getPointInput` ends prematurely, returning a `nil` value to the main function. This causes the program to issue a `princ` message of “Incomplete information to draw a boundary.”

The call to `princ` near the end of the program serves as a prompt. Upon application load, the prompt informs users what they need to type to initiate the drawing of a garden path. The final `princ` without a string argument forces the program to exit quietly, meaning the value of the main function’s final expression is not returned. If the final suppressing `princ` were omitted, the prompt would display twice.

Filling the Gaps in the Program

For the code in this new file to work correctly, you must write three more function definitions. The main garden path code contains calls to three custom functions:

- `gp:getPointInput`
- `gp:getUserInput`
- `gp:drawOutline`

For now, you will just write stubbed-out function definitions. A stubbed-out function serves as a placeholder for the complete function that is to follow. It allows you to try out pieces of your code before adding all the detail needed to complete the application.

To define stubbed-out functions for the application

- 1 Position your cursor at the top of the program code in the text editor window and press ENTER a couple of times to add blank lines.
- 2 Enter the following code, beginning where you inserted the blank lines:

```
;;; Function gp:getPointInput will get path location and size
(defun gp:getPointInput()
  (alert
   "Function gp:getPointInput will get user drawing input"
  )
  ;; For now, return T, as if the function worked correctly.
  T
)

;;; Function gp:getDialogInput will get path parameters
(defun gp:getDialogInput ()
  (alert
   "Function gp:getDialogInput will get user choices via a dialog"
  )
  ;;For now, return T, as if the function worked correctly.
  T
)

;;; Function gp:drawOutline will draw the path boundary
(defun gp:drawOutline ()
  (alert
   (strcat "This function will draw the outline of the polyline"
           "\nand return a polyline entity name/pointer."
  )
  )
  ;; For now, simply return a quoted symbol. Eventually, this
  ;; function will return an entity name or pointer.
  'SomeEname
)
```

Right before the end of each input function is a line of code that contains only a `T`. This is used as a return value to the calling function. All AutoLISP functions return a value to the function that called them. The letter `T` is the symbol for “true” in AutoLISP, and adding it causes the function to return a true value. The way *gpmain.lsp* is structured, each input function it calls must return a value other than `nil` (which indicates “no value”) for the program to proceed to the next step.

An AutoLISP function will, by default, return the value of the last expression evaluated within it. In the stubbed-out functions, the only expression is a call to the `alert` function. But `alert` always returns `nil`. If this is left as the last

expression in `gp:getPointInput`, it will always return `nil`, and you will never pass through the `if` to the `gp:getDialogInput` function.

For a similar reason, the end of the `gp:DrawOutline` function returns a quoted symbol (`'SomeEname`) as a placeholder. A quoted symbol is a LISP construct that is not evaluated. (If you are curious about how the LISP language works, there are a number of good books available, mentioned at the end of this tutorial.)

Letting Visual LISP Check Your Code

VLISP has a powerful feature for checking your code for syntactical errors. Use this tool before trying to run the program. You can catch common typing errors such as missing parentheses or missing quotation marks, and other syntactical problems.

To check the syntax of your code



- 1 Make sure the text editor window containing *gpmain.lsp* is the active window. (Click in the title bar of the window to activate it.)
- 2 From the VLISP menu, choose Tools ► Check Text in Editor.
- 3 The Build Output window appears with the results of the syntax check. If VLISP did not detect any errors, the window contains text similar to the following:

```
[CHECKING TEXT GPMAIN.LSP loading...]  
.....  
; Check done.
```

If you have problems and need help, refer to the “Developing Programs with Visual LISP” chapter of the *Visual LISP Developer’s Guide*. See if you can determine where the problem is located. If you are spending too much time locating the problem, use the sample *gpmain.lsp* file provided in the *lesson1* directory to continue with the tutorial.

To use the supplied *gpmain.lsp* program (if necessary)

- 1 Close the text editor window containing the *gpmain.lsp* code you entered.
- 2 Choose File ► Open File from the VLISP menu, and open the *gpmain.lsp* file in the `\Tutorial\VisualLISP\lesson1` directory.
- 3 Choose File ► Save As and save the file in your `\Tutorial\VisualLISP\MyPath` directory as *gpmain.lsp*, replacing the copy you created.

Running the Program with Visual LISP

Running AutoLISP programs in VLISP allows you to use the many debugging features of VLISP to investigate problems that may occur in your application.

To load and run the program



- 1 With the text editor window active, choose Tools ► Load Text in Editor from the VLISP menu.
- 2 At the `_ $` prompt in the VLISP Console window, enter `(C:GPath)`.
The Console window expects commands to be entered in AutoLISP syntax, so all function names must be enclosed in parentheses.
- 3 Press ENTER or click OK in response to the message windows. The final message should read “Congratulations – your program is complete!”

NOTE If AutoCAD is minimized when you run `gpath`, you will not see the prompts until you restore the AutoCAD window (using either the taskbar or ALT+TAB).

Wrapping Up Lesson 1

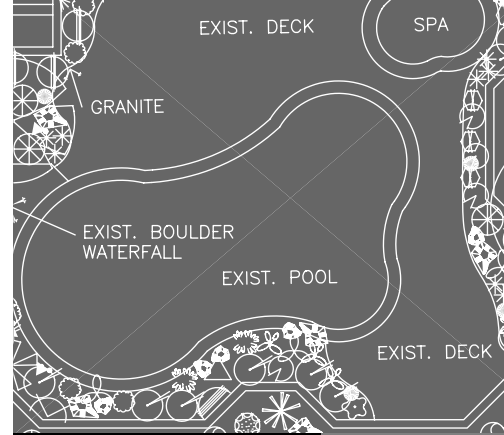
In this lesson, you

- Defined program goals.
- Learned the value of stub functions.
- Learned about naming functions to identify them as specific to your application or as general functions to be used over and over.
- Learned how to use VLISP to check your code.
- Learned how to load and run a program in VLISP.

You are done with this lesson. Save your program file again to be certain you have the latest revisions.

Using Visual LISP Debugging Tools

This lesson teaches you how to use several valuable VLISP debugging tools that speed up the development of AutoLISP programs. You will also learn the difference between local and global variables, and when to use them. Your program will become more active—prompting users to enter some information. The information will be stored in a list and you'll begin to understand the power of using lists within your AutoLISP programs. After all, LISP got its name because it is a LIST Processing language.



In This Chapter

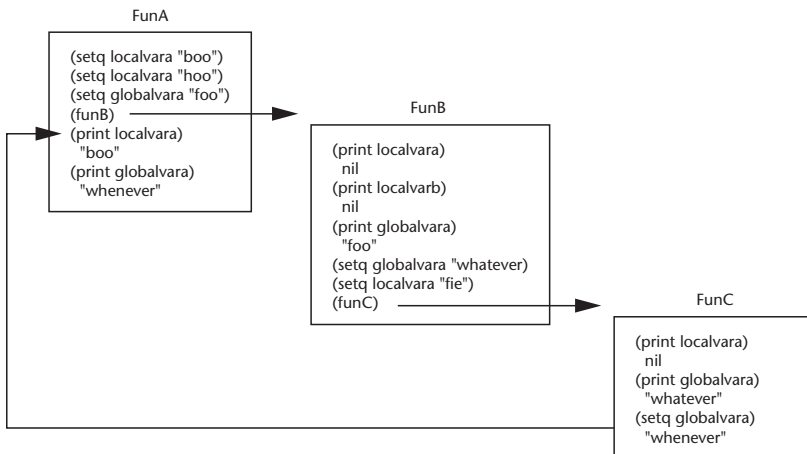
2

- Differentiating between Local and Global Variables
- Using Association Lists to Bundle Data
- Examining Program Variables
- Revising the Program Code
- Commenting Program Code
- Setting a Breakpoint and Using More Watches
- Wrapping Up Lesson 2

Differentiating between Local and Global Variables

This lesson discusses the use of local variables versus global document variables. Global variables are accessible by all functions loaded within a document (an AutoCAD drawing). These variables may retain their value after the program that defined them completes. Sometimes, this is what you want. You'll see an example of this later in the tutorial.

Local variables retain their value only as long as the function that defined them is running. After the function finishes running, the local variable values are automatically discarded, and the system reclaims the memory space the variable used. This is known as automatic garbage collection, and is a feature of most LISP development environments, such as VLISP. Local variables use memory more efficiently than global variables.



Another big advantage is that local variables make it easier to debug and maintain your applications. With global variables, you are never sure when or in which function the variable's value might be modified; with local variables you don't have as far to trace. You usually end up with fewer side effects (that is, one part of the program affecting a variable from another part of the program).

Because of the advantages cited, this tutorial uses local variables almost exclusively.

NOTE If you have been working with AutoLISP for some time, you may have developed the practice of using global variables during development to examine your program while you are building it. This practice is no longer necessary, given the powerful debugging tools of VLISP.

Using Local Variables in the Program

Refer to the **gp:getPointInput** function you created in Lesson 1:

```
(defun gp:getPointInput()  
  (alert  
    "Function gp:getPointInput will get user drawing input"  
  )  
  ;; For now, return T, as if the function worked correctly.  
  T  
)
```

So far, the function does not do much work. You will now begin to build on it by adding functions to get input from the user, which will define the start point, endpoint, and width of the path.

It is a good practice when creating AutoLISP programs to emulate the behavior of AutoCAD. For this reason, instead of asking the user to indicate the width by selecting a point in the drawing in respect to the centerline of a linear shape, your program should ask for a selection of the half width.

Once the **gp:getPointInput** function is complete, the variables, as well as the values assigned to them, will no longer exist. Therefore, you will store user-supplied values in local variables. Here's what the function might look like:

```
(defun gp:getPointInput(/ StartPt EndPt HalfWidth)  
  (if (setq StartPt (getpoint "\nStart point of path: "))  
    (if (setq EndPt (getpoint StartPt "\nEndpoint of path: "))  
      (if (setq HalfWidth (getdist EndPt "\nHalf width of path: "))  
        T  
      )  
    )  
  )  
)
```

The local variables are declared following the slash character, in the **defun** statement that begins the function. The first call to **getpoint** prompts the user to indicate a start point. The endpoint is then acquired in relation to the chosen start point. While selecting the endpoint, the user will observe a rubber-band line extending from the start point. Similarly, while setting the half width value, the user will view another rubber-band line, this time representing distance, emanating from the endpoint.

To see how `gp:getPointInput` works

- 1 Type the `gp:getPointInput` code into the VLISP Console window.
- 2 With the Console window cursor following the last parenthesis of the block of code (or on the next line below it), press ENTER and you will replace any previously loaded version of the `gp:getPointInput` function.
- 3 Execute the function from the Console window by entering `(gp:getPointInput)` at the Console prompt.
- 4 Pick points when prompted, and enter a half width value.

Examining the `gp:getPointInput` Function

When you ran the `gp:getPointInput` function, control was automatically passed from VLISP to AutoCAD. You responded to three prompts, after which control was passed back from AutoCAD to VLISP, and a τ symbol displayed in the Console window.

Within the program, here's what happens:

- 1 VLISP waits for you to pick the first point.
- 2 When you pick the first point, the program stores the value of your selection (a list containing three coordinate values—an *X*, *Y*, and *Z* value) into the `startPt` variable.
- 3 The first `if` function examines the result to determine whether a valid value was entered or no value was entered. When you pick a start point, control is passed to the next `getpoint` function.
- 4 When you pick an endpoint, the point value is stored in the `endPt` variable.
- 5 The result of this statement is examined by the next `if` statement, and control is passed to the `getdist` function.
- 6 The `getdist` function acts in a similar fashion when you pick a point on the screen or enter a numeric value. The result of the `getdist` function is stored in the `halfwidth` variable.
- 7 Program flow reaches the τ nested deeply within the function. No other functions follow this, so the function ends, and the value τ is returned. This is the τ you see at the Console window.

You need some way to return values from one function to another. One way to do this is to create a list of the values retrieved from `gp:getPointInput`, as highlighted in the following code:

```

(defun gp:getPointInput ( / StartPt EndPt HalfWidth )
  (if (setq StartPt (getpoint "\nStart point of path: "))
    (if (setq EndPt (getpoint StartPt "\nEndpoint of path: "))
      (if (setq HalfWidth (getdist EndPt "\nHalf width of path: "))
        (list StartPt EndPt HalfWidth)
      )
    )
  )
)

```

Copy this version of `gp:getPointInput` into the Console window and press ENTER. Here's an opportunity to try another feature of the Console window.

To use the Console window history feature to run `gp:getPointInput`

- 1 Press TAB. This invokes the Console history command, cycling through any commands previously entered in the Console window. If you go too far, press SHIFT+TAB to cycle in the other direction.
- 2 When you see `(gp:getPointInput)` at the Console prompt, press ENTER to execute the function once again.
- 3 Respond to the prompts as before.

The function returns a list containing two nested lists and a real (floating point) value. The return values look like the following:

```
((4.46207 4.62318 0.0) (7.66688 4.62318 0.0) 0.509124)
```

These values correspond to the `StartPt`, `EndPt`, and `HalfWidth` variables.

Using Association Lists to Bundle Data

The previous example works, but you can do better. In the next exercise, you will build an association list, or `assoc` list (after the LISP function that deals with association lists). In an association list, the values you are interested in are associated with key values. Here is a sample association list:

```
((10 4.46207 4.62318 0.0) (11 7.66688 4.62318 0.0) (40 . 1.018248))
```

In the sample association list, the key values are the numbers 10, 11, and 40. These key values serve as a unique index within the list. This is the mechanism AutoCAD uses to return entity data to AutoLISP if you access an entity from within your program. A key value of 10 indicates a start point, a key value of 11 is typically an endpoint.

What are the advantages of an association list? For one thing, unlike the regular list, the order of the values returned does not matter. Look at the first list again:

```
((4.46207 4.62318 0.0) (7.66688 4.62318 0.0) 0.509124)
```

Look at the return values; it is not apparent which sublist is the start point and which is the endpoint. Furthermore, if you modify the function in the future, any other function that relies on data returned in a specific order may be adversely affected.

Using an association list, the order of the values does not matter. If the order of an association list changes, you can still tell which value defines what. For example, an 11 value is still an endpoint, regardless of where it occurs within the overall list:

```
((11 7.66688 4.62318 0.0)      ; order of list  
(40 . 1.018248)              ; has been  
(10 4.46207 4.62318 0.0))    ; modified
```

Putting Association Lists to Use

When you use association lists, you should document what your key values represent. For the garden path, the key values of 10, 11, 40, 41, and 50 will mean the following:

- 10 indicates the 3D coordinate of the start point of the path.
- 11 indicates the 3D coordinate of the endpoint of the path.
- 40 indicates the width (not the half width) of the path.
- 41 indicates the length of the path, from start to end.
- 50 indicates the primary vector (or angle) of the path.

The following is an updated version of the **gp:getPointInput** function. Within it, an AutoLISP function called **cons** (short for construct a list) builds the keyed sublists that belong to the association list. Copy this version to the Console window, press ENTER, and run (**gp:getPointInput**) again:

```
(defun gp:getPointInput(/ StartPt EndPt HalfWidth)  
  (if (setq StartPt (getpoint "\nStart point of path: "))  
      (if (setq EndPt (getpoint StartPt "\nEndpoint of path: "))  
          (if (setq HalfWidth (getdist EndPt "\nHalf width of path: "))  
              ;; if you've made it this far, build the association list  
              ;; as documented above. This will be the return value  
              ;; from the function.
```

```

        (list
          (cons 10 StartPt)
          (cons 11 EndPt)
          (cons 40 (* HalfWidth 2.0))
          (cons 50 (angle StartPt EndPt))
          (cons 41 (distance StartPt EndPt))
        )
      )
    )
  )
)

```

Notice that, when building the list, the program converts the half width specified by the user into a full width by multiplying its value by 2.

The Console window shows output similar to the following:

```

_ $ (gp:getPointInput)
((10 2.16098 1.60116 0.0) (11 12.7126 7.11963 0.0) (40 . 0.592604)
(50 . 0.481876) (41 . 11.9076))
_ $

```

Storing the Return Value of gp:getPointInput in a Variable

Now try something else. Call the function again, but this time store the return value in a variable named `gp_PathData`. To do this, enter the following at the Console window prompt:

```
(setq gp_PathData (gp:getPointInput))
```

To view the value of the variable you just set, enter its name at the Console window prompt:

```
_ $ gp_PathData
```

VLISP returns data like the following:

```
((10 2.17742 1.15771 0.0) (11 13.2057 7.00466 0.0) (40 . 1.12747)
(50 . 0.487498) (41 . 12.4824))
```

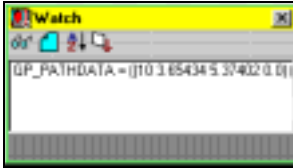
Examining Program Variables

VLISP provides you with an entire toolkit of programming and debugging tools. One of the most valuable tools is a Watch, which lets you examine variables in more detail than appears in the VLISP Console window. You can also watch local variables within functions as the function executes.

To watch the value of a variable

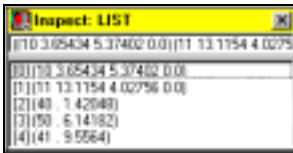


- 1 Choose Debug ► Add Watch from the VLISP menu. VLISP displays a dialog box titled “Add Watch.”
- 2 Enter the name of the variable you wish to examine. For this example, specify `gp_PathData`, the variable you just set from the Console window. VLISP displays a Watch window:



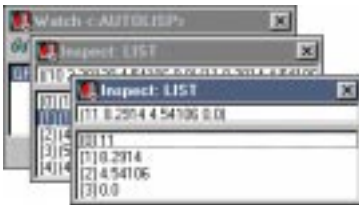
VLISP displays the value of the variable on a single line within the Watch window—the base window shown in the illustration. In this case, the value of the variable is a long list, and you cannot see its entire value. You can resize the Watch window by dragging its border, but there is a better alternative.

- 3 Double-click on the variable name in the Watch window. This opens an Inspect window:



The Inspect window indicates the data type of the variable you are inspecting (in this case, a list), and the value of the variable. For lists, Inspect displays each list item on its own line.

- 4 Double-click on the line with the association list key 11. VLISP opens another Inspect window:



- 5 When you are done inspecting variables, close all the Inspect windows but keep the Watch window open.

Revising the Program Code

Now that you've seen how to use association lists in AutoLISP code, you can use this method in writing the completed version of the `gp:getPointInput` function. Using the following code, replace or modify the version of `gp:getPointInput` you previously saved in `gpmain.lsp`.

NOTE If you need or want to type the code into `gpmain.lsp`, rather than copy it from another file, you can save time by leaving out the comments (all lines that begin with semicolons). But don't get used to the idea of writing code without comments!

```
;;;-----  
;;;      Function: gp:getPointInput                               ;  
;;;-----  
;;;  Description: This function asks the user to select three   ;  
;;;                points in a drawing, which will determine the ;  
;;;                path location, direction, and size.         ;  
;;;-----  
;;;  If the user responds to the get functions with valid data, ;  
;;;  use startPt and endPt to determine the position, length,   ;  
;;;  and angle at which the path is drawn.                     ;  
;;;-----  
;;;  The return value of this function is a list consisting of: ;  
;;;  (10 . Starting Point) ;; List of 3 reals (a point) denoting ;  
;;;                        ;; starting point of garden path.    ;  
;;;  (11 . Ending Point)  ;; List of 3 reals (a point) denoting ;  
;;;                        ;; ending point of garden path.      ;  
;;;  (40 . Width)         ;; Real number denoting boundary      ;  
;;;                        ;; width.                             ;  
;;;  (41 . Length)       ;; Real number denoting boundary      ;  
;;;                        ;; length.                            ;  
;;;  (50 . Path Angle)   ;; Real number denoting the angle     ;  
;;;                        ;; of the path, in radians.         ;  
;;;-----  
(defun gp:getPointInput(/ StartPt EndPt HalfWidth)  
  (if (setq StartPt (getpoint "\nStart point of path: "))  
      (if (setq EndPt (getpoint StartPt "\nEndpoint of path: "))  
          (if (setq HalfWidth (getdist EndPt "\nHalf width of path: "))  
              ;; if you've made it this far, build the association list  
              ;; as documented above. This will be the return value  
              ;; from the function.  
              (list  
                (cons 10 StartPt)  
                (cons 11 EndPt)  
                (cons 40 (* HalfWidth 2.0))  
                (cons 50 (angle StartPt EndPt))  
                (cons 41 (distance StartPt EndPt))  
              )  
          )  
      )  
  )  
)  
)  
)  
)
```

Next, you need to update the main function, `C:GPath`, in `gpmain.lsp`. Modify it to look like the following code:

```
(defun C:GPath (/ gp_PathData)
  ;; Ask the user for input: first for path location and
  ;; direction, then for path parameters. Continue only if you
  ;; have valid input. Store the data in gp_PathData.
  (if (setq gp_PathData (gp:getPointInput))
      (if (gp:getDialogInput)
          (progn
            ;; At this point, you have valid input from the user.
            ;; Draw the outline, storing the resulting polyline
            ;; pointer in the variable called PolylineName.
            (setq PolylineName (gp:drawOutline))
            (princ "\nThe gp:drawOutline function returned <")
            (princ PolylineName)
            (princ ">")
            (Alert "Congratulations - your program is complete!")
          ) ;_ end of progn
        (princ "\nFunction cancelled.")
      ) ;_ end of if
      (princ "\nIncomplete information to draw a boundary.")
    ) ;_ end of if
  (princ) ; exit quietly
);_ end of defun
```

If you are copying and pasting the code, add the following comments as a header preceding `C:GPath`:

```

;;;*****
;;;      Function: C:GPath          The Main Garden Path Function  ;
;;;-----
;;; Description: This is the main garden path function. It is a ;
;;;              C: function, meaning that it is turned into an ;
;;;              AutoCAD command called GPATH. This function ;
;;;              determines the overall flow of the garden path ;
;;;              program. ;
;;;*****
;;; The gp_PathData variable is an association list of the form: ;
;;; (10 . Starting Point) - List of 3 reals (a point) denoting ;
;;;                          starting point of the garden path. ;
;;; (11 . Ending Point)   - List of 3 reals (a point) denoting ;
;;;                          endpoint of the garden path. ;
;;; (40 . Width)          - Real number denoting boundary ;
;;;                          width. ;
;;; (41 . Length)        - Real number denoting boundary ;
;;;                          length. ;
;;; (50 . Path Angle)    - Real number denoting the angle of ;
;;;                          the path, in radians. ;
;;; (42 . Tile Size)     - Real number denoting the size ;
;;;                          (radius) of the garden path tiles. ;
;;; (43 . Tile Offset)   - Spacing of tiles, border to border. ;
;;; ( 3 . Object Creation Style) ;
;;;                          - Object creation style indicates how ;
;;;                          the tiles are to be drawn. The ;
;;;                          expected value is a string and one ;
;;;                          one of three values (string case ;
;;;                          is unimportant): ;
;;;                          "ActiveX" ;
;;;                          "Entmake" ;
;;;                          "Command" ;
;;; ( 4 . Polyline Border Style) ;
;;;                          - Polyline border style determines ;
;;;                          the polyline type to be used for ;
;;;                          path boundary. The expected value ;
;;;                          one of the following (string case is ;
;;;                          unimportant): ;
;;;                          "Pline" ;
;;;                          "Light" ;
;;;*****

```

To test the code revisions

- 1 Save the updated file.
- 2 Use the Check feature to search for any syntactical errors.
- 3 Format the code, to make it more readable.
- 4 Load the code, so that VLISP redefines the earlier versions of the functions.
- 5 To run the program, enter (**c:gpath**) at the Console prompt.

If the program does not run successfully, try fixing it and running it again. Repeat until you are too frustrated to continue. If all else fails, you can copy the correct code from the *Tutorial\VisualLISP\Lesson2* directory.

Commenting Program Code

VLISP treats any AutoLISP statement beginning with a semicolon as a comment. The last two code examples contained a lot of comments. A comment in an AutoLISP program is something you write for yourself, not for the program. Commenting code is one of the best programming practices you can establish for yourself. Why write comments?

- To explain the code to yourself when you are editing the program nine months in the future, adding all those features your users have been asking you about. Memory fades, and the most apparent sequence of functions can easily turn into an unrecognizable tangle of parentheses.
- To explain the code to others who inherit the responsibility of updating the program. Reading someone else's code is an extremely frustrating experience, especially if the code contains very few comments.

VLISP contains some utilities that help you as you comment your code. Notice some comments in the examples begin with three semicolons (`;;;`), sometimes two (`;;`), and sometimes just one (`;`). Refer to “Applying Visual LISP Comment Styles” in the *Visual LISP Developer's Guide* to see how VLISP treats the different comments.

To save space and trees, the remaining code examples in this tutorial do not include all the comments in the sample source files. It is assumed you have already established the beneficial habit of extensive commenting and will do so without any prompting.

Setting a Breakpoint and Using More Watches

A breakpoint is a symbol (point) you place in source code to indicate where you want a program to stop executing temporarily. When you run your code, VLISP proceeds normally until it encounters a breakpoint. At that point, VLISP suspends execution and waits for you to tell it what to do. It hasn't halted the program for good—it has placed it in a state of suspended animation.

While your program is suspended, you can

- Step through your code, function by function, or expression by expression.
- Resume normal execution of your program at any point.
- Alter the value of variables dynamically, and change the results of the program being executed.
- Add variables to the Watch window.

Using the Debug Toolbar

The Debug toolbar contains several tools you will employ as you work through this section. By default, this toolbar is attached to the View and Tools toolbars, and appears as a single VLISP toolbar:



— Debug toolbar grips

The Debug toolbar is the left-most set of icons. Most of the items on the toolbar are inactive until you run your program in debugging mode (that is, with one or more breakpoints defined).

If you haven't done so already, detach the Debug toolbar from its position at the top of the screen. To do this, grab and drag it by the two vertical grips at the left of the toolbar. You can detach any of the VLISP toolbars and position them on your screen where they are most effective for your style of work.

The Debug toolbar is divided into three main groups of buttons, each consisting of three buttons. When you run a program in debugging mode, the toolbar looks like the following:



- The first three buttons allow you to step through your program code.
- The next three buttons determine how VLISP should proceed whenever it has stopped at a breakpoint or an error.
- The next three buttons set or remove a breakpoint, add a Watch, and jump to the position within your source code where the last break occurred.

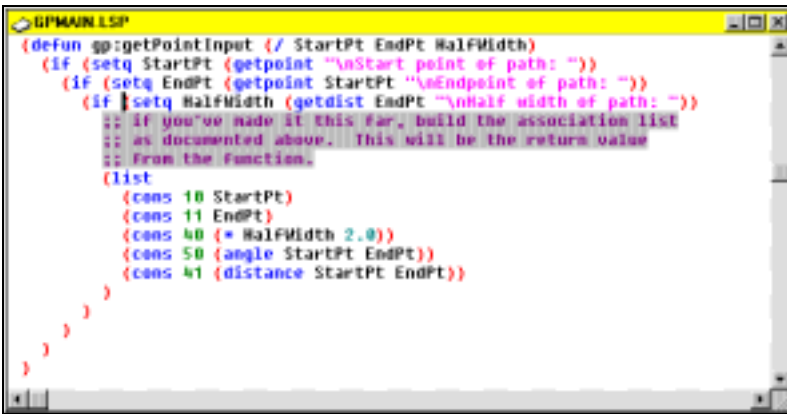
The last button on the Debug toolbar is a Step Indicator. It does not execute any function but provides a visual indication of where your cursor is positioned as you step through your code. When you are not running in debugging mode, this button appears blank.

To set a breakpoint

- 1 In the VLISP editor window containing *gpmain.lsp*, position your cursor just in front of the opening parenthesis of the `setq` function of the following line of code, within the `gp:getPointInput` function:

```
(setq HalfWidth (getdist EndPt "\nHalf width of path: "))
```

- 2 Click the mouse once. The position is illustrated in the following screen snapshot:



```
(defun gp:getPointInput (/ StartPt EndPt HalfWidth)
  (if (setq StartPt (getpoint "\nStart point of path: "))
    (if (setq EndPt (getpoint StartPt "\nEndpoint of path: "))
      (if (setq HalfWidth (getdist EndPt "\nHalf width of path: "))
        ;; If you've made it this far, build the association list
        ;; as documented above. This will be the return value
        ;; from the function.
        (list
         (cons 10 StartPt)
         (cons 11 EndPt)
         (cons 40 (* HalfWidth 2.0))
         (cons 50 (angle StartPt EndPt))
         (cons 41 (distance StartPt EndPt)))
      )
    )
  )
)
```



- 3 With the text insertion point set, press the Toggle Breakpoint button on the Debug toolbar.

The Toggle Breakpoint button acts as a toggle, alternating between on and off states. If there is no breakpoint at the cursor position, it sets one; if there is already a breakpoint there, it removes it.



- 4 Press the Load Active Edit Window button on the Tools toolbar to load the file.
- 5 Run the `(c:GPath)` function from the VLISP Console prompt.
VLISP executes the program normally up to the breakpoint. In this case, it will prompt you for the first two points—the start point and endpoint of the path.
- 6 Specify the start point and endpoint when prompted.

To step through the code from the breakpoint



- 1 Press the Step Over button.

After you press the Step Over button, control passes to AutoCAD and you are prompted to specify the width of the path.

- 2 Reply to the prompt.

After you specify the width, control passes back to VLISP. Notice where your cursor is and what the step indicator button shows.

```
(defun gp:getPointInput (/ StartPt EndPt HalfWidth)
  (if (setq StartPt (getpoint "\nStart point of path: "))
    (if (setq EndPt (getpoint StartPt "\nEndpoint of path: "))
      (if (setq HalfWidth (getdist EndPt "\nHalf width of path: "))
        ;; If you've made it this far, build the association list
        ;; as documented above. This will be the return value
        ;; from the function.
        (list
         (cons 10 StartPt)
         (cons 11 EndPt)
         (cons 40 (= HalfWidth 2.0))
         (cons 50 (angle StartPt EndPt))
         (cons 41 (distance StartPt EndPt))
        )
      )
    )
  )
)
```

VLISP evaluates the entire highlighted expression, then stops at the end of the expression.

- 3 Press the Step Over button again. VLISP jumps to the beginning of the next block of code, and highlights the entire block.

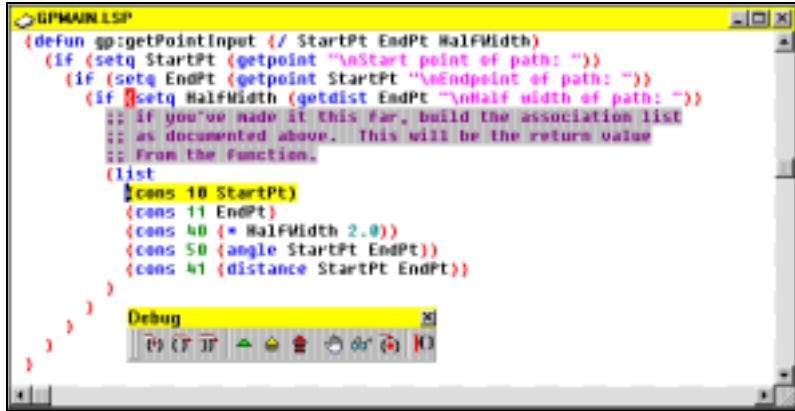
```
(defun gp:getPointInput (/ StartPt EndPt HalfWidth)
  (if (setq StartPt (getpoint "\nStart point of path: "))
    (if (setq EndPt (getpoint StartPt "\nEndpoint of path: "))
      (if (setq HalfWidth (getdist EndPt "\nHalf width of path: "))
        ;; If you've made it this far, build the association list
        ;; as documented above. This will be the return value
        ;; from the function.
        (list
         (cons 10 StartPt)
         (cons 11 EndPt)
         (cons 40 (= HalfWidth 2.0))
         (cons 50 (angle StartPt EndPt))
         (cons 41 (distance StartPt EndPt))
        )
      )
    )
  )
)
```



- 4 Press the Step Into (not Step Over) button.

NOTE During this exercise, if you make an incorrect selection and skip a step or two, you can restart the exercise very easily. First, press the Reset button from the Debug toolbar. This terminates the execution of any VLISP code, and resets the VLISP system to the top level. Next, start over at step 1.

Now the first `cons` function is highlighted, and VLISP is stopped right before the function (notice the Step Indicator button).



Watching Variables As You Step through a Program

While you step through your program, you can add variables to the Watch window and change their values.

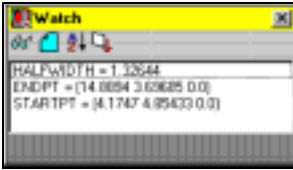
If you do not see the Watch window, simply press the Watch Window button on the toolbar to bring it back.



If your Watch window stills contains the variable `gp_PathData`, press the Clear Window button displayed at the top of the Watch window.

To add variables to the Watch window

- 1 Double-click on any occurrence of `startpt` in the VLISP text editor window. This is the name of the variable whose value you want to track.
- 2 Press the Add Watch button in the Watch window, or right-click and choose Add Watch.
- 3 Repeat this process for the variables `endpt` and `halfwidth`. Your Watch window should resemble the following:



If you are debugging a program that isn't working correctly, use breakpoints in combination with watches to make sure your variables contain the values you expect.

If a variable does not contain the value you think it should, you can change the value and see how it affects the program. For example, say that you expect the `halfwidth` value to be a whole number. But because you weren't careful about picking the points during the input selections, you ended up with a value like 1.94818.

To change the value of a variable while the program is running

- 1 Enter the following at the Console prompt:

```
(setq halfwidth 2.0)
```

Note that the value in the Watch window changes. But can you be sure the new value will be used when the `width` sublist (`40 . width`) is created in the association list? Add one more expression to the Watch window to test this.

- 2 Choose Debug ► Watch Last Evaluation from the VLISP menu.

This adds a variable named `*Last-Value*` to your Watch window.

`*Last-Value*` is a global variable in which VLISP automatically stores the value of the last expression evaluated.

- 3 Step through the program (pressing either the Step Into or Step Over button) until the expression responsible for building the `width` sublist is evaluated. The code for this action is:

```
(cons 40 (* Halfwidth 2.0))
```

If you overrode the value of `halfwidth` as specified, the evaluation of this expression should return `(40 . 4.0)` in the Watch window.

Stepping Out of the `gp:getPointInput` Function and into `C:Gpmain`

There is one more point to illustrate: what happens to the value of the local variables in `gp:getPointInput` after you exit the function.

To exit `gp:getPointInput` and return control to `c:gpath`



- 1 Press the Step Out button.

VLISP steps to the very end of the `gp:getPointInput` function and stops just before exiting.



- 2 Press the Step Into button.

Control returns to `c:gpmain`, the function that called `gp:getPointInput`.



Examine the values of the variables in the Watch window. Because they are variables local to the `gp:getPointInput` function, `endpt` and `startpt` are `nil`. VLISP automatically reclaimed the memory occupied by these variables. Normally, the third local function variable `halfwidth` also contains a value of `nil`, but due to debugging activity, it was overridden globally in the Console window and still possesses the value 2.0 in the Watch window. Also the global `*last-value*` variable displays the association list constructed by `gp:getPointInput`.

Your first debugging session is complete. But don't forget your program is still in suspended animation.

To complete this lesson



- 1 Press the Continue button on the Debug toolbar. Respond to the prompts. This runs the program to completion.

- 2 Choose Debug ► Clear All Breakpoints from the VLISP menu. Respond "yes" to the prompt. This removes all the breakpoints within your code.

Remember: you can remove individual breakpoints by positioning the cursor at the breakpoint and pressing the Toggle Breakpoint button.

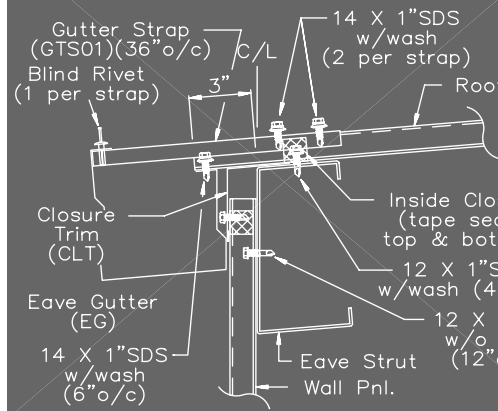
Wrapping Up Lesson 2

In this lesson, you

- Learned about local and global variables.
- Set and removed breakpoints in a program.
- Stepped through a program while it was executing.
- Watched and dynamically changed the value of program variables during execution.
- Saw how local variables are reset to `nil` after the function that defined them completes its run.

The tools you learned in this lesson will be part of your daily work if you intend to develop AutoLISP applications with VLISP.

Drawing the Path Boundary



In This Lesson

3

In this lesson, you will expand your program so it actually draws something within AutoCAD—the polyline outline of the garden path. To draw the border, you must create some utility functions that are not specific to a single application but are general in nature and may be recycled for later use. You will also learn about writing functions that accept arguments—data that is passed to the function from the outside—and why the use of arguments is a powerful programming concept. By the end of the lesson, you will draw an AutoCAD shape parametrically, which means dynamically drawing a shape based on the unique data parameters provided by the user.

- Planning Reusable Utility Functions
- Drawing AutoCAD Entities
- Enabling the Boundary Outline Drawing Function
- Wrapping Up Lesson 3

Planning Reusable Utility Functions

Utility functions perform tasks common to many applications you will be writing. These functions form a tool kit you can use over and over again.

When you create a function as part of a tool kit, spend some time documenting it thoroughly. In your comments, also note the features you would like to add to the function in the future, should time permit.

Converting Degrees to Radians

You will now create a function to prevent you from repetitively typing an equation. It looks like this:

```
(defun Degrees->Radians (numberOfDegrees)
  (* pi (/ numberOfDegrees 180.0)))
```

This function is called **Degrees->Radians**. The function name indicates its purpose.

Why do you need a function to convert angular measurements? Behind the scenes, AutoCAD uses radian angular measurement to keep track of angles, whereas most people think in terms of degrees. This function in your toolkit allows you to think in degrees, and lets AutoLISP convert those numbers to radians.

To test the utility function

- 1 Enter the following at the VLISP Console prompt:

```
(defun Degrees->Radians (numberOfDegrees)
  (* pi (/ numberOfDegrees 180.0)))
```

- 2 Enter the following at the VLISP Console prompt:

```
(degrees->radians 180)
```

The function returns the number 3.14159. According to how this function works, 180 degrees is equivalent to 3.14159 radians. (Does that number ring a bell? If so, treat yourself to a piece of pi.)

To use this function within your program, simply copy the function definition from the Console window into your *gpmain.lsp* file. You can paste it anywhere in the file, as long as you do not paste it into the middle of an existing function.



To clean up your work, select the text you just pasted in, then press the Format Selection button; VLISP will properly indent and format the code.

Next, add some comments describing the function. When you have fully documented the function, your code should look something like this:

```
;;;-----;
;;;      Function: Degrees->Radians      ;
;;;-----;
;;; Description: This function converts a number representing an ;
;;;              angular measurement in degrees, into its radian ;
;;;              equivalent. There is no error checking on the ;
;;;              numberOfDegrees parameter -- it is always ;
;;;              expected to be a valid number. ;
;;;-----;
(defun Degrees->Radians (numberOfDegrees)
  (* pi (/ numberOfDegrees 180.0))
)
```

Converting 3D Points to 2D Points

Another useful function in the garden path program converts 3D points to 2D points. AutoCAD usually works with 3D coordinates, but some entities, such as lightweight polylines, are always meant to be 2D. The points returned by the `getpoint` function are 3D, so you need to create a function to convert them.

To convert a 3D point to a 2D point

- 1 Enter the following at the Console window prompt:

```
(defun 3dPoint->2dPoint (3dpt)(list (car 3dpt) (cadr 3dpt)))
```

- 2 Test the function by entering the following at the Console prompt:

```
(3dpoint->2dpoint (list 10 20 0))
```

This works, but there is another consideration for the garden path application. Although it often doesn't matter whether a number is an integer or a real in LISP functions, this isn't the case with ActiveX functions, which you'll use later in this lesson. ActiveX functions require real numbers. You can easily modify the function to ensure it returns reals instead of integers.

- 3 Enter the following code at the Console prompt:

```
(defun 3dPoint->2dPoint (3dpt)(list (float(car 3dpt))
  (float(cadr 3dpt))))
```

- 4 Run the function again:

```
(3dpoint->2dpoint (list 10 20 0))
```

Notice the return values are now reals (indicated by the decimal values).

- 5 Test the function again, this time using the **getpoint** function. Enter the following at the Console prompt:

```
(setq myPoint(getpoint))
```

- 6 Pick a point in the AutoCAD graphics window.

The **getpoint** function returns a 3D point.

- 7 Enter the following at the Console prompt:

```
(3dPoint->2Dpoint myPoint)
```

Note the 2D point returned.

Now add the function to the *gpmain.lsp* file, just as you did with **Degrees->Radians**. The new code should look like the following:

```
;;;-----;
;;; Function: 3dPoint->2dPoint ;
;;;-----;
;;; Description: This function takes one parameter representing a ;
;;;              3D point (list of three integers or reals), and ;
;;;              converts it into a 2D point (list of two reals). ;
;;;              There is no error checking on the 3D point ;
;;;              parameter -- it is assumed to be a valid point. ;
;;;-----;
;;; To do: Add some kind of parameter checking so that this ;
;;;       function won't crash a program if it is passed a ;
;;;       null value, or some other kind of data type than a ;
;;;       3D point. ;
;;;-----;
(defun 3dPoint->2dPoint (3dpt)
  (list (float(car 3dpt)) (float(cadr 3dpt)))
)
```

Note that the function heading includes a comment about some work you should do on this function in the future. If you want to earn some extra credit, think about how you would go about foolproofing this function so that invalid data does not make it crash.

Hint: **numberp** and **listp** functions...

```
(listp '(1 1 0)) => T
(numberp 3.4) => T
```

Drawing AutoCAD Entities

Most AutoLISP programs draw entities using one of several methods:

- ActiveX functions
- The **entmake** function
- The **command** function

This lesson focuses on entity creation via ActiveX. In Lesson 5, you will implement the **entmake** and AutoCAD command alternatives.

Creating Entities Using ActiveX Functions

The newest way of creating entities is by using the ActiveX functions within VLISP. ActiveX has several advantages over **entmake** and **command**.

- ActiveX functions are faster.
- ActiveX function names indicate the action they perform, resulting in easier readability, maintenance, and bug-fixing.

You will see an example of an ActiveX function later in this lesson.

Using entmake to Build Entities

The **entmake** function allows you to build an entity by gathering values for things such as coordinate location and orientation, layer, and color into an association list, then asking AutoCAD to build the entity for you. The association list you build for the **entmake** function looks very much like the association list you get back when you call the **entget** function. The difference is that **entget** returns information about an entity, while **entmake** builds a new entity from raw data.

Using the AutoCAD Command Line

When AutoLISP first appeared in AutoCAD, the only available means for entity creation was the **command** function. This allows an AutoLISP programmer to code just about any command that can be executed from the AutoCAD Command prompt. This is reliable, but it is not as fast as ActiveX methods and does not provide the flexibility of **entmake**.

Enabling the Boundary Outline Drawing Function

After the last lesson, the `gp:drawOutline` function looked like the following:

```
;;;-----;
;;;      Function: gp:drawOutline                               ;
;;;-----;
;;; Description: This function draws the outline of the       ;
;;;              garden path.                                 ;
;;;-----;
(defun gp:drawOutline ()
  (alert
    (strcat "This function will draw the outline of the polyline "
            "\nand return a polyline entity name/pointer."
            )
    )
  )
  ;; For now, simply return a quoted symbol. Eventually, this
  ;; function will return an entity name or pointer.
  'SomeEname
)
```

As it exists, the code does not do much. However, using the association list information stored in the variable `gp_PathData`, you have enough information to calculate the points for the path boundary. You now have to determine how to pass the information in that variable to `gp:drawOutline`.

Remember `gp_PathData` is a local variable defined within the `C:GPath` function. In AutoLISP, local variables declared in one function are visible to any function called from that function (refer to “Differentiating between Local and Global Variables” on page 14 for clarification). The `gp:drawOutline` function is called from within `C:GPath`. You can refer to the `gp_PathData` variable in `gp:drawOutline`, but this is not a good programming practice.

Why? When the two functions using the same variable are defined in the same file, as in the examples shown so far, it is not too difficult to figure out where the variable is defined and what it is used for. But if the functions are defined in different files—as is often the case—you would have to search through both files to figure out what `gp_PathData` represents.

Passing Parameters to Functions

A better way to convey information from one function to another is to pass parameters to the called function. Design the function so it expects to receive a number of values. Remember the `Degrees->Radians` function? This function is passed a parameter named `numberOfDegrees`:

```
(defun Degrees->Radians (numberOfDegrees)
  (* pi (/ numberOfDegrees 180.0)))
```

When you call the function, it expects you to pass it a number. The number within **Degrees->Radians** is declared as the parameter named `numberOfDegrees`. For example:

```
_ $ (degrees->radians 90)
1.5708
```

In this case, the number 90 is assigned to the parameter `numberOfDegrees`.

You can also pass a variable to a function. For example, you might have a variable called `aDegreeValue` that contains the number 90. The following commands set `aDegreeValue` and pass the variable to **Degrees->Radians**:

```
_ $ (setq aDegreeValue 90)
90
_ $ (degrees->radians aDegreeValue)
1.5708
```

Working with an Association List

You can pass the association list in the `gp_PathData` variable to the **gp:drawOutline** function by invoking the function as follows:

```
(gp:drawOutline gp_PathData)
```

Simple enough, but you also need to figure out how to process the information stored in the association list. The VLISP Inspect feature can help you determine what to do.

To use the VLISP Inspect feature to analyze your association list

- 1 Load the code that is in the text editor window.
- 2 Enter the following expression at the Console prompt:

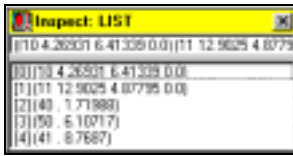
```
(setq BoundaryData (gp:getPointInput))
```

VLISP will store the information you provide in a variable named `BoundaryData`.
- 3 Respond to the prompts for start point, endpoint, and half width.
- 4 Select the `BoundaryData` variable name in the Console window by double-clicking it.



- 5 Choose View ► Inspect from the VLISP menu.

VLISP displays a window like the following:



The Inspect window shows you each sublist within the `BoundaryData` variable.

- 6 Enter the following at the VLISP Console prompt:

```
(assoc 50 BoundaryData)
```

The `assoc` function returns the entry in the association list that is identified by the specified key. In this example, the specified key is 50; this is associated with the angle of the garden path (see “Putting Association Lists to Use” on page 18 for a list of the key-value pairs defined for this application).

- 7 Enter the following at the VLISP Console prompt:

```
(cdr (assoc 50 BoundaryData))
```

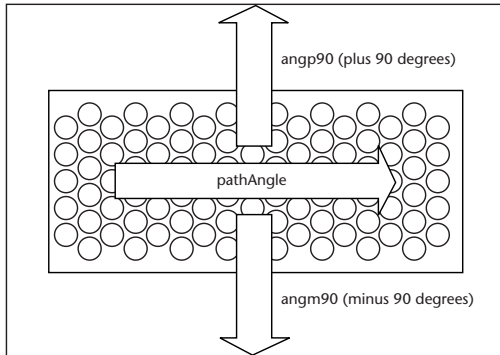
The `cdr` function returns the second element, and any remaining elements after that, from a list. In this example, `cdr` retrieves the angle value, which is the second and last element in the entry returned by the `assoc` function.

By this point, you should have no trouble understanding the following code fragment:

```
(setq PathAngle (cdr (assoc 50 BoundaryData))
      Width      (cdr (assoc 40 BoundaryData))
      HalfWidth  (/ Width 2.00)
      StartPt    (cdr (assoc 10 BoundaryData))
      PathLength (cdr (assoc 41 BoundaryData)))
```

Using Angles and Setting Up Points

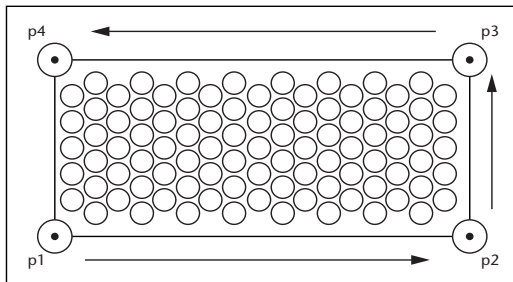
There are still a couple of issues remaining. First, you need to figure out how to draw the path at any angle the user specifies. From the `gp:getPointInput` function, you can easily establish the primary angle of the path. To draw it, you need a couple of additional vectors perpendicular to the primary angle.



This is where the **Degrees->Radians** function is useful. The following code fragment demonstrates how you can set up your two perpendicular vectors using the `PathAngle` variable as an argument passed to the **Degrees->Radians** function:

```
(setq angp90 (+ PathAngle (Degrees->Radians 90))
      angm90 (- PathAngle (Degrees->Radians 90)))
```

With the data you now have in hand, you can establish the four corner points of the path using **polar** function:



```
(setq p1 (polar StartPt angm90 HalfWidth)
      p2 (polar p1 PathAngle PathLength)
      p3 (polar p2 angp90 Width)
      p4 (polar p3 (+ PathAngle (Degrees->Radians 180)))
```

The **polar** function returns a 3D point at a specified angle and distance from a point. For instance, **polar** locates `p2` by projecting `p1` a distance of `PathLength` along a vector oriented at an angle of `PathAngle`, counter-clockwise from the `x`-axis.

Understanding the ActiveX Code in `gp:drawOutline`

The `gp:drawOutline` function issues ActiveX calls to display the path's polyline border in AutoCAD. The following code fragment uses ActiveX to draw the border:

```
;; Add polyline to the model space using ActiveX automation.
(setq pline (vla-addLightweightPolyline
  *ModelSpace* ; Global Definition for Model Space
  VLADataPts ; vertices of path boundary
  ) ;_ end of vla-addLightweightPolyline

  ) ;_ end of setq

(vla-put-closed pline T)
```

How do you make sense of this code? An essential resource is the *ActiveX and VBA Reference*, which describes the methods and properties accessible to ActiveX clients such as this garden path application. The “Working with ActiveX” chapter of the *Visual LISP Developer's Guide* explains how to translate the VBA™ syntax in the *ActiveX and VBA Reference* into ActiveX calls in AutoLISP syntax.

For the moment, though, you can gain a rudimentary understanding by scrutinizing the pattern of the two `vla-` calls in the preceding example. The names of all AutoLISP ActiveX functions that work on AutoCAD objects are prefixed with `vla-`. For example, `addLightweightPolyline` is the name of an ActiveX method, and `vla-addLightweightPolyline` is the AutoLISP function that invokes this method. The `vla-put-closed` call updates the `closed` property of the `pline` object, the polyline drawn by `vla-addLightweightPolyline`.

The Automation objects that factor into AutoLISP ActiveX calls abide by a few standard rules:

- The first argument to a `vla-put`, `vla-get`, or `vla-` method call is the object being modified or queried, for example, `*ModelSpace*` in the first function call and `pline` in the second call.
- The return value of a `vla-` method call is a *VLA-object*, which can be used in subsequent calls. For example, `vla-addLightweightPolyline` yields a return object, `pline`, that is altered in the next ActiveX call.
- The ActiveX object model is structured hierarchically. Objects are traversed from the application object at the topmost level down to individual drawing primitives, such as polyline and circle objects. Thus, the `gp:drawOutline` function is not yet complete, because the `*ModelSpace*` automation object must first be accessed via the root application object.

Ensuring That ActiveX Is Loaded

ActiveX functionality is not automatically enabled when you start AutoCAD or VLISP, so your programs must ensure that ActiveX is loaded. The following function call accomplishes this:

```
(vl-load-com)
```

If ActiveX support is not yet available, executing **vl-load-com** initializes the AutoLISP ActiveX environment. If ActiveX is already loaded, **vl-load-com** does nothing.

Obtaining a Pointer to Model Space

When you add entities through ActiveX functions, you need to identify the model space or paper space in which the entity is to be inserted. (In ActiveX terminology, entities are objects, but this tutorial will continue using the term entity.) To tell AutoCAD which space the new entities should occupy, you need to obtain a pointer to that space. Unfortunately, obtaining a pointer to model space is not a simple, single-shot function. The following code fragment shows how the operation needs to be set up:

```
(vla-get-ModelSpace (vla-get-ActiveDocument  
  (vlax-get-Acad-Object)))
```

Working from the inside out, the **vlax-get-Acad-Object** function retrieves a pointer to AutoCAD. This pointer is passed to the **vla-get-ActiveDocument** function, which retrieves a pointer to the active drawing (document) within AutoCAD. The Active Document pointer is then passed to the **vla-get-ModelSpace** function that retrieves a pointer to the model space of the current drawing.

This is not the kind of expression you want to type over and over. For example, look at how much more complicated the code for adding a polyline using ActiveX appears when the entire model space expression is used:

```
(setq pline (vla-addLightweightPolyline  
  (vla-get-ModelSpace  
    (vla-get-ActiveDocument  
      (vlax-get-Acad-Object)  
    )  
  )  
  VLADataPts)  
)  
(vla-put-closed pline T)
```

The function is definitely less understandable. Not only that, but within every expression within your program where an entity is created, you repeat the same set of nested functions. This demonstrates one of the few excellent

uses for global variables. The garden path application can add a lot of entities to model space (think of all the tiles in the path), so, set up a global variable to store the pointer to the model space, as in the following code:

```
(setq *ModelSpace* (vla-get-ModelSpace (vla-get-ActiveDocument  
(vlax-get-Acad-Object))))
```

You can use the variable `*ModelSpace*` anytime you call an ActiveX entity creation function. The only tricky thing with this scheme is the `*ModelSpace*` variable must be ready to go before you start drawing. For this reason, the `setq` establishing this variable will be called at the time the application is loaded, immediately after the call to `vl-load-com`. These calls will be placed before any `defun` in the program file. As a result, they are executed as soon as the file is loaded.

Constructing an Array of Polyline Points

The last issue to deal with is how to transform the individual point variables—`p1`, `p2`, `p3`, and `p4`—into the format required for the `vla-addLightweightpolyline` function. First, get some help on the topic.

To obtain information on a function



- 1 Press the Help button on the VLISP toolbar.
- 2 Enter `vla-addLightweightpolyline` in the Enter Item Name dialog box, and press OK. (The Help system is not case sensitive, so do not worry about how you capitalize the function name.)

Online Help states that `AddLightWeightPolyline` requires you to specify the polyline vertices as an array of doubles in the form of a variant. Here is how Help describes this parameter:

The array of 2D WCS coordinates specifying the vertices of the polyline. At least two points (four elements) are required for constructing a lightweight polyline. The array size must be a multiple of 2.

A variant is an ActiveX construct that serves as a container for various types of data. Strings, integers, and arrays can all be represented by variants. The variant stores data along with the information identifying the data.

So far, you have four points, each in the format (x, y, z). The challenge is to convert these four points into a list of the following form:

```
(x1 y1 x2 y2 x3 y3 x4 y4)
```

The **append** function takes multiple lists and concatenates them. To create a list of the four points in the proper format for the ActiveX function, you can use the following expression:

```
(setq polypoints (append (3dPoint->2dPoint p1)
                        (3dPoint->2dPoint p2)
                        (3dPoint->2dPoint p3)
                        (3dPoint->2dPoint p4)))
```

Writing the **3dPoint->2dPoint** function four times is a bit cumbersome. You can reduce the code further by using the **mapcar** and **apply** functions. When selected, **mapcar** executes a function on individual elements in one or more lists, and **apply** passes a list of arguments to the specified function. The resulting code looks like the following:

```
(setq polypoints (apply 'append (mapcar '3dPoint->2dPoint
                                       (list p1 p2 p3 p4))))
```

Before the call to **mapcar**, the list of points is in this form:

```
((x1 y1 z1) (x2 y2 z2) (x3 y3 z3) (x4 y4 z4))
```

After **mapcar** you have a list of points in the following form:

```
((x1 y1) (x2 y2) (x3 y3) (x4 y4))
```

And finally, after applying the **append** function on the list returned from **mapcar**, you end up with the following:

```
(x1 y1 x2 y2 x3 y3 x4 y4)
```

Constructing a Variant from a List of Points

So far, the data in the **polypoints** variable is in a list format suitable for many AutoLISP calls. However, the data is to be supplied as an input parameter to an ActiveX call that expects a variant array of doubles. You can use another utility function to make the required conversion from list to variant:

```
(defun gp:list->variantArray (ptsList / arraySpace sArray)
  ; allocate space for an array of 2d points stored as doubles
  (setq arraySpace (vlax-make-safearray
                    vlax-vbdouble ; element type
                    (cons 0
                          (- (length ptsList) 1)
                          ) ; array dimension
                    )
  )
  (setq sArray (vlax-safearray-fill arraySpace ptsList))
  ; return array variant
  (vlax-make-variant sArray)
  )
```

The following actions take place in `gp:list->variantArray`:

- The `v1ax-make-safearray` function is called to allocate an array of doubles (`v1ax-vdouble`). The `v1ax-make-safearray` function also requires you to specify the lower and upper index boundaries of the array. In `gp:list->variantArray`, the call to `v1ax-make-safearray` specifies a start index of 0 and sets the upper limit to one less than the number of elements passed to it (`ptsList`).
- The `v1ax-safearray-fill` function is called to populate the array with the elements in the point list.
- The `v1ax-make-variant` is called to convert the safearray into a variant. As the last function call in `gp:list->variantArray`, the return value is passed to the calling function.

The following is an example of a function call that invokes `gp:list->variantArray` to convert a list to a variant array of doubles:

```
; data conversion from list to variant
(setq VLADDataPts (gp:list->variantArray polypoints))
```

Putting It All Together

You now have all the code you need to draw the outline of the garden path.

To update your code

- 1 Replace your old code for the `gp:drawOutline` function with the following:

```
;;;-----
;;;      Function: gp:drawOutline
;;;-----
;;; Description: This function will draw the outline of the garden
;;;              path.
;;;-----
;;; Note: No error checking or validation is performed on the
;;; BoundaryData parameter. The sequence of items within this
;;; parameter does not matter, but it is assumed that all sublists
;;; are present and contain valid data.
;;;-----
(defun gp:drawOutline (BoundaryData / VLADDataPts PathAngle
  Width HalfWidth StartPt PathLength
  angm90  angp90    p1 p2
  p3 p4    polypoints pline
  )
```

```

;; extract the values from the list BoundaryData
(setq PathAngle (cdr (assoc 50 BoundaryData))
Width (cdr (assoc 40 BoundaryData))
HalfWidth (/ Width 2.00)
StartPt (cdr (assoc 10 BoundaryData))
PathLength (cdr (assoc 41 BoundaryData))
angp90 (+ PathAngle (Degrees->Radians 90))
angm90 (- PathAngle (Degrees->Radians 90))
p1 (polar StartPt angm90 HalfWidth)
p2 (polar p1 PathAngle PathLength)
p3 (polar p2 angp90 Width)
p4 (polar p3 (+ PathAngle (Degrees->Radians 180)) PathLength)
polypoints (apply 'append
(mapcar '3dPoint->2dPoint (list p1 p2 p3 p4))
)
)
;; ***** data conversion *****
;; Notice, polypoints is in AutoLISP format, consisting of a list
;; of the 4 corner points for the garden path.
;; The variable needs to be converted to a form of input parameter
;; acceptable to ActiveX calls.
(setq VLADDataPts (gp:list->variantArray polypoints))

;; Add polyline to the model space using ActiveX automation.
(setq pline (vla-addLightweightPolyline
*ModelSpace*; Global Definition for Model Space
VLADDataPts
) ;_ end of vla-addLightweightPolyline

) ;_ end of setq
(vla-put-closed pline T)
;; Return the ActiveX object name for the outline polyline
;; The return value should look something like this:
;; #<VLA-OBJECT IAcadLWPolyline 02351a34>
pline
) ;_ end of defun

```

Note that **gp:drawOutline** now returns the variable `pline`, not the quoted symbol `'SomeEname` used in the stubbed-out version of the function.

- 2 Format the code you just entered by selecting it and pressing the Format Selection button on the VLISP toolbar.

- 3 Enable ActiveX and add the global variable assignment for the pointer to model space, as described earlier. Scroll to the top of the text editor window and add the following code before the first **defun**:

```
;;;-----  
;;; First step is to load ActiveX functionality. If ActiveX support  
;;; already exists in document (can occur when Bonus tools have been  
;;; loaded into AutoCAD), nothing happens. Otherwise, ActiveX  
;;; support is loaded.  
;;;-----  
  
(vl-load-com)  
  
;;; In Lesson 4, the following comment and code is moved to utils.lsp  
;;;-----  
;;; For ActiveX functions, we need to define a global variable that  
;;; "points" to the Model Space portion of the active drawing. This  
;;; variable, named *ModelSpace* will be created at load time.  
;;;-----  
(setq *ModelSpace*  
      (vla-get-ModelSpace  
        (vla-get-ActiveDocument (vlax-get-acad-object))  
        ) ;_ end of vla-get-ModelSpace  
      ) ;_ end of setq
```

Note how the above code lives outside of any **defun**. Because of this, VLISP automatically executes the code at the time you load the file.

- 4 Look for the following line in the **c:GPath** function:

```
(setq PolylineName (gp:drawOutline))
```

Change it to the following:

```
(setq PolylineName (gp:drawOutline gp_PathData))
```

The **gp:drawOutline** function is now expecting a parameter—the list containing the polyline boundary data—and this change fulfills that requirement.

- 5 Add the **gp:list->variantArray** function shown in “Constructing a Variant from a List of Points” on page 45 to the end of *gpmain.lsp*.

Try loading and running the revised program. VLISP takes control away from AutoCAD before you see the end result, so switch back to the AutoCAD window after control returns to VLISP. If the program ran correctly, you should see a border for the garden path. If you find errors, debug the code and try again.

Wrapping Up Lesson 3

In this lesson, you

- Wrote utility functions that can be reused in other applications.
- Added entity creation logic to your program.
- Learned how to use ActiveX functions.
- Learned how to work with association lists.
- Enabled your program to draw a garden path border.

If you're confused about anything from this lesson, it is recommended you go through it once again before moving on to Lesson 4. (If you decide to do so, copy the completed code from your *Lesson2* directory so that you start the lesson from the correct place.) And if all else fails, you can always copy the code from the *Tutorial\VisualLISP\Lesson3* directory.

Creating a Project and Adding the Interface

In this lesson, you will accomplish two major tasks: creating a VLISP project and adding a dialog-based interface to your application. In the process, you will split the single AutoLISP file you worked with so far (*gpmain.lsp*) into several smaller files, reinforcing the concept of code modularity.

From this lesson on, the tutorial provides more general descriptions of the tasks you need to perform, unless new topics are covered. Also, the code fragments will be minimally documented to save space. This, however, does not absolve you from your moral obligation to fully document your code!

In This Lesson

4

- Modularizing Your Code
- Using Visual LISP Projects
- Adding the Dialog Box Interface
- Interacting with the Dialog Box from AutoLISP Code
- Providing a Choice of Boundary Line Type
- Cleaning Up
- Running the Application
- Wrapping Up Lesson 4

Modularizing Your Code

As a result of the work you did in Lesson 3, your *gpmain.lsp* file was getting rather large. This is not a problem for VLISP, but it is easier to maintain the code if you split things up into files containing logically related functions. It's also easier to debug your code. For example, if you have a single file with 150 functions, a single missing parenthesis can be difficult to find.

In the tutorial, the files will be organized as follows:

Tutorial file organization

File name	Contents
GP-IO.LSP	All input and output (I/O) functions) such as getting user input. Also contains the AutoLISP code required for the dialog box interface you will be adding.
UTILS.LSP	Includes all generic functions that can be used again on other projects. Also contains load-time initializations.
GPDRAW.LSP	All drawing routines—the code that actually creates the AutoCAD entities.
GPMAIN.LSP	The basic C:GPath function.

To split *gpmain.lsp* into four files

- 1 Create a new file, then cut and paste the following functions from *gpmain.lsp* into the new file:

- **gp:getPointInput**
- **gp:getDialogInput**

Save the new file in your working directory as *gp-io.lsp*.

- 2 Create a new file, then cut and paste the following functions from *gpmain.lsp* into the new file:

- **Degrees->Radians**
- **3Dpoint->2Dpoint**
- **gp:list->variantArray**

Also, at the beginning of the file, insert the lines of code to establish ActiveX functionality (`v1-load-com`) and commit global variable assignment (`*ModelSpace*`).

Save the file as *utils.lsp*.

- 3 Create a new file, then cut and paste the following function from *gpmain.lsp* into the new file:

■ `gp:drawOutline`

Save this file as *gpdraw.lsp*.

- 4 After stripping the code out of *gpmain.lsp*, save it and check it. Only the original function, `c:GPath`, should remain in the file.



Your VLISP desktop is starting to get crowded. You can minimize any window within VLISP and it stays accessible. Press the Select Window button on the toolbar to choose a window from a list, or choose Window from the VLISP menu and select a window to view.

Using Visual LISP Projects

The VLISP project feature provides a convenient way to manage the files that make up your application. And with the project feature, you can open a single project file instead of individually opening every LISP file in the application. Once the project is open, getting to its constituent files is a double-click away.

To create a VLISP project

- 1 Choose Project ► New Project from the VLISP menu.
- 2 Save the file in your *Lesson4* directory, using the name *gpath.prj*.
After you save the file, VLISP displays the Project Properties dialog box.



3 Press the [Un]Select All button on the left in the Project Properties dialog box.



4 Press the button containing an arrow pointing to the right. This adds all the selected files to your project.

In the Project Properties dialog box, the list box on the left shows all LISP files that reside in the same directory as your project file and are *not* included in that project. The list box on the right lists all the files that make up the project. When you add the selected files to the project, those file names move from the left box to the right box.

5 In the list box on the right side of the dialog box, select *gpmain*, then press the Bottom button. This moves the file to the bottom of the list.

VLISP loads project files in the order they are listed. Because the prompt that tells users the name of the command is located at the end of the *gpmain.lsp* file, you need to move this file to the bottom of the list. Loading this file last results in the prompt displayed to users. The *utils.lsp* file should be loaded first because it contains initialization code for the application. Therefore, select *utils* in the dialog's list box and press the Top button.

6 Press OK.



VLISP adds a small project window to your VLISP desktop. The window lists the files in your project. Double-click on any file to open the file in the VLISP text editor (if it is not already open) and make it the active editor window.

Adding the Dialog Box Interface

The next part of this lesson concerns adding a dialog box interface to the garden path application. To do this, you will be working with another language, dialog control language (DCL).

Currently, your `gpath` function only accepts input at the Command line. You included a stubbed-out function (`gp:getDialogInput`) with the intention of adding a dialog box interface. Now is the time to add the interface.

There are two steps in creating a functional dialog interface:

- Define the appearance and contents of the dialog boxes.
- Add program code to control dialog behavior.

The description and format of a dialog box is defined in a `.dcl` file. In the Visual LISP Developer's Guide, DCL is described in chapter 11, "Designing Dialog Boxes," chapter 12, "Managing Dialog Boxes," and chapter 13, "Programmable Dialog Box Reference."

Program code that initializes default settings and responds to user interaction will be added to `gp:getDialogInput`.

Defining the Dialog Box with DCL

Begin by taking a look at the dialog box you need to create.



The dialog box contains the following elements:

- Two sets of radio buttons.
One set of buttons determines the polyline style of the boundary, and the other set of buttons specifies the tile entity creation method (ActiveX, **entmake**, or **command**). Only one radio button in a set can be selected at one time.
- Edit boxes for specifying the radius of tiles and the spacing between tiles.
- A standard set of OK and Cancel buttons.

Dialog box components are referred to as *tiles* in DCL. Writing the complete contents of a dialog box DCL file may seem overwhelming. The trick is to sketch out what you want, break it down into sections, then write each section.

To define the dialog box

- 1 Open a new file in the VLISP text editor window.
- 2 Enter the following statement in the new file:

```
label = "Garden Path Tile Specifications";
```

This DCL statement defines the title of the dialog box window.

- 3 Define the radio buttons for specifying polyline type by adding the following code:

```
: boxed_radio_column { // defines the radio button areas
  label = "Outline Polyline Type";
  : radio_button { // defines the lightweight radio button
    label = "&Lightweight";
    key = "gp_lw";
    value = "1";
  }
  : radio_button { // defines the old-style polyline radio button
    label = "&Old-style";
    key = "gp_hw";
  }
}
```

The `boxed_radio_column` DCL directive defines a box boundary and allows you to specify a label for the set of buttons. Within the boundary, you specify the radio buttons you need by adding `radio_button` directives. Each radio button requires a label and a key. The key is the name by which your AutoLISP code can refer to the button.

Notice that the radio button labeled "lightweight" is given a value of 1. A value of 1 (a string, not an integer) assigned to a button makes it the default choice in a row of buttons. In other words, when you first display the dialog, this button will be selected. Also notice that in DCL files, double-slash characters, not semicolons as in AutoLISP, indicate a comment.

- 4 Define the radio column for the selection of the entity creation style by adding the following code:

```
: boxed_radio_column {          // defines the radio button areas
  label = "Tile Creation Method";
  : radio_button {              // defines the ActiveX radio button
    label = "&ActiveX Automation";
    key = "gp_actx";
    value = "1";
  }
: radio_button {                // defines the (entmake) radio button
  label = "&Entmake";
  key = "gp_emake";
}
: radio_button {                // defines the (command) radio button
  label = "&Command";
  key = "gp_cmd";
}
}
```

- 5 Add the following code to define the edit box tiles that allow users to enter the numbers specifying tile size and spacing:

```
: edit_box {                    // defines the Radius of Tile edit box
  label = "&Radius of tile";
  key = "gp_trad";
  edit_width = 6;
}
: edit_box {                    // defines the Spacing Between Tiles edit box
  label = "S&pacings between tiles";
  key = "gp_spac";
  edit_width = 6;
}
```

Notice that this definition does not set any initial values for the edit boxes. You will set default values for each edit box in your AutoLISP program.

- 6 Add the following code for the OK and Cancel buttons:

```
: row {                          // defines the OK/Cancel button row
  : spacer { width = 1; }
  : button {                      // defines the OK button
    label = "OK";
    is_default = true;
    key = "accept";
    width = 8;
    fixed_width = true;
  }
  : button {                      // defines the Cancel button
    label = "Cancel";
    is_cancel = true;
    key = "cancel";
    width = 8;
    fixed_width = true;
  }
  : spacer { width = 1; }
}
```

Both buttons are defined within a row, so they line up horizontally.

- 7 Scroll to the beginning of the text editor window and insert the following statement as the first line in your DCL:

```
gp_mainDialog : dialog {
```

- 8 The `dialog` directive requires a closing brace, so scroll to the end of the file and add the brace as the last line of DCL code:

```
}
```

Saving a DCL File

Before saving the file containing your DCL, consider the fact that AutoCAD must be able to locate your DCL file during runtime. For this reason, the file must be placed in one of the AutoCAD Support File Search Path locations. (If you are unsure about these locations, choose Tools ► Options from the AutoCAD menu and examine the Support File Search Path locations under the Files tab.)

For now, you can save the file in the AutoCAD *Support* directory.

To save your DCL file

- 1 Choose File ► Save As from the VLISP menu.
- 2 In the Save As Type field of the Save As dialog box, choose DCL Source Files from the pull-down menu.
- 3 Change the Save In path to *<AutoCAD directory>\Support*.
- 4 Enter the file name *gpdialog.dcl*.
- 5 Press Save.

Notice VLISP changes the syntax coloring scheme after you save the file. VLISP is designed to recognize DCL files and highlight the different types of syntactical elements.

Previewing a Dialog Box

VLISP provides a preview feature for checking the results of your DCL coding.

To preview a dialog box defined with DCL

- 1 Choose Tools ► Interface Tools ► Preview DCL in Editor from the VLISP menu.
- 2 Press OK when prompted to specify a dialog name.

In this case, your DCL file defines just a single dialog box, so there is no choice to be made. As you create larger and more robust applications, however, you may end up with DCL files containing multiple dialog boxes. This

is where you can select which one to preview.

- 3 If the dialog box displays successfully, press any button to end the dialog.

VLISP passes control to AutoCAD to display the dialog box. If AutoCAD finds syntactical errors, it displays one or more message windows identifying the errors.

If AutoCAD detects DCL errors and you are unable to figure out how to fix them, copy the *gpdialog.dcl* file in your *Tutorial\VisualLISP\Lesson4* directory and save it in the *Support* directory.

Interacting with the Dialog Box from AutoLISP Code

You now need to program your AutoLISP function to interact with the dialog box. The stubbed-out function, `gp:getDialogInput`, is where this activity will take place. This function now lives in the *gp-io.lsp* file, which you earlier extracted from *gpmain.lsp*.

Developing a dialog box interface can be confusing the first few times you do it. It involves planning ahead and asking yourself such questions as:

- Does the dialog box need to be set up with default values?
- What happens when the user presses a button or enters a value?
- What happens when the user presses Cancel?
- If the dialog (*.dcl*) file is missing, what needs to occur?

Setting Up Dialog Values

When you run the complete garden path application, notice that the dialog box always starts up with ActiveX as the default object creation method and Lightweight as the polyline style. Something more interesting occurs with the default tile size—the values change depending on the width of the path. The following code fragment shows how to set up the default values to be displayed in the dialog box:

```
(setq  objectCreateMethod "ACTIVE"
      plineStyle "LIGHT"
      tilerad (/ pathWidth 15.0)
      tilespace (/ tilerad 5.0)
      dialogLoaded T
      dialogShow T
) ;_ end of setq
```

For the moment, don't worry about what purpose the `dialogLoaded` and `dialogShow` variables serve. This becomes apparent in the next two sections.

Loading the Dialog File

Your program first needs to load the DCL file using the `load_dialog` function. This function searches for dialog files according to the AutoCAD support file search path, unless you specify a full path name.

For every `load_dialog` function there should be a corresponding `unload_dialog` function later in the code. You will see this in a moment. For now, take a look at how you need to load in your dialog:

```
;; Load the dialog box. Set up error checking to make sure
;; the dialog file is loaded before continuing
(if (= -1 (setq dcl_id (load_dialog "gpdialog.dcl")))
    (progn
      ;; There's a problem - display a message and set the
      ;; dialogLoaded flag to nil
      (princ "\nCannot load gpdialog.dcl")
      (setq dialogLoaded nil)
    ) ;_ end of progn
  ) ;_ end of if
```

The `dialogLoaded` variable indicates if the dialog loaded successfully. In the code where you set up the initial values for the dialog box, you set `dialogLoaded` to an initial value of `t`. As you can see in the code fragment above, `dialogLoaded` is set to `nil` if there is a problem with the load.

Loading a Specific Dialog into Memory

It was noted earlier that a single DCL file may contain multiple dialog box definitions. The next step in using a dialog is to specify which dialog box definition to display. The following code demonstrates this:

```
(if (and dialogLoaded
      (not (new_dialog "gp_mainDialog" dcl_id))
    ) ;_ end of and
    (progn
      ;; There's a problem...
      (princ "\nCannot show dialog gp_mainDialog")
      (setq dialogShow nil)
    ) ;_ end of progn
  ) ;_ end of if
```

Notice how the `and` function is used to test if the dialog was loaded and if the call to `new_dialog` was successful. If there are multiple expressions evaluated within an `and` function call, evaluation of subsequent expressions is terminated with the first expression that evaluates to `nil`. In this case, if the

`dialogLoaded` flag is `nil` (meaning the load function in the previous section failed), VLISP does not attempt to perform the `new_dialog` function.

Notice that the code also accounts for the possibility that something might not be working properly with the DCL file, and sets the `dialogShow` variable to `nil` if that is the case.

The `new_dialog` function simply loads the dialog into memory—it does not display it. The `start_dialog` function displays the dialog box. All dialog box initialization, such as setting tile values, creating images or lists for list boxes, and associating actions with specific tiles must take place after the `new_dialog` call and before the `start_dialog` call.

Initializing the Default Dialog Values

If everything worked successfully in loading the dialog, you are ready to start setting up the values that will be displayed to users. A successful load is indicated if the flag variables `dialogLoaded` and `dialogShow` are both `T` (true).

Now set the initial values for the tile radius and spacing. The `set_tile` function assigns a value to a tile. An edit box deals with strings rather than numbers, so you need to use the `rtos` (convert Real TO String) function to convert your tile size variable values into strings in decimal format with a precision of two digits. The following code handles this conversion:

```
(if (and dialogLoaded dialogShow)
    (progn
      ;; Set the initial state of the tiles
      (set_tile "gp_trad" (rtos tileRad 2 2))
      (set_tile "gp_spac" (rtos tileSpace 2 2))
```

Assigning Actions to Tiles

A DCL definition does nothing more than define a lifeless dialog box. You connect this lifeless dialog box to your dynamic AutoLISP code with the `action_tile` function, as demonstrated by the following code:

```
;; Assign actions (the functions to be invoked) to dialog buttons
(action_tile
  "gp_lw"
  "(setq plineStyle \"Light\")"
)
(action_tile
  "gp_hw"
  "(setq plineStyle \"Pline\")"
)
```

```

(action_tile
  "gp_actx"
  "(setq objectCreateMethod \"ActiveX\")"
)
(action_tile
  "gp_emake"
  "(setq objectCreateMethod \"Entmake\")"
)
(action_tile
  "gp_cmd"
  "(setq objectCreateMethod \"Command\")"
)
(action_tile "cancel" "(done_dialog) (setq UserClick nil)")
(action_tile
  "accept"
  (strcat "(progn (setq tileRad (atof (get_tile \"gp_trad\")))"
    "(setq tileSpace (atof (get_tile \"gp_spac\")))"
    "(done_dialog) (setq UserClick T))")
  )
)

```

Notice all the quotes around the AutoLISP code. When you write an AutoLISP **action_tile** function, your code is essentially telling a tile, “here, remember this string, then pass it back to me when the user selects you.” The string (anything within double-quotation marks) is dormant until the user selects the tile. At that time, the tile passes the string to AutoCAD, which converts the string into functioning AutoLISP code and executes the code.

For example, consider the following **action_tile** expression, which is connected to the lightweight polyline radio button:

```

(action_tile
  "gp_lw"
  "(setq plineStyle \"Light\")"
)

```

The code assigns the string "(setq plineStyle \"Light\")" to the radio button. When a user picks the button, the string is passed back to AutoCAD and transformed directly into the following AutoLISP expression:

```
(setq plineStyle "Light")
```

Look at one more code fragment. The following is the **action_tile** expression assigned to the OK button:

```

(action_tile
  "accept"
  (strcat "(progn (setq tileRad (atof (get_tile \"gp_trad\")))"
    "(setq tileSpace (atof (get_tile \"gp_spac\")))"
    "(done_dialog) (setq UserClick T))")
  )
)

```

When a user presses the OK button, the lengthy string assigned to the button is passed to AutoCAD and turned into the following AutoLISP code:

```
(progn
  (setq tileRad (atof (get_tile "gp_trad")))
  (setq tileSpace (atof (get_tile "gp_spac")))
  (done_dialog)
  (setq UserClick T)
)
```

This code does several things: It retrieves the current values from the tiles whose key values are `gp_trad` (the tile radius) and `gp_spac` (the tile spacing value). Then `atof` converts the number string into a real number. The dialog is terminated with the `done_dialog` function, and a value of `T`, or true, is assigned to the variable `UserClick`.

You're done assigning actions to the buttons. The next thing to do is to put it all in motion.

Starting the Dialog

The `start_dialog` function displays a dialog box and accepts user input. The `start_dialog` function requires no arguments.

```
(start_dialog)
```

Control passes to users when you issue `start_dialog`. Users can make choices within the dialog box, until they press the OK or Cancel buttons.

Unloading the Dialog

When a user presses the OK or Cancel button, you need to unload the dialog. Like `start_dialog`, `unload_dialog` is another simple function.

```
(unload_dialog dcl_id)
```

Determining What to Do Next

If the user pressed OK, you must build a list containing the values set by the user's interaction with the dialog. This list is what `gp:getDialogInput` will return to its calling function. If the user pressed Cancel, the function returns `nil`:

```
(if UserClick          ; User clicked Ok
    ;; Build the resulting data
    (progn
      (setq Result (list
                    (cons 42 tileRad)
                    (cons 43 TileSpace)
                    (cons 3 objectCreateMethod)
                    (cons 4 plineStyle)
                    )
            )
    )
)
```

Putting the Code Together

With the examples above, and a few additional lines, you have the code needed to complete the `gp:getDialogInput` function.

To put `gp:getDialogInput` together

- 1 Open your copy of *gp-io.lsp* in a VLISP text editor window.
- 2 Delete the code in `gp:getDialogInput` (the `defun gp:getDialogInput` statement and everything after it).
- 3 Enter the following `defun` statement as the first line of code in the `gp:getDialogInput` function:

```
(defun gp:getDialogInput (pathwidth / dcl_id objectCreateMethod
                        plineStyle tilerad tilespace result UserClick
                        dialogLoaded dialogShow)
```

The function expects a single argument (`pathwidth`), and establishes a number of local variables.

- 4 Following the code you added in step 3, enter the sample code from each of the following sections of this chapter:
 - “Setting Up Dialog Values”
 - “Loading the Dialog File”
 - “Loading a Specific Dialog into Memory”
 - “Initializing the Default Dialog Values”
 - “Assigning Actions to Tiles”

NOTE Enter just the first code example from “Assigning Actions to Tiles,” not the fragments in the explanations that follow. Those fragments just repeat pieces of the example.

- “Starting the Dialog”
- “Unloading the Dialog”
- “Determining What to Do Next”

5 After the last line of code, add the following:

```
)  
)  
  Result;  
) ;_ end of defun
```



6 Format the code you entered by choosing Tools ► Format Code in Editor from the VLISP menu.

Updating a Stubbed-Out Function

You have now revised the `gp:getDialogInput` function. Whenever you modify a stubbed-out function, you should always check a couple of things:

- Has the `defun` statement changed? That is, does the function still take the same number of arguments?
- Does the function return something different?

In the case of `gp:getDialogInput`, the answer to both questions is yes. The function now accepts the parameter of the path width (to set the default tile size and spacing). And instead of returning `T`, which is the value the stubbed-out version of the function returned, `gp:getDialogInput` now returns an association list containing four new values.

Both changes affect the code that calls the function and the code that handles the return values from the functions. Replace your previous version of the `C:GPath` function in `gpmain.lsp` with the following code:

```
(defun C:GPath (/ gp_PathData gp_dialogResults)  
  ;; Ask the user for input: first for path location and  
  ;; direction, then for path parameters. Continue only if you  
  ;; have valid input. Store the data in gp_PathData.  
  (if (setq gp_PathData (gp:getPointInput))  
      (if (setq gp_dialogResults (gp:getDialogInput (cdr(assoc 40  
                                                             gp_PathData))))
```

```

(progn
  ;; Now take the results of gp:getPointInput and append this
  ;; to the added information supplied by gp:getDialogInput.

  (setq gp_PathData (append gp_PathData gp_DialogResults))

  ;; At this point, you have all the input from the user.
  ;; Draw the outline, storing the resulting polyline
  ;; "pointer" in the variable called PolylineName.
  (setq PolylineName (gp:drawOutline gp_PathData))
  ) ;_ end of progn
(princ "\nFunction cancelled.")
) ;_ end of if
(princ "\nIncomplete information to draw a boundary.")
) ;_ end of if
(princ) ; exit quietly
) ;_ end of defun

```

Take a look at the boldface lines in the revision of the main **C:GPath** function. There are two essential changes to make the program work correctly:

- When the **gp:getDialogInput** function is invoked, the path width is passed to it. This is done by extracting the value associated with the key 40 index of the **gp_PathData** association list.
- The association list returned by **gp:getPointInput** is assigned to a variable called **gp_dialogResults**. If this variable has a value, its content needs to be appended to the association list values already stored in **gp_PathData**.

There are additional changes in the code resulting from the replacement of placeholders in the stubbed-out version. The easiest thing to do is copy this code from the online tutorial and paste it into your file.

Providing a Choice of Boundary Line Type

One requirement specified for the garden path application was to allow users to draw the boundary outline as either a lightweight polyline or an old-style polyline. The first version of **gp:drawOutline** you wrote always used a lightweight polyline to draw the boundary. Now that the dialog box interface is ready to go, you can build in the option for drawing an old-style polyline as well. To accomplish this, **gp:drawOutline** must determine what kind of polyline to draw, and then it must draw it.

The necessary changes to **gp:drawOutline** are included in the following code fragment. Make the modification from the *gpdraw.lsp* file indicated in bold:

```

(setq PathAngle (cdr (assoc 50 BoundaryData))
      Width (cdr (assoc 40 BoundaryData))
      HalfWidth (/ Width 2.00)
      StartPt (cdr (assoc 10 BoundaryData))
      PathLength (cdr (assoc 41 BoundaryData))
      angp90 (+ PathAngle (Degrees->Radians 90))
      angm90 (- PathAngle (Degrees->Radians 90))
      p1 (polar StartPt angm90 HalfWidth)
      p2 (polar p1 PathAngle PathLength)
      p3 (polar p2 angp90 Width)
      p4 (polar p3 (+ PathAngle (Degrees->Radians 180))
              PathLength)
      poly2Dpoints (apply 'append
                          (mapcar '3dPoint->2dPoint (list p1 p2 p3 p4))
                          )
      poly3Dpoints (mapcar 'float (append p1 p2 p3 p4))
      ;; get the polyline style
      plineStyle (strcase (cdr (assoc 4 BoundaryData)))
) ;_ end ofsetq
;; Add polyline to the model space using ActiveX automation
(setq pline (if (= plineStyle "LIGHT")
               ;; create a lightweight polyline
               (vla-addLightweightPolyline
                *ModelSpace* ; Global Definition for Model Space
                (gp:list->variantArray poly2Dpoints) ;data conversion
                ) ;_ end of vla-addLightweightPolyline
               ;; or create an old-style polyline
               (vla-addPolyline
                *ModelSpace*
                (gp:list->variantArray poly3Dpoints) ;data conversion
                ) ;_ end of vla-addPolyline
               ) ;_ end of if
) ;_ end ofsetq

```

Typing the changes into your code can be very tricky, as you not only need to add code but also to delete some existing lines and rearrange others. It is recommended you copy the entire `setq` statement from the online tutorial and paste it into your code.

Cleaning Up

If you have not done so already, delete the following chunk of code from the `C:GPPath` function in `gpmain.lsp`:

```

(princ "\nThe gp:drawOutline function returned <")
(princ PolylineName)
(princ ">")
(Alert "Congratulations - your program is complete!")

```

You had been using this code as a placeholder, but now that `gp:drawOutline` is functioning, you no longer need it.

Running the Application

Before running your program, save all the files you changed, if you have not already done so. You can choose File ► Save All from the VLISP menu, or use the ALT+SHIFT+S keyboard shortcut to save all your open files.

The next thing you must do is reload all the files in VLISP.

To load and run all the files in your application

- 1 If the project file you created earlier in this lesson is not already open, choose Project ► Open Project from the VLISP menu, then enter the project file name *gpath*; do not include the *.prj* extension. If VLISP does not find the project file, press the Browse button and choose the file from the Open Project dialog box. Click Open.



- 2 Press the Load Source Files button in the project window.
- 3 Enter the **(C:GPath)** command at the VLISP Console prompt to run the program. If you have some debugging to do, try using the tools you learned in Lessons 2 and 3. And remember, if all else fails, you can always copy the code from the *Tutorial\VisualLISP\Lesson4* directory.

Also, try drawing the path using both lightweight and old-style polylines. After drawing the paths, use the AutoCAD **list** command to determine whether or not your program is drawing the correct entity types.

Wrapping Up Lesson 4

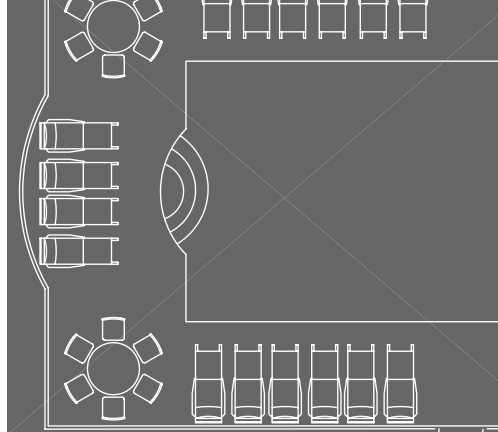
In this lesson, you

- Modularized your code by dividing it among four files.
- Organized your code modules in a VLISP project.
- Learned to define a dialog box with Dialog Control Language (DCL).
- Added AutoLISP code to set up and handle input in the dialog box.
- Modified your code to provide users with a choice of boundary line type.

Now you have a program that draws a garden path boundary. In the next lesson, you will add the tiles to the garden path. In the process, you will be introduced to more VLISP program development tools.

Drawing the Tiles

By the end of this lesson, your application will meet the basic requirements stated in Lesson 1. You will add the functionality for drawing tiles within the boundary of the garden path and provide this function using several different methods of entity creation. You will also learn some keyboard shortcuts and new editing tools.



In This Lesson

5

- Introducing More Visual LISP Editing Tools
- Adding Tiles to the Garden Path
- Testing the Code
- Wrapping Up Lesson 5

Introducing More Visual LISP Editing Tools

Open your copy of *gpdraw.lsp* in a VLISP text editor window, if the file is not already open. There are a couple of things about this code that are typical of much of the code you will be developing with VLISP. First, there are many parentheses and parentheses within parentheses. Second, there are many function calls, and some of those functions have very long names (`vla-addLightweightPolyline`, for example). VLISP provides some editing tools to help you deal with these common features of AutoLISP code.

Matching Parentheses

VLISP provides a parenthesis matching feature to help you find the close parenthesis that corresponds to an open parenthesis.

To match an open parenthesis with its corresponding close parenthesis

- 1 Place your cursor in front of the opening parenthesis that precedes the `setq` function call.
- 2 Press CTRL+SHIFT+]. (Double-clicking also does the trick.)

VLISP finds the closing parenthesis that matches the one you chose, and selects all the code in between. Not only does this ensure you typed in the correct number of parentheses, it also makes it easy to copy or cut the selected text. This might have come in handy when you updated this call at the end of Lesson 4.

Why else might you want to do this? You can copy a chunk of code to the VLISP Console window, paste it there, and try it out. Or maybe you have figured out how to replace 50 lines of code with three really marvelous lines of much better code. You can quickly select the old code using the parentheses matching tool, then eliminate it with a single keystroke. It is a lot quicker to let VLISP find an entire block than for you to hunt down every last closing parenthesis.

There is a corresponding key command for matching and selecting backward. To try this, put your cursor after a closing parenthesis, then either double-click or press CTRL+SHIFT+[. VLISP searches for the corresponding opening parenthesis, and selects it along with the enclosed code.

Both commands are also available by choosing Edit ► Parentheses Matching from the VLISP menu.

Completing a Word Automatically

Imagine you are adding some new functionality to your program using the following code:

```
ObjectCreationStyle (strcase (cdr (assoc 3 BoundaryData)))
(if (equal ObjectCreationStyle "COMMAND")
    (progn
      (setq firstCenterPt(polar rowCenterPt (Degrees->Radians 45)
distanceOnPath))
      (gp:Create_activeX_Circle)
    )
)
```

(Don't worry about what this code actually does, if anything. It is only an example that includes several long variable and function names.)

VLISP can save you some keystrokes by completing words for you.

To use the Visual LISP Complete Word by Match feature

- 1 Scroll to the bottom of the *gpdraw.lsp* file and enter the following code:

```
ObjectCreationStyle (strcase (cdr (assoc 3 BoundaryData)))
  (if (equal Ob
```

- 2 Press CTRL+SPACEBAR.

VLISP just saved you seventeen keystrokes as it searched within the current file and found the closest match to the last two letters you typed.

- 3 Complete the line of code so that it looks like the following:

```
(if (equal ObjectCreationStyle "COMMAND")
```

- 4 Add the following lines:

```
(progn
  (setq firstCenterPt(p
```

- 5 Press CTRL+SPACEBAR.

VLISP matches the most recent “p” word, which happens to be `progn`. However, the word you need is `polar`. If you keep pressing CTRL+SPACEBAR, VLISP cycles through all the possible matches in your code. Eventually, it will come around to `polar`.

- 6 Delete all the code you just entered; it was for demonstration purposes only.

The Complete Word by Match feature is also available from the VLISP Search menu.

Completing a Word by Apropos

If you have worked with AutoLISP before, you may have had to type in an expression similar to the one shown below:

```
(setq myEnt (ssname mySelectionSet ssIndex))
```

Often, it is confusing to keep track of all the selection set functions: **ssname**, **ssget**, **sslenth**, and so on. VLISP can help, using its Complete Word by Apropos feature.

To use the Visual LISP Complete Word by Apropos feature

- 1 Scroll to the bottom of the *gpdraw.lsp* file and enter the following on a blank line:

```
(setq myEnt (ent
```

- 2 Press CTRL+SHIFT+SPACEBAR.

VLISP displays a list of all AutoLISP symbols that begin with the letters *ent*.

Use the cursor keys (the up and down arrow keys) to move through the list. Select **ENTGET**, then press ENTER.

VLISP replaces the *ent* you typed with **ENTGET**.

- 3 Delete the code.

Getting Help with a Function

The code that adds a lightweight polyline to the drawing calls a function named **vla-addLightweightPolyline**. Not only is that a lengthy term to write, but there are several functions whose names begin with **vla-add** that you will use to create entities. Rather than consulting a manual to look up the function name every time you create a program, let VLISP help.

To get help with using a function

- 1 Enter the following on a blank line:

```
(vla-add
```

- 2 Press CTRL+SHIFT+SPACEBAR.

- 3 Scroll through the list until you find **vla-addLightweightPolyline**.

- 4 Double-click on **vla-addLightweightPolyline**.

VLISP displays the Symbol Service dialog box for the selected function.



- 5 Press the Help button in the Symbol Service dialog box. (For ActiveX functions, you will be directed to the **ActiveX and VBA Reference**.)

- 6 Delete the changes you made to *gpdraw.lsp*; these were for demonstration purposes only. Also, close the Symbol Service and Apropos windows.

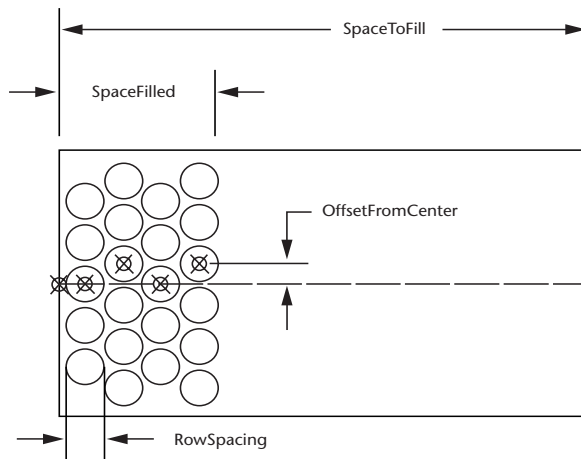
Adding Tiles to the Garden Path

You now have a path boundary and are ready to fill it with tiles. You will need to apply some logic and work through a little geometry.

Applying Some Logic

One thing you need to do is determine how to space out the tiles and draw them. If this were a simple rectilinear grid of tiles, you could use the AutoCAD **ARRAY** command to fill in the tiles. But for the garden path, you need to have each row of tiles offset from the previous row.

This row-offset pattern is a repeating pattern. Think of how you might go about laying the tiles if you were building the actual path. You would probably be inclined to start at one end and just keep laying down rows until there wasn't any more space left.



Here is the logic in pseudo-code:

```
At the starting point of the path
Figure out the initial row offset from center (either centered on
the path or offset by one "tile space").
While the space of the boundary filled is less than the space to
fill,
  Draw a row of tiles.
  Reset the next start point (incremented by one "tile space").
  Add the distance filled by the new row to the amount of space
  filled.
  Toggle the offset (if it is centered, set it up off-center, or
  vice versa).
  Go back to the start of the loop.
```

Applying Some Geometry

There are only a few dimensions you need to know to draw the garden path. The half width is easy: it is just half the width of the path. You already defined the code to obtain this width from users and saved it in an association list.

Tile spacing is also easy; it is twice the radius (that is, the diameter) plus the space between the tiles. The dimensions are also obtained from users.

Row spacing is a little trickier, unless you are really sharp with trigonometry. Here is the formula:

$$\text{Row Spacing} = (\text{Tile Diameter} + \text{Space between Tiles}) * (\text{the sine of } 60 \text{ degrees})$$

Drawing the Rows

See if you can make sense of the following function. Compare it to the pseudo-code and try to catch the geometric calculations just described. There may be a few AutoLISP functions that are new to you. If you need help with these functions, refer to the [AutoLISP Reference](#). For now, just read the code; do not write anything.

```
(defun gp:Calculate-and-Draw-Tiles (BoundaryData / PathLength
  TileSpace TileRadius SpaceFilled SpaceToFill
  RowSpacing offsetFromCenter
  rowStartPoint pathWidth pathAngle
  ObjectCreationStyle TileList)
  (setq PathLength (cdr (assoc 41 BoundaryData))
        TileSpace (cdr (assoc 43 BoundaryData))
        TileRadius (cdr (assoc 42 BoundaryData))
        SpaceToFill (- PathLength TileRadius)
        RowSpacing (* (+ TileSpace (* TileRadius 2.0))
                      (sin (Degrees->Radians 60)))
        ) ;_ end of *
```

```

SpaceFilled RowSpacing
offsetFromCenter 0.0
offsetDistance (/ (+ (* TileRadius 2.0) TileSpace) 2.0)
rowStartPoint (cdr (assoc 10 BoundaryData))
pathWidth (cdr (assoc 40 BoundaryData))
pathAngle (cdr (assoc 50 BoundaryData))
ObjectCreationStyle (strcase (cdr (assoc 3 BoundaryData)))
) ;_ end of setq

;; Compensate for the first call to gp:calculate-Draw-tile Row
;; in the loop below.
(setq rowStartPoint
  (polar rowStartPoint
    (+ pathAngle pi)
    (/ TileRadius 2.0)
  ) ;_ end of polar
) ;_ end of setq
;; Draw each row of tiles.
(while (<= SpaceFilled SpaceToFill)
  ;; Get the list of tiles created, adding them to our list.
  (setq tileList (append tileList
    (gp:calculate-Draw-TileRow
      (setq rowStartPoint
        (polar rowStartPoint
          pathAngle
          RowSpacing
        ) ;_ end of polar
      ) ;_ end of setq
      TileRadius
      TileSpace
      pathWidth
      pathAngle
      offsetFromCenter
      ObjectCreationStyle
    ) ;_ end of gp:calculate-Draw-TileRow
  ) ;_ end of append
  ;; Calculate the distance along the path for the next row.
  SpaceFilled (+ SpaceFilled RowSpacing)
  ;; Alternate between a zero and a positive offset
  ;; (causes alternate rows to be indented).
  offsetFromCenter
  (if (= offsetFromCenter 0.0)
    offsetDistance
    0.0
  ) ;_ end of if
) ;_ end of setq
) ;_ end of while
;; Return the list of tiles created.
tileList
) ;_ end of defun

```

A couple of sections from the code may need a little extra explanation.

The following code fragment occurs right before the **while** loop begins:

```
;; Compensate for the very first start point!!
(setq rowStartPoint(polar rowStartPoint
  (+ pathAngle pi)(/ TileRadius 2.0)))
```

There are three pieces to the puzzle of figuring out the logic behind this algorithm:

- The `rowStartPoint` variable starts its life within the **gp:calculate-and-draw-tiles** function by being assigned the point the user selected as the start point of the path.
- The very first argument passed to the **gp:calculate-draw-tile-row** function does the following:

```
(setq rowStartPoint(polar rowStartPoint pathAngle RowSpacing))
```

Another way of stating this is: At the time the

gp:calculate-draw-tile-row function is called, the `rowStartPoint` variable is set to one `RowSpacing` distance beyond the current `rowStartPoint`.

- The `rowStartPoint` argument is used within **gp:calculate-draw-tile-row** as the starting point for the centers of the circles in the row.

To compensate for the initial forward shifting of the `rowStartPoint` during the drawing of the first row (that is, the first cycle through the **while** loop), you will want to shift `rowStartPoint` slightly in the opposite direction. The aim is to avoid the appearance of a large margin of empty space between the path boundary and the first row. Half the `TileRadius` is a sufficient amount by which to move the point. This can be achieved by using **polar** to project `rowStartPoint` along a vector oriented 180 degrees from the `PathAngle`. If you think about it, this places the point temporarily outside the path boundary.

The next fragment (modified for readability) may be a little puzzling:

```
(setq tileList (append tileList
  (gp:calculate-draw-tile-row
    (setq rowStartPoint
      (polar rowStartPoint pathAngle RowSpacing)
    ) ;_ end of setq
    TileRadius TileSpace pathWidth pathAngle
    offsetFromCenter ObjectCreationStyle
  )))
```

In essence, there is **setq** wrapped around an **append** wrapped around the call to **gp:calculate-draw-tile-row**.

The `gp:calculate-Draw-TileRow` function will return the Object IDs for each tile drawn. (The Object ID points to the tile object in the drawing.) You are drawing the tiles row by row, so the function returns the Object IDs of one row at a time. The `append` function adds the new Object IDs to any existing Object IDs stored in `tileList`.

Near the end of the function, you can find the following code fragment:

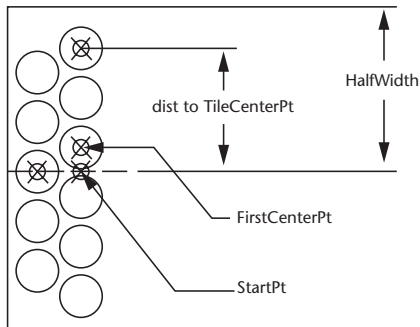
```
(setq offsetFromCenter
  (if (= offsetFromCenter 0.0)
      offsetDistance
      0.0)
  )
)
```

This is the offset toggle, which determines whether the row being drawn should begin with a circle centered on the path or offset from the path. The pseudo-code for this algorithm follows:

```
Set the offset amount to the following:
  If the offset is currently zero, set it to the offset distance;
  Otherwise, set it back to zero.
```

Drawing the Tiles in a Row

Now that you have the logic for drawing the path, the next step is to figure out how to draw the tiles in each row. In the following diagram, there are two cases shown: a row where the offset from the center of the path is equal to 0.0, and a case where the offset is not equal to zero. Take a look at the diagram, then read the pseudo-code that follows.



```
Set up variables for StartPoint, angp90, angm90, and so on.
Set the variable FirstCenterPoint to the StartPoint + offset amount
  (which may be 0.0).
Set the initial value of TileCenterPt to FirstCenterPoint.
```

```

(Comment: Begin by drawing the circles in the angp90 direction.)
While the distance from the StartPoint to the TileCenterPt is less
than the HalfWidth:
    Draw a circle (adding to the accumulating list of circles).
    Set TileCenterPt to the next tile space increment in the angp90
    direction.
End While

Reset the TileCenterPoint to the FirstCenterPoint + the tile space
increment at angm90.
While the distance from the StartPoint to the TileCenterPt is less
than the HalfWidth:
    Draw a circle (adding to the accumulating list of circles).
    Set TileCenterPt to the next tile space increment in the angm90
    direction.
End While

Return the list of circles.

```

Looking at the Code

Now look at the code for the `gp:calculate-Draw-TileRow` function:

```

(defun gp:calculate-Draw-TileRow (startPoint TileRadius
    TileSpace pathWidth pathAngle offsetFromCenter
    ObjectCreationStyle / HalfWidth TileDiameter
    ObjectCreationFunction angp90 angm90
    firstCenterPt TileCenterPt TileList)
  (setq HalfWidth (- (/ pathWidth 2.00) TileRadius)
    Tilespace (+ (* TileRadius 2.0) TileSpace)
    TileDiameter (* TileRadius 2.0)
    angp90 (+ PathAngle (Degrees->Radians 90))
    angm90 (- PathAngle (Degrees->Radians 90))
    firstCenterPt (polar startPoint angp90 offsetFromCenter)
    tileCenterPt firstCenterPt
    ObjectCreationStyle(strcase ObjectCreationStyle)
    ObjectCreationFunction
    (cond
      ((equal ObjectCreationStyle "ACTIVEX")
        gp:Create_activeX_Circle
      )
      ((equal ObjectCreationStyle "ENTMAKE")
        gp:Create_entmake_Circle
      )
      ((equal ObjectCreationStyle "COMMAND")
        gp:Create_command_Circle
      )
    )
  )

```

```

(T
  (alert (strcat "ObjectCreationStyle in function
                gp:calculate-Draw-TileRow"
                "\nis invalid. Contact developer for assistance."
                "\n      ObjectCreationStyle set to ACTIVEX"
                )
        )
  )
  (setq ObjectCreationStyle "ACTIVEX")
)
)
)
;; Draw the circles to the left of the center.
(while (< (distance startPoint tileCenterPt) HalfWidth)
  ;; Add each tile to the list to return.
  (setq tileList
    (cons
      (ObjectCreationFunction tileCenterPt TileRadius)
      tileList
    )
  )
  ;; Calculate the center point for the next tile.
  (setq tileCenterPt
    (polar tileCenterPt angp90 TileSpacing)
  )
);_ end of while

;; Draw the circles to the right of the center.
(setq tileCenterPt
  (polar firstCenterPt angm90 TileSpacing))
(while (< (distance startPoint tileCenterPt) HalfWidth)
  ;; Add each tile to the list to return.
  (setq tileList
    (cons
      (ObjectCreationFunction tileCenterPt TileRadius)
      tileList
    )
  )
  ;; Calculate the center point for the next tile.
  (setq tileCenterPt (polar tileCenterPt angm90 TileSpacing))
);_ end of while

;; Return the list of tiles.
tileList
) ;_ end of defun

```

The AutoLISP code logic follows the pseudo-code, with the following addition:

```
(setq ObjectCreationFunction
  (cond
    ((equal ObjectCreationStyle "ACTIVE")
      gp:Create_activeX_Circle
    )
    ((equal ObjectCreationStyle "ENTMAKE")
      gp:Create_entmake_Circle
    )
    ((equal ObjectCreationStyle "COMMAND")
      gp:Create_command_Circle
    )
  )
  (T
    (alert
      (strcat
        "ObjectCreationStyle in function gp:calculate-Draw-TileRow"
        "\nis invalid. Contact the developer for assistance."
        "\n      ObjectCreationStyle set to ACTIVE"
      ) ;_ end of strcat
    ) ;_ end of alert
    (setq ObjectCreationStyle "ACTIVE")
  )
) ;_ end of cond
) ;_ end of setq
```

Remember the specification to allow users to draw the tiles (circles) using either ActiveX, the **entmake** function, or the **command** function? The `ObjectCreationFunction` variable is assigned one of three functions, depending on the `ObjectCreationStyle` parameter (passed from **C:GPath** and through **gp:Calculate-and-Draw-Tiles**). Here are the three functions as they will be defined in *gpdraw.lsp*:

```
(defun gp:Create_activeX_Circle (center radius)
  (vla-addCircle *ModelSpace*
    (vlax-3d-point center) ; convert to ActiveX-compatible 3D point
    radius
  )
) ;_ end of defun

(defun gp:Create_entmake_Circle(center radius)
  (entmake
    (list (cons 0 "CIRCLE") (cons 10 center) (cons 40 radius))
  )
  (vlax-ename->vla-object (entlast))
)

(defun gp:Create_command_Circle(center radius)
  (command "_CIRCLE" center radius)
  (vlax-ename->vla-object (entlast))
)
)
```

The first function draws a circle using an ActiveX function and returns an ActiveX object.

The second function draws a circle using `entmake`. It returns an entity name converted into an ActiveX object.

The third function draws a circle using `command`. It also returns an entity name converted into an ActiveX object.

Testing the Code

If you've made it this far, you have earned a shortcut.

To test the code

- 1 Close all the active windows within VLISP, including any open project windows.
- 2 Copy the entire contents of the *Tutorial\VisualLISP\Lesson5* directory to your *MyPath* tutorial directory.
- 3 Open the project file *gpath5.prj* using Select Project ► Open Project from the VLISP menu bar.
- 4 Load the project source files.
- 5 Activate (switch to) the AutoCAD window and issue the `gpath` command to run the program.
- 6 Run `gpath` to draw the garden path three times, each time using a different entity creation method. Do you notice a difference in the speed with which the path is drawn with each method?

Wrapping Up Lesson 5

You started this lesson by learning VLISP editing features that helped you

- Match the parentheses in your code.
- Find and complete a function name.
- Obtain online help for a function.

You finished the lesson by building code that draws the tiles in the garden path. You now have a program that meets the requirements established at the very beginning of this tutorial.

At this point, you probably have acquired enough experience with VLISP to venture off on your own. But if you are up to it, there are two more lessons in this tutorial that demonstrate the use of reactor functions and other advanced features of the VLISP environment.

Acting with Reactors

In this lesson, you will learn about reactors and how to attach them to drawing events and entities. Reactors allow your application to be notified by AutoCAD when particular events occur. For example, if a user moves an entity that your application has attached a reactor to, your application will receive notification that the entity has moved. You can program this to trigger additional operations, such as moving other entities associated with the one the user moved, or perhaps updating a text tag that records revision information on the altered drawing feature. In effect, it is like setting up your application with a pager and telling AutoCAD to beep the application when something happens.

In This Lesson

6

- Reactor Basics
- Designing Reactors for the Garden Path
- Test Driving Your Reactors
- Wrapping Up Lesson 6

Reactor Basics

A reactor is an object you attach to the drawing editor, or to specific entities within a drawing. Extending the metaphor of the pager, the reactor object is an automatic dialer that knows how to call your pager when something significant happens. The pager within your application is an AutoLISP function called by the reactor; such a function is known as a callback function.

NOTE The complexity of the application code and the level of expertise required for these final two lessons is much higher than Lessons 1 through 5. There is a great deal of information presented, but it is not all explained at the same level of detail as in the previous lessons. If you are a beginner, don't worry if you don't get it the first time. Consider this just a first taste of some of the very powerful but more technically difficult features of VLISP.

Reactor Types

There are many types of AutoCAD reactors. Each reactor type responds to one or more AutoCAD events. Reactors are grouped into the following categories:

Editor Reactors	Notify your application each time an AutoCAD command is invoked.
Linker Reactors	Notify your application every time an ObjectARX application is loaded or unloaded.
Database Reactors	Correspond to specific entities or objects within a drawing database.
Document Reactors	Notify your application in MDI mode of a change to the current drawing document, such as opening of a new drawing document, activating a different document window, and changing a document's lock status.
Object Reactors	Notify you each time a specific object is changed, copied, or deleted.

With the exception of editor reactors, there is one type of reactor for each reactor category. Editor reactors encompass a broad class of reactors: for example, DXF™ reactors that notify an application when a DXF file is imported or exported, and Mouse reactors that notify of mouse events such as double-clicks.

Within the reactor categories, there are many specific events to which you can attach a reactor. AutoCAD allows users to perform many different kinds of actions, and it is up to you to determine the actions that you are interested in. Once you have done this, you can attach your reactor “auto-dialer” to the event, then write the callback function that is triggered when the event occurs.

Designing Reactors for the Garden Path

To implement reactor functionality in the garden path application, start by handling just a few events, rather than trying to cover all possible user actions.

Selecting Reactor Events for the Garden Path

For the tutorial, set the following goals:

- When a corner point (vertex) of the garden path boundary is repositioned, redraw the path so that the outline remains rectilinear. In addition, redraw the tiles based on the new size and shape.
- When the garden path boundary is erased, erase the tiles as well.

Planning the Callback Functions

For each reactor event, you must plan the function that will be invoked when the event occurs. The following pseudo-code outlines the logical sequence of events that should occur when users drag one of the polyline vertices to a new location:

```
Defun gp:outline-changed
  Erase the tiles.
  Determine how the boundary changed.
  Straighten up the boundary.
  Redraw new tiles.
End function
```

There is a complication, though. When the user begins dragging the outline of a polyline vertex, AutoCAD notifies your application by issuing a **:vlr-modified** event. However, at this point the user has just begun dragging one of the polyline vertices. If you immediately invoke the **gp:outline-changed** function, you will interrupt the action that the user is in the midst of. You would not know where the new vertex location will be, because the user has not yet selected its position. And finally, AutoCAD will not allow your function to modify the polyline object while the user is still

dragging it. AutoCAD has the polyline object open for modification, and leaves it open until the user finishes repositioning the object.

You need to change your approach. Here is the updated logic:

```
When the user begins repositioning a polyline vertex,  
  Invoke the gp:outline-changed function  
  Defun gp:outline-changed.  
    Set a global variable that stores a pointer to the polyline  
    being modified by the user.  
  End function  
When the command completes,  
  Invoke the gp:command-ended function  
  Defun gp:command-ended.  
    Erase the tiles.  
    Get information on the previous polyline vertex locations.  
    Get information on the new polyline vertex locations.  
    Redefine the polyline (straighten it up).  
    Redraw the tiles.  
  End function
```

When a user completes modifying a path outline, AutoCAD notifies your application by issuing a `:vlr-commandEnded` event, if you have established an editor reactor.

The use of a global variable to identify the polyline the user changed is necessary because there is no continuity between the `gp:outline-changed` and `gp:command-ended` functions. In your application, both functions are invoked and executed independently of one another. The global variable stores important information set up in one function and accessed in the other.

Now consider what to do if the user erases the garden path boundary. The ultimate objective is to erase all the tiles. The following pseudo-code outlines the logic:

```
When the user begins to erase the boundary,  
  Invoke the gp:outline-erased function  
  Defun gp:outline-erased.  
    Set a global variable that stores a pointer to the reactor  
    attached to the polyline currently being erased.  
  End function  
When the erase is completed,  
  Invoke the gp:command-ended function  
  Defun gp:command-ended.  
    Erase the tiles that belonged to the now-deleted polyline.  
  End function
```

Planning for Multiple Reactors

Users may have several garden paths on the screen, and may be erasing more than one. You need to plan for this possibility.

The reactor associated with an entity is an object reactor. If there are several entities in the drawing, there may also be several object reactors, one for each entity. A specific editing event, such as the `erase` command, can trigger many callbacks, depending on how many of the selected entities have reactors attached. Editor reactors, on the other hand, are singular in nature. Your application should only attach a single `:v1r-commandEnded` event reactor.

The event sequence for both modifications—changing a vertex location and erasing a polyline—ends up with actions that need to be performed within the `gp:command-ended` function. Determine which set of actions to perform for each condition. The following pseudo-code outlines the logic:

```
Defun gp:command-ended (2nd version)
  Retrieve the pointer to the polyline (from a global variable).
  Conditional:
    If the polyline has been modified then:
      Erase the tiles.
      Get information on the previous polyline vertex locations.
      Get information on the new polyline vertex locations.
      Redefine the polyline (straighten it up).
      Redraw the tiles.
    End conditional expression.
    If the polyline has been erased then:
      Erase the tiles.
    End conditional expression.
  End Conditional
End function
```

Attaching the Reactors

The next step in planning a reactor-based application is to determine how and when to attach reactors. You need to attach two object reactors for the polyline border of any garden path (one to respond to a modification event, the other to respond to erasure), and one editor reactor to alert your application when users complete their modification to the polyline. Object reactors are attached to entities, while editor reactors are registered with AutoCAD.

There is another consideration to account for. To recalculate the polyline outline—return it to a rectilinear shape—after the user modifies it, you must know what the vertex configuration was before the modification. This information cannot be determined once the polyline has been modified. By that time you can only retrieve information on what the new configuration is. So how do you solve this? You could keep this information in a global variable, but there is a major problem with that idea. Users can draw as many garden paths as they want, and every new path would require a new global variable. This would get very messy.

Storing Data with a Reactor

You can solve the problem of saving the original configuration by taking advantage of another feature of VLISP reactors—the ability to store data within a reactor. When the user first draws a path boundary, you attach a reactor to the boundary, along with the data you need to save. This entails modifying your main program function, `C:GPath`, as follows:

```
Defun C:GPath
  Do everything that is already done in the garden path
  (and don't break anything).

  Attach an object reactor to the polyline using these parameters:
    A pointer to the polyline just drawn,
    A list of data that you want the reactor to record,
    A list of the specific polyline object events to be tracked,
    along with the LISP callback functions to be invoked.
  End of the object reactor setup.

  Attach editor reactor to the drawing editor using the
  following parameters:
    Any data you want attached to the reactor (in this case, none)
    A list of the specific editor reactor events to be tracked,
    along with the LISP callback functions to be invoked.
  End of the editor reactor setup.
End function
```

Updating the C:GPath Function

Update the `C:GPath` function by adding reactor creation logic.

To add reactor creation logic to C:GPath

- 1 Replace your version of `gpmain.lsp` with the updated version shown below. Copy this code from the `<AutoCAD directory>\Tutorial\VisualLISP\Lesson6` directory:

```
(defun C:GPath (/
  gp_PathData
  gp_dialogResults
  PolylineName
  tileList
)
(setvar "OSMODE" 0) ;; Turn off object snaps.
;|
;| Lesson 6 adds a stubbed-out command reactor to AutoCAD.
;| However, it would be undesirable to react to every
;| drawing of a circle should the COMMAND tile creation
;| method be chosen by the user. So, disable the
;| *commandReactor* in case it exists.
;|
```

```

(if *commandReactor*
  (progn
    (setq *commandReactor* nil)
    (vlr-remove-all :VLR-Command-Reactor)
  )
)

;; Ask the user for input: first for path location and
;; direction, then for path parameters. Continue only if you
;; have valid input. Store the data in gp_PathData.
(if (setq gp_PathData (gp:getPointInput))
  (if (setq gp_dialogResults
    (gp:getDialogInput
      (cdr (assoc 40 gp_PathData))
    ) ;_ end of gp:getDialogInput
  ) ;_ end ofsetq

    (progn
      ;; Now take the results of gp:getPointInput and append this to
      ;; the added information supplied by gp:getDialogInput.

      (setq gp_PathData (append gp_PathData gp_DialogResults))

      ;; At this point, you have all the input from the user.
      ;; Draw the outline, storing the resulting polyline "pointer"
      ;; in the variable called PolylineName.

      (setq PolylineName (gp:drawOutline gp_PathData))

      ;; Next, it is time to draw the tiles within the boundary.
      ;; The gp_tileList contains a list of the object pointers for
      ;; the tiles. By counting up the number of points (using the
      ;; length function), we can print out the results of how many
      ;; tiles were drawn.
      (princ "\nThe path required ")
      (princ
        (length
          (setq tileList (gp:Calculate-and-Draw-Tiles gp_PathData))
        ) ;_ end of length
      ) ;_ end of princ
      (princ " tiles.")

      ;; Add the list of pointers to the tiles (returned by
      ;; gp:Calculate-and-Draw-Tiles) to gp_PathData. This will
      ;; be stored in the reactor data for the reactor attached
      ;; to the boundary polyline. With this data, the polyline
      ;; "knows" what tiles (circles) belong to it.

      (setq gp_PathData
        (append (list (cons 100 tileList))
          ; all the tiles
          gp_PathData
        ) ;_ end of append
      ) ;_ end ofsetq
    )
  )
)

```

```

;; Before we attach reactor data to an object, let's look at
;; the function vlr-object-reactor.
;; vlr-object-reactor has the following arguments:
;; (vlr-object-reactor owner's data callbacks)
;; The callbacks Argument is a list comprised
;; '(event_name . callback_function).
;;
;; For this exercise we will use all arguments
;; associated with vlr-object-reactor.
;; These reactor functions will execute only if
;; the polyline in PolylineName is modified or erased.

(vlr-object-reactor

  ;; The first argument for vlr-object-reactor is
  ;; the "Owner's List" argument. This is where to
  ;; place the object to be associated with the
  ;; reactor. In this case, it is the vlaObject
  ;; stored in PolylineName.

  (list PolylineName)

  ;; The second argument contains the data for the path
  gp_PathData

  ;; The third argument is the list of specific reactor
  ;; types that we are interested in using.
  '
    (
      ;; reactor that is called upon modification of the object.
      (:vlr-modified . gp:outline-changed)
      ;; reactor that is called upon erasure of the object.
      (:vlr-erased . gp:outline-erased)
    )
  ) ;_ end of vlr-object-reactor

;; Next, register a command reactor to adjust the polyline
;; when the changing command is finished.
(if (not *commandReactor*)
  (setq *commandReactor*
    (VLR-Command-Reactor
      nil ; No data is associated with the command reactor
      '(
        (:vlr-commandWillStart . gp:command-will-start)
        (:vlr-commandEnded . gp:command-ended)
      )
    ) ;_ end of vlr-command-reactor
  )
)

```

```

;; The following code removes all reactors when the drawing is
;; closed. This is extremely important!!!!!!!
;; Without this notification, AutoCAD may crash upon exiting!
(if (not *DrawingReactor*)
    (setq *DrawingReactor*
        (VLR-DWG-Reactor
         nil ; No data is associated with the drawing reactor
         '(:vlr-beginClose . gp:clean-all-reactors)
         )
        ) ;_ end of vlr-DWG-reactor
    )
) ;_ end of progn
(princ "\nFunction cancelled.")
) ;_ end of if
(princ "\nIncomplete information to draw a boundary.")
) ;_ end of if
(princ) ; exit quietly
) ;_ end of defun

;; Display a message to let the user know the command name.
(princ "\nType GPATH to draw a garden path.")
(princ)

```

- 2 Review the code modifications and comments describing what each new statement does. This tutorial shows all modified code in boldface.

Adding Reactor Callback Functions

The reactor callback functions add a substantial amount of code to your application. This code is provided for you in the *Lesson6* directory.

To add the reactor callback functions to your program

- 1 Copy the *gpreact.lsp* file from the *Tutorial\VisualLISP\Lesson6* directory to your *MyPath* working directory.
- 2 Open the *GPath* project (if it is not already open), and press the Project Properties button in the *gpath* project window.
- 3 Add the *gpreact.lsp* file to your project.
- 4 The *gpreact.lsp* file can reside anywhere in the order of files between *utils.lsp*, which must remain first, and *gpmain.lsp*, which should remain as the last file. Move any files, if necessary, then press OK.
- 5 Open the *gpreact.lsp* file by double-clicking on the file name within the *gpath* project window.



Read through the comments in the file to help you understand what it is doing. Note that all the callback functions are stubbed out; the only functionality they perform is to display alert messages when they are fired.

The last function in the file is so important it deserves a heading of its own.

Cleaning Up After Your Reactors

Reactors are indeed very active. When you design an application that relies on them, you could very well spend a great deal of time crashing your program and possibly crashing AutoCAD as well. It helps to have a tool available to remove all the reactors you have added, if necessary.

The *gpreact.lsp* file includes a function `gp:clean-all-reactors` that doesn't do much on its own. Instead, it makes a call to the `CleanReactors` function. Add this function to your *utils.lsp* file by copying the following code to the end of the file:

```
;;;-----;
;;;      Function: CleanReactors                               ;
;;;-----;
;;;  Description: General utility function used for cleaning up ;
;;;                reactors. It can be used during debugging, as ;
;;;                well as cleaning up any open reactors before  ;
;;;                a drawing is closed.                         ;
;;;-----;
(defun CleanReactors ()
  (mapcar 'vlr-remove-all
    '( :VLR-AcDb-reactor
      :VLR-Editor-reactor
      :VLR-Linker-reactor
      :VLR-Object-reactor
    )
  )
)
```

Test Driving Your Reactors

By now, you should have all the necessary pieces in place to use some live reactors.

To test the reactor code



- 1 Load all the source code from your project. (Press the Load Source Files button in the `gpath` project window.)
- 2 Fire up the `c:GPath` function and give it a spin.
The program will draw a garden path for you, just as you were able to in Lesson 5. You won't see anything interesting at first.
- 3 Try the following actions after you draw the path:
 - Move a polyline vertex. Pick the polyline and turn on its grips, then drag a vertex to a new location.
 - Stretch the polyline.

- Move the polyline.
- Erase the polyline.

Examine the messages that appear. You are watching the behind-the-scenes activities of a powerful capability.

(If your application is not working correctly and you do not want to take the time to debug it right now, you can run the sample code provided in the *Tutorial\VisualLISP\Lesson6* directory. Use the *Gpath6* project in that directory.)

NOTE Because of the reactor behavior, you may notice that after testing a reactor sequence in AutoCAD, you cannot return to VLISP by pressing ALT+TAB, or by clicking to activate the VLISP window. If this happens, simply enter `vlisp` at the AutoCAD Command prompt to return to VLISP.

Examining Reactor Behavior in Detail

With a stack of scrap paper, start tracing the reactor events within the application. Here is an example of the kinds of things you should track:

Command: erase Object: Polyline border
Reactor sequence
<ol style="list-style-type: none"> 1. Type in erase command 2. Callback: GP:COMMAND-WILL-START 3. Select objects: pick a polyline 4. Hit Enter (object selection complete) 5. Callback: GP:OUTLINE-ERASED 6. Callback: GP:OUTLINE-CHANGED 7. Callback: GP:COMMAND-ENDED

Draw ten garden paths, then track the following Command/Object combinations, selecting the polylines in succession:

- Erase / Polyline border (path 1).
- Erase / Circle within a polyline (path 2).
- Erase / Two polylines (paths 3 and 4).
- Move / Polyline border (path 5).
- Move / Circle within a polyline (path 6).
- Move / Two polylines and several circles (paths 7 and 8).
- Move Vertex (via grips) / Polyline border (path 9).
- Stretch / Polyline border (path 10).

This exercise will give you a good understanding of what is happening behind the scenes. At any time throughout Lesson 7 when the reactor functionality becomes confusing, refer to your “reactor-trace sheets.”

Wrapping Up Lesson 6

In this lesson, you were introduced to AutoCAD reactors and how to implement them using VLISP. You designed a plan for adding reactors to the garden path application, and added some of the code your program will need to implement the plan.

You have now been through some very new and exciting stuff, from the AutoLISP perspective. Reactors can add a great deal of functionality to an application, but remember—the more powerful your programs can be, the harder they can crash.

Another thing to keep in mind is that the way your application is designed, the reactor functionality is not persistent from one drawing session to the next. If you save a drawing that contains a garden path hooked up to reactors, the reactors will not be there the next time you open the drawing. You can learn about adding persistent reactors by reviewing the “Transient Versus Persistent Reactors” topic in the *Visual LISP Developer’s Guide*, and then reading about the referenced functions in the *AutoLISP Reference*.

Putting It All Together

In Lesson 6, you learned the basic mechanics behind reactor-based applications. In Lesson 7, you will add functionality to this knowledge and create a garden path that knows how and when to modify itself. After testing your application and determining that it works satisfactorily, you will create a VLISP application from your VLISP project.

You should consider this part of the tutorial as the advanced topics section. If you are a beginner, you may not understand all the AutoLISP code presented here. There are several AutoLISP books listed at the end of this lesson that provide more thorough information on some of the advanced AutoLISP concepts presented here.



In This Lesson

7

- Planning the Overall Reactor Process
- Adding the New Reactor Functionality
- Redefining the Polyline Boundary
- Wrapping Up the Code
- Building an Application
- Wrapping Up the Tutorial
- LISP and AutoLISP Books

Planning the Overall Reactor Process

You need to define several new functions in this lesson. Rather than present you with details on all aspects of the new code, this lesson presents an overview and points out the concepts behind the code. At the end of the lesson, you will have all the source code necessary to create a garden path application identical to the sample program you ran in Lesson 1.

NOTE When you are in the midst of developing and debugging reactor-based applications, there is always the potential of leaving AutoCAD in an unstable state. This can be caused by several situations, such as failing to remove a reactor from deleted entities. For this reason, it is recommended that before beginning Lesson 7, you should close VLISP, save any open files as you do so, exit AutoCAD, then restart both applications.

Begin by loading the project as it existed at the end of Lesson 6.

Two obvious pieces of work remain to be done in the garden path application:

- Writing the object reactor callbacks.
- Writing the editor reactor callbacks.

You also need to consider how to handle the global variables in your program. Often, it is desirable to have globals retain a value throughout an AutoCAD drawing session. In the case of reactors, however, this is not the case. To illustrate this, imagine a user of your garden path application has drawn several garden paths in a single drawing. After doing this, the user erases them, first one at a time, then two at a time, and so on, until all but one path is erased.

Lesson 5 introduced a global variable `*reactorsToRemove*`, responsible for storing pointers to the reactors for the polylines about to be erased. When `*reactorsToRemove*` is declared in `gp:outline-erased`, the event lets you know the polyline is about to be erased. The polyline is not actually removed until the `gp:command-ended` event fires.

The first time the user deletes a polyline, things work just as you would expect. In `gp:outline-erased`, you store a pointer to the reactor. When `gp:command-ended` fires, you remove the tiles associated with the polyline to which the reactor is attached, and all is well. Then, the user decides to erase two paths. As a result, your application will get two calls to `gp:outline-erased`, one for each polyline about to be erased. There are two potential problems you must anticipate:

- When you **setq** the `*reactorsToRemove*` variable, you must add a pointer to a reactor to the global, making sure not to overwrite any values already stored there. This means `*reactorsToRemove*` must be a list structure, so you can append reactor pointers to it. You can then accumulate several reactor pointers corresponding to the number of paths the user is erasing within a single erase command.
- Every time **gp:command-will-start** fires, indicating a new command sequence is beginning, you should reinitialize the `*reactorsToRemove*` variable to `nil`. This is necessary so that the global is not storing reactor pointers from the previous erase command.

If you do not reinitialize the global variable or use the correct data structure (in this case, a list), you will get unexpected behavior. In the case of reactors, unexpected behavior can be fatal to your AutoCAD session.

Here is the chain of events that needs to occur for users to erase two garden paths with a single erase command. Note how global variables are handled:

- Initiate the **erase** command. This triggers the **gp:command-will-start** function. Set `*reactorsToRemove*` to `nil`.
- Select two polylines; your application is not yet notified.
- Press ENTER to erase the two selected polylines.
 - Your application gets a callback to **gp:outline-erased** for one of the polylines. Add its reactor pointer to the null global, `*reactorsToRemove*`.
 - Your application gets a callback to **gp:outline-erased** for the second of the polylines. Append its reactor pointer to the `*reactorsToRemove*` global that already contains the first reactor pointer.
- AutoCAD deletes the polylines.
- Your callback function **gp:command-ended** fires. Eliminate any tiles associated with the reactor pointers stored in `*reactorsToRemove*`.

In addition to the `*reactorsToRemove*` global, your application also includes a `*polyToChange*` global, which stores a pointer to any polyline that will be modified. Two additional globals for the application will be introduced later in this lesson.

Reacting to More User-Invoked Commands

When writing a reactor-based application, you need to handle any command that affects your objects in a significant way. One of your program design activities should be to review all possible AutoCAD editing commands and determine how your application should respond to each one. The format of the reactor-trace sheet shown near the end of Lesson 6 is very good for this purpose. Invoke the commands you expect your user to use, and write down

the kind of behavior with which your application should respond. Other actions to plan for include

- Determine what to do when users issue **UNDO** and **REDO** commands.
- Determine what to do when users issue the **OOPS** command after erasing entities linked with reactors.

To prevent a very complex subject from becoming very, *very* complex, the tutorial does not try to cover all the possibilities that should be covered, and the functionality within this lesson is kept to an absolute minimum.

Even though you won't be building in the complete functionality for these extra commands, examine what a few additional editing functions would require you to do:

- If users stretch a polyline boundary (using the **STRETCH** command) several things should happen. It could be stretched in any direction, not just on the major or minor axis, so the boundary may end up in a very odd shape. In addition, you need to take into consideration how many vertices have been stretched. A situation where only one vertex is stretched will result in a polyline quite different from one in which two vertices are moved. In any case, the tiles must be erased and new positions recalculated once you determine the adjustments needed to the boundary.
- If users move a polyline boundary, all the tiles should be erased, then redrawn in the new location. This is a fairly simple operation, because the polyline boundary did not change its size or shape.
- If users scale a polyline boundary, you need to make a decision. Should the tiles be scaled up as well, so that the path contains the same number of tiles? Or, should the tile size remain the same and the application add or remove tiles, depending on whether the polyline was scaled up or down?
- If users rotate a polyline boundary, all the tiles should be erased, then redrawn in the new orientation.

To begin, though, just plan for the following:

- Warn the user upon command-start that the selected edit command (such as *stretch*, *move*, or *rotate*) will have detrimental effects on a garden path.
- If the user proceeds, erase the tiles and do not redraw them.
- Remove the reactors from the path outline.

NOTE In addition to user-invoked AutoCAD commands, entities may also be modified or deleted through AutoLISP or ObjectARX applications. The example provided in the Garden Path tutorial does not cover programmatic manipulation of the garden path polyline boundary, such as through (`entdel <polyline entity>`). In this case, the editor reactor events `:vlr-commandWillStart` and `:vlr-commandEnded` will not be triggered.

Storing Information within the Reactor Objects

One other important aspect of the application you need to think about is what kind of information to attach to the object reactor that is created for each polyline entity. In Lesson 6, you added code that attached the contents of `gp_PathData` (the association list) to the reactor. You expanded the data carried within `gp_PathData` by adding a new keyed field (100) to the association list. This new sublist is a list of pointers to all the circle entities assigned to a polyline boundary.

Because of the work that needs to be done to recalculate the polyline boundary, four additional key values should be added to `gp_pathData`:

```
;;; StartingPoint                                     ;
;;; (12 . BottomStartingPoint)      15-----14      ;
;;; (15 . TopStartingPoint)         |                 | ;
;;; EndingPoint                     10  ----pathAngle----> 11 ;
;;; (13 . BottomEndingPoint)       |                 | ;
;;; (14 . TopEndingPoint)           12-----13      ;
;;;                                     ;
```

These ordered points are necessary to recalculate the polyline boundary whenever the user drags a corner grip to a new location. This information already exists within the `gp:drawOutline` function in `gpdraw.lsp`. But look at the return value of the function. Currently, only the pointer to the polyline object is returned. So you need to do three things:

- Assemble the perimeter points in the format required.
- Modify the function so that it returns the perimeter point lists and the pointer to the polyline.
- Modify the `c:GPath` function so that it correctly deals with the new format of the values returned from `gp:drawOutline`.

Assembling the perimeter point lists is simple. Look at the code in **gp:drawOutline**. The local variable `p1` corresponds to the key value 12, `p2` to 13, `p3` to 14, and `p4` to 15. You can add the following function call to assemble this information:

```
(setq polyPoints(list
  (cons 12 p1)
  (cons 13 p2)
  (cons 14 p3)
  (cons 15 p4)
))
```

Modifying the function so that it returns the polyline perimeter points and the polyline pointer is also easy. As the last expression within **gp:drawOutline**, assemble a list of the two items of information you want to return.

```
(list pline polyPoints)
```

To add program logic to save the polyline perimeter points

- 1 Modify **gp:drawOutline** by making the changes shown in boldface in the following code (don't overlook the addition of the `polyPoints` local variable to the **defun** statement):

```
(defun gp:drawOutline (BoundaryData / PathAngle
  Width HalfWidth StartPt PathLength
  angm90 angp90 p1 p2
  p3 p4 poly2Dpoints
  poly3Dpoints plineStyle pline
  polyPoints
)
;; extract the values from the list BoundaryData.
(setq PathAngle (cdr (assoc 50 BoundaryData))
  Width (cdr (assoc 40 BoundaryData))
  HalfWidth (/ Width 2.00)
  StartPt (cdr (assoc 10 BoundaryData))
  PathLength (cdr (assoc 41 BoundaryData))
  angp90 (+ PathAngle (Degrees->Radians 90))
  angm90 (- PathAngle (Degrees->Radians 90))
  p1 (polar StartPt angm90 HalfWidth)
  p2 (polar p1 PathAngle PathLength)
  p3 (polar p2 angp90 Width)
  p4 (polar p3 (+ PathAngle
    (Degrees->Radians 180)) PathLength)
  poly2Dpoints (apply 'append
    (mapcar '3dPoint->2dPoint (list p1 p2 p3 p4))
  )
  poly3Dpoints (mapcar 'float (append p1 p2 p3 p4))
;; get the polyline style.
```

```

plineStyle (strcase (cdr (assoc 4 BoundaryData)))

;; Add polyline to the model space using ActiveX automation.
pline      (if (= plineStyle "LIGHT")
            ;; create a lightweight polyline.
            (vla-addLightweightPolyline
             *ModelSpace* ; Global Definition for Model Space
             (gp:list->variantArray poly2Dpoints)
             ;data conversion
            ) ;_ end of vla-addLightweightPolyline
            ;; or create a regular polyline.
            (vla-addPolyline
             *ModelSpace*
             (gp:list->variantArray poly3Dpoints)
             ;data conversion
            ) ;_ end of vla-addPolyline
            ) ;_ end of if
polyPoints (list
            (cons 12 p1)
            (cons 13 p2)
            (cons 14 p3)
            (cons 15 p4)
            )
) ;_ end ofsetq
(vla-put-closed pline T)

(list pline polyPoints)
) ;_ end of defun

```

- 2 Modify the **C:GPath** function (in *gpmain.lsp*). Look for the line of code that currently looks like this:

```
(setq PolylineName (gp:drawOutline gp_PathData))
```

Change it so it appears as follows:

```
(setq PolylineList      (gp:drawOutline gp_PathData)
      PolylineName      (car PolylineList)
      gp_pathData       (append gp_pathData (cadr PolylineList))
) ;_ end ofsetq

```

The *gp_PathData* variable now carries all the information required by the reactor function.

- 3 Add *PolylineList* to the local variables section of the **C:GPath** function definition.

Adding the New Reactor Functionality

In Lesson 6, you hooked up callback function `gp:command-will-start` to the reactor event `:vlr-commandWillStart`. As it currently exists, the function displays some messages and initializes two global variables, `*polyToChange*` and `*reactorsToRemove*`, to `nil`.

To add functionality to the `gp:command-will-start` callback function

- 1 Open your `gpreact.lsp` file.
- 2 In the `gp:command-will-start` function, add two variables to the `setq` function call by modifying it as follows:

```
;; Reset all four reactor globals to nil.
(setq *lostAssociativity* nil
      *polyToChange* nil
      *reactorsToChange* nil
      *reactorsToRemove* nil)
```

- 3 Replace the remaining code in `gp:command-will-start`, up to the last `princ` function call, with the following code:

```
(if (member (setq currentCommandName (car command-list))
           '( "U"      "UNDO"      "STRETCH"  "MOVE"
             "ROTATE"  "SCALE"   "BREAK"   "GRIP_MOVE"
             "GRIP_ROTATE" "GRIP_SCALE" "GRIP_MIRROR"))
    ) ;_ end of member
  (progn
    (setq *lostAssociativity* T)
    (princ "\nNOTE: The ")
    (princ currentCommandName)
    (princ " command will break a path's associativity .")
  ) ;_ end of progn
) ;_ end of if
```

This code checks to see if the user issued a command that breaks the associativity between the tiles and the path. If the user issued such a command, the program sets the `*lostAssociativity*` global variable and warns the user.

As you experiment with the garden path application, you may discover additional editing commands that can modify the garden path and cause the loss of associativity. Add these commands to the quoted list so that the user is aware of what will happen. When this function fires, the user has started a command but has not selected any entities to modify. The user could still cancel the command, leaving things unchanged.

Adding Activity to the Object Reactor Callback Functions

In Lesson 6, you registered two callback functions with object reactor events. The `gp:outline-erased` function was associated with the `:vlr-erased` reactor event, and `gp:outline-changed` was associated with the `:vlr-modified` event. You need to make these functions do what they are intended to do.

To make the object reactor callback functions do what they are intended to do

- 1 In `gpreact.lsp`, change `gp:outline-erased` so it appears as follows:

```
(defun gp:outline-erased (outlinePoly reactor parameterList)
  (setq *reactorsToRemove*
        (cons reactor *reactorsToRemove*))
  (princ)
) ;_ end of defun
```

There is just one operation performed here. The reactor attached to the polyline is saved to a list of all reactors that need to be removed. (Remember: though reactors are attached to entities, they are separate objects entirely, and their relationships to entities need to be managed just as carefully as regular AutoCAD entities.)

- 2 Change `gp:outline-changed` to reflect the following code:

```
(defun gp:outline-changed (outlinePoly reactor parameterList)
  (if *lostAssociativity*
    (setq *reactorsToRemove*
          (cons reactor *reactorsToRemove*))
    (setq *polytochange* outlinePoly
          *reactorsToChange* (cons reactor *reactorsToChange*)))
  (princ)
)
```

There are two categories of functions that can modify the polyline outline. The first category contains those commands that will break the path's associativity with its tiles. You checked for this condition in `gp:command-will-start` and set the `*lostAssociativity*` global variable accordingly. In this case, the tiles need to be erased, and the path is then in the user's hands. The other category is the grip mode of the `STRETCH` command, where associativity is retained and you need to straighten out the outline after the user has finished dragging a vertex to a new location.

The `*polytochange*` variable stores a VLA-Object pointer to the polyline itself. This will be used in the `gp:command-ended` function when it comes time to recalculate the polyline border.

Designing the gp:command-ended Callback Function

The `gp:command-ended` editor reactor callback function is where most action takes place. Until this function is called, the garden path border polylines are “open for modify;” that is, users may still be manipulating the borders in AutoCAD. Within the reactor sequence, you have to wait until AutoCAD has done its part of the work before you are free to do what you want to do.

The following pseudo-code illustrates the logic of the `gp:command-ended` function:

```
Determine the condition of the polyline.
  CONDITION 1 - POLYLINE ERASED (Erase command)
    Erase the tiles.
  CONDITION 2 - LOST ASSOCIATIVITY (Move, Rotate, etc.)
    Erase the tiles.
  CONDITION 3 - GRIP_STRETCH - REDRAW AND RE-TILE
    Erase the tiles.
    Get the current boundary data from the polyline.
    If it is a lightweight polyline,
      Process boundary data as 2D
    Else
      Process boundary data as 3D
    End if
  Redefine the polyline border (pass in parameters of the current
    boundary configuration, as well as the old).
  Get the new boundary information and put it into the format
    required for setting back into the polyline entity.
  Regenerate the polyline.
  Redraw the tiles (force ActiveX drawing).
  Put the revised boundary information back into the reactor
    named in *reactorsToChange*.
End function
```

The pseudo-code is relatively straightforward, but there are several important details buried in the pseudo-code, and they are things you would not be expected to know at this point.

Handling Multiple Entity Types

The first detail is that your application may draw two kinds of polylines: old-style and lightweight. These different polyline types return their entity data in different formats. The old-style polyline returns a list of twelve reals: four sets of *X*, *Y*, and *Z* points. The lightweight polyline, though, returns a list eight reals: four sets of *X* and *Y* points.

You need to do some calculations to determine the revised polyline boundary after a user moves one of the vertices. It will be a lot easier to do the calculations if the polyline data has a consistent format.

The Lesson 7 version of the *utils.lsp* file contains functions to perform the necessary format conversions: **xyzList->ListOfPoints** extracts and formats 3D point lists into a list of lists, and **xyList->ListOfPoints** extracts and formats 2D point lists into a list of lists.

To add the code for converting polyline data into a consistent format

- 1 If you have a copy of *utils.lsp* open in a VLISP text editor window, close it.
- 2 Copy the version of *utils.lsp* from the *Tutorial\VisualLISP\Lesson7* directory into your working directory.

In addition to the two functions that reformat polyline data, *utils.lsp* contains additional utility functions needed in handling user alterations to the garden path.

- 3 Open *utils.lsp* in a VLISP text editor window and review the new code.

Using ActiveX Methods in Reactor Callback Functions

The second detail appearing in the pseudo-code shows up near the end, at the step for redrawing the tiles. Here is the pseudo-code statement:

```
Redraw the tiles (force ActiveX drawing)
```

The parenthetical phrase says it all: force ActiveX drawing. Why is this required? Why can't the application use the object creation preference stored in the association sublist?

The answer is you cannot use the **command** function for entity creation within a reactor callback function. This has to do with some internal workings of AutoCAD. You need to force the tile drawing routine to use ActiveX. You will hear more about this issue later in this lesson.

Handling Nonlinear Reactor Sequences

The final important detail deals with a quirk in the command/reactor sequence in AutoCAD when users modify a polyline by using the specialized GRIP commands. These commands, such as GRIP_MOVE and GRIP_ROTATE, are available from a shortcut menu after you select the grip of an object and right-click. The reactor sequence is not as linear as a simple MOVE or ERASE command. In effect, the user is changing to a different command while in the midst of another. To demonstrate this situation, you can load the code

from Lesson 6 that traces the sequence of reactor events. Or simply review the following annotated VLISP Console window output to see what happens:

```
;; To start, select the polyline and some of the circles by using a
;; crossing selection box. The items in the selection set--
;; the chosen circles and the polyline--are now shown with grips on.
;; To initiate the sequence, click on one of the polyline grips:
(GP:COMMAND-WILL-START #<VLR-Command-reactor> (GRIP_STRETCH))

;; Now change the command to a move by right-clicking and choosing
;; MOVE from the pop-up menu. Notice that the command-ended
;; reactor fires in order to close out the GRIP_STRETCH command
;; without having fired an object reactor event:
(GP:COMMAND-ENDED #<VLR-Command-reactor> (GRIP_STRETCH))
(GP:COMMAND-WILL-START #<VLR-Command-reactor> (GRIP_MOVE))

;; Now drag the outline (and the selected circles) to a new location.
(GP:OUTLINE-CHANGED #<VLA-OBJECT IAcadLWPolyline 028f3188>
  #<VLR-Object-reactor> nil)
(GP:COMMAND-ENDED #<VLR-Command-reactor> (GRIP_MOVE))
```

This demonstrates that you cannot be certain your object reactor callbacks will be called in all cases.

There is a related quirk in this sequence. Even during the final command-ended callback, the circles that are still part of the grip selection set cannot be deleted. These circles are still open by AutoCAD. If you attempt to erase them during the command-ended callback, you can crash AutoCAD. To get around this, you can use another global variable to store a list of pointers to the tile objects until they can be deleted.

To process nonlinear reactor sequences

1 Add the following function to the *gpreact.lsp* file:

```
(defun gp:erase-tiles (reactor / reactorData tiles tile)
  (if (setq reactorData (vlr-data reactor))
      (progn
        ;; Tiles in the path are stored as data in the reactor.
        (setq tiles (cdr (assoc 100 reactorData)))
        ;; Erase all the existing tiles in the path.
        (foreach tile tiles
          (if (and (null (member tile *Safe-to-Delete*))
                  (not (vlax-erased-p tile)))
              (progn
                (vla-put-visible tile 0)
                (setq *Safe-to-Delete* (cons tile *Safe-to-Delete*)))
              )
          )
        )
      (vlr-data-set reactor nil)
  )
)
```

This new function will be used in the first phase of erasing tiles. Notice that the tiles are not actually erased: they are made invisible and are added to a global variable named **Safe-to-Delete**.

- 2 Add the following function to the *gpreact.lsp* file:

```
(defun Gp:Safe-Delete (activeCommand)
  (if (not (equal
    (strcase (substr activeCommand 1 5))
    "GRIP_"
  )
  )
  (progn
    (if *Safe-to-Delete*
      (foreach Item *Safe-to-Delete*
        (if (not (vlax-erased-p Item))
          (vla-erase item)
        )
      )
    )
    (setq *Safe-to-Delete* nil)
  )
)
```

This function can be invoked at a time when a GRIP_MOVE or GRIP_STRETCH command is not being executed.

Coding the command-ended Function

Now that you have seen the pseudo-code and handled some important details, replace the stubbed-out code in the **gp:command-ended** reactor callback with the following:

```
(defun gp:command-ended (reactor      command-list
  /      objReactor
  reactorToChange reactorData
  coordinateValues currentPoints
  newReactorData  newPts
  tileList
  )
  (cond
    ;; CONDITION 1 - POLYLINE ERASED (Erase command)
    ;; If one or more polyline borders are being erased (indicated
    ;; by the presence of *reactorsToRemove*), erase the tiles
    ;; within the border, then remove the reactor.
    (*reactorsToRemove*
     (foreach objReactor *reactorsToRemove*
       (gp:erase-tiles objReactor)
     )
     (setq *reactorsToRemove* nil)
    )
  )
```

```

;; CONDITION 2 - LOST ASSOCIATIVITY (Move, Rotate, etc.)
;; If associativity has been lost (undo, move, etc.), then
;; erase the tiles within each border
;;
((and *lostassociativity* *reactorsToChange*)
 (foreach reactorToChange *reactorsToChange*
  (gp:erase-tiles reactorToChange)
 )
 (setq *reactorsToChange* nil)
 )

;; CONDITION 3 - GRIP_STRETCH
;; In this case, the associativity of the tiles to the path is
;; kept, but the path and the tiles will need to be
;; recalculated and redrawn. A GRIP_STRETCH can only be
;; performed on a single POLYLINE at a time.
((and (not *lostassociativity*)
 *polytochange*
 *reactorsToChange*
 (member "GRIP_STRETCH" command-list)
 ;; for a GRIP_STRETCH, there will be only one reactor in
 ;; the global *reactorsToChange*.
 (setq reactorData
  (vlr-data (setq reactorToChange
   (car *reactorsToChange*)
  )
 )
 )
 )

;; First, erase the tiles within the polyline border.
(gp:erase-tiles reactorToChange)
;; Next, get the current coordinate values of the polyline
;; vertices.
(setq coordinateValues
 (vlax-safearray->list
  (vlax-variant-value
   (vla-get-coordinates *polyToChange*)
  )
 )
 )

;; If the outline is a lightweight polyline, you have
;; 2d points, so use utility function xyList->ListOfPoints
;; to convert the coordinate data into lists of
;; ((x y) (x y) ...) points. Otherwise, use the
;; xyzList->ListOfPoints function that deals
;; with 3d points, and converts the coordinate data into
;; lists of ((x y z) (x y z) ...) points.
(setq CurrentPoints
 (if (= (vla-get-ObjectName *polytochange*) "AcDbPolyline")
  (xyList->ListOfPoints coordinateValues)
  (xyzList->ListOfPoints coordinateValues)
 )
 )

```

```

;; Send this new information to RedefinePolyBorder -- this
;; will return the new Polyline Border
(setq NewReactorData
  (gp:RedefinePolyBorder CurrentPoints reactorData)
)
;; Get all the border Points and ...
(setq newpts (list (cdr (assoc 12 NewReactorData))
  (cdr (assoc 13 NewReactorData))
  (cdr (assoc 14 NewReactorData))
  (cdr (assoc 15 NewReactorData))
)
)
;; ...update the outline of the polyline with the new points
;; calculated above. If dealing with a lightweight polyline,
;; convert these points to 2D (since all the points in
;; newpts are 3D), otherwise leave them alone.
(if (= (cdr (assoc 4 NewReactorData)) "LIGHT")
  (setq newpts (mapcar '(lambda (point)
    (3dPoint->2dPoint Point)
  )
    newpts
  )
)
)
)
)
;; Now update the polyline with the correct points.
(vla-put-coordinates
 *polytochange*
)
;; For description of the list->variantArray see utils.lsp.
(gp:list->variantArray (apply 'append newpts))
)
)
;; Now use the current definition of the NewReactorData,
;; which is really the same as the garden path data
;; structure. The only exception is that the field (100)
;; containing the list of tiles is nil. This is OK since
;; gp:Calculate-and-Draw-Tiles does not require this field
;; to draw the tiles. In fact this function creates the tiles
;; and returns a list of drawn tiles.
(setq tileList (gp:Calculate-and-Draw-Tiles
  ;; path data list without correct tile list
  NewReactorData
  ;; Object creation function
  ;; Within a reactor this *MUST* be ActiveX
  "ActiveX"
)
)
)

```


defined in *gpdraw.lsp*.) Here is the part of the function that declares the parameters and local variables:

```
(defun gp:Calculate-and-Draw-Tiles (BoundaryData /
  PathLength      TileSpace
  TileRadius      SpaceFilled
  SpaceToFill     RowSpacing
  offsetFromCenter rowStartPoint
  pathWidth       pathAngle
  ObjectCreationStyle TileList)
```

Notice only that one parameter is currently specified, and `ObjectCreationStyle` is identified as a local variable. Review how the `ObjectCreationStyle` variable is set, which is a little farther into the function:

```
(setq ObjectCreationStyle (strcase (cdr (assoc 3 BoundaryData))))
```

The `ObjectCreationStyle` is currently set internally within the function by retrieving the value tucked away in the `BoundaryData` variable (the association list). But now you need to be able to override that value.

To modify `gp:Calculate-and-Draw-Tiles` to accept an object creation style argument

- 1 Add the `ObjectCreationStyle` variable to the function argument.
- 2 Remove `ObjectCreationStyle` from the local variables.

The `defun` statement for the function should look like the following:

```
(defun gp:Calculate-and-Draw-Tiles (BoundaryData
  ObjectCreationStyle
  / PathLength      TileSpace
  TileRadius      SpaceFilled
  SpaceToFile     RowSpacing
  offsetFromCenter rowStartPoint
  pathWidth       pathAngle
  TileList)      ; remove ObjectCreationStyle from locals
```

Note that if you declare a variable both as a parameter (before the slash) and as a local variable (after the slash), VLISP will point this out to you. For example, if you declare `ObjectCreationStyle` as both a parameter and a variable, then use the VLISP syntax checking tool on the `gp:Calculate-and-Draw-Tiles` function, the following message will appear in the Build Output window:

```
; *** WARNING: same symbol before and after / in arguments list:
OBJECTCREATIONSTYLE
```

- 3 Modify the first `setq` expression within `gp:Calculate-and-Draw-Tiles` so that it looks like the following (changes indicated in bold):

```
(setq
  PathLength (cdr (assoc 41 BoundaryData))
  TileSpace (cdr (assoc 43 BoundaryData))
  TileRadius (cdr (assoc 42 BoundaryData))
  SpaceToFill (- PathLength TileRadius)
  RowSpacing (* (+ TileSpace (* TileRadius 2.0))
    (sin (Degrees->Radians 60))
  )
  SpaceFilled RowSpacing
  offsetFromCenter 0.0
  offsetDistance /(+(* TileRadius 2.0)TileSpace)2.0)
  rowStartPoint cdr (assoc 10 BoundaryData))
  pathWidth cdr (assoc 40 BoundaryData))
  pathAngle cdr (assoc 50 BoundaryData))
) ;_ end of setq
(if (not ObjectCreationStyle)
  (setq ObjectCreationStyle (strcase (cdr (assoc 3 BoundaryData)))))
)
```

The original assignment statement for `ObjectCreationStyle` has been removed. The code now checks to see if a value has been provided for `ObjectCreationStyle`. If `ObjectCreationStyle` is not set (that is, the value is `nil`), the function assigns it a value from the `BoundaryData` variable.

There is one more series of changes you need to make to `gp:Calculate-and-Draw-Tiles`.

Modifying Other Calls to `gp:Calculate-and-Draw-Tiles`

In the reactor callback, a hard-coded string "Activex" is passed to `gp:Calculate-and-Draw-Tiles` as the `ObjectCreationStyle` argument. But what about the other times `gp:Calculate-and-Draw-Tiles` is invoked?

If you remember back to Lesson 4, it was pointed out that whenever you change a stubbed-out function, you need to ask the following questions:

- Has the function call (invocation) changed? That is, does the function still take the same number of arguments?
- Does the function return something different?

The same questions need to be asked any time you make a significant change to a working function as you build, refine, and update your applications. In this case, you need to find any other functions in your project that invoke `gp:Calculate-and-Draw-Tiles`. VLISP has a feature that helps you do this.

To find all calls to `gp:Calculate-and-Draw-Tiles` in your project

- 1 In the VLISP text editor window, double-click on the word `gp:Calculate-and-Draw-Tiles` within the `gpdraw.lsp` file.
- 2 Choose Search ► Find from the VLISP menu.
Because you preselected the function name, it is already listed as the string to search for.
- 3 Select the Project button listed under Search in the Find dialog box.
When you select this option, the Find dialog box expands at the bottom, and you can select the project to be searched.
- 4 Specify your current project name, then press the Find button.
VLISP displays the results in the Find output window:



```
<Find output>
Search in 5 Files in project gpath6 ...
E:/Program Files/ACAD2000/Tutorial/VisualISP/lesson6/GPDRW.lsp
;;;      drawing function gp:Calculate-and-Draw-Tiles.
;;;      Function: gp:Calculate-and-Draw-Tiles
;;;      (defun gp:Calculate-and-Draw-Tiles (BoundaryData /
E:/Program Files/ACAD2000/Tutorial/VisualISP/lesson6/GPMAIN.lsp
      (setq tilelist (gp:Calculate-and-Draw-Tiles gp_PathData))
      ;; gp:Calculate-and-Draw-Tiles) to the gp_PathData variable.
5 occurrences found
```

- 5 Look at the results in the Find Output window and determine whether there are any other locations in your code where you make a call to `gp:Calculate-and-Draw-Tiles`. There should only be one: a location within `gpmain.lsp`.
- 6 In the Find Output window, double-click on the line of code calling `gp:Calculate-and-Draw-Tiles`.
VLISP activates a text editor window and takes you right to that line of code in `gpmain.lsp`. The code currently appears as follows:

```
(setq tilelist (gp:Calculate-and-Draw-Tiles gp_PathData))
```

- 7 Replace the line of code with the following:

```
(setq tilelist (gp:Calculate-and-Draw-Tiles gp_PathData nil))
```

Why `nil`? Take another look at the pseudo-code:

If `ObjectCreationStyle` is `nil`, assign it from the `BoundaryData`.

Passing `nil` as a parameter to `gp:Calculate-and-Draw-Tiles` causes that function to check the user's choice of how to draw the tiles (as determined by the dialog box selection and stored in `gp_PathData`). Subsequent calls from the command-ended reactor callback, however, will override this behavior by forcing the use of `ActiveX`.

Congratulations! You now have the basic reactor functionality in place. If you prefer, copy the *gpmain.lsp* and *gpdraw.lsp* files from the *Tutorial\VisualLISP\Lesson7* into your working directory and examine the completed, debugged code.

Your celebration party may be somewhat short-lived. There is still a lot of work to be done, and it is all triggered from this rather innocuous looking fragment of code in the **gp:Command-ended** function:

```
(setq NewReactorData
      (gp:RedefinePolyBorder CurrentPoints reactorData)
) ;_ end of setq
```

Redefining the Polyline Boundary

You have worked hard to get to this point, and your brain has probably had enough new concepts, terms, commands, and imperatives for a while. With that in mind, it is recommended that you copy the sample code supplied with the tutorial, rather than entering it on your own.

To copy the code used to redefine the polyline boundary

- 1 Copy the file *gppoly.lsp* from the *Tutorial\VisualLISP\Lesson7* directory into your working directory.
- 2 In the project window for your project, press the Project Properties button.
- 3 Add the *gppoly.lsp* file to the project.
- 4 Press OK to accept the project with the additional file.
- 5 In the project window, double-click the *gppoly.lsp* file to open it.

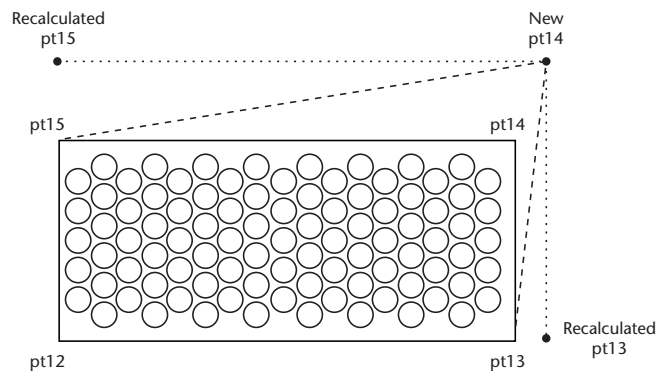
Looking at the Functions in *gppoly.lsp*

The file *gppoly.lsp* contains a number of functions required for straightening a polyline when a single grip has been stretched. Only some of these functions will be explained in depth in this tutorial.

NOTE This section of the Garden Path tutorial contains some of the most complex code and concepts in the entire lesson. If you are a beginner, you may want to jump ahead to the “Building an Application” section later in this chapter.

The functions within the *gppoly.lsp* file are organized in a way that you may have noticed in other AutoLISP source code files. The highest-level function, often the main, or **c:** function (in this case, **gp:Redefine-PolyBorder**), is located at the bottom of the file. The functions called within the main function are defined above it within the source file. This convention goes back to the old days of programming, when some development environments required that files be organized this way. With VLISP, this is a matter of personal style; there is no requirement that you organize your functions in any specific sequence.

Before diving into the details, step back and look at what needs to be done to recalculate and draw the garden path boundary. The following illustration shows an example of a garden path, along with the association list key points stored in the reactor data:



In this example, the 12 key point is the lower-left corner, 13 is lower-right, and so on. If the user moves the upper-right point (the 14 key point), the program will need to recalculate two existing points—the lower-right (13) and upper-left (15).

Understanding the **gp:RedefinePolyBorder** Function

The following pseudo-code shows the logic behind the main function, **gp:RedefinePolyBorder**:

```
Function gp:RedefinePolyBorder
  Extract the previous polyline corner points (12, 13, 14, and 15
    key values).
  Find the moved corner point by comparing the previous
    polyline corner points with the current corner points.
    (The one "misfit" point will be the point that moved.)
```

```
Set the new corner points by recalculating the two points
adjacent to the moved point.
Update the new corner points in the reactor data (that will
be stored back in the reactor for the modified polyline).
Update other information in the reactor data. (Start point,
endpoint, width, and length of path need to be recalculated.)
```

Understanding the `gp:FindMovedPoint` Function

The `gp:FindMovedPoint` function contains some very powerful LISP expressions dealing with list manipulation. Essentially, what this function does is compare the list of the current polyline points (after the user dragged one to a new location) to the previous points, and return the keyed list (the 13 <xvalue> <yvalue>) for the moved point.

The best way to figure out how this function works is to step through the code and watch the values that it manipulates. Set a breakpoint right at the first expression (`setq result . . .`) and watch the following variables while you step through the function:

- `KeyListToLookFor`
- `PresentPoints`
- `KeyedList`
- `Result`
- `KeyListStatus`
- `MissingKey`
- `MovedPoint`

The `mapcar` and `lambda` functions will be examined in the following section. For now, however, follow the comments in the code to see if you can understand what is happening within the functions.

Understanding the `gp:FindPointInList` Function

The function header in the source code explains how `gp:FindPointInList` transforms the information it works with. Like the previous function, `Gp:FindMovedPoint`, this function uses LISP's list manipulation capabilities to perform the work. When operating with lists, you will often see the `mapcar` and `lambda` functions used together as they are here. At first, these are strange and confusing functions, with names that do not indicate what they do. Once you learn how to use them, however, you will find them to be two of the most powerful functions within the AutoLISP repertoire. What follows is a brief overview of `mapcar` and `lambda`.

The **mapcar** function applies (maps) an expression to every item in a list. For example, given a list of the integers 1, 2, 3, and 4, **mapcar** can be used to apply the **1+** function to add 1 to each number in the list:

```
_$ (mapcar '1+ '(1 2 3 4))  
(2 3 4 5)
```

An initial definition for **mapcar** is that it maps the function given in the first parameter to the successive items in the second parameter—the list. The resulting value from a **mapcar** operation is the list transformed by whatever function or expression was applied to it. (Actually, **mapcar** can do more than that, but for now this definition will suffice.)

In the supplied example, every value in the list '(1 2 3 4) was passed to the **1+** function. Essentially, **mapcar** performed the following operations, assembling the resulting values in a list:

```
(1+ 1)  -> 2  
(1+ 2)  -> 3  
(1+ 3)  -> 4  
(1+ 4)  -> 5
```

Here is another example of **mapcar**, this time using the **null** function to test whether or not the values in a list are null (not true) values:

```
_$ (mapcar 'null (list 1 (= 3 "3") nil "Steve"))  
(nil T T nil)
```

What happened in this code was essentially the following:

```
(null 1)      -> nil  
(null (= 3 "3")) -> T  
(null nil)   -> T  
(null "Steve") -> nil
```

You can use many existing AutoLISP functions within a **mapcar**. You can also use your own functions. For example, imagine you have just created a very powerful function named **equals2**:

```
_$ (defun equals2(num) (= num 2))  
EQUALS2  
_$ (mapcar 'equals2 '(1 2 3 4))  
(nil T nil nil)
```

OK, so **equals2** is not all that powerful. But it is in such cases that **lambda** comes in handy. You can use **lambda** in cases where you do not want or need to go through the overhead of defining a function. You will sometimes see **lambda** defined as an anonymous function. For example, instead of defining a function called **equals2**, you could write a **lambda** expression to perform the same operation without the overhead of the function definition:

```
_$ (mapcar '(lambda (num) (= num 2)) '(1 2 3 4))  
(nil T nil nil)
```

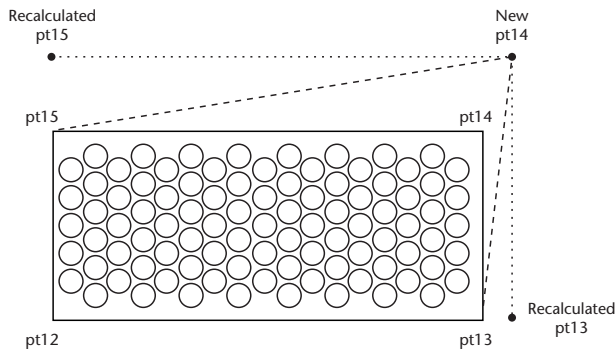
What happened in the code was this:

```
(= 1 2)    -> nil  
(= 2 2)    -> T  
(= 3 2)    -> nil  
(= 4 2)    -> nil
```

With this knowledge, see if the `gp:FindPointInList` function makes sense. Again, review the comments within the source code.

Understanding the `gp:recalcPolyCorners` Function

The key to understanding how `gp:recalcPolyCorners` works is to revisit the diagram showing what the key values of 12 through 15 represent:



In the diagram, the user moved the corner point associated with the key value of 14. This means the corner points associated with 13 and 15 need to be recalculated.

Point 15 needs to be moved along the current vector defined by point 12 to point 15 until it lines up with the new point 14. The vectors from 12 to 15, and from 14 to 15, must be perpendicular to each other. The same operation must be applied to recalculate the new location for point 13.

Now take another look at the code to see if it makes sense.

Understanding the `gp:pointEqual`, `gp:rtos2`, and `gp:zeroSmallNum` Functions

These three functions are required to get around one of the quirks of programming in an AutoCAD system, which, as you are well aware, allows you a great deal of precision. Occasionally, though, numbers are not quite precise

enough, due to the rounding up or down of floating point values defining geometric positions. You must be able to compare one set of points with other points, so you must deal with these cases.

Have you ever noticed that occasionally, when you list the information associated with an AutoCAD entity, you see a value such as $1.0e-017$? This number is *almost* zero, but when you are comparing it to zero within a LISP program, *almost* does not count.

Within the garden path, you need to be able to compare numbers without having to worry about the fact that $1.0e-017$ is not quite zero. The `gp:pointEqual`, `gp:rtos2`, and `gp:zeroSmallNum` functions handle any discrepancies in rounding when comparing point lists.

This completes your tour of the functions in *gppoly.lsp*.

Wrapping Up the Code

So far, you have done the following in this lesson:

- Modified the `gp:drawOutline` function so that it returns the polyline perimeter points in addition to the pointer to the polyline. You added this information to the `gp_PathData` variable. This variable is stored with the reactor data in the object reactor attached to every garden path.
- Updated the reactor functions in *gpreact.lsp*.
- Added functions `xyzList->ListOfPoints`, `xyList->ListOfPoints`, and other utility functions to the *utils.lsp* file.
- Updated the `gp:Calculate-and-Draw-Tiles` function so that `ObjectCreationStyle` is now a parameter to the function rather than a local variable.
- Modified the call to `gp:Calculate-and-Draw-Tiles` in the `C:GPath` function within the *gpmain.lsp* file.
- Added *gppoly.lsp* to your project, and examined the functions within it.

Give the completed application a try. Save your work, then load in the project sources, run the `Gpath` function, and try stretching and moving the garden path boundary. Remember: if something is not working and you are unable to debug the problem, you can load the completed code from the *Tutorial\VisualLISP\Lesson7* directory.

Building an Application

The final act within this tutorial is to take your garden path code and turn it into a standalone application. This way, it can be distributed as a single executable to any user or customer. Fortunately, this last set of tasks is probably the easiest in this entire tutorial, as VLISP does practically all the work for you.

NOTE It is recommended that you proceed with building an application only if your code is in good working form. Make sure that you have tested your application using the original source files, and that you are satisfied with the results.

Starting the Make Application Wizard

To assist you in creating standalone applications, VLISP provides the Make Application wizard.

To run the Make Application wizard

- 1 To start the wizard, choose File ► Make Application ► New Application Wizard from the VLISP menu.
- 2 Select Expert mode and press Next.
The wizard prompts you to specify the directory in which to store files created by Make Application, and to name your application. Make Application produces two output files: a *.vlx* file containing your program executable, and a *.prv* file containing the options you specify to Make Application. The *.prv* file is also known as a *make* file. You can use the make file to rebuild your application, when necessary.
- 3 Specify your *Tutorial\VisualLISP\MyPath* directory as the application location, and call the application **gardenpath**. VLISP uses the application name in the output file names (in this instance, *gardenpath.vlx* and *gardenpath.prv*.)
Press Next to continue.
- 4 The application options are not covered in this tutorial. Accept the defaults and press Next. (For information on separate namespace applications, see “Running an Application in Its Own Namespace” in the Visual LISP Developer’s Guide.)
- 5 In this step, the wizard is prompting you to identify all the AutoLISP source code files that make up the application. You could individually select the LISP source files, but there is an easier way. Change the pull-down file type

selection box so that “Visual LISP Project file” is shown, then press the Add button. Select the *Gpath* project file and press Open.

NOTE Depending on how you worked through the tutorial, you may have several *Gpath* project files showing up. Select the most recent file. If you copied in the completed source code from Lesson 7, the project name to select should be *Gpath7.prj*.

After selecting the project file, press Next to continue.

- 6 One advantage of compiled VLX applications is that you can compile your dialog control files (*.dcl*) into the complete application. This reduces the number of individual source files your end users need to deal with, and eliminates any of the search path problems when loading a DCL file.

Change the pull-down file type selection box so that “DCL files” is shown, then press the Add button. Select the *gpdialog.dcl* file, then press Open.

Press Next to continue building the application.

- 7 Compilation options are not covered in this tutorial. Accept the defaults and press Next. (For information on compile options, see “Optimizing Application Code” in the *Visual LISP Developer’s Guide*.)
- 8 The final step is used to review the selections you’ve made. At this point, you can select Finish. VLISP will begin the build process, displaying the results in the Build Output window. Several intermediate files will be produced, as your individual source code files are compiled into a format that can be linked into the single VLX application.

When it is all complete, you’ll have an executable file named *gardenpath.vlx*. To test it, do the following:

- From the Tools menu in AutoCAD, select Load Application.
- Load the *gardenpath.vlx* application that was just created and is found in the *Tutorial\VisualLISP\MyPath* directory.
- Run the **gpath** command.

Wrapping Up the Tutorial

You have finally made it to the end of the path! As you have discovered, a lot of material was covered in this tutorial. Both AutoLISP concepts and VLISP operations have been introduced. The “garden path revisited” was designed to give you a sampling of many topics and concepts. You might be interested in more information. The following is a brief bibliography of some common LISP and AutoLISP books.

LISP and AutoLISP Books

AutoLISP Books

AutoLISP: Programming for Productivity, William Kramer, Autodesk Press, ISBN 0-8273-5832-6.

Essential AutoLISP, Roy Harkow, Springer-Verlag, ISBN 0-387-94571-7.

AutoLISP in Plain English: A Practical Guide for Non-Programmers, George O. Head, Ventana Press, ISBN: 1566041406.

General LISP Books

LISP, 3rd Edition, Patrick Henry Winston and Berthold Klaus Paul Horn, Addison-Wesley Publishing Company, ISBN 0-201-08319-1.

ANSI Common Lisp, Paul Graham, Prentice Hall, ISBN 0-13-370875-6.

Looking at LISP, Tony Hasemer, Addison-Wesley Publishing Company, ISBN 0-201-12080-1.

Common LISP, The Language, Second Edition, Guy L. Steele, Jr., Digital Press, ISBN 1-55558-041-6.

Index

Symbols

- `()`, matching, 70
- `//`, as DCL comment code, 56
- `;`, as AutoLISP comment code, 21, 24
- `>` button (Project Properties dialog box), 54
- `_ $` prompt, in Console window, 12
- `{}`, in DCL, 58
- 2D points
 - converting 3D points to, 35–36
 - converting lists of, to lists of lists, 105
- 3D points
 - at specified angle and distance from a point, 41
 - converting lists of, to list of lists, 105
 - converting to 2D points, 35–36
- 3dPoint->2dPoint function, 35–36

A

- action_tile function, 61–63
- actions, assigning to tiles, 61–63
- ActiveX functions
 - command function and, 37, 81
 - entity creation. *See* object creation
 - entmake function and, 37, 81
 - global variables and, 44
 - loading ActiveX, 43, 47
 - model space pointer and, 43–44, 47–48
 - object creation using, 37, 42–49, 57, 80, 105, 110–114
 - object model structure for, 42
 - passing parameters to, 38
 - radio button for, 56
 - reactors and, 105, 110–114
 - real numbers required by, 35
 - return values from, 35, 42
 - translating VBA syntax into ActiveX calls, 42
- variants
 - constructing from a list of points, 44–46
 - defined, 44
- vla- prefix and, 42

- Add Watch button (Watch window), 29
- Add Watch command (Debug menu), 20
- adding
 - files to projects, 114
 - reactor callback functions, 91
 - variables to Watch window, 20, 29–30
 - See also* assigning; attaching; creating; defining
- analyzing. *See* inspecting; watching variables
- angles
 - converting degrees to radians, 34–35, 40–41
 - drawing path at any angle, 40–41
- anonymous functions, 117–118
- append function, 45, 77
- applications, 120
 - creating, 120–121
- apply function, 45
- arguments. *See* parameter passing
- arrays of polyline points, constructing, 44–45
- arrow (>) button (Project Properties dialog box), 54
- assigning
 - actions to tiles, 61–63
 - See also* adding; attaching
- assoc function, 40
- association lists (assoc lists), 17–19
 - advantages, 18
 - in gp:getPointInput function, 17–19, 20, 22–23, 31
 - inspecting (parsing), 39–40
 - key values, 17–18, 40
 - passing to functions, 39–40
 - using, 18–19
- associativity between tiles and path, breaking, 98, 102, 103, 104
- atof function, 63
- attaching
 - reactors, 87
 - See also* adding; assigning; detaching

AutoCAD
command line, 37
crashing, 92, 96, 106
DCL files and, 58
emulating in AutoLISP programs, 15
entities. *See* entities
minimized window, gpath prompts and, 12
reactors and. *See* reactors
waiting for control to return from, 7

AutoLISP code
checking, 8, 11
color syntax coding, 8, 58
commenting, 21, 24
debugging. *See* debugging
modularizing, 52–53
revising, 21–23, 112–114
selecting blocks of, 70
testing, 23, 81
automatic garbage collection, 14
automatic word completion, 71–72
Automation objects, 42

B

blocks of code, selecting, 70
boundaries
ActiveX and, 42–48
angle of path and, 40–41
dialog box creation for, 56–58
dialog box defaults for, 59
erasing, reactors and, 85–94, 96–97, 103, 107
line styles
default type, 59
defining radio buttons for specifying, 56–57
drawing specified style, 66–67
reactors and, 104–105, 107–110
modifying
GRIP commands and, 106, 107
grip mode of STRETCH command and, 98, 103, 114–119
reactors and, 85–94, 98–119
vertices
comparing, 119
determining, 40–41, 118
finding moved points, 116
finding points in lists, 116–118
reactors and moving, 85–94, 98–119
variant array of, 44–46
See also points
See also gp:drawOutline function
BoundaryData variable, 39–40, 111
boxed_radio_column DCL directive, 56
braces ({}), in DCL, 58
breakpoints, 24–31
clearing all, 31
clearing individual, 26, 31

breakpoints (*continued*)
cursor and, 27
Debug toolbar and, 25–26, 27
defined, 24
setting, 26–27
stepping through code from, 27–29
Btm button (Project Properties dialog box), 54
Build Output window
application building and, 121
syntax checking and, 11
building. *See* creating
buttons
assigning actions for dialog boxes, 62–63
creating for dialog boxes, 56, 57
Debug toolbar, 25–26, 27–29, 31
Project Properties dialog box, 53–54
VLISP toolbar, 25, 44, 53
Watch window, 20

C

C: (prefix), 9
C:GPath function, 7–8, 22–23, 48, 65–66, 67, 88–91, 101
Calculate-and-Draw-Tiles function, 74–77, 110–114
Calculate-Draw-TileRow function, 76–81
callback functions
ActiveX and, 105, 110–114
adding, 91
adding functionality to, 102–103
defined, 84
editor reactors and, 86–87, 96, 98, 104–107
object reactors and, 86, 96, 98, 103, 106, 111–119
planning and designing, 85–86, 104
See also specific callback functions
Cancel button
defining in DCL, 56
procedure when user presses, 63–64
case sensitivity, Help system and, 44
cdr function, 40
changing
boundaries
GRIP commands and, 106, 107
grip mode of STRETCH command and, 98, 103, 114–119
reactors and, 85–94, 98–119
value of variables while program is running, 30
See also clearing; detaching; erasing; removing
Check Text in Editor command (Tools menu), 11
checking code. *See* syntax checking
circles. *See* tiles (garden path)
clean-all-reactors function, 92
CleanReactors function, 92

- Clear All Breakpoints command (Debug menu), 31
- clearing
 - all breakpoints, 31
 - individual breakpoints, 25, 31
 - See also* detaching; erasing; removing
- command function
 - ActiveX and, 37, 80
 - entity creation via, 57, 81, 105
 - radio button for, 57
 - reactors and, 88, 105
- command line, in AutoCAD, 37
- command-ended function. *See*
 - gp:command-ended function
- commands
 - entering in Console window, 12
 - See also specific commands*
- command-will-start function, 97, 102, 103
- comments
 - AutoLISP, 21, 24
 - DCL, 56
- comparing floating point values, 119
- Complete Word by Apropos feature, 72
- Complete Word by Match command (Search menu), 71
- completing words automatically, 71–72
- concatenating lists, 45
- Console window
 - entering commands in, 12
 - history command, 17
- constructing. *See* creating
- Continue button (Debug toolbar), 31
- converting
 - degrees to radians, 34–35, 40–41
 - lists of points to variant arrays, 44–46
 - real numbers to strings, 61
 - strings to real numbers, 63
 - 3D point lists to list of lists, 105
 - 3D points to 2D points, 35–36
 - 2D point lists to list of lists, 105
- corner points. *See* boundaries, vertices
- crashing AutoCAD, reactors and, 92, 94, 106
- creating
 - applications, 120–121
 - boundaries. *See* boundaries
 - breakpoints, 26–27
 - buttons in dialog boxes, 56
 - dialog box default values, 59
 - dialog boxes, 56–58
 - entities (objects). *See* objects, creating
 - files, 8, 120–121
 - projects, 53–55
 - tiles. *See* tiles (garden path)
 - variant arrays of polyline points, 44–46
 - See also* adding; assigning; attaching; defining

- curly braces {}, in DCL, 58
- cursor, breakpoints and, 27

D

- database reactors, 84
- DCL (dialog control language)
 - AutoCAD and, 58
 - comment code for, 56
 - defining dialog boxes, 55–58
 - loading files, 60–61
 - previewing files, 58–59
 - saving files, 58
 - See also* dialog boxes
 - sources of information on, 55
 - syntax coloring scheme and, 58
 - unloading files, 63
 - VLX applications and, 121
- .dcl file extension, 55, 121
- Debug menu
 - Add Watch command, 20
 - Clear All Breakpoints command, 31
 - Watch Last Evaluation command, 30
- Debug toolbar
 - breakpoints and, 25–26, 27
 - Continue button, 31
 - described, 25–26
 - Reset button, 29
 - Step Indicator, 26, 27
 - Step Into button, 28, 30, 31
 - Step Out button, 27, 30–31
 - Step Over button, 27–28
 - stepping through code and, 25, 27–32
 - Toggle Breakpoint button, 26, 31
- debugging, 14–32
 - association lists and, 17–19
 - breakpoints and, 24–28
 - changing value of variables while program is running, 30
 - commenting and, 24
 - global variables and, 14–15, 30, 31
 - local variables and, 14–17, 29–30
 - modularizing, 52
 - revising code, 21–23
 - stepping through code, 27–31
 - watching variables, 20, 29–30
- defaults, dialog box values, 56, 59, 61
- defining
 - dialog boxes, 55–58
 - functions. *See* defun function statement
 - goals, 6
 - See also* adding; creating
- defun function statement
 - ActiveX loading and, 47
 - described, 9
 - global variable assignment and, 47–48
- defun function statement
 - stubbed-out functions and, 10–11

defun function statement (*continued*)
 variable declaration and, 15

Degrees->Radians function, 34–35, 38–39, 41

deleting. *See* clearing; detaching; erasing;
 removing

designing
 programs, 5–12
 reactor callback functions, 85–86, 104
See also planning

detaching
 toolbars, 25
See also attaching; clearing; erasing;
 removing

dialog boxes, 55–64
 assigning actions to tiles, 61–63
 default values, 56, 59, 61
 defining, 55–58
 initializing, 61
 interfacing with, from AutoLISP code,
 59–64
 loading, 60–61
 overview, 55
 previewing, 58–59
 saving, 58
 starting, 63
 unloading, 63
 VLX applications and, 121
See also specific dialog boxes

dialog control language (DCL). *See* DCL (dialog
 control language)

DialogInput function. *See* gp:getDialogInput
 function

dialogLoaded variable, 60

dialogShow variable, 60, 61

directories
 source code, 3
 working directory, 3

displaying program output, 9

document reactors, 84

done_dialog function, 63

double slash (//), as DCL comment code, 56

doubles, arrays of, 44–46

drawing. *See* creating

drawOutline function. *See* gp:drawOutline
 function

DXF reactors, 84

E

Edit menu, Parentheses Matching command, 70

editor reactors
 callback functions and, 86–87, 96, 98,
 104–110
 described, 84, 86, 87
 types of, 84
 where to attach, 87

EndPt variable, 16, 17, 29–30, 31

Enter Expression to Watch dialog box, 20

entget function, entmake function versus, 37

entities. *See* objects

entmake
 entity creation via, 80

entmake function
 ActiveX and, 37, 56, 80
 entget function versus, 37
 entity creation via, 37
 radio button for, 56

erase-tiles function, 106–107

erasing
 boundaries and tiles, reactors and, 85–94,
 96–97, 103, 107
 tiles and redrawing when boundaries
 change, reactors and, 85–94, 98,
 104–119
See also clearing; detaching; removing

error checking
 color coding and, 8, 58
 syntax checker, 11
See also debugging

error messages, displaying, 9

executing. *See* running

exiting
 programs quietly, 9
 stepping through code, 30–31

F

File menu
 Make Application command, 120–121
 New File command, 8
 Save All command, 68
 Save As command, 58

files
 adding to projects, 114
 creating, 8
 creating application, 120–121
 loading
 DCL files, 60–61
 program files, 12, 68
 project files, 55, 68
 make, 120
 managing via projects, 53–55
 modularizing, 52–53
 organizing contents of source code, 115
 saving all, 68
 saving DCL, 58
 unloading DCL, 63

Find command (Search menu), 113

finding. *See* searching

FindMovedPoint function, 116

FindPointInList function, 116–118

floating point values
 comparing, 119
See also real numbers

Format AutoLISP in Editor command (Tools
 menu), 9

formatting, 8
 forward slashes (*//*), as DCL comment code, 56
 functions
 anonymous, 117–118
 callback. *See* callback functions
 checking, 8, 11
 commenting, 21, 24
 debugging. *See* debugging
 defining. *See* defun function statement
 exiting quietly, 9
 help for, 44, 72–73
 loading, 12
 naming
 finding and completing names, 72–73
 organizing within source code files, 115
 passing parameters to. *See* parameter passing
 polar, 41, 76
 return values. *See* return values
 revising, 21–23, 112–114
 running, 12
 on individual elements in lists, 45,
 116–118
 stubbed-out
 defining, 10–11
 updating, 65–66
 testing, 23, 81
 variables. *See* variables
See also specific functions; utility functions

G

garbage collection, automatic, 14
 garden path
 angle of, 40–41
 illustrated, 6, 77
 width of, 59, 65, 66
 See also boundaries; gp:drawOutline
 function; gp:getDialogInput
 function; gp:getPointInInput
 function; tiles (garden path)
gardenpath.prv file, 120
gardenpath.vlx file, 7, 120–121
 getDialogInput function. *See* gp:getDialogInput
 function
 getdist function, 16
 getpoint function, 16
 getPointInInput function. *See* gp:getPointInInput
 function
 global variables, 14–15
 ActiveX entity creation and, 44
 debugging and, 14–15, 30, 31
 defined, 14
 defun function statement and, 47–48
 local versus, 14
 model space and, 43, 47–48
 reactors and, 87, 96–97, 102, 103, 106,
 107
 uses for, 44

global variables (*continued*)
 See also specific global variables
 goals, defining, 6
 gp: prefix, 9
 gp:Calculate-and-Draw-Tiles function, 74–77,
 110–114
 gp:Calculate-Draw-TileRow function, 76–81
 gp:clean-all-reactors function, 92
 gp:command-ended function, 85–87, 104–105,
 107–110
 code for, 107–110
 designing, 85–86, 104
 gp:outline-changed function and, 85–86
 gp:outline-erased function and, 96, 97
 multiple entity types and, 104–105
 multiple reactors and, 86–87
 gp:command-will-start function, 97, 102, 103
 gp:drawOutline function, 38–48
 ActiveX and, 42–48
 angle of path and, 40–41
 basic code for, 46–48
 boundary line style selection code for,
 66–67
 passing parameters to, 39–48
 pointer to model space, 42–44, 47–48
 purpose of, 9, 11
 reactors and, 99–101
 return values
 polyline perimeter point list, 99–101
 polyline pointer, 43–47
 stubbing-out, 10–11
 vertices
 setting up, 40–41
 variant array of, 44–47
 See also boundaries
 gp:erase-tiles function, 106–107
 gp:FindMovedPoint function, 116
 gp:FindPointInList function, 116–118
 gp:getDialogInput function
 adding dialog box to, 55–66
 purpose of, 9
 return value, 65
 stubbing-out, 10–11
 gp:getPointInInput function, 16–31
 angles and, 40
 breakpoints and, 24–28
 how it works, 16–17
 local variables in, 15, 30–31
 purpose of, 9–11
 return values
 as 3D points, 35–36
 association lists of, 17–19, 20, 22–23,
 31
 regular lists of, 16–17
 storing in variables, 19
 revising code, 21–23
 stepping through, 25–31

gp:getPointInput function (*continued*)
 stubbing-out, 10–11
 watching variables, 20, 29–30
 gp:list->variantArray function, 45–46
 gp:outline-changed function, 85–86, 103
 gp:outline-erased function, 96, 97, 103
 gp:pointEqual function, 119
 gp:recalcPolyCorners function, 118
 gp:RedefinePolyBorder function, 115
 gp:rtos2 function, 119
 gp:Safe-Delete function, 107
 gp:zeroSmallNum function, 119
 gp_dialogResults variable, 66
 gp_PathData variable
 described, 22–23
 gp:drawOutline function and, 39, 48
 gp:getDialogInput function and, 19, 66
 reactors and, 88–91, 99–101
 watching value of, 20
 gp_spac variable, 63
 gp_trad variable, 63
 gpath command, 7, 9
 GPath function. *See* C:GPath function
 gpath.prj file, 53
 gpdialog.dcl file, 58, 59, 121
 gpdraw.lsp file, 66, 70, 71, 72, 73, 80
 gp-io.lsp file, 52
 gpmain.lsp file
See also C:GPath function
 modularizing, 52–53
 position in project files, 54
 reactors and, 88–91
 return values and, 10
 using the supplied, 11
 gppoly.lsp, 114–119
 gpreact.lsp file, 91–92, 102–104, 106–107
 GRIP commands, reactors and, 106, 107
 grip mode of STRETCH command, reactors and,
 98, 103, 114–119

H

HalfWidth variable, 16, 17, 29–31
 Help button (VLISP toolbar)
 help, for functions, 44–45, 72–73
 highlighting blocks of code, 70
 history command, 17

I

I-beam cursor, in Debug toolbar, 27
 initializing dialog boxes, 61
 input. *See* gp:getDialog Input function;
 gp:getPointInput function; *entries*
beginning with get, vla-get, and vlax-get
 Inspect command (View menu), 39–40

Inspect window
 Add Watch command, 20
 Inspect command, 39
 inspecting
 association lists, 39–40
 variables, 19, 20
 installing tutorial, 3
 integers, ActiveX functions and, 35
 Interface Tools command (Tools menu), 58, 59

K

key values
 AutoCAD and, 17–18
 in associated lists, 17–18, 40

L

lambda function, 116–118
 Last-Value variable, 31, 30
 lightweight boundary lines. *See* boundaries, line
 styles
 line styles. *See* boundaries, line styles
 linker reactors, 84
 list->variant Array function, 45–46
 lists
 association. *See* association lists
 comparing lists of points, 119
 concatenating, 45
 converting 2D point lists to list of, 105
 converting 3D point lists to list of, 105
 converting lists of points to variant arrays,
 45–46
 executing functions on individual elements
 in, 116, 118
 finding points in, 116, 118
 of return values, 16
 association lists. *See* association lists
 passing to functions
 association lists, 39, 40
 regular lists, 45
 testing for null values in, 117
 Load Text in Editor command (Tools menu), 12
 load_dialog function, 60
 loading
 ActiveX, 43, 47
 DCL files, 60–61
 functions, 12, 43, 47
 programs, 12, 68
 project files, 54, 68
See also unloading
 selected code, 3, 39, 70, 106
 local variables, 14–16
 declaring, 15, 111
 defined, 14
 global versus, 14

local variables (*continued*)
nil return values and, 31
See also specific local variables
watching, 20, 29
lostAssociativity variable, 102, 103

M

Make Application command (File menu),
120–121
make files, 120
managing files via projects, 53–55
mapcar function, 45, 116, 117
matching parentheses, 70
memory, local and global variables and, 14
messages, displaying program, 9
minimized windows
AutoCAD window, gpath prompts and, 12
VLISP, accessing, 53
model space
global variables and, 43–44, 47, 48
obtaining a pointer to, 43–44, 47, 48
obtaining a pointer to global variables
model space and, 43
ModelSpace variable, 42, 44, 47, 53
modifying. *See* changing
modularizing, 52, 53
Mouse reactors, 84
moving
boundaries. *See* boundaries, modifying
toolbars, 25

N

naming functions, 9
finding and completing names, 72–73
new features, 2
New File command (File menu), 8
New Project command (Project menu), 53, 54,
55
new_dialog function, 61
NewReactorData parameter, 111
nil parameters, 113
nil return values, 10, 31
non-linear reactor sequences, 106–107
null values, testing for, in lists, 117
numbers
ActiveX functions and, 35
comparing floating point values, 119
converting real numbers to strings, 61
converting strings to real numbers, 63
integer versus real, 35
passing to functions, 38, 39

O

object reactors
callback functions and, 99, 103, 106

object reactors (*continued*)
described, 84, 87
where to attach, 87
ObjectARX applications, reactors and, 84, 99
ObjectCreationFunction variable, 80
ObjectCreationStyle parameter, 80, 110–114
ObjectCreationStyle variable, 110, 111
objects
ActiveX object model structure, 42
Automation objects, 42
creating
dialog box for, 56–58
with ActiveX, 37, 42–48, 57, 80, 105,
110–114
with command function, 57, 81
with entmake function, 37, 57
handling multiple types of, 104–105
key values and, 17–18
reactors and. *See* object reactors
returning entity data to AutoLISP, 17–18
VLA-objects, 42
objects. *See* entities (objects)
OK button
assigning actions to, 62
defining in DCL, 57
procedure when user presses, 62–64
Open Project command (Project menu), 68
opening
project files, 54, 55, 68
See also loading
organizing functions within source code files,
115
outline. *See also* gp:drawOutline function
outline-changed function, 85, 86, 103, 106
outline-erased function, 86, 96, 97, 103
output
displaying program, 11
vla-put functions, 42
See also gp:drawOutline function

P

parameter passing, 38, 39, 40
association lists, 39, 40
declaring variables as both parameters and
local variables, 111
lists, 45
nil parameters, 113
numbers, 38, 39
variables, 39
See also specific parameters
parentheses [()], matching, 70
Parentheses Matching command (Edit menu),
70
parsing association lists, 39, 40
passing parameters
ActiveX functions, 42
passing parameters. *See* parameter passing

path. *See* garden path
 PathAngle variable, 40, 41
 PathLength variable, 41
 pausing program execution. *See* breakpoints
 planning

- reactor callback functions, 85–86, 104
- reactors, 95, 102
- See also* designing
- utility functions, 33–36

 pline variable, 42, 43, 47
 pointEqual function, 118
 pointers

- to model space, 43–44, 47
- VLISP pointer, 7

 PointInput function. *See* gp:getPointInput function
 points

- 3D points at specified angle and distance from a point, 41
- comparing, 118
- converting 3D points to 2D points, 35–36
- converting lists of 2D/3D points to list of lists, 105
- determining corner, 41, 118
- finding in lists, 116, 117, 118
- finding moved, 116
- reactors and moving, 85–94, 97–119
- variant array of, 44–46

 polar function, 41, 76
 polyline borders. *See* boundaries
 PolylineList variable, 101
 polyPoints variable, 100, 101
 polyToChange variable, 97, 102, 103
 previewing dialog boxes, 58
 princ function, 9
 .prj file extension, 68
 programs. *See* code; functions
 Project menu

- New Project command, 53–54
- Open Project command, 68

 Project Properties button (project window), 91, 114
 Project Properties dialog box, 53, 54
 project window, 55, 114
 projects

- adding files to, 114
- creating, 53–55
- described, 53
- loading and running all files in, 68
- searching, 113

 prompts

- _\$_ prompt in Console window, 12
- displaying program, 9
- minimized AutoCAD window and, 12

 .prv file extension, 120

Q

quiet exit, 9
 quoted symbols, 11

R

radians, converting degrees to, 34–35, 41
 radio buttons

- in dialog boxes, 56, 57
- See also* buttons

 radio_column DCL directive, 56
 radius of tiles

- default, 57, 61
- specifying, 56

 reactors, 83–114

- ActiveX and, 105, 110–114
- attaching, 87
- AutoLISP applications and, 113
- for boundaries, 85–94, 97–119
- callback functions
 - ActiveX and, 105, 110–114
 - adding, 91
 - adding functionality to, 102–110
 - defined, 84
 - editor reactors and, 86–87, 96, 97–98, 104–107
 - object reactors and, 86, 96, 97, 103, 106, 110–119
 - planning and designing, 85–87, 104
 - See also specific callback functions*
- crashing AutoCAD and, 92, 94, 96, 106
- defined, 84
- editor
 - callback functions and, 97–98, 104–107
 - described, 84, 86, 87
 - where to attach, 87
- editors
 - types of, 84
- event selection, 85
- global variables and, 87, 96–97, 102–103, 106
- GRIP commands and, 106, 107
- grip mode of STRETCH command and, 98, 103, 114–119
- multiple, 86–87
- multiple entity types and, 104–105
- non-linear sequences of, 105–107
- object
 - callback functions and, 86, 96, 97, 103, 106, 111–119
 - described, 84, 87
 - where to attach, 87
- ObjectARX applications and, 84, 99
- planning the overall process, 96–101
- removing, 92, 96
- returning to VLISP from AutoCAD and, 93

- reactors (*continued*)
 - storing data with, 88, 99–101
 - testing, 92–94
 - tiles and, 85–94, 96–97, 98, 103–119
 - trace sheets for, 93–94, 97, 105
 - transient versus persistent, 94
 - types of, 84, 86, 87
- *reactorsToRemove* variable, 96
- real numbers
 - ActiveX functions and, 35
 - converting real numbers to strings, 61, 63
 - floating point values, 119
- recalcPolyCorners function, 118
- RedefinePolyBorder function, 115
- reference works. *See* resources
- reformatting, 9
- regular lists, 16
- removing
 - reactors, 92, 96
 - See also* clearing; detaching; erasing
- requirements, 2
- Reset button (Debug toolbar), 29
- resources
 - DCL, 55
 - LISP and AutoLISP, 122
- return values
 - ActiveX functions, 43
 - defined, 10
 - gp:drawOutline function, 46, 47, 99–101
 - gp:getDialogInput function, 19
 - gp:getPointInput function.
 - See* gp:getPointInput function, return values
 - lists of
 - association. *See* association lists
 - regular, 16–17
 - nil return, 10, 31
 - storing in variables, 19
 - true return, 10
- revising code, 21–23, 112–114
- rotating boundaries. *See* boundaries, modifying
- rows. *See* tiles (garden path)
- rowStartPoint variable, 76
- rtos function, 61
- rtos2 function, 118, 119
- running
 - and stepping through, 27–31
 - breakpoints with, 24–31
 - dialog boxes, 63
 - functions, 12
 - on individual elements in lists, 45, 117
 - garden path example, 7
 - programs, 12, 68
 - project files, 68
 - tutorial, 2
 - VLISP, 7
- S**
 - Safe-Delete function, 107
 - *Safe-to-Delete* function, 107
 - Save All command (File menu), 68
 - Save As command (File menu), 8
 - saving
 - all files, 68
 - DCL files, 58
 - scaling boundaries. *See* boundaries, modifying
 - Search menu
 - Complete Word by Match command, 71
 - Find command, 113
 - searching
 - for closest match to complete a word, 71
 - for moved points, 116
 - for points in a list, 116–118
 - projects, 113
 - Select Window button (VLISP toolbar), 53
 - semicolon (;), as AutoLISP comment code, 21, 24
 - set_tile function, 61
 - setting. *See* adding; assigning; creating; defining
 - slashes (/), as DCL comment code, 56
 - source code
 - CD containing, 3
 - directories, 3
 - organizing functions within files of, 115
 - spacing of rows, 74
 - spacing of tiles, 74
 - default, 59, 65
 - specifying, 57
 - start_dialog function, 61, 63
 - starting
 - dialog boxes, 63
 - See also* creating; running
 - StartPt variable, 16, 17, 29, 31
 - Step Indicator (Debug toolbar), 26, 27
 - Step Into button (Debug toolbar), 25, 28, 30, 31
 - Step Out button (Debug toolbar), 25, 31
 - Step Over button (Debug menu), 30
 - Step Over button (Debug toolbar), 25, 28
 - stepping through code, 27–32
 - Debug toolbar and, 25, 27–32
 - exiting, 31–32
 - from breakpoints, 27, 28–29, 30, 31
 - watching variables while, 29–30
 - storing
 - data with reactors, 88, 99–101
 - return values in variables, 19
 - stretching boundaries, reactors and, 98, 103, 114–119
 - strings
 - converting real numbers to, 61
 - converting to real numbers, 63

stubbed-out functions
 defining, 10–11
 updating, 65–66
suspending program execution. *See* breakpoints
symbols
 quoted, 11
 T, 10
syntax checking
 automated, 11
 color coding and, 8, 58

T

T (symbol), 10
testing
 code, 23, 81
 reactor code, 92–94
 See also debugging; syntax checking
“3D”, index entries beginning with. *See Symbols section of this index*
tileList variable, 77
tiles (dialog box component), 56
tiles (garden path)
 erasing when boundary is erased, 85–94, 96–97, 103, 107
 first row of, 76
 illustrated, 6, 73
 radius and tile spacing
 default, 59, 65
 specifying, 57
 reactors and, 85–94, 96–97, 103–119
 redrawing when boundary is modified, 85–94, 98, 104–119
 row spacing, 74
 row-offset pattern, 73–74, 76–77
Toggle Breakpoint button (Debug toolbar), 25, 26, 31
toolbars
 moving, 25
 See also Debug toolbar; VLISP toolbar
Tools menu
 Check Text in Editor command (Tools menu), 11
 Format AutoLISP in Editor command (Tools menu), 9
 Interface Tools command, 58
 Load Text in Editor command, 12
Top button (Project Properties dialog box), 54
trace sheets, reactor, 93–94, 97, 105
true return value, 10
tutorial, overview, 3
“2D”, index entries beginning with. *See Symbols section of this index*

U

unload_dialog function, 60, 63

unloading
 DCL files, 63
utility functions
 3dPoint->2dPoint, 35–36
 gp:list->variantArray function, 45–46
 xyList->ListOfPoints, 105
 xyzList->ListOfPoints, 105
utils.lsp file, 52, 53, 54, 91, 92, 105, 105

V

values
 for radio buttons in DCL, 56
 See also key values; return values; variables
variables
 breakpoints and, 24
 changing value while program is running, 30
 declaring, 38
 passing to functions, 39
 storing return values in, 19
variants (ActiveX)
 constructing arrays of polyline points, 44–46
 defined, 44
VBA syntax, translating into AutoLISP, 42
vertices. *See* boundaries, vertices; points
View menu
 Inspect command, 39–40
Visual LISP Tutorial, 1
Visual LISP. *See* entries beginning with VLISP
vla- (prefix), 42
vla-addLightweightPolyline function, 42, 70, 72
vla-get functions, 42
vla-get-ActiveDocument function, 43
vla-get-ModelSpace function, 43
VLA-objects, 42
vla-put functions, 42
vla-put-closed function, 42
vlax-get-Acad-Object function, 43
vlax-make-safearray function, 46
vlax-make-variant function, 46
vlax-safearray-fill function, 46
vlisp command, 7
VLISP development environment
 accessing minimized windows, 53
 described, 2
 reactors and returning from AutoCAD to, 93
 VLISP pointer in, 7
 waiting for control to return from AutoCAD, 7
VLISP menu
 Debug command. *See* Debug menu
 Edit command. *See* Edit menu
 File command. *See* File menu
 Project command. *See* Project menu
 Search command. *See* Search menu

- VLISP menu (*continued*)
 - Tools command. *See* Tools menu
 - View command. *See* View menu
 - Window command, 53
- VLISP toolbar
 - Help button, 72
 - Select Window button, 53
- vl-load-com function, 43
 - vlr-commandEnded events, 87–91, 99
 - vlr-commandWillStart events, 99, 102
 - vlr-erased event, 103
 - vlr-modified event, 85
 - vlr-modified events, 103
- vlr-object-reactor function, 88–91
- VLX applications, 2, 120–121
- .vlx file extension, 120

W

- Watch command (Debug menu), Add, 20
- Watch Last Evaluation command (Debug menu), 30
- Watch window
 - Add Watch button in, 29
 - Add Watch command and, 20
 - Watch Last Evaluation command and, 30
- watching variables, 20, 29–30
 - adding variables to Watch window, 20, 29–30
 - Debug toolbar and, 25
 - defined, 19
 - while stepping through code, 29–30
- width of path, 59, 65, 66
- Window command (VLISP menu), 53
- windows. *See* minimized windows; project window; *specific windows*
- words, completing automatically, 71–72
- working directory, 3

X

- xyList->ListOfPoints function, 105
- xyzList->ListOfPoints function, 105