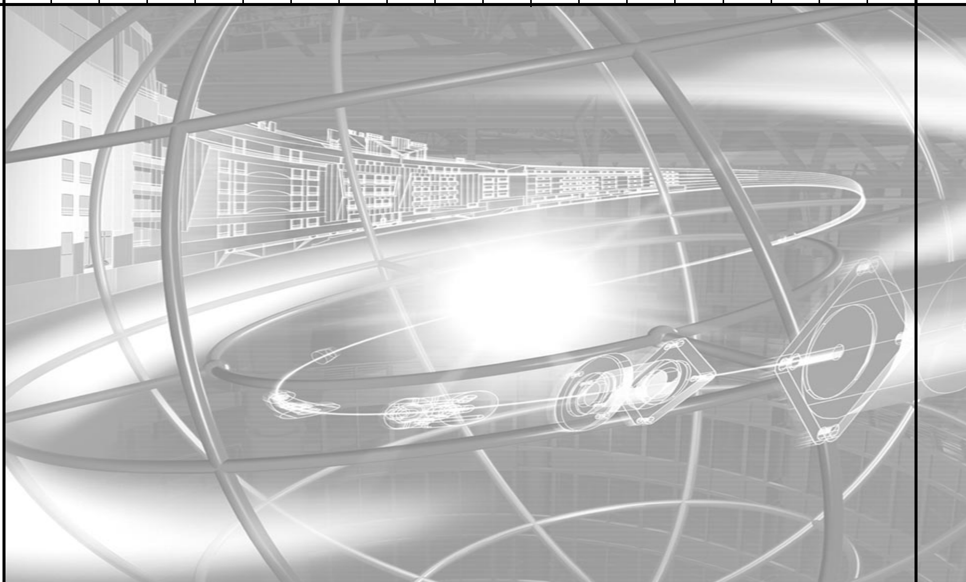
Autodesk

AutoCAD® 2000



VISUAL LISP™ 教程

00120-010000-5080 April 22, 1999

版权所有 © 1999 Autodesk, Inc.
保留所有权利

AUTODESK, INC. 对于本出版物中的资料及其实用性不承担任何明确或暗示的担保，包括且不仅限于适销性和针对特定用途的适用性的担保。

任何因购买和使用这些资料而造成的特殊、间接、偶然或必然的损害，AUTODESK, INC. 均不负任何责任。无论采取何种诉讼形式 AUTODESK, INC. 唯一且仅担负的责任总额，以不超过本出版物的售价为限。

Autodesk, Inc., 保留修订及改进产品的权利。本出版物仅描述其出版时的产品内容，并不反映产品未来的情况。

Autodesk 商标

以下列出了 Autodesk, Inc. 在美国和／或其他国家的注册商标：3D Plan、3D Props、3D Studio、3D Studio MAX、3D Studio VIZ、3DSurfer、ADE、ADI、Advanced Modeling Extension、AEC Authority（徽标）、AEC-X、AME、Animator Pro、Animator Studio、ATC、AUGI、AutoCAD、AutoCAD Data Extension、AutoCAD Development System、AutoCAD LT、AutoCAD Map、Autodesk、Autodesk Animator、Autodesk（徽标）、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、Drafix、Education by Design、Generic、Generic 3D Drafting、Generic CADD、Generic Software、Geodyssey、Heidi、HOOPS、Hyperwire、Inside Track、Kinetix、MaterialSpec、Mechanical Desktop、Multimedia Explorer、NAAUG、Office Series、Opus、PeopleTracker、Physique、Planix、RadioRay、Rastation、Softdesk、Softdesk（徽标）、Solution 3000、Tech Talk、Texture Universe、The AEC Authority、The Auto Architect、TinkerTech、*WHIP!*、*WHIP!*（徽标）、Woodbourne、WorkCenter 和 World-Creating Toolkit.

以下列出了 Autodesk, Inc. 在美国和／或其他国家的商标：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、AutoCAD 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（徽标）、ClearScale、Concept Studio、Content Explorer、cornerStone Toolkit、Dancing Baby（图像）、Design Your World、Design Your World（徽标）、DesignCenter、Designer's Toolkit、DWG Linking、DWG Unplugged、DXF、Exegis、FLI、FLIC、GDX Driver、Generic 3D、Heads-up Design、Home Series、Kinetix（徽标）、MAX DWG、ObjectARX、ObjectDBX,Ooga-Chaka、Photo Landscape、Photoscape、Plugs and Sockets、PolarSnap、Powered with Autodesk Technology、Powered with Autodesk Technology（徽标）、ProjectPoint、Pro Landscape、QuickCAD、SchoolBox、SketchTools、Suddenly Everything Clicks、Supportdesk、The Dancing Baby、Transforms Ideas Into Reality、Visual LISP、Volo 和 Where Design Connects.

第三方产品商标

Élan License Manager 是 Élan Computer Group, Inc. 的商标。Microsoft、Visual Basic、Visual C++ 和 Windows 是 Microsoft Corporation 在美国和／或其他国家的注册商标； Visual FoxPro 和 Microsoft Visual Basic Technology 徽标是 Microsoft Corporation 在美国和／或其他国家的商标。所有其他商标名称、产品名称或商标均属于其各自的持有人。

第三方软件程序说明

ACIS ® 版权所有 1994, 1997, 1999 Spatial Technology, Inc.、Three-Space Ltd. 和 Applied Geometry Corp. 保留所有权利。

版权所有 1996 Microsoft Corporation 保留所有权利。

International CorrectSpell™ Spelling Correction System 版权所有 1995 Lernout & Hauspie Speech Products, N.V 保留所有权利。

InstallShield™ 3.0. 版权所有 1997 InstallShield Software Corporation 保留所有权利。

部分版权所有 1991-1996 Arthur D. Applegate 保留所有权利。

本软件的一部分基于 Independent JPEG Group 的工作成果。

出自 Bitstream ® typeface library 的字体。版权所有 1992。

出自 Payne Loving Trust 的字体。版权所有 1996。保留所有权利。

本产品的许可管理部分基于 Élan License Manager 版权所有 1989, 1990, 1998 Élan Computer Group, Inc. 的工作成果，保留所有权利。

Autodesk 在此对 Perceptual Multimedia, Inc. 开发的 Visual LISP 花园小路教程表示感谢。

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.

1 2 3 4 5 6 7 8 9 10

目录

绪论 1

重访花园小路：使用 Visual LISP 2
教程概述 3

第一课 **设计程序，开始编程 5**

定义整个程序的目标 6
Visual LISP 入门 7
Visual LISP 代码格式 8
代码分析 9
填补程序 9
让 Visual LISP 检查您的代码 11
在 Visual LISP 中运行程序 12
第一课回顾 12

第二课 **使用 Visual LISP 调试工具 13**

局部变量和全局变量的区别 14
    在程序中使用局部变量 15
    检测函数 gp:getPointInput 16
使用关联表来捆绑数据 17
    使用关联表 18
    将 gp:getPointInput 的返回值保存到变量中 19
检查程序变量 19
修改程序代码 21
给程序代码加注释 24
设置断点并进行更多监视 24
    使用“调试”工具栏 25
    单步执行代码 27

单步执行程序时监视变量 29
步出函数 gp:getPointInput 并步入 C:Gpmain 30
第二课回顾 32

第三课 **绘制小路边界 33**
计划可重用工具函数 34
将度转换为弧度 34
将三维点转换为二维点 35
绘制 AutoCAD 图元 37
用 ActiveX 函数创建图元 37
使用 entmake 函数创建图元 37
使用 AutoCAD 命令行 37
编写绘制边界轮廓的函数 38
将参数传给函数 39
使用关联表 39
使用角度和设置点 40
理解 gp:drawOutline 中的 ActiveX 代码 42
确保加载 ActiveX 43
获取指向模型空间的指针 43
构造多段线端点数组 44
由点表构造变体 45
代码集成 46
第三课回顾 49

第四课 **创建工程，添加界面 51**
模块化代码 52
使用 Visual LISP 工程 53
添加对话框界面 55
用 DCL 定义对话框 55
保存 DCL 文件 58
预览对话框 58
用 AutoLISP 代码与对话框交互 59
设置对话框的值 59
加载对话框文件 60
将指定对话框加载到内存中 60
初始化缺省的对话框值 61
给控件指定动作 61
启动对话框 63
卸载对话框 63
确定下一步的动作 64
代码集成 64
更新简单空函数 65
提供可选的边界线类型 66

收尾工作 67
运行应用程序 68
第四课回顾 68

第五课 **绘制砖 69**

更多的 Visual LISP 编辑工具 70
括号匹配 70
自动完成词语 71
按系统匹配完成词语 72
获取函数帮助 72
在花园小路中添加砖 73
应用逻辑知识 73
几何分析 74
绘制一排砖 74
绘制每一排砖 77
阅读代码 78
测试代码 81
第五课回顾 81

第六课 **使用反应器 83**

反应器基础 84
反应器类型 84
设计花园小路应用程序的反应器 85
选择花园小路应用程序的反应器事件 85
规划回调函数 85
规划多重反应器 86
附着反应器 87
用反应器保存数据 88
更新 C:GPath 函数 88
添加反应器的回调函数 91
清除反应器 92
测试反应器 92
详细检查反应器的行为 93
第六课回顾 94

第七课 **程序集成 95**

规划反应器整体过程 96
响应更多的用户调用命令 97
将信息保存在反应器对象中 99
添加新的反应器功能 102
实现对象反应器的回调函数 103
设计回调函数 gp:command-ended 104

处理多种图元类型 104
在反应器回调函数中使用 ActiveX 方法 105
处理非线性的反应器序列 105
编写 command-ended 函数 107
更新 gp:Calculate-and-Draw-Tiles 110
修改其他调用 gp:Calculate-and-Draw-Tiles 函数的代码 112
重新定义多段线边界 114
查看 gppoly.lsp 文件中的函数 114
理解 gp:RedefinePolyBorder 函数 115
理解 gp:FindMovedPoint 函数 116
理解 gp:FindPointInList 函数 116
理解 gp:recalcPolyCorners 函数 118
理解 gp:pointEqual、gp:rtos2 和 gp:zeroSmallNum 函数 118
代码回顾 119
生成应用程序 120
使用“生成应用程序”向导 120
教程回顾 121
LISP 和 AutoLISP 参考书 122
AutoLISP 参考书 122
普通 LISP 参考书 122

绪论

本章内容包括

- 重访花园小路：使用 Visual LISP
- 教程概述

本教程用于示范 AutoCAD® 的 Visual LISP™ 编程环境的强大性能，介绍 AutoLISP® 语言的一些新特性。

本教程的目标是实现用自动绘图工具绘制花园小路。这些工具最大可能地节省了绘图时间，尽显参数化编程的威力。您将学习如何创建一个绘图程序，来自动生成复杂的形——这样就可以避免每次都要从头开始重复绘图工作。

重访花园小路：使用 Visual LISP

本教程是为精通 AutoCAD 的用户编写的，并假定您对 LISP 或 AutoLISP 比较熟悉。同时，您应了解基本的 Windows® 文件管理任务，如创建目录、复制文件、浏览硬盘或网络文件系统等。

如果您熟悉 AutoLISP 并使用过以前版本的“花园小路 (AutoLISP)”教程，那么您将注意到下面几个区别：

- 引入了 Visual LISP (VLISP™) 环境。该环境提供了创建 AutoLISP 应用程序所需的编辑、调试和其他工具。而以前版本的花园小路 (AutoLISP) 教程只讲授 AutoLISP 语言概念，而不涉及 VLISP 开发工具。
- 增加了对 AutoLISP 中新的 ActiveX™ 和反应器，以及 VLISP 提供的其他几个 AutoLISP 语言扩展的介绍。
- 重新设计了整个教程。即使您对以前版本的教程比较熟悉，您也会看到完全不同的源代码，同时得到更多指导。
- 花园小路教程可以以两种方式执行。一是 LISP 文件以解释型语言方式运行（可以是加载到单个文档中的多个 LISP 文件或函数），二是将程序代码编译为 VLX 应用程序（.vlx 文件）。VLX 应用程序在一个独立的名称空间中运行，它可以与应用程序加载的文档进行交互。

教程概述

在本教程中，您的目标是给 AutoCAD 添加一个新命令，该命令能够绘制一条铺圆形砖的花园小路。本教程共分为七课，随着课程不断深入，在完成单个任务上的指导将越来越简略。如果您有任何疑问，可以参考 VLISP 文档。

第四课和第五课超越了基本 AutoLISP 概念，相当于中级水平。而第六课和第七课则包括了相当复杂的高级编程技术，是为精通 AutoLISP 的开发者准备的。

绘制花园小路的每一步中的所有源代码都可以在 AutoCAD 安装光盘上找到，但只有在选择“完全”安装选项（或“自定义”安装选项并选取了“样例”条目）时才会安装这些教程文件。如果您已经安装了 AutoCAD 但没有安装样例，请重新运行安装程序，选择“自定义”安装选项，然后选取“样例”一项即可。

源代码文件所在的目录结构和教程的课程计划一致：

<AutoCAD 目录 >*Tutorial**VisualLISP**Lesson1*

<AutoCAD 目录 >*Tutorial**VisualLISP**Lesson2*

等等。

建议您不要修改 AutoCAD 提供的源代码样例文件。如果您的程序运行不正确，您可能需要将提供的源代码复制到您自己的工作目录中。在整个教程中设定的工作目录是：

<AutoCAD 目录 >*Tutorial**VisualLISP**MyPath*

如果您使用不同的工作目录，则应该将其替换成您的工作目录名。

最后，请先仔细阅读 Visual LISP 开发人员手册中的“Visual LISP 入门”一节。它简要介绍了一些完成本教程所必须了解的概念。

设计程序，开始编程

本课内容包括

1

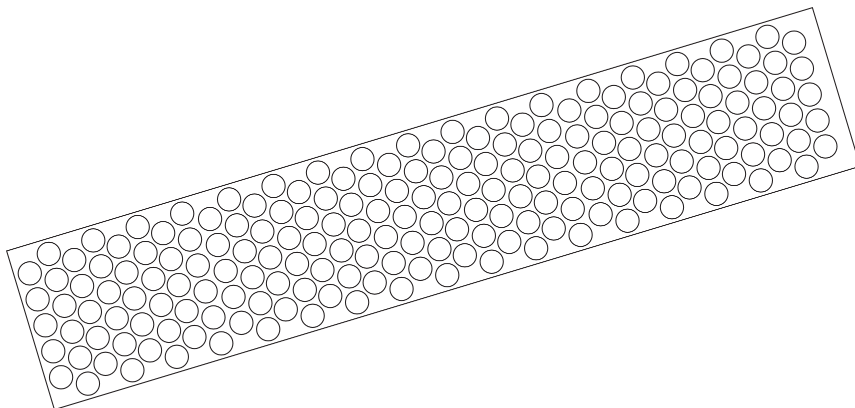
在第一课中，您首先定义整个应用程序要完成什么样的工作。然后利用 Visual LISP (VLISP) 开发环境创建 LISP 文件，开始为您的应用程序编写 AutoLISP 代码。

在这一过程中，您将开始发现 VLISP 是如何帮您开发应用程序的。

- 定义整个程序的目标
- Visual LISP 入门
- Visual LISP 代码格式
- 代码分析
- 填补程序
- 让 Visual LISP 检查您的代码
- 在 Visual LISP 中运行程序
- 第一课回顾

定义整个程序的目标

开发 AutoLISP 程序的出发点是为了实现某些 AutoCAD 操作的自动化，它可能是为了加快重复性绘图工作的步伐，或简化一系列复杂操作。在本教程中，您的程序要画的花园小路是一个复杂的形，它由许多部件组成，而且部件的数目和用户最初的输入有关。它完成后的样子大致如此：



要绘制这条花园小路，您的程序必须完成这些操作：

- 给定起点、端点和宽度，画一个矩形的边界。该边界的方向可以是任何二维方向，而且路的大小不受限制。
- 提示用户输入砖的大小和砖的间距。这些圆形砖将填充在边界线以内，不会重叠，也不会超出边界线。
- 要将砖成排交叉放置。

如果想看看程序是如何工作的，可以运行 AutoCAD 提供的该应用程序的完整版本。

运行所提供样例的步骤

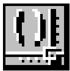
1 从 AutoCAD“工具”菜单选择“加载应用程序”。
2 从 Tutorial\VisualLISP 目录下选取 gardenpath.vlx，然后选择“加载”。
3 单击“关闭”。
4 在命令提示处，输入 **gpath**。
5 响应前两个提示，在 AutoCAD 图形窗口拾取起点和端点。
6 提示输入小路半宽时 (Half Width of Path) 输入 **2**。
7 显示“Garden Path Tile Specifications”对话框时单击 OK 按钮。

Visual LISP 入门

现在您已看到应用程序的运行情况，您可以开始在 VLISP 中开发该应用程序了。但首先，您可能应看看 VLISP 等待控制从 AutoCAD 返回时是怎样的（您可能已遇到过这种情况）。

观察 Visual LISP 如何等待控制从 AutoCAD 返回的步骤

1 在 AutoCAD 命令提示处，输入 **vlisp** 启动 Visual LISP。
2 切换回 AutoCAD 窗口（从任务栏上选取 AutoCAD，或按 ALT+TAB 并选择 AutoCAD），并在 AutoCAD 命令提示处输入 **gpath**。
3 在响应 **gpath** 的提示前，切换回 VLISP 窗口。

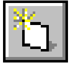


在 VLISP 窗口中，鼠标指针显示为 VLISP 符号状，您无法选取任何命令，也无法在 VLISP 窗口的任何位置输入文本。指针符号是为了提示您，在重新继续 VLISP 中的工作之前必须先完成 AutoCAD 中的操作。无论什么时候见到这种 VLISP 指针时，都请想到这点。

4 返回 AutoCAD 窗口，响应 **gpath** 的所有提示。

现在您可以开始编写“花园小路”应用程序了。

用 **VLISP** 开发应用程序的步骤



1 从 VLISP 的“文件”菜单，选择“新建文件”。

2 在文本编辑窗口（窗口标题为 < 未命名 0>）中输入下述代码，可省略其中的注释：

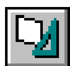
```
;;; 函数 C:GPath 是程序主函数，定义
;;; AutoCAD 的 GPATH 命令。
(defun C:GPath ()
 ;; 要求用户输入：首先输入路径位置和方向，
 ;; 然后输入路径参数，只有在输入了有效参数
 ;; 之后才继续工作。
 (if (gp:getPointInput)      ;
   (if (gp:getDialogInput)
     (progn
       ;; 在此处已获取用户的有效输入。
       ;; 画出轮廓，将结果多段线对象名存入名为
       ;; PolylineName 的变量。
       (setq PolylineName (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)  ; 静默退出
)
;;; 显示一条消息，让用户知道命令名。
(princ "\nType gpath to draw a garden path.")
(princ)
```

3 从菜单上选择“文件”➢“另存为”，将代码保存为新文件 <AutoCAD 目录 >*\Tutorial\VisualLISP\MyPath\gpmain.lsp.*

4 检查输入的代码是否正确。

Visual LISP 代码格式

VLISP 可识别组成 AutoLISP 程序文件的各种字符和词，并将字符着以不同颜色。这样您可以很快地发现那些有错的地方。例如，如果您忘了输入文本字符串后面的闭引号，接着输入的所有字符都会显示成洋红色，因为洋红色代表文本字符串。当您输入闭引号后，VLISP 将把随后的文本根据它们所表示的语言元素种类正确着色。



当您输入文本时，VLISP 也会通过添加空格和设置缩进来设置其格式。如果要 VLISP 设置您从其他文件复制到 VLISP 文本编辑器的代码的格式，请从 VLISP 菜单中选择“工具”➢“设置编辑器中代码的格式”。

代码分析

defun 函数用来定义新函数。注意主函数名为 **C:GPath**，前缀 **C:** 使得可以从 AutoCAD 命令行调用该函数，**GPath** 是用户在 AutoCAD 命令行运行该应用程序时需输入的名称。用来获取用户输入的函数被命名为 **gp:getPointInput** 和 **gp:getDialogInput**，用来画花园小路轮廓的函数是 **gp:drawOutline**。这些函数名加上前缀 gp: 是为了说明它们是用于“花园小路”应用程序的函数。虽然这并不是必要的，但这是一个好的命名习惯，它可将专用于应用程序的函数和您经常使用的通用工具函数区分开来。

在主函数中，如果程序运行成功，**princ** 表达式会显示程序的结果，如果程序碰到非预期事件，它会显示警告信息。例如，正如您在第二课中所看到的，如果用户按 ENTER 而不是在屏幕上拾取点，则将使 **gp:getPointInput** 函数调用提前结束，将 nil 值返回主函数中。这会使程序发出一个 **princ** 消息：Incomplete information to draw a boundary。

程序倒数第二次调用 **princ** 的目的是给用户一个提示。加载该应用程序时，该提示告诉用户需要键入什么命令来绘制花园小路。最后不带字符串参数调用 **princ** 则是使程序静默退出，即不返回主函数最后一个表达式的值。如果省略最后一个 **princ**，则前一个 princ 的提示将显示两次。

填补程序

为了让该新文件的代码正确运行，您还必须完成三个函数的定义。主程序中包含对下述三个自定义函数的调用：

- **gp:getPointInput**
- **gp:getUserInput**
- **gp:drawOutline**

现在您可以只给出函数最简单的定义。函数的简单定义就象是占位符，目的是使整个大程序完整。这样，您就可以在给应用程序加上所有细节之前运行已有代码。

给应用程序加上函数简单定义的步骤

1 将光标定位到文本编辑窗口中程序代码的顶部，键入 ENTER 数次以添加一些空行。

2 在插入空行的位置，输入如下代码：

```
;;; 函数 gp:getPointInput 将获取路径的位置和尺寸
(defun gp:getPointInput()
 (alert
  "Function gp:getPointInput will get user drawing input"
 )
 ;; 现在返回 T，就好象该函数正确运行一样。
 T
)

;;; 函数 gp:getDialogInput 将获取路径参数
(defun gp:getDialogInput ()
 (alert
  "Function gp:getDialogInput will get user choices via a dialog"
 )
 ;; 现在返回 T，就好象该函数正确运行一样。
 T
)

;;; 函数 gp:drawOutline 将绘制小路边界
(defun gp:drawOutline ()
 (alert
  (strcat "This function will draw the outline of the polyline"
                             "\nand return a polyline entity name/pointer."
  )
 )
 ;; 现在返回某引用符号，
 ;; 最后该函数应返回图元名或指针
 'SomeEname
)
```

每个输入函数的最后一行代码仅包括一个 T，这是用来返回一个值给调用它的函数。所有的 AutoLISP 函数都返回一个值给调用它们的函数，字母 T 在 AutoLISP 中是表示 True 的符号，加上它可使函数返回值 True。在 gpmain.lsp 中，程序已规定它调用的每个输入函数都必须返回一个不是 nil 的值（nil 表示“没有值”），否则程序无法执行下一步。

在缺省情况下，AutoLISP 函数将返回函数内最后一个计算的表达式的值。在函数的最简单的定义中，调用的唯一表达式是函数 **alert**，但函数 **alert** 总是返回 nil。如果它是 **gp:getPointInput** 中最后一个表达式，该函数将总是返回 nil，您永远也无法通过 **if** 执行到下面的 **gp:getDialogInput** 语句。

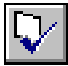
由于类似的原因，函数 **gp:DrawOutline** 的最后返回一个引用符号 ('SomeEname) 给调用它的函数。所谓引用符号，实际上是一种不被求值的 LISP 结构（如果您对 LISP 语言感兴趣，在本教程的最后列出了一些很好的参考书）。

让 Visual LISP 检查您的代码

VLISP 提供了一个强大的功能，可以检查代码中的语法错误。在尝试运行程序之前，请先使用该工具，这样您可发现一般的输入错误（如丢掉括号或引号等）和其他一些语法问题。

进行代码语法检查的步骤

1 确认包含 gpmain.lsp 的文本编辑窗口是活动窗口（单击该窗口的标题栏可使它成为活动窗口）。



2 从 VLISP 菜单中选择“工具”➢“检查编辑器中的文字”。

3 出现“< 编译输出 >”窗口并显示语法检查的结果。如果 VLISP 没有查到任何错误，窗口将包含类似以下的文本：

[检查文字 GPMAIN.LSP 正在加载 ...]
......
; 检查完成。

如果您碰到了问题并需要帮助，可参考 Visual LISP 开发人员手册中的“用 Visual LISP 开发程序”一章。看看您能否确定问题出在何处。如果这要花掉您太多时间，可使用 lesson1 目录下提供的样例文件 gpmain.lsp 继续学习本教程。

（在必要情况下）使用提供的 gpmain.lsp 程序的步骤

1 关闭包含输入的 gpmain.lsp 代码的文本编辑窗口。

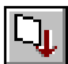
2 从 VLISP 菜单上选取“文件”➢“打开文件”，并打开 \Tutorial\VisualLISP\lesson1 目录下的 gpmain.lsp 文件。

3 选择“文件”➢“另存为”，并将该文件保存为目录 \Tutorial\VisualLISP\MyPath 下的 gpmain.lsp 文件，覆盖您创建的文件。

在 Visual LISP 中运行程序

在 VLISP 中运行 AutoLISP 程序，您就可使用 VLISP 的许多调试功能来研究应用程序中可能出现的问题。

加载和运行程序的步骤



1 在文本编辑窗口活动的情况下，从 VLISP 菜单上选取“工具”➢“加载编辑器中的文字”。
2 在 VLISP 控制台窗口的 _$ 提示处输入 **(C:GPath)**。

   控制台窗口认为命令是按 AutoLISP 语法输入的，所以所有函数名都必须包括在括号里。
3 按 ENTER 键或单击 OK 按钮以响应消息对话框，最后的一条消息应该是 Congratulations - your program is complete!。

注意 如果您运行 **gpath** 时 AutoCAD 已被最小化，在您恢复 AutoCAD 窗口（用任务栏或 ALT+TAB）前将不会看到提示。

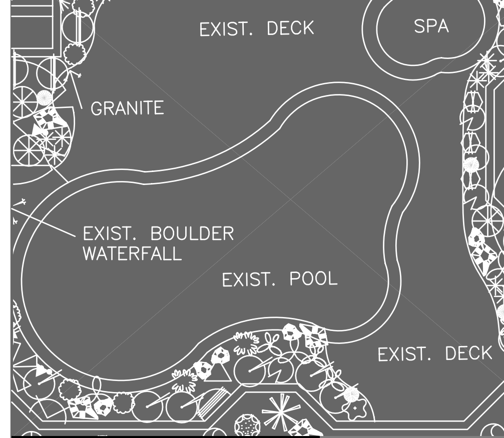
第一课回顾

在本课中，您

- 确定了程序的目标。
- 了解了简单定义函数的作用。
- 学会了如何命名函数以标明它是专用于您应用程序还是被反复使用的通用函数。
- 学会了如何使用 VLISP 检查代码。
- 学会了如何在 VLISP 中加载和运行程序。

您已完成了本课的学习，请再次保存您的程序文件以确保它是最新的版本。

使用 **Visual LISP** 调试工具



本课内容包括

2

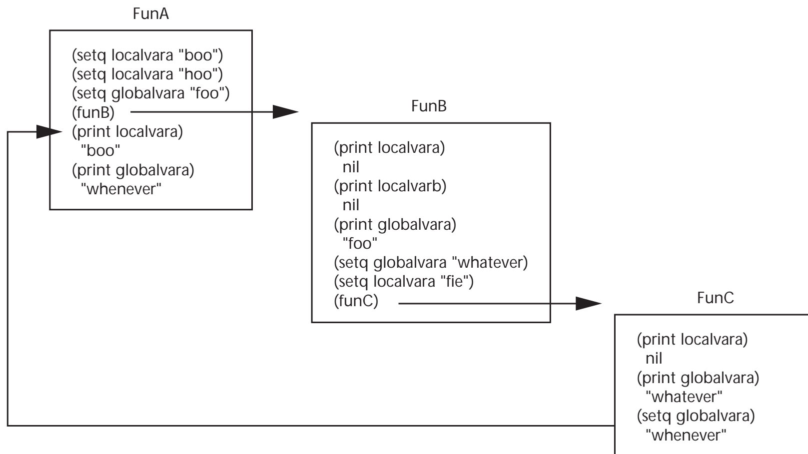
- 局部变量和全局变量的区别
- 使用关联表来捆绑数据
- 检查程序变量
- 修改程序代码
- 给程序代码加注释
- 设置断点并进行更多监视
- 第二课回顾

本课讲授如何使用一些有用的 VLISP 调试工具，它们可加快 AutoLISP 程序开发的速度。您还将学习局部和全局变量的不同之处，以及在何时使用它们。同时，您的程序将更加活跃，可以提示用户输入某些信息，这些信息被保存在表中。您将开始了解表在 AutoLISP 程序中的作用。毕竟，LISP 是由于它是一个表处理 (LISt Processing) 语言才得名的。

局部变量和全局变量的区别

本课讨论如何使用局部变量和全局文档变量。在文档（AutoCAD 图形）中加载的所有函数都能访问的变量是全局变量，这些变量在定义它们的程序结束以后还会保持它们的值。有些时候，这正是您所想要的——在本教程的后面就有一个这样的例子。

而局部变量只在定义它们的函数运行时才会有它们的值。函数运行完毕后局部变量的值被自动放弃，同时系统还会收回变量使用的内存空间。这被称为“自动无用数据收集”，大多数 LISP 开发环境（如 VLISP 等）都会提供该功能。局部变量的内存利用效率要比全局变量高。



局部变量的另一个大优点是，它使得应用程序的调试和维护更为容易。使用全局变量，您无法确定在哪个函数中修改了这个变量的值，而使用局部变量的话，跟踪就比较容易。一般来说，这样做造成的相互影响（即程序的某部分影响程序其他部分中的变量）也比较小。

正是由于上面提到的优点，本教程基本上只使用局部变量。

注意 如果您以前用过 AutoLISP，那您可能有过在开发中使用全局变量，以便在构建程序时检查该程序的经验。现在已经不需要采用这种方法了，因为 VLISP 提供了强大的调试工具。

在程序中使用局部变量

回头看一下您在第一课中创建的函数 **gp:getPointInput**：

```
(defun gp:getPointInput()

  (alert
    "Function gp:getPointInput will get user drawing input"
  )
  ;; 现在返回 T，就好象该函数正确运行一样
  T
)
```

迄今为止，该函数并没有做什么工作。您现在将开始完成该函数，即往函数中添加相应函数调用，以获取用户输入，来定义小路的起点、端点和宽度。

在创建 AutoLISP 程序时的一种好习惯是仿效 AutoCAD 的做法。所以，最好不要让用户以中心线端点为参照拾取一点来指定小路宽度，而应该让他输入半宽。

当 **gp:getPointInput** 函数完成运行时，其中的变量以及赋给变量的值都将不再存在。因此，您可以将用户提供的值保存在局部变量中。下面是函数的实现代码：

```
(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
      )
    )
  )
)
```

在函数开始的 **defun** 语句里，斜线后面是局部变量的声明。函数中第一次 **getpoint** 调用提示用户指定起点，然后要求用户依据输入的起点指定端点。在输入端点时，用户会看到一条从起点延伸出来的拖引线。类似地，在设置小路半宽值时，用户也会看到一条由端点伸出的拖引线，但这次它是表示距离。

查看 **gp:getPointInput** 如何工作的步骤

1 在 VLISP 控制台窗口输入 **gp:getPointInput** 的代码。
2 将控制台窗口光标放在代码块的最后一个括号后（或它下面的那行），然后按 ENTER 键，这样您就可以替换您以前加载的任何版本的 **gp:getPointInput**。
3 在控制台提示处输入 **(gp:getPointInput)**，执行该函数。
4 依次输入点以响应提示，然后输入小路半宽值。

检测函数 gp:getPointInput

当您运行 **gp:getPointInput** 函数时，VLISP 会自动把控制传给 AutoCAD。您响应三个提示后，AutoCAD 会自动把控制传回给 VLISP，最后函数会在控制台窗口显示符号 T。

程序流程：

1 VLISP 等候用户输入第一个点。
2 当用户拾取第一个点时，程序将所输入的点（包括 X、Y 和 Z 三个坐标值的表）保存在变量 StartPt 中。
3 第一个 **if** 函数检查结果以判断是输入了有效值还是没输入值。用户拾取起点后，控制传给下一个 **getpoint** 函数。
4 当用户拾取端点时，该点的值将被保存在变量 Endpt 中。
5 该语句的结果由第二个 **if** 语句检查，然后控制会传给 **getdist** 函数。
6 **getdist** 函数的行为类似，用户需要在屏幕上拾取一个点或输入一个数值。函数 **getdist** 的结果保存在变量 HalfWidth 中。
7 最后程序流将到达被层层嵌套在函数中的 T 语句。该语句后没有其他函数，所以，整个函数结束并返回值 T。这也就是您在控制台窗口看到的 T。

您需要采用一些方法来从一个函数返回值到另一个函数。其中一种方法是创建一个 **gp:getPointInput** 返回值的表，如下述代码中的突出部分所示：

```
(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)
          )
   )
 )
)
```

将该版本的 **gp:getPointInput** 函数复制到控制台窗口并按 ENTER 键，这时您有机会尝试控制台窗口的另一个功能。

利用控制台窗口的历史功能运行 gp:getPointInput 的步骤

1 按 TAB 键，这将调用控制台的历史记录功能，可以在以前输入控制台的所有命令之间循环切换。按 SHIFT + TAB 可按相反方向循环。

2 当您在控制台提示处看到 **(gp:getPointInput)** 时，按 ENTER 键可再次执行该函数。

3 和以前一样响应提示。

该函数返回的表包括两个嵌套表和一个实数（浮点数），返回的表和下表类似：

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

这些值分别代表变量 StartPt、EndPt 和 HalfWidth。

使用关联表来捆绑数据

虽然前面的例子可以完成所需工作，但您可以用更好的方法。在下一个练习中，您将创建一个关联表。在关联表中，您所感兴趣的值是和组码关联起来的。下面就是一个关联表的例子：

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

在该样例中，组码是数字 10、11 和 40。这些组码在表中的作用是作为具有唯一性的索引使用。在从程序中访问 AutoCAD 图元时，AutoCAD 就用这种机制将图元数据返回给 AutoLISP。一般组码 10 代表起点，而组码 11 则代表端点。

那么使用关联表的好处是什么呢？和普通表相比，关联表中数据值的顺序是没有明确含义的。再看一下去掉了组码的第一个表：

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

该值中无法明显看出哪个子表是起点，哪个是端点。另外，如果您将来修改了该函数，那些依赖于该函数返回数据顺序的函数就可能受到影响，可能不能正确运行。

而使用关联表的话，返回值的顺序就没有关系了。如果关联表的顺序改变了，您仍能辨别每个值所定义的量。例如，值 11 所关联的仍是端点，不论它在整个表中所处的位置如何：

```
((11 7.66688 4.62318 0.0)     ; 表的顺序
 (40 . 1.018248)              ; 已经
 (10 4.46207 4.62318 0.0))    ; 改变
```

使用关联表

如果使用关联表，您最好在注释中记录组码的含义。对“花园小路”，组码 10、11、40、41 和 50 的含义如下：

- 10 表示小路起点的三维坐标。
- 11 表示小路端点的三维坐标。
- 40 表示小路的宽度（不是半宽）。
- 41 表示小路从起点到端点的长度。
- 50 表示小路的方向（或角度）。

下面是 **gp:getPointInput** 函数修改后的版本。在该版本中，调用了名为 **cons**（construct a list 的缩写）的 AutoLISP 函数来为关联表中的子表加上关键字。可将该版本的函数定义复制到控制台窗口，按 ENTER 键，然后输入 **(gp:getPointInput)** 再次运行该函数：

```
(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: "))
         ;; 按前述文档创建关联表，
         ;; 该关联表将成为函数的返回值。
```

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

注意在创建表时，程序将用户指定的半宽乘以 2，转换成所需的小路全宽。

控制台窗口显示的输出和下面所列类似：

```
_$ (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))
_$
```

将 **gp:getPointInput** 的返回值保存到变量中

现在再做其他一些尝试。再次调用该函数，但这次将返回值保存到名为 gp_PathData 的变量中。在控制台窗口提示处输入如下命令可完成该工作：

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

如果想查看您刚设置的变量的值，可在控制台窗口提示处输入其名称：

```
_$ gp_PathData
```

VLISP 返回的数据和下面类似：

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

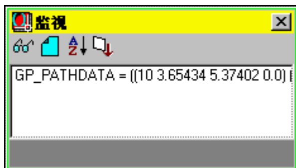
检查程序变量

VLISP 为您提供了用于编写和调试程序的全套工具，监视工具就是其中最有价值的工具之一。它可让您查看变量的信息，远比在 VLISP 控制台窗口显示的信息更详细。您还可以在函数执行时监视函数中的局部变量。

监视变量值的步骤

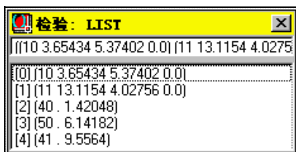
1 从 VLISP 菜单中选择“调试”➢“添加监视”，VLISP 将显示一个标题为“添加监视”的对话框。

2 输入您要监视的变量名。在这里，请输入您刚在控制台窗口设置的变量 gp_PathData。VLISP 会显示一个“监视”窗口：



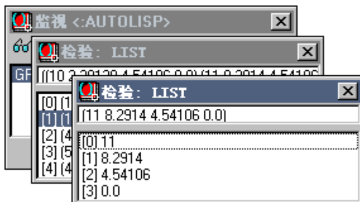
VLISP 会在“监视”窗口（上图显示的基本窗口）中将变量值显示在一行中。在本例中，变量值是一个较长的表，您无法看到它整个值。您可以拖动“监视”窗口的边界调整窗口的大小来查看该行，但还有更好的办法。

3 双击“监视”窗口中的变量名，这将打开一个“检验”窗口：



“检验”窗口指出您所检验变量的数据类型（在本例中是表），以及该变量的值。如果变量是表，“检验”每行显示表的一个条目。

4 双击关联表中关键字为 11 的行，VLISP 将显示另一个“检验”窗口：



5 您检验完变量后，可关闭所有“检验”窗口，但请保持“监视”窗口为打开。

修改程序代码

既然您已看到了如何在 AutoLISP 代码中使用关联表，现在可以用该方法编写完整的 **gp:getPointInput** 函数。可用下面的代码，替换或修改您以前保存在文件 gpmain.lsp 中的函数 **gp:getPointInput**。

注意 如果要将代码键入文件 gpmain.lsp，而不是将它从其他文件中复制过来，您可以不键入注释（所有以分号 (;) 开头的行）以节约时间。但写程序时千万不要也不加注释！

```
;;;--------------------------------------------------------------;
;;;    函数：gp:getPointInput                                    ;
;;;--------------------------------------------------------------;
;;;  说明：该函数让用户在图形中选取三个点，                      ;
;;;        以确定小路的位置、方向和尺寸。                        ;
;;;--------------------------------------------------------------;
;;;  如果用户输入有效值响应 get 函数，                           ;
;;;  用 startPt 和 endPt 来确定所画小路的                        ;
;;;  位置、长度和角度。                                          ;
;;;--------------------------------------------------------------;
;;;  该函数的返回值是包括如下内容的表：                          ;
;;;  (10 . 起点 )    ;; 含三个实数的表，用来指定                 ;
;;;                  ;;  花园小路的起点                          ;
;;;  (11 . 端点 )    ;; 含三个实数的表，用来指定                 ;
;;;                  ;;  花园小路的端点                          ;
;;;  (40 . 宽度 )    ;; 实数，用来指定小路边界宽度               ;
;;;  (41 . 长度 )    ;; 实数，用来指定小路边界长度               ;
;;;  (50 . 小路方向 ) ;; 实数，以弧度为单位                      ;
;;;                  ;;  指定小路方向                            ;
;;;--------------------------------------------------------------;
(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: "))
        ;; 按前述文档创建关联表，
        ;; 该关联表将成为函数的返回值。
        (list
          (cons 10 StartPt)
          (cons 11 EndPt)
          (cons 40 (* HalfWidth 2.0))
          (cons 50 (angle StartPt EndPt))
          (cons 41 (distance StartPt EndPt))
) ) ) ) )
```

然后，您必须更新主函数：*gpmain.lsp* 中的 **C:GPath**，请将它修改成和下述代码一样：

```
(defun C:GPath (/ gp_PathData)
  ;; 要求用户输入：首先输入路径位置和方向，
  ;; 然后输入路径参数，只有在输入了有效参数之后
  ;; 才继续工作。将数据保存在 gp_PathData 变量中。
  (if (setq gp_PathData (gp:getPointInput))
    (if    (gp:getDialogInput)
      (progn
            ;; 在此处已获取用户的有效输入。
            ;; 画出轮廓，将结果多段线指针存入名为
            ;; PolylineName 的变量。
            (setq PolylineName (gp:drawOutline))
            (princ "\nThe gp:drawOutline function returned <")
            (princ PolylineName)
            (princ ">")
            (Alert "Congratulations - your program is complete!")
        ) ;_ 结束 progn
       (princ "\nFunction cancelled.")
    ) ;_ 结束 if
    (princ "\nIncomplete information to draw a boundary.")
  ) ;_ 结束 if
 (princ)                              ; 静默退出
);_ 结束 defun
```

如果您是复制和粘贴代码，请将下述注释加在 C:GPath 前面：

```
;;;**************************************************************
;;;              函数 : C:GPath       花园小路的主函数
;;;--------------------------------------------------------------
;;;  说明 : 这是花园小路的主函数。它是 C: 函数，
;;;           说明它已转换成名为 GPATH 的 AutoCAD 命令。
;;;           该函数决定花园小路应用程序的整个流程。
;;;**************************************************************
;;; 变量 gp_PathData 是如下格式的关联表：
;;;  (10 . 起点 )    ;; 含三个实数的表，用来指定
;;;                    花园小路的起点
;;;  (11 . 端点 )    ;; 含三个实数的表，用来指定
;;;                    花园小路的端点 ;
;;;  (40 . 宽度 )    ;; 实数，用来指定小路边界宽度
;;;  (41 . 长度 )    ;; 实数，用来指定小路边界长度
;;;  (50 . 小路方向 ) ;; 实数，以弧度为单位指定
;;;                    花园小路的方向
;;;  (42 . 砖尺寸 )   - 实数，用来指定花园小路
;;;                    砖的尺寸（单位为弧度）
;;;  (43 . 砖的间距 ) - 砖的间距，边界到边界的距离
;;;  ( 3 . 对象的创建方法 )
;;;                  - 对象的创建方法是指绘制砖的方法，
;;;                    其值是一个字符串，
;;;                    它应是下述三值之一（字符串中
;;;                    字符的大小写无关紧要）：
;;;                       "ActiveX"
;;;                       "Entmake"
;;;                       "Command"
;;;  ( 4 . 多段线边界样式 )
;;;                  - 多段线边界样式确定小路边界所用
;;;                    多段线的类型，
;;;                    其值应是下述字符串之一（字符串中
;;;                    字符的大小写无关紧要）：
;;;                       "Pline"
;;;                       "Light"
;;;**************************************************************
```

测试修改后代码的步骤

1 保存更新后的文件。

2 用检查功能查找语法错误。

3 设置代码格式，增强其可读性。

4 加载代码，使 VLISP 重新定义该函数，覆盖原有版本。

5 在控制台提示处输入 **(c:gpath)**，运行该程序。

如果程序运行不成功，可试着改正错误再运行它。如果多次修改后还不能正确运行，您可以从 Tutorial\VisualLISP\Lesson2 目录下复制正确的代码。

给程序代码加注释

VLISP 将任何以分号开头的 AutoLISP 语句看作注释。上面的两个代码样例中就包含了许多注释。AutoLISP 程序中的注释是写给自己而不是写给程序的。给代码加上注释是编程最应该养成的好习惯。给程序加注释有如下好处：

- 如果您很长时间以后再编辑程序给程序加上用户要求的新功能时，注释可为您解释代码的含义。随着时间的推移，记忆是会消褪的，那些最明显的函数序列都会变得难以理解。
- 如果有其他人要接替您，负责更新和维护程序，这些注释可向他们解释代码的含义。阅读其他人的程序是很费劲的事，尤其在代码中注释很少时更加困难。

VLISP 中包括了一些工具，可以在您为代码加入注释时给您以帮助。注意例子中的注释有时开头有三个分号 (;;;)，有时是两个 (;;)，而有时则只有一个 (;)。如果想知道 VLISP 如何对待这些不同的注释，请参见 Visual LISP 开发人员手册中的“应用 Visual LISP 注释样式”。

为了节省空间，本教程后面的代码样例不再包括样例源文件中的所有注释。我们假设您已建立了给代码添加注释的好习惯，并假设您不需要提示就可以做到这一点。

设置断点并进行更多监视

断点是您放到源代码中的一种符号，它用来指定您想要程序暂停运行的位置。当您运行代码时，VLISP 会正常地执行代码，直到它碰到断点。在断点处，VLISP 会挂起程序的运行，等待您的指令来决定下一步的工作。此时 VLISP 并没有终止程序，它只是让程序处于挂起状态而已。

当程序挂起时，您可以

- 单步执行代码，一个函数接一个函数，或一个表达式接一个表达式。
- 在任意处使程序恢复正常运行。
- 动态修改变量的值，还可以修改正在运行的程序的结果。
- 将变量添加到“监视”窗口中。

使用“调试”工具栏

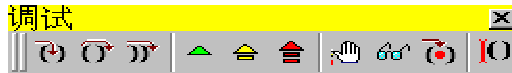
“调试”工具栏中包含了几个工具，您在学习本节时会用到它们。缺省情况下，该工具栏附着在“视图”和“工具”工具栏上，看起来就象单个 VLISP 工具栏：

“调试”工具栏控制条

“调试”工具栏包括最左边的那几个图标。该工具栏上的大多数条目在平时都是禁用的，只有在调试模式（例如定义了一个或几个断点）下运行程序时它们才会变为可用状态。

如果您尚未将“调试”工具栏从屏幕顶部拖下来，那您现在可以试试。要拖动它，只需用鼠标点中工具栏最左边的两个竖条并拖动即可。用这种拖动方法可以将 VLISP 的任何工具栏拖到屏幕的任意位置，以满足工作的需要。

“调试”工具栏上的按钮分为三组，每组包括三个按钮。在调试模式下运行程序时，该工具栏如下图所示：



- 前面三个按钮让您单步执行程序代码。
- 接下来的三个按钮用来确定在 VLISP 在断点处或产生错误时停下来时，下一步该如何继续。
- 接下来的三个按钮用来设置或删除断点、添加监视以及跳转到您程序代码中的最近一次运行暂停处。

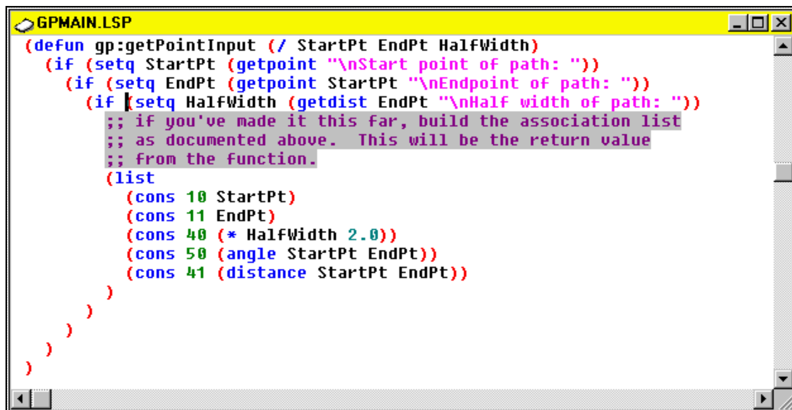
“监视”工具栏上的最后一个按钮是一个单步调试指示器，它不执行任何操作，但提供了一个视觉化指示器，用于在单步执行代码时指示光标的位置。不是运行在调试模式下时，该按钮显示为空白。

设置断点的步骤

1 在包括 *gpmain.lsp* 的 VLISP 编辑窗口中，将光标放到下面一行代码（它是函数 **gp:getPointInput** 中的一行代码）的 setq 函数的开括号正前面：

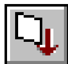
(setq HalfWidth (getdist EndPt "\nHalf width of path: "))

2 单击鼠标。该位置如下图所示：



3 当文本插入点处于该位置时，单击“调试”工具栏上的“切换断点”按钮。

“切换断点”按钮的作用就象一个开关，在开和关两个状态之间互相切换。如果光标处没有断点，它设置一个断点，而如果这里已经设置了断点，则它删除该断点。



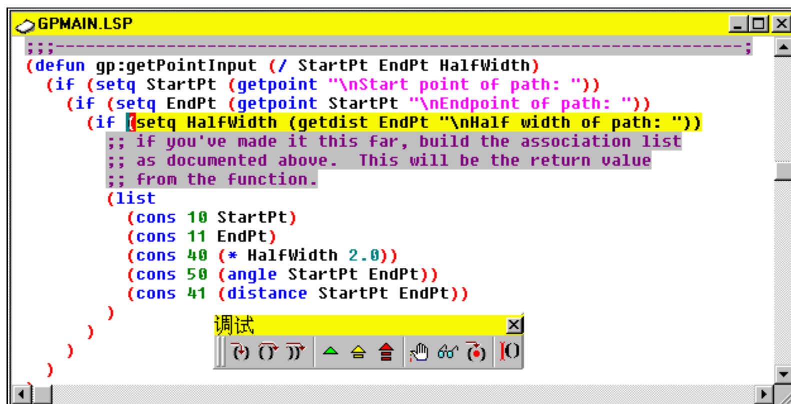
4 单击“工具”工具栏上的“加载活动编辑窗口”按钮以加载程序。

5 从 VLISP 控制台提示处运行 **(C:GPath)** 函数。

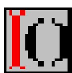
VLISP 在碰到断点以前将正常执行程序。在本例中，它将提示您输入前两个点：小路的起点和端点。

6 当程序提示时依次指定起点和端点。

在您指定了这些点后，VLISP 会挂起程序的执行，并将焦点返回到文本编辑窗口，亮显断点处那行代码：



请注意：



■ 光标正好位于断点处。这点可能很难注意到，所以 VLISP 提供了另一个线索。
■ 在“调试”工具栏上，单步调试指示器图标显示为一个红色的 I 光标在一对括号的前面，这表示 VLISP 调试器是暂停在表达式之前。

单步执行代码

有三个按钮可用来单步执行代码，它们是“调试”工具栏上最左边的三个按钮。它们的功能依次说明如下：

■ 步入亮显表达式。
■ 步出亮显表达式，执行到表达式最后。
■ 步出当前所暂停的函数，执行到该函数最后。

在您作出选择之前，请看一下亮显代码、光标的位置和单步调试指示器按钮。概括地说，亮显的表达式包括一个 **getdist** 函数（它嵌套在 **setq** 函数里），光标位置处在亮显代码块的前面。

从断点处单步执行代码的步骤

1 单击“下一表达式”按钮。

在您单击按钮之后，控制被传给 AutoCAD，它提示您指定小路宽度。

2 响应提示。

在指定了宽度之后，控制又被传回给 VLISP。请注意您光标的位置和单步调试指示器按钮显示的状态：

GPMAIN.LSP

```
(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))
        )
      )
    )
  )
)
```

调试

VLISP 对整个亮显表达式进行了求值，然后暂停在表达式最后。

3 再次单击“下一表达式”按钮，VLISP 跳到下一块代码的前面，并亮显整个代码块：

GPMAIN.LSP

```
(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))
        )
      )
    )
  )
)
```

调试

4 单击 “下一嵌套表达式”按钮（而不是 “下一表达式”按钮）。

注意 在这个练习中，如果您选错了按钮，跳过了某步，您可以很容易地重新开始。首先，单击“调试”工具栏上的“重置”按钮，这将停止任何 VLISP 代码的运行，并将 VLISP 系统重新设置到首层状态。然后，您就可以从第一步开始执行了。

现在第一个 **cons** 函数被亮显，VLISP 正好暂停在该函数前（请注意单步调试指示器按钮）：

```
GPMAIN.LSP
(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))
        )
      )
    )
  )
)
调试
```

单步执行程序时监视变量

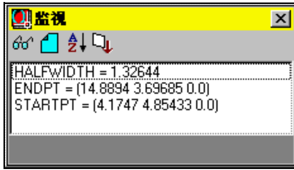
当您单步执行程序时，可以将变量添加到“监视”窗口中，并改变它们的值。

如果您没有看到“监视”窗口，只需单击工具栏上的“监视窗口”按钮就可显示该窗口。

如果“监视”窗口中仍包括变量 gp_PathData，请单击“监视”窗口顶部的“清除窗口”按钮将该变量从“监视”窗口中清除。

将变量添加到“监视”窗口中的步骤

1. 双击 VLISP 文本编辑窗口中任意一处 StartPt，这是您想跟踪其值的变量名。
2. 单击“监视”窗口中的“添加监视”按钮，或单击鼠标右键并选择“添加监视”。
3. 对变量 EndPt 和 HalfWidth 重复上述操作，您的“监视”窗口看起来应和下图类似：



如果调试的程序运行不正确，可结合使用断点和监视工具，来确定变量的值是否和预想的一致。

如果变量的值和预想的不一致，您可以改变它的值以观察它对程序的影响。例如，假设您预想 halfwidth 的值是个整数，但您在输入时没有注意拾取点，使得输入了类似于 1.94818 的一个值。

在程序运行时改变变量的值的步骤

1 在控制台提示处输入如下命令：

```
(setq halfwidth 2.0)
```

注意“监视”窗口中的值也跟着改变了。但这可能还不能使您确认关联表中创建的子表 (40 . width) 用了这个新值，可以再向“监视”窗口中添加一个表达式来验证这点。

2 从 VLISP 菜单上选择“调试”➢“监视最新结果”。

这将一个名为 *Last-Value* 的变量添加到“监视”窗口中。*Last-Value* 是一个全局变量，VLISP 将最近求值的表达式的值自动保存在该变量中。

3 单步执行程序（单击“下一嵌套表达式”或“下一表达式”按钮），直到用于构建 width 子表的表达式被计算。执行该动作的代码是：

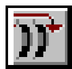
```
(cons 40 (* HalfWidth 2.0))
```

如果您已如前所述重新设置了 **HalfWidth** 的值，对该表达式的求值结果应该是“监视”窗口中所显示的 (40 . 4.0)。

步出函数 gp:getPointInput 并步入 C:Gpmain

还有一点需要演示的是：当您退出 **gp:getPointInput** 时，函数中局部变量的值将会有何变化。

退出 **gp:getPointInput** 函数，将控制返回到 **c:gpath** 的步骤

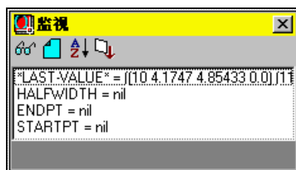


1 单击“跳出”按钮。

VLISP 将执行到函数 **gp:getPointInput** 的最后，并暂停在即将退出该函数处。

2 单击“下一嵌套表达式”按钮。

控制将返回到调用 **gp:getPointInput** 的函数 **c:gpmain**。



检查“监视”窗口中变量的值。由于 endpt 和 startpt 只是函数 **gp:getPointInput** 的局部变量，所以它们的值变为 nil，VLISP 会自动收回这些变量占用的内存。正常情况下，第三个函数局部变量 **HalfWidth** 的值也应该是 nil，但由于在调试时，曾在控制台窗口中全局地修改它的值，所以它在“监视”窗口中的值仍为 2.0。全局变量 *LastValue* 显示的值则是 **gp:getPointInput** 创建的关联表。

您的第一次调试任务已经结束了，但不要忘记您的程序仍处于挂起状态。

结束本课的步骤

1 单击“调试”工具栏上的“继续”按钮，响应程序的提示。这样程序将运行完毕。

2 从 VLISP 菜单上选择“调试” ➤ “清除所有断点”，VLISP 提示时选择“是”。这将删除代码中的所有断点。

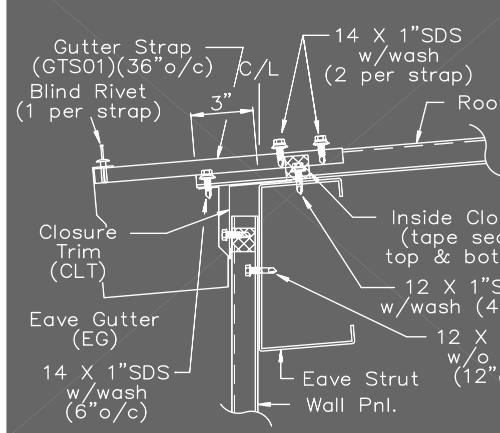
记住您可以将光标放在断点处，然后按“切换断点”按钮来删除单个断点。

第二课回顾

在本课中，您

- 学习了局部变量和全局变量。
- 设置和删除程序中的断点。
- 单步执行程序。
- 在程序执行时监视并动态修改程序中变量的值。
- 看到了在定义局部变量的函数执行完毕后，局部变量的值是如何被重置为 nil 的。

如果您打算用 Visual LISP 开发 AutoLISP 应用程序，那么在本课中学会的这些工具将成为您日常工作的一部分。



绘制小路边界

本课内容包括

3

- 计划可重用工具函数
- 绘制 AutoCAD 图元
- 编写绘制边界轮廓的函数
- 第三课回顾

在本课中，您将继续扩展程序，让它在 AutoCAD 中真正画一些东西——花园小路的多段线轮廓。为了画这条边界，您需要创建一些工具函数，它们并非只能用于该应用程序，而是通用函数，以后还会用上它们。您还要学习编写带有形式参数的函数。形式参数是函数的引用者传给函数的数据，使用形式参数是一个很重要的编程概念。在本课的最后还将绘制一个参数化的 AutoCAD 形，也就是说，根据用户提供的数据参数动态地绘制形。

计划可重用工具函数

工具函数可实现较通用的功能，因此许多应用程序都可以用上它们。这些函数可形成一个可供用户反复使用的工具集。

创建工具集中的函数时，应花些时间为它写一个完整的文档，在注释中也应该注上将来要加上的功能。

将度转换为弧度

现在要创建一个函数，以避免重复键入同一个常用的等式，该函数如下所示：

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

该函数名为 **Degrees->Radians**。其函数名就说明了函数的作用。

为什么需要函数来进行角度的单位转换呢？这是因为 AutoCAD 在底层实现时是用弧度作为角度的单位，而大多数人是用度作为角度的单位来考虑问题的。您工具集里的该函数就可让您用度作单位，而让 AutoLISP 将度转换为弧度。

测试工具函数的步骤

1 在 VLISP 控制台提示处输入如下代码：

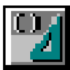
```
(defun Degrees->Radians (numberOfDegrees)
(* pi (/ numberOfDegrees 180.0)))
```

2 在 VLISP 控制台提示处输入如下命令：

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

该函数返回实数 3.14159。根据函数的功能，180 度等于弧度的 3.14159 。

如果要在程序中使用该函数，只需将函数定义从控制台窗口复制到 gpmain.lsp 文件中即可。您可将它粘贴到文件的任意位置，只要不将它粘贴到现有函数中间就行。



为了让代码的可读性更好，可以选取刚粘贴的文本，然后单击“设置选定代码的格式”按钮，VLISP 将适当缩进，设置代码格式。

然后，加上一些注释来说明该函数。编写了完整的函数文档以后，您的代码应和下述代码类似：

```
;;;-------------------------------------------------------------
;;;        函数：Degrees->Radians
;;;-------------------------------------------------------------
;;; 说明：本函数将一个以度为单位表示角度的数字
;;;           转换为以弧度为单位表示的角度。
;;;           本函数不检查参数 numberOfDegrees，
;;;           而始终认为它是一个有效数字。
;;;-------------------------------------------------------------
(defun Degrees->Radians (numberOfDegrees)
 (* pi (/ numberOfDegrees 180.0))
)
```

将三维点转换为二维点

花园小路程序中的另一个有用的工具函数可以将三维点转换为二维点。AutoCAD 通常使用三维坐标，但某些图元，如优化多段线等，总是使用二维坐标。由于 **getpoint** 函数返回的点都是三维点，所以您需要创建一个函数来转换它们。

将三维点转换为二维点的步骤

1 在控制台窗口提示处输入如下代码：

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

2 在控制台提示处输入如下命令来测试该函数：

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

这样就可以了，但对花园小路应用程序还有另一个问题要考虑。尽管对 LISP 函数来说，一个数是整数还是实数一般无关紧要，但对本课中您随后就会用到的 ActiveX 函数却不是这样——ActiveX 函数要求使用实数。您很容易就可修改该函数，以确保它返回的是实数而不是整数。

3 在控制台提示处输入如下代码：

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

4 再次运行该函数：

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

注意现在返回的值就是（用十进制表示的）实数了。

5 再测试一下该函数，这次用 **getpoint** 函数测试。在控制台提示处输入如下命令：

(setq myPoint(getpoint))

6 在 AutoCAD 图形窗口中拾取一点。

getpoint 函数返回一个三维点。

7 在控制台提示处发出如下命令：

(3dPoint->2Dpoint myPoint)

注意返回的是二维点。

现在把该函数加到 gpmain.lsp 文件中，添加的方法和添加 **Degrees->Radians** 函数一样。新代码应该和下述代码类似：

```
;;;--------------------------------------------------------------
;;; 函数：3dPoint->2dPoint
;;;--------------------------------------------------------------
;;; 说明：本函数有一个参数，表示一个三维点
;;;         （由三个整数或实数组成的表），函数将它
;;;         转换为二维点（由两个实数组成的表）。
;;;         本函数并不检查参数 3dpt，而是
;;;         总认为它是一个有效点。
;;;--------------------------------------------------------------
;;; 要添加的功能：加上一些参数检查功能，
;;;      这样即使传给它空值或不是三维点的值，
;;;      函数也不会导致程序崩溃。
;;;--------------------------------------------------------------
(defun 3dPoint->2dPoint (3dpt)
  (list (float(car 3dpt)) (float(cadr 3dpt)))
)
```

注意该函数头上有一个注释，说明将来这个函数要加上的功能。如果您想得到更好的声誉，就应该多考虑怎样使该函数变得更健壮，以确保无效的数据也不会导致它崩溃。

提示：numberp 和 listp 函数 ...

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

绘制 **AutoCAD** 图元

大多数 AutoLISP 程序用下述方法之一绘制图元：

- ActiveX 函数
- **entmake** 函数
- **command** 函数

本课主要讲如何用 ActiveX 创建图元。在第五课您将学习如何用 **entmake** 和 AutoCAD **command** 命令创建图元。

用 **ActiveX** 函数创建图元

创建图元的最新方法是在 VLISP 中使用 ActiveX 函数。和 **entmake** 与 **command** 相比，用 ActiveX 函数有如下几个优点：

- ActiveX 函数生成图元的速度更快。
- ActiveX 的函数名本身就说明它们所完成的操作，所以写出的代码可读性更好，更易于维护和修改错误。

在本课的后面部分您将会看到一个 Active 函数的例子。

使用 **entmake** 函数创建图元

使用 **entmake** 函数创建图元的步骤是：先将图元有关数据（如坐标位置、方向、图层、颜色等）收集起来组成一个关联表，然后请求 AutoCAD 创建图元。为 **entmake** 函数创建的关联表和调用 **entget** 函数所获取的关联表非常相象，不同之处在于 **entget** 返回某图元的信息，而 **entmake** 则是利用关联表的数据创建新图元。

使用 **AutoCAD** 命令行

AutoLISP 一开始在 AutoCAD 中出现时，它创建图元的唯一方法就是用 **command** 函数。该函数可使 AutoLISP 程序员编写的代码包括任何可在 AutoCAD 命令提示处执行的命令。这种方法也是可靠的，但它的速度没有 ActiveX 方法那么快，适应性也不如 **entmake** 那么强。

编写绘制边界轮廓的函数

上一课结束后，编写的 **gp:drawOutline**

```
;;;--------------------------------------------------------------
;;;     函数：gp:drawOutline
;;;--------------------------------------------------------------
;;; 说明：本函数绘制花园小路的轮廓。
;;;--------------------------------------------------------------
(defun gp:drawOutline ()
  (alert
    (strcat "This function will draw the outline of the polyline "
      "\nand return a polyline entity name/pointer."
    )
  )
  ;; 现在仅返回某引用符号，
  ;; 最后该函数应返回某图元名或指针
  'SomeEname
)
```

显然，上述代码并不能完成什么实际工作。然而，利用变量 gp_PathData 中存储的关联表信息，您完全可以算出小路的边界点。现在的问题是如何将该变量中的信息传给 **gp:drawOutline**。

请记住 gp_PathData 是 **C:GPath** 函数中定义的局部变量。在 AutoLISP 中，在某函数中定义的局部变量在该函数调用的所有函数中都是可见的（详细信息请参见第 14 页的“局部变量和全局变量的区别”）。**gp:drawOutline** 是在 **C:GPath** 中调用的。您可以在 **gp:drawOutline** 中引用变量 gp_PathData，但这不是一种好的编程方法。

为什么呢？如果使用同一变量的两个函数定义在同一个文件中（如前例所示），找到在哪里定义该变量、哪里使用该变量并不太困难，但如果这些函数在不同的文件中定义，那么，要确定 gp_PathData 代表的含义则必须搜索多个文件。

将参数传给函数

将信息从一个函数传给另一个函数，更好的办法是将参数传给被调用的函数。可以在设计函数时就确定要传给它哪些参数。还记得 **Degrees->Radians** 函数吗？该函数就有一个名为 numberOfDegrees 的参数：

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

在调用该函数时，必须给它提供一个数。函数 **Degrees->Radians** 将这个数声明成名为 numberOfDegrees 的参数。例如：

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

在该例中，数 90 被赋给参数 numberOfDegrees。

您也可将变量传给函数作为参数。例如，假设有一个名为 aDegreeValue 的参数，其值为 90，以下命令将设置 aDegreeValue，并将该变量传给 **degrees->radians**：

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

使用关联表

通过使用如下方式调用函数 gp:drawOutline 可以将变量 gp_PathData 中的关联表传给函数 **gp:drawOutline**：

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

这虽然很简单，但您还需要解决存储在该关联表中的信息的问题。VLISP 的“检验”功能可以帮您确定该怎么做。

使用 VLISP 的“检验”功能分析关联表的步骤

1 加载文本编辑窗口中的代码。

2 在控制台提示处输入如下表达式：

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

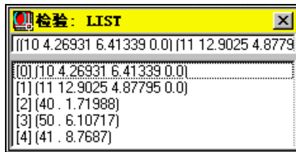
VLISP 将把您提供的信息保存到名为 BoundaryData 的变量中。

3 响应提示，输入起点、端点和半宽。

4 在控制台窗口中双击变量名 BoundaryData 以选择它。

5 从 VLISP 菜单中选择“视图”➢“检验”。

VLISP 显示如下窗口：



“检验”窗口将显示变量 BoundaryData 中的所有子表。

6 在 VLISP 控制台提示处输入如下命令：

```
(assoc 50 BoundaryData)
```

assoc 函数返回关联表中和指定组码相关联的条目。在本例中，指定的组码是 50，它和花园小路的角度相关联（关于该应用程序中组码和值的对应列表，请参见第 18 页的“使用关联表”）。

7 在 VLISP 控制台提示处输入如下命令：

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

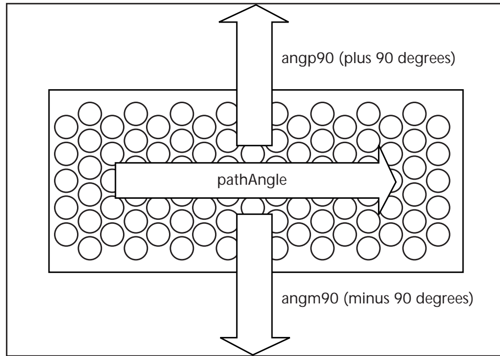
cdr 函数返回表中除掉第一个元素之外的所有其他元素。在本例中，**cdr** 返回角度值，因为它是 **assoc** 函数所返回条目的第二个元素，同时也是最后一个元素（该条目中只有两个元素）。

有了上述说明，您应该很容易理解下述代码：

```
(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))
```

使用角度和设置点

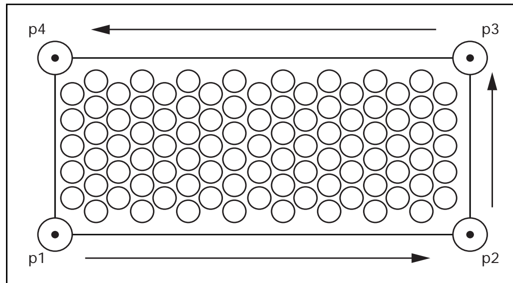
至此，还有一些工作有待完成。首先，您需要考虑如何按照用户指定的任何角度绘制小路。可以通过 **gp:getPointInput** 函数很容易算出小路的主方向。为了绘制小路，还需要算出和小路主方向垂直的另外一些向量。



这里我们就需要使用函数 **Degrees->Radians** 了。下述代码调用函数 **Degrees->Radians** 计算出了两个和主方向垂直的向量。在调用函数 **Degrees->Radians** 时，使用了变量 PathAngle 作为函数的参数：

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

有了这些数据，就可以调用 **polar** 函数计算出小路的四个角点的坐标了：




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

polar 函数返回距指定点指定角度和指定距离的三维点。例如，**polar** 将 p2 放置在使 p1 和 p2 的距离为 PathLength，且 p1 到 p2 的方向角（从 X 轴开始逆时针计算）为 PathAngle 的位置。

理解 gp:drawOutline 中的 ActiveX 代码

gp:drawOutline 函数可以调用 ActiveX 方法在 AutoCAD 中显示小路的多段线边界。以下代码使用 ActiveX 绘制边界：

```
;; 使用 ActiveX automation 将多段线添加到模型空间。
(setq pline   (vla-addLightweightPolyline
        *ModelSpace*  ; 模型空间的全局定义
        VLADataPts  ; 小路边界的角点
        ) ;_ 结束 vla-addLightweightPolyline

 ) ;_ 结束 setq

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

怎样理解这些代码呢？最重要的一份资料是 ActiveX and VBA Reference，它描述了 Active 客户端（如该花园小路应用程序等）可以访问的所有方法和特性。Visual LISP 开发人员手册中的“使用 ActiveX”一章解释了如何将 ActiveX and VBA Reference 中的 VBA™ 语法转换为 AutoLISP 中调用 ActiveX 的语法。

现在，您仍旧可以通过仔细观察上例中的两个带前缀 **vla-** 的函数，来初步了解 ActiveX 方法。AutoLISP 中所有操作 AutoCAD 对象的 ActiveX 函数都有一个前缀 **vla-**，例如，某 ActiveX 方法名为 addLightweightPolyline，那么调用该方法的 AutoLISP 函数名为 **vla-addLightweightPolyline**。调用 **vla-put-closed** 可更新 pline 对象的 closed 特性，而多段线的绘制则由函数 **vla-addLightweightPolyline** 完成。

AutoLISP ActiveX 对 Automation 对象的调用要遵守下述几个标准规则：

- **vla-put**、**vla-get** 或 **vla-** 方法的第一个参数是要修改或查询的对象。例如，第一个函数调用中的第一个参数是 *ModelSpace*，而第二个函数调用中的第一个参数是 pline。
- **vla-** 方法的返回值是 VLA 对象，可将它用于随后的函数调用中。例如，**vla-addLightweightPolyline** 返回的是对象 pline，它被用在下一个 ActiveX 调用中。
- ActiveX 对象模型具有层次化的结构。对象要从最顶层的应用程序对象一直遍历到最底层的单个图形图元（如多段线和圆等对象）。所以，我们并没有完成 **gp:drawOutline** 函数，因为必须先通过顶部的应用程序对象访问 *ModelSpace* Automation 对象。

确保加载 **ActiveX**

AutoCAD 或 Visual LISP 启动时并没有自动加载 ActiveX 功能，所以，如果要使用 ActiveX，就必须确保已加载了 ActiveX。下述函数可以完成该任务：

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

如果没有加载 ActiveX 支持程序，那么运行 **vl-load-com** 可以初始化 AutoLISP ActiveX 环境。如果已加载 ActiveX，**vl-load-com** 将不做任何工作。

获取指向模型空间的指针

通过 ActiveX 函数添加图元时，需要指定图元是插入在模型空间还是图纸空间（在 ActiveX 术语中，图元也是对象，但在本教程中为方便起见，我们继续使用术语图元）。为了告诉 AutoCAD 图元要加入到哪个空间，您需要先获取指向该空间的指针。不幸的是，获取一个指向模型空间的指针并不容易，无法通过调用单独一个函数来完成。下述代码显示了如何完成该操作：

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

从里到外来分析，**vlax-get-Acad-Object** 函数检索指向 AutoCAD 的指针，然后该指针被传给 **vla-get-ActiveDocument** 函数，它返回的是指向 AutoCAD 中活动图形（文档）的指针。接着该指针又被传给 **vla-get-ModelSpace** 函数，它返回指向当前图形模型空间的指针。

您可能不想多次重复键入这样的表达式。例如，请看下面的复杂样例。它用 ActiveX 来绘制多段线，其中就用了上述整个关于模型空间的表达式：

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

这让函数更难懂了。不仅如此，而且在程序中每一个创建图元的表达式中，都必须重复地输入同样的嵌套函数调用。这里就是应使用全局变量的一个极好的例子（虽然这种情况很少）——因为花园小路应用程序要将许多图元添加到模型空间中（想想小路中所有的砖就知道了），所以使用一个全局变量来保存指向模型空间的指针就很有用。下述代码可完成该工作：

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

这样，在调用 ActiveX 图元创建函数时就可以随时使用变量 *ModelSpace* 。唯一要注意的是必须在开始绘图以前先设置变量 *ModelSpace*。因为这个原因，可在加载该应用程序时，调用 **vl-load-com** 之后，马上调用 **setq** 语句来设置该变量。由于把该语句放在了程序文件中任何 **defun** 语句以前，所以在加载该文件时，马上就会执行该语句。

构造多段线端点数组

最后一个要解决的问题是如何将单个的点变量 p1、p2、p3 和 p4 转换成函数 **vla-addLightweightpolyline** 所要求的格式。首先看一看与该主题有关的帮助信息。

获取函数信息的步骤

1 在 VLISP 工具栏上单击 “帮助”按钮。

2 在“输入项目名称”对话框中输入 **vla-addLightweightpolyline**，然后单击“确定”按钮。（帮助系统不区分大小写，所以不用担心函数名的哪些字母应该大写。）

在线帮助指明函数 **AddLightWeightPolyline** 要求您以变体的方式将多段线端点指定为双精度实数数组。下面是帮助中对该参数的描述：

二维 WCS 坐标数组指定多段线的端点。创建优化多段线要求至少有两个点（四个元素），数组中实数的个数必须是 2 的倍数。

变体是一种 ActiveX 结构。它就象一个容器，可包含各种类型的数据。字符串、整数和数组都可以用变体来表示。变体在保存数据的同时，还保存了指示数据类型的信息。

现在，您已经有了四个点，每个点的格式为 (x, y, z)。要解决的问题是将这四个点转换为下列形式的表：

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

append 函数可以用多个表作为参数，并把它们连接起来形成一个表。用下述表达式，您可以将四个点转换成 ActiveX 函数所需要的表：

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

写四次 **3dPoint->2dPoint** 可能有点麻烦，您可以用 **mapcar** 和 **apply** 函数进一步简化减少代码。**mapcar** 针对一个或多个表中的单个元素执行某函数，而 **apply** 则将一列参数传给指定函数。这样修改后的代码如下所示：

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

在调用 **mapcar** 之前，四个点组成的表为如下格式：

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

调用 **mapcar** 之后，四个点组成的表格式如下：

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

最后将 **append** 函数应用到由 **mapcar** 返回的表上，您将得到下表：

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

由点表构造变体

现在，变量 polypoints 中的数据是以表的形式保存的，这适合于许多 AutoLISP 函数调用。然而，提供给 ActiveX 函数的输入参数应该是包含双精度实数数组的变体。您可以用另一个工具函数来完成由表到变体的转换：

```
(defun gp:list->variantArray (ptsList / arraySpace sArray)
 ; 给以双精度实数表示的二维点数组分配空间
 (setq arraySpace (vlax-make-safearray
                    vlax-vbdouble ; 元素类型
                    (cons 0
                          (- (length ptsList) 1)
                          ) ; 数组维数
                    )
       )
 (setq sArray (vlax-safearray-fill arraySpace ptsList))
 ; 返回数组变体
 (vlax-make-variant sArray)
 )
```

gp:list->variantArray 函数中执行了如下操作：

- 调用了 **vlax-make-safearray** 函数来为双精度数组 vlax-vbdouble 分配空间。**vlax-make-safearray** 函数还要求您指定数组下标的上下边界。在 **gp:list->variantArray** 中，调用 **vlax-make-safearray** 创建的数组的起始下标为 0，而下标上界为传给它的元素（ptsList）数目减 1。
- 调用 **vlax-safearray-fill** 函数将数组中的元素设置为点表中的相应元素。
- 调用 **vlax-make-variant** 函数将 safearray 转换成变体。在 **gp:list->variantArray** 完成最后一个函数调用后，将返回值传给调用它的函数。

下面是一个调用 **gp:list->variantArray** 函数的例子，它将表转换为包含双精度实数数组的变体：

```
; 将数据由表转换为变体
(setq VLADataPts (gp:list->variantArray polypoints))
```

代码集成

现在您有了绘制花园小路轮廓所需的所有代码。

更新代码的步骤

1 用下述代码代替函数 **gp:drawOutline** 的旧代码：

```
;;;--------------------------------------------------------------
;;;    函数：gp:drawOutline
;;;--------------------------------------------------------------
;;; 说明：本函数将绘制花园小路的轮廓。
;;;--------------------------------------------------------------
;;; 注意：函数没有对参数 BoundaryData 进行错误有效性检查。
;;; 该参数中条目的顺序无关紧要，但假设其中包含了所有子表，
;;; 并且假设子表中包含的数据是有效的。
;;; --------------------------------------------------------------
(defun gp:drawOutline (BoundaryData / VLADataPts PathAngle
              Width  HalfWidth  StartPt PathLength
              angm90  angp90     p1 p2
              p3  p4     polypoints pline
             )
```

```
 ;; 从表 BoundaryData 中提取数据
 (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))
           )
 )
 ;; ***** 数据转换 *****
 ;; 注意，polypionts 是 AutoLISP 格式，
 ;; 包含花园小路四个角点组成的表。
 ;; 必须将该变量转换为 ActiveX 函数能
 ;; 接受的输入参数形式。
 (setq VLADataPts (gp:list->variantArray polypoints))
 ;; 使用 ActiveX automation 将多段线添加到模型空间。
 (setq              pline (vla-addLightweightPolyline
        *ModelSpace*; 模型空间全局定义
        VLADataPts
         ) ;_ 结束 vla-addLightweightPolyline
 ) ;_ 结束 setq
 (vla-put-closed pline T)
 ;; 返回轮廓多段线的 ActiveX 对象名
 ;; 返回值的形式应如下所示：
 ;; #<VLA-OBJECT IAcadLWPolyline 02351a34>
 pline
) ;_ 结束 defun
```

注意现在 **gp:drawOutline** 返回的是变量 pline，而不是在简单空函数版本中使用的引用符号 'SomeEname。

2 选取您刚输入的代码，单击 VLISP 工具栏上的“设置选定代码的格式”按钮，设置这些代码的格式。

3 如前面所述那样加载 ActiveX 功能，并添加一个全局变量保存指向模型空间的指针。滚动文本编辑窗口到其顶部，在第一个 **defun** 前面加上如下代码：

```
;;;--------------------------------------------------------------
;;; 第一步是加载 ActiveX 功能。如果文档已加载了 ActiveX 支持
;;; 程序（当 AutoCAD 已加载 Bonus 工具时可能会出现这种情况），
;;; 则不做任何工作。否则加载 ActiveX 支持程序。
;;;--------------------------------------------------------------

(vl-load-com)

;;; 在第四课，如下注释和代码被移到文件 utils.lsp 中
;;;--------------------------------------------------------------
;;; 为了使用 ActiveX 函数，我们要定义一个全局变量，包括指向
;;; 活动文档的模型空间部分的指针。该变量名为 *ModelSpace*，
;;; 在加载时就会创建该变量
;;;--------------------------------------------------------------
(setq *ModelSpace*
       (vla-get-ModelSpace
          (vla-get-ActiveDocument (vlax-get-acad-object))
       ) ;_ 结束  vla-get-ModelSpace
) ;_ 结束 setq
```

请注意上述代码是在任何 **defun** 语句之外，正因为这个原因，VLISP 在您加载该文件时就会自动执行该代码。

4 在 **C:GPath** 函数中查找如下代码行：

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

Change it to the following:

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

gp:drawOutline 函数现在需要一个参数——包含多段线边界数据的表，而该改动满足了这个要求。

5 将第 45 页的“由点表构造变体”中所示的函数 **gp:list->variantArray** 添加到 gpmain.lsp 文件的最后。

现在可以加载和运行修改后的程序了。注意 VLISP 在您看到最后结果以前不会把控制交给 AutoCAD，所以在控制返回 VLISP 后要切换回 AutoCAD 窗口。如果程序运行正确，您应该可看到花园小路的边界。如果出现错误，请调试代码，然后再试。

第三课回顾

在本课中，您

- 编写了工具函数，可以在其他应用程序中重用这些工具函数。
- 给程序添加了创建图元部分。
- 学习了怎样使用 ActiveX 函数。
- 学习了怎样使用关联表。
- 使程序能绘制花园小路边界。

如果您对本课中的任何内容有点糊涂，建议您在学习第四课以前再看一遍本课。（如果您需要如此，请将您第二课目录中的全部代码复制过来，这样您才能正确地开始本课学习。）如果程序总是运行不正确，和以前类似，只需复制 Tutorial\VisualLISP\Lesson3 目录下的代码就可以了。

创建工程，添加界面

本课内容包括

4

- 模块化代码
- 使用 Visual LISP 工程
- 添加对话框界面
- 用 AutoLISP 代码与对话框交互
- 提供可选的边界线类型
- 收尾工作
- 运行应用程序
- 第四课回顾

在本课中，您将完成两个主要任务：创建一个 VLISP 工程和给应用程序添加一个对话框界面。在此过程中，您将把现在的单个 AutoLISP 文件 (gpmain.lsp) 拆分为几个小文件，以加强对代码模块性概念的认识。

从本课开始，教程对需要完成的任务给以更一般的描述，除非涉及新的主题。同时，教程中的代码也尽量压缩以节约空间。然而，这并不是说您在编写代码时也可以如此，您还是应该在代码中添加足够的注释文档。

模块化代码

完成第三课后，您的 gpmain.lsp 文件变得相当大了。VLISP 处理这种大文件并没有任何困难，但如果您将文件中定义函数按功能大致分类，然后将文件拆分成几个小文件，同类函数放在一个文件中，这样代码维护将变得更容易一些，而且调试也更方便。例如，如果您的某个文件中有 150 个函数，那么连一个括号不匹配的错误都将难以定位。

在本教程中，文件按下述方式组织：

教程文件组织方式	
文件名	内容
GP-IO.LSP	所有输入和输出 (I/O) 函数，如获取用户输入等。还包括即将添加的对话框界面的 AutoLISP 代码。
UTILS.LSP	包括所有工具函数，它们可用于其他工程。还包括加载时的初始化代码。
GPDRAW.LSP	所有的绘图代码，它们完成 AutoCAD 图元的实际创建工作。
GPMAIN.LSP	主程序 C:GPath 函数。

将 gpmain.lsp 分为四个文件的步骤

1 创建一个新文件，然后在 gpmain.lsp 文件中剪切下述函数，并将它们粘贴到新文件中：

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

将新文件保存在您的工作目录下，取名为 gp-io.lsp。

2 创建一个新文件，然后在 gpmain.lsp 文件中剪切下述函数，并将它们粘贴到新文件中：

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

同时，在文件的开始处，插入加载 ActiveX 功能的代码 (**vl-load-com**)，并设置全局变量 (*ModelSpace*)。

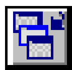
将文件保存为 utils.lsp。

3 创建一个新文件，然后在 gpmain.lsp 文件中剪切下述函数，并将它们粘贴到新建文件中：

- **gp:drawOutline**

将文件保存为 gpdraw.lsp。

4 将上述函数剔除出 gpmain.lsp 后，保存该文件并检查一下，现在该文件中应该只剩下原来的 **C:GPath** 函数。



现在您的 VLISP 桌面开始显得有点挤了。您可以最小化 VLISP 中的任何窗口，同时还可访问它：单击工具栏上的“选择窗口”按钮从列表中选择要显示的窗口，或从 VLISP 菜单上选择“窗口”，再选择要查看的窗口。

使用 Visual LISP 工程

VLISP 的工程功能为您管理组成应用程序的多个文件提供了一个方便的途径。使用工程功能，您不用打开应用程序中的所有文件，只要打开单个工程文件即可。打开工程文件后，如果要打开工程中的文件，只需双击它即可。

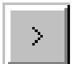
创建 VLISP 工程的步骤

1 从 VLISP 菜单上选择“工程” ➢ “新建工程”。

2 将文件保存在 Lesson4 目录下，命名为 gpath.prj。

保存文件之后，VLISP 将显示“工程特性”对话框。

3 单击“工程特性”对话框左边的“全部（不）选择”按钮。



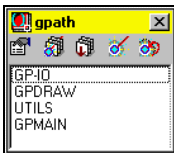
4 单击包含指向右边箭头的按钮，这将使所有选定的文件添加到您的工程中。

在“工程特性”对话框中，左边的列表框显示的是您工程文件所在目录中不属于该工程的所有 LISP 文件，右边的列表框显示的是组成该工程的所有文件。当您将所选文件添加到工程中时，这些文件名将从左边的列表框移到右边的列表框。

5 在对话框右边的列表框中选择 gpmain，然后单击“底部”按钮，则可将文件移到列表底部。

VLISP 加载工程文件的顺序和文件在列表中的顺序相同。因为显示命令名称提示的代码位于 gpmain.lsp 文件末尾，所以应将该文件移到列表底部。最后加载该文件将导致向用户显示提示。而 utils.lsp 文件应该首先加载，因为它包括了应用程序的初始化代码。所以，请在对话框中的列表框中选择 utils 并单击“顶部”按钮。

6 单击“确定”。



VLISP 在 VLISP 桌面上显示一个小的工程窗口，该窗口列出了工程中的所有文件。双击任意文件就可在 VLISP 文本编辑器中打开它（如果它尚未被打开的话），并使它成为活动的编辑窗口。

添加对话框界面

本课下一部分的主要内容是给花园小路应用程序添加对话框界面。为了完成该任务，您需要使用另一种语言－－对话框控制语言 (DCL)。

目前 **gpath** 函数仅能从命令行接受输入，但您包含了一个名为 **gp:getDialogInput** 的简单空函数，它的目的就是为了添加对话框界面。现在是添加该界面的时候了。

对话框界面的创建可以分为两个步骤：

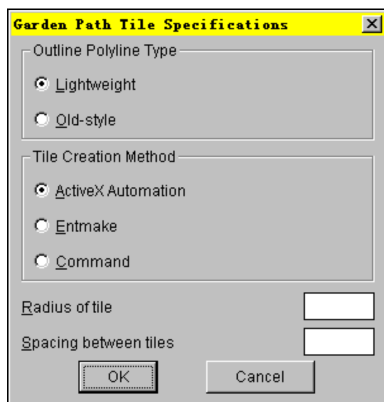
- 定义对话框的外观和内容。
- 添加程序代码，控制对话框的行为。

对话框的外观和所包含内容是用 .dcl 文件定义的。在 Visual LISP 开发人员手册中的第十一章“设计对话框”、第十二章“管理对话框”和第十三章“可编程对话框参考”都阐述了 DCL 文件。

还需要编写程序代码来初始化对话框的缺省设置，以及定义和用户交互的方法。这些代码将被添加到函数 **gp:getDialogInput** 中。

用 DCL 定义对话框

首先请看一下您要创建的对话框。



该对话框包括下述元素：

- 两个单选钮集。
  一个单选钮集确定小路边界的多段线类型，另一个则指定砖图元的创建方法（ActiveX、**entmake** 或 **command**）。任何时刻每个单选钮集中都只能有一个单选钮被选中。
- 文本框，用于指定砖的半径以及砖与砖之间的间距。
- 包括“OK”和“Cancel”按钮的标准按钮集。

在 DCL 中对话框部件被称为控件，写出一个对话框的完整 DCL 文件可能会比较难以理解，我们采用的方法是先规划出您想要设计的对话框，将其分为几小块，然后分别写出每一块。

定义对话框的步骤

1 在 VLISP 文本编辑窗口中打开一个新文件。

2 在新文件中输入如下语句：

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

该 DCL 语句定义了对话框窗口的标题。

3 添加下述代码，定义指定多段线类型的单选钮：

```
: boxed_radio_column {    // 定义单选钮区域
  label = "Outline Polyline Type";
  : radio_button {        // 定义优化多段线单选钮
    label = "&Lightweight";
    key = "gp_lw";
    value = "1";
   }
: radio_button {       // 定义普通多段线单选钮
  label = "&Old-style";
  key = "gp_hw";
 }
}
```

DCL 语句 boxed_radio_column 定义了一个带边框的矩形区域，而且您可以给按钮集指定标签。用 radio_button 语句在边框里添加单选钮。每个单选钮要求有一个标签和一个关键字，您的 AutoLISP 代码就是用关键字来引用按钮的。

注意标签为 "Highweigh" 的单选钮的值被设为 1。给按钮设置值 1（是字符串而不是整数）将使它成为按钮列中的缺省选择。换句话说，当您第一次显示该对话框时，将选定该按钮。同时请注意，在 DCL 文件中，注释以两个斜线开头，而不是使用 AutoLISP 中的分号。

4 添加如下代码，定义选择图元创建模式的单选钮集：

```
: boxed_radio_column {    // 定义单选钮区域
 label = "Tile Creation Method";
 : radio_button {        // 定义 ActiveX 单选钮
  label = "&ActiveX Automation";
  key = "gp_actx";
  value = "1";
  }
: radio_button {         // 定义 (entmake) 单选钮
 label = "&Entmake";
 key = "gp_emake";
 }
: radio_button {         // 定义 (command) 单选钮
 label = "&Command";
 key = "gp_cmd";
 }
}
```

5 添加如下代码，定义让用户输入砖尺寸和间距的文本框：

```
: edit_box {     // 定义用于输入砖半径的文本框
 label = "&Radius of tile";
 key = "gp_trad";
 edit_width = 6;
}
: edit_box {     // 定义用于输入砖间距的文本框
 label = "S&pacing between tiles";
 key = "gp_spac";
 edit_width = 6;
}
```

注意该定义并没有给文本框设置任何初值，给每个文本框设置缺省值的工作将由 AutoLISP 程序完成。

6 添加下述代码，为对话框添加 OK 和 Cancel 按钮：

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

两个按钮是定义在同一行内，所以它们水平地排成一行。

7 滚动文本编辑窗口，在 DCL 文件的最前面插入下述语句作为第一行：

```
gp_mainDialog : dialog {
```

8 该 dialog 语句需要一个和它匹配的闭大括号，所以请滚动到文件最后，将闭大括号加入文件作为 DCL 代码的最后一行：

```
}
```

保存 DCL 文件

在保存 DCL 文件之前，请注意在 AutoCAD 运行时必须能够找到您的 DCL 文件。所以，文件必须放到某个 AutoCAD 支持文件搜索路径中。如果您无法确定这些路径，请从 AutoCAD 菜单上选择“工具”➢“选项”，检查“文件”选项卡中的“支持文件搜索路径”。

现在可以把文件保存在 AutoCAD support 目录下了。

保存 DCL 文件的步骤

1 从 VLISP 菜单上选择“文件”➢“另存为”。

2 在“另存为”对话框的“存为类型”框中，从下拉列表中选择“DCL 源文件”。

3 将保存路径改成 <AutoCAD 目录 >*Support*。

4 输入文件名 gpdialog.dcl。

5 单击“保存”。

请注意在您保存文件之后，VLISP 改变了语法着色的配置。VLISP 可以识别 DCL 文件，并按代码在文件中所处的语法元素类别分别着色。

预览对话框

VLISP 提供了预览功能，可以检查 DCL 编码的结果。

预览用 DCL 定义的对话框的步骤

1 从 VLISP 菜单上选择“工具”➢“界面工具”➢“预览编辑器中的 DCL”。

2 在提示指定对话框名时单击“确定”。

本例中的 DCL 文件只定义了一个对话框，所以无需选择。如果您创建更大的应用程序，DCL 文件中可能包括多个对话框定义，这时，可以在这里选择预览哪个对话框。

3 如果成功显示了对话框，单击任意按钮可终止对话框的预览。

VLISP 将控制传给 AutoCAD 来显示对话框，如果 AutoCAD 发现了语法错误，它会显示一个或多个消息窗口来指出这些错误。

如果 AutoCAD 检测到 DCL 文件中的错误，而您又不知道如何更正这些错误，那也没关系，只需将 Tutorial\VisualLISP\Lesson4 目录下的 gpdialog.dcl 文件复制过来，保存到 Support 目录下即可。

用 AutoLISP 代码与对话框交互

现在需要编写 AutoLISP 函数来与对话框交互。前面编写的简单空函数 **gp:getDialogInput** 就是为执行该功能而预留的。该函数已从 gpmain.lsp 文件中移到 gp-io.lsp 文件中了。

开始开发对话框界面时，您可能会觉得有点糊涂。它需要您预先计划，回答如下问题：

- 该对话框是否需要在初始化时设置缺省值？
- 当用户单击按钮或输入值时要进行什么工作？
- 当用户单击“取消”按钮时要进行什么工作？
- 如果找不到对话框文件 (.dcl)，应作出什么反应？

设置对话框的值

运行完整的花园小路应用程序时，请注意对话框在一开始总是以 ActiveX 作为缺省的对象创建方法，而以优化多段线作为缺省多段线类型。更有意思的是砖的缺省尺寸还和路的宽度有关——宽度变了，砖的缺省尺寸也会跟着变。下面的代码段显示了如何设置开始时显示在对话框中的缺省值：

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

现在暂时不要管变量 dialogLoaded 和 dialogShow 的作用，在下两节中它们的作用就会变得很明显了。

加载对话框文件

您的程序首先需要调用 **load_dialog** 函数来加载 DCL 文件。如果您没有提供完整的路径，该函数将在 AutoCAD 支持文件搜索路径中查找对话框文件。

对代码中的每个 **load_dialog** 函数，后面都必须有一个 **unload_dialog** 函数和它相对应（您马上就会看到该函数）。现在先看一下加载对话框的方法：

```
;; 加载对话框。进行错误检查以确保
;; 在继续下一步以前先加载对话框文件
(if (= -1 (setq dcl_id (load_dialog "gpdialog.dcl")))
  (progn
    ;; 加载文件出现了问题，显示错误消息，并将
    ;; 标志 dialogLoaded 设为 nil
    (princ "\nCannot load gpdialog.dcl")
    (setq dialogLoaded nil)
  ) ;_ 结束 progn
) ;_ 结束 if
```

dialogLoaded 变量指示是否成功加载了对话框。在设置对话框的初始值时，dialogLoaded 的值设为 T。如果加载对话框文件出现错误，dialogLoaded 将被设为 nil，如上述代码段所示。

将指定对话框加载到内存中

前面已经提到，一个 DCL 文件中可能包含多个对话框定义。使用对话框的下一步是指出要显示哪个对话框定义，如下代码完成的就是这个工作：

```
(if (and dialogLoaded
         (not (new_dialog "gp_mainDialog" dcl_id))
    ) ;_ 结束 and
  (progn
    ;; 出现问题。
    (princ "\nCannot show dialog gp_mainDialog")
    (setq dialogShow nil)
  ) ;_ 结束 progn
) ;_ 结束 if
```

请注意上述代码中是怎样用 **and** 函数测试是否加载了对话框，以及是否成功调用了 **new_dialog** 函数。如果在 **and** 函数调用中要对多个表达式进行求值，在碰到第一个求值结果为 nil 的表达式后，将停止对后续表达式的求值。所以在本例中，如

果标志 dialogLoaded 为 nil（表示上一节中的函数调用加载 DCL 文件失败），VLISP 将不会去调用 **new_dialog** 函数。

还要注意的是上述代码还考虑到了使用 DCL 文件时可能会碰到错误，如果是这种情况，变量 dialogShow 将被设为 nil。

new_dialog 函数只是将对话框加载到内存中，而不会显示它。**start_dialog** 负责显示对话框。所有对话框的初始化工作，如设置控件值、创建图像或给列表框创建列表、给控件设置相关联的动作等，必须在调用 **new_dialog** 之后和调用 **start_dialog** 之前完成。

初始化缺省的对话框值

如果加载对话框的所有工作都已成功，就可以开始设置显示给用户的初始值了。加载成功的标志是标志变量 dialogLoaded 和 dialogShow 的值都是 T（真）。

现在来设置砖半径和间距的初始值。**set_tile** 函数将值赋给控件。因为文本框处理的是字符串而不是数字，所以需要用 **rtos** 函数（用于将实数转换为字符串）来将砖尺寸变量的值按十进制格式转换为字符串，转换时采用的精度为带两位小数。如下代码可完成该转换工作：

```
(if (and dialogLoaded dialogShow)
   (progn
     ;; 设置砖的初始值
     (set_tile "gp_trad" (rtos tileRad 2 2))
     (set_tile "gp_spac" (rtos tileSpace 2 2))
```

给控件指定动作

DCL 文件只能定义静态的对话框，必须在 AutoLISP 代码中调用 **action_tile** 函数给静态的对话框的控件定义相关联的动作，才能控制对话框。如下代码就给控件指定了相应动作：

```
;; 给对话框按钮指定动作（要调用的函数）
(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))"
  )
)
```

请注意 AutoLISP 代码中的所有被引号引起来的字符串。当您调用 **action_tile** 函数时，实质上是告诉控件："记住该字符串，当用户选取你时将它返回给我"。该字符串（双引号里的任何字符集合）在用户选取该控件之前保持静止，而在用户选取该控件后，控件会将该字符串传给 AutoCAD，AutoCAD 再将该字符串转换成 AutoLISP 代码并执行该代码。

例如，考察如下 **action_tile** 表达式，它在程序中是和优化多段线单选钮相关联的：

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

上述代码将字符串 "(setq plineStyle \"Light\")" 和单选钮相关联，当用户选择该按钮时，该字符串将传回 AutoCAD，然后直接转换为如下 AutoLISP 表达式：

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

再看一段代码。下面是赋给"OK"按钮的 **action_tile** 表达式：

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

用户单击“OK”按钮时，赋给该按钮的冗长字符串将被传给 AutoCAD，并转换为如下 AutoLISP 代码：

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

该代码完成的工作包括：首先获取关键字为 gp_trad（砖半径）和 gp_spac（砖间距）的控件的当前值，然后调用 **atof** 函数将数字字符串转换为实数。最后调用 **done_dialog** 函数结束该对话框，并将变量 UserClick 的值设为 T（真）。

您现在已给所有的按钮关联上相应动作了，下一步就是让对话框工作起来。

启动对话框

start_dialog 函数显示对话框并接受用户输入，它不需要参数。

```
(start_dialog)
```

调用 **start_dialog** 函数时，控制交给了用户。用户可以在对话框中作出选择，直到单击“OK”或“Cencel”按钮。

卸载对话框

用户单击“OK”或“Cancel”按钮时，需要卸载该对话框。和 **start_dialog** 一样，**unload_dialog** 也是一个简单函数。

```
(unload_dialog dcl_id)
```

确定下一步的动作

如果用户单击了“OK”按钮，则必须构造一个表，其中包含用户在对话框中输入的信息。**gp:getDialogInput** 函数返回给调用它的函数的值就是该表。如果用户单击的是“Cancel”按钮，函数将返回 nil：

```
(if UserClick        ; 用户单击“OK”
 ;; 构造结果表
 (progn
   (setq Result (list
       (cons 42 tileRad)
       (cons 43 TileSpace)
       (cons 3 objectCreateMethod)
       (cons 4 plineStyle)
       )
    )
  )
)
```

代码集成

有了上面的例子，再加上另外几行代码，您就可以完成 **gp:getDialogInput** 函数了。

gp:getDialogInput 代码集成的步骤

1 在 VLISP 文本编辑窗口中打开 gp-io.lsp 文件。

2 删除 **gp:getDialogInput** 函数中的代码（defun gp:getDialogInput 语句和其后的所有代码）。

3 输入如下 defun 语句，作为 **gp:getDialogInput** 函数的第一行代码：

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

该函数有一个参数 (pathwidth)，同时还声明了几个局部变量。

4 在第三步加入的代码后面，输入本章如下几节中的代码样例：

- 设置对话框的值
- 加载对话框文件
- 将指定对话框加载到内存中
- 初始化缺省的对话框值
- 给控件指定动作

注意 请只需输入“给控件指定动作”中的第一段代码样例，而不要输入后面用于解释代码的代码段，这些代码只是第一段代码样例的重复。

- 启动对话框
- 卸载对话框
- 确定下一步的动作

5 在最后一行代码后，加入：

```
  )
 )
 Result;
) ;_ 结束 defun
```

6 从 VLISP 菜单上选择“工具”➢“设置编辑器中代码的格式”，设置您输入代码的格式。

更新简单空函数

现在您完成了 **gp:getDialogInput** 函数。在修改任何一个简单空函数时，都应该就如下方面进行检查：

- 是否修改了 defun 语句？也就是说，函数的参数数目是否仍相同？
- 函数的返回值是否有了不同？

对于 **gp:getDialogInput**，两个问题的答案都是“是”。该函数现在以小路宽度作为参数，用来设置砖尺寸和间距的缺省值。该函数的简单空函数返回 T，而现在它返回的是包含四个新值的关联表。

这两个修改将影响调用该函数的函数和处理该函数返回值的代码。所以需要用下述代码替换以前的 **C:GPath** 函数（在 *gpmain.lsp* 中）：

```
(defun C:GPath (/ gp_PathData gp_dialogResults)
 ;; 要求用户输入：先是小路的位置和方向，
 ;; 然后是小路参数。只有在获取了有效值后才继续执行。
 ;; 数据保存在 gp_PathData 中。
 (if (setq gp_PathData (gp:getPointInput))
  (if (setq gp_dialogResults (gp:getDialogInput (cdr(assoc 40
                  gp_PathData))))
```

```
(progn
 ;; 现在先取得 gp:getPointInput 的结果，然后将
 ;; gp:getDialogInput 提供的信息附加到该结果后面。

 (setq gp_PathData (append gp_PathData gp_DialogResults))

 ;; 现已取得所有用户输入信息。
 ;; 绘制小路轮廓，并将指向结果多段线的指针
 ;; 保存到名为 PolylineName 的变量中。
 (setq PolylineName (gp:drawOutline gp_PathData))
) ;_ 结束 progn
 (princ "\nFunction cancelled.")
 ) ;_ 结束 if
 (princ "\nIncomplete information to draw a boundary.")
 ) ;_ 结束 if
 (princ) ; 静默退出
) ;_ 结束 defun
```

请注意修改后的主函数 **C:GPath** 中的黑体字。为了让程序正确运行，这里做出了两个重要修改：

- 调用函数 **gp:getDialogInput** 时，将小路宽度作为参数传给了它。其中小路宽度是通过将关联表 gp_PathData 中组码为 40 的关联值取出得到的。
- **gp:getPointInput** 函数返回的关联表赋给了名为 **gp_dialogResults** 的变量。如果该变量有了值，还要将它包含的值附加到已存到变量 **gp_PathData** 中的关联表后。

修改后的 **gp:getPointInput** 还有另外一些变化。最简单的办法是复制在线教程上的代码，然后粘贴到您的文件中。

提供可选的边界线类型

开发花园小路应用程序的一个要求是允许用户选择绘制边界多段线的类型：是优化多段线还是普通多段线。第一个版本的 **gp:drawOutline** 总是用优化多段线来绘制边界。但现在对话框界面已能让用户选择使用普通多段线。为了达到这个目的，**gp:drawOutline** 函数必须先确定使用哪种类型的多段线，然后再绘制该多段线：

对 **gp:drawOutline** 函数的必要改动包括在下述代码中，所需改动已用黑体字标出，请修改您的 gpdraw.lsp 文件：

```
(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))
   ;; 获取多段线类型
   plineStyle   (strcase (cdr (assoc 4 BoundaryData)))
) ;_ 结束 setq
;; 用 ActiveX Automation 将多段线添加到模型空间
(setq pline (if (= plineStyle "LIGHT")
    ;; 创建优化多段线
    (vla-addLightweightPolyline
        *ModelSpace*     ; 模型空间的全局定义
        (gp:list->variantArray poly2Dpoints) ; 数据转换
     ) ;_ 结束 vla-addLightweightPolyline
    ;; 或创建普通多段线
    (vla-addPolyline
        *ModelSpace*
        (gp:list->variantArray poly3Dpoints) ;data conversion
    ) ;_ 结束 vla-addPolyline
   ) ;_ 结束 if
) ;_ 结束 setq
```

修改代码时需要很小心，因为不仅需要添加所需代码，还要删除某些已有的代码，或者将一些代码重新排列。建议您先从在线教程上复制整个 **setq** 语句，然后再将它粘贴到您的代码中。

收尾工作

如果您尚未删除不再需要的代码，那么请从 gpmain.lsp 文件的 **C:GPath** 函数中删除如下代码：

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

这些代码以前是作为占位符用的，既然现在 **gp:drawOutline** 已经可以工作，不再需要这些代码了。

运行应用程序

在运行应用程序以前，别忘了先保存这些文件。可以从 VLISP 菜单上选择“文件”➢“全部保存”，或按下 ALT+SHIFT+S 快捷键保存所有打开的文件。

下一步必须做的是在 VLISP 中重新加载所有文件。

加载并运行应用程序中的所有文件的步骤

1 如果您现在没有打开本课前面章节中创建的工程文件，请从 VLISP 菜单中选择“工程”➢“打开工程”，然后输入工程文件名 gpath，注意不要包括 .prj 扩展名。如果 VLISP 没找到该工程文件，可单击“浏览”按钮，并从“打开工程”对话框选择文件，再单击“打开”按钮。

2 单击工程窗口中的“加载源文件”按钮。

3 在 VLISP 控制台提示处输入 **(C:GPath)** 命令，运行该程序。如果需要调试程序，可以使用您在第二课和第三课中学到的工具。记住，如果运行程序失败，您可以将代码从 Tutorial\VisualLISP\Lesson4 目录下复制过来。

同时，请用优化多段线和普通多段线两种方法绘制小路。在绘制完小路后，可用 AutoCAD 的 **list** 命令来判断程序所绘制的图元类型是否正确。

第四课回顾

在本课中，您

- 将代码分为四个文件，使代码模块化。
- 用 VLISP 工程来组织代码模块。
- 学习了用对话框控制语言 (DCL) 定义对话框。
- 添加 AutoLISP 代码来设置对话框并处理对话框中的输入信息。
- 修改代码，允许用户选择边界线类型。

现在您的程序可绘制花园小路的边界了。在下一课中，您将给花园小路添上砖。在该过程中将会介绍更多的 VLISP 程序开发工具。

绘制砖

本课内容包括

5

- 更多的 Visual LISP 编辑工具
- 在花园小路中添加砖
- 测试代码
- 第五课回顾

本课结束时，您的应用程序将能达到第一课中提到的基本要求。您要给程序添上在花园小路的边界里绘制砖的函数，而且该函数还能提供几种不同的方法来创建图元。您还会学到一些键盘快捷键和新的编辑工具。

更多的 Visual LISP 编辑工具

如果您没有在 VLISP 编辑窗口中打开 gpdraw.lsp 文件，请先打开该文件。有些事情您在用 VLISP 开发程序时经常会遇到。首先，有不少括号，而且括号中还有括号。第二，有许多函数调用，而且有些函数的名称很长（例如 **vla-addLightweightPolyline** 函数）。VLISP 提供了一些编辑工具，帮您处理这些 AutoLISP 代码的共同特征。

括号匹配

VLISP 提供了括号匹配功能，可帮您查找和开括号相对应的闭括号。

匹配开括号的闭括号的步骤

1 将光标放在 **setq** 函数调用前的开括号前面。

2 按 CTRL＋SHIFT＋]（双击也可以完成同样的功能）。

VLISP 将找到和您所选的开括号相匹配的闭括号，并选中两者间的代码。这不仅使您能检查输入的括号数目是否正确，而且还简化了复制或剪切选定文本的操作。在第四课最后更新该函数调用时，这个功能就很有用。

您还会在什么时候用到这一功能呢？您可以将某段代码复制到 VLISP 控制台窗口。或者，您可能想用 3 行很精彩的代码来替换原来的 50 行代码。用括号匹配工具可以很快选中原有代码，然后按一个键就能删除它们。这一功能可让 VLISP 很快就找到一整块代码，而不用一行一行地查找最后的闭括号。

同样也有相应的键盘命令来向前匹配和选取代码。将光标放到某闭括号后面，双击或按 CTRL＋SHIFT＋[，VLISP 将查找相应的开括号，并选中其中的代码。

从 VLISP 菜单上选择“编辑”➤“括号匹配”，也可以使用这两个命令。

自动完成词语

假设您用如下代码给程序添加新功能：

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

（不必管代码实际干什么，它只是一个包含有几个长变量名和函数名的代码样例。）

VLISP 可为您自动完成词语，让您少键入一些字符。

使用 Visual LISP 的按历史匹配完成词语功能的步骤

1 滚动窗口到 gpdraw.lsp 文件的底部，输入如下代码：

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

2 按 CTRL＋SPACEBAR。

VLISP 将根据您最后键入的两个字符，在当前文件中搜索，并找到最接近的匹配项，这可让您少键入 17 个字符。

3 完成该行代码，如下所示：

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

4 输入如下代码行：

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

5 按 CTRL＋SPACEBAR。

VLISP 将匹配最近输入的词语，它是 progn。然而，您需要的词语是 polar。如果您继续按 CTRL＋SPACEBAR，VLISP 将在您代码中的所有可能匹配项之间循环。最后，您会找到 polar。

6 删除您刚才输入的所有代码，它们仅用于示范。

从 VLISP 的“搜索”菜单也能访问匹配完成词语功能。

按系统匹配完成词语

如果您以前用过 AutoLISP，您可能键入过类似下述语句的表达式：

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

有时，关于选择集的函数（如 **ssname**、**ssget**、**sslength** 等）很容易让人混淆。在这种情况下，VLISP 的按系统匹配完成词语功能可以帮助您。

使用 Visual LISP 的按系统匹配完成词语功能的步骤

1. 滚动窗口到 gpdraw.lsp 文件的底部，在空行上输入如下代码：

   ```
   (setq myEnt (ent
   ```

2. 按 CTRL＋SHIFT＋SPACEBAR。

   VLISP 显示以字符 ent 开头的所有 AutoLISP 符号。

   使用方向键（向上和向下的方向键）在列表中移动，选择 ENTGET，然后按 ENTER。

   VLISP 将用 ENTGET 替换您输入的 ent。

3. 删除这些代码。

获取函数帮助

在图形中绘制优化多段线的代码中调用了一个名为 **vla-addLightweightPolyline** 的函数。不仅键入这个冗长的函数名很麻烦，而且用来创建图元的几个函数的名字都是以 **vla-add** 开头的。然而，在编写程序代码时并不用总去手册中查找函数名，VLISP 可以帮助您。

获取使用函数的帮助的步骤

1. 在某空行中输入如下代码：

   ```
   (vla-add
   ```

2. 按 CTRL＋SHIFT＋SPACEBAR。
3. 在列表中滚动，直到找到 vla-addLightweightPolyline。
4. 双击 vla-addLightweightPolyline。

   VLISP 显示所选函数的“符号服务”对话框。

5. 在“符号服务”对话框中单击“帮助”按钮。（对 ActiveX 函数，您将被引导至 ActiveX and VBA Reference 中。）

6 删除您在 gpdraw.lsp 文件中所做的修改，它们仅用于示范。同时请关闭“符号服务”和“自动匹配”窗口。

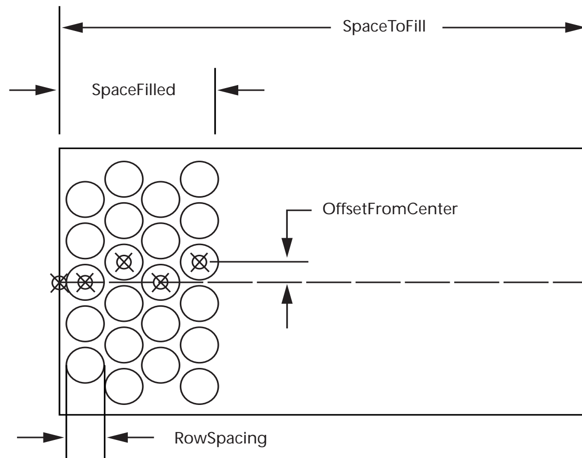
在花园小路中添加砖

现在您已经绘制了小路的边界，并准备着手用砖填充小路。您将需要应用一些逻辑知识，还得做一些几何分析。

应用逻辑知识

您要做的第一件事情是确定怎样摆放和绘制砖。如果是简单的矩形砖，您可以用 AutoCAD 的 ARRAY 命令来绘制。但对花园小路，您得让每排砖都相对于前一排砖有一点偏移。

这种每排之间的偏移构成了漂亮的、不断重复的编排图案。想想在修建真正的小路时是怎样摆放砖的。您可能会先从小路的某个端点出发，一排接一排地放，直到剩下的空间不够放一排砖为止。



下面是该过程的逻辑伪代码：

从小路的起点开始
给出从路中间开始计算的当前那排砖的偏移值，该偏移值为 0（砖放在路中间）或一个“砖间距”长度。
当待填充的空间大于每排砖填充的空间时，
 画一排砖。
 重新设置下一步的起点（增加一个“砖间距”）
 将这排砖填充的距离累加到已填充距离中。
 切换偏移值（如果是在路中间，则切换成让它偏移，
 或相反操作，从偏移切换为在路中间）。
 回到循环的起点

几何分析

绘制花园小路时只需知道几个尺寸。半宽很容易理解：它就是小路宽度的一半。您已经编写了从用户处获取该宽度的代码，并已将它保存在关联表中。

砖间距也容易理解：它是半径的两倍（即直径）再加上砖之间的距离。该尺寸也是由用户输入的。

排间距则有点麻烦，您得精通三角函数。它的公式如下：

排间距 = (砖直径 + 砖之间的距离) * (60 度的正弦值)

绘制一排砖

看看您是否能理解如下函数的含义。把它和前面的伪代码相比较，尽量理解刚讲解过的三角函数计算。其中可能有一些 AutoLISP 函数是您第一次碰到。如果您需要这些函数的帮助信息，请参见 AutoLISP 参考。现在您只需阅读这些代码，而不用编写任何代码。

```
(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))
                ) ;_ 结束 *
```

```
    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)))
 ) ;_ 结束 setq

 ;; 对下面循环中第一次调用 gp:calculate-Draw-tileRow
 ;; 进行补充。
 (setq rowStartPoint
    (polar rowStartPoint
       (+ pathAngle pi)
       (/ TileRadius 2.0)
    ) ;_ 结束 polar
 ) ;_ 结束 setq
 ;; 绘制每一排砖。
 (while (<= SpaceFilled SpaceToFill)
  ;; 获取所创建砖的列表，将它们加入到我们的表中。
  (setq tileList   (append tileList
         (gp:calculate-Draw-TileRow
          (setq rowStartPoint
              (polar rowStartPoint
                   pathAngle
                   RowSpacing
              ) ;_ 结束 polar
          ) ;_ 结束 setq
          TileRadius
          TileSpace
          pathWidth
          pathAngle
          offsetFromCenter
          ObjectCreationStyle
         ) ;_  结束 gp:calculate-Draw-TileRow
        ) ;_ 结束 append
   ;; 计算沿小路的距离以绘制下一排砖。
   SpaceFilled   (+ SpaceFilled RowSpacing)
   ;; 在 0 和正的偏移量间切换
   ;; （使排与排之间成锯齿状）。
   offsetFromCenter
       (if (= offsetFromCenter 0.0)
            offsetDistance
            0.0
       ) ;_ 结束 if
  ) ;_ 结束 setq
 ) ;_ 结束 while
 ;; 返回所创建砖的列表。
 tileList
) ;_ 结束 defun
```

可能还需要对这段代码的某几节做些解释。

如下代码段正好位于 **while** 循环开始之前：

```
;; 对第一个起点进行补偿！
(setq rowStartPoint(polar rowStartPoint
  (+ pathAngle pi)(/ TileRadius 2.0)))
```

该语句中有三个难点需要解释一下，它们也是该算法应用到的逻辑知识：

- rowStartPoint 变量是在函数 **gp:Calculate-and-Draw-Tile** 中声明的，开始赋给它的初值为用户选择的小路起点。
- 传给第一次调用 **gp:calculate-Draw-TileRow** 函数的参数还进行了如下操作：

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

  换句话说，在调用 **gp:calculate-Draw-TileRow** 函数时，变量 rowStartPoint 被设为距当前 rowStartPoint 沿小路方向一个 RowSpacing 远处。
- 在 **gp:calculate-Draw-TileRow** 函数中，参数 rowStartPoint 用作该排圆砖中间那块砖的圆心（可能还要做一下偏移）。

为了对绘制第一排砖（**while** 的第一个循环）所用的 rowStartPoint 的初始偏移进行一些补偿，您需要将 rowStartPoint 向相反的方向稍微作一些偏移，其目的是避免在小路边界和第一排砖之间有太大的空隙。这时偏移半个 TileRadius 就已经足够了，可使用 **polar** 函数将 rowStartPoint 沿着和 PathAngle 相差 180 度（即反方向）的方向进行偏移。仔细想一想，您会发现该点会暂时落在小路边界之外。

下面一段代码（为方便阅读，已稍作改动）可能也较难理解：

```
(setq tileList (append tileList
            (gp:calculate-Draw-TileRow
             (setq rowStartPoint
               (polar rowStartPoint pathAngle RowSpacing)
             ) ;_ 结束 setq
             TileRadius TileSpace pathWidth pathAngle
             offsetFromCenter ObjectCreationStyle
            )))
```

实质上，这儿先调用 **gp:calculate-Draw-TileRow** 函数，然后再调用包在它外面的 **append** 函数，最后是调用包在最外面的 **setq** 函数。

gp:calculate-Draw-TileRow 函数将返回所绘制的每一块砖的 Object ID（该 Object ID 指向图形中的砖对象）。因为您是一排接一排地绘制砖，所以该函数每次返回一排砖的 Object ID。**Append** 函数将新的 Object ID 附加到保存在 tileList 中的现有 Object ID 中。

在靠近函数的最后处，您会发现如下代码段：

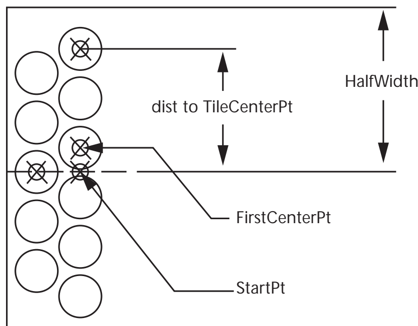
```
(setq offsetFromCenter
  (if (= offsetFromCenter 0.0)
      offsetDistance
      0.0
  )
)
```

它的作用是切换偏移值，偏移值可确定要画的那排砖中的第一块是应该在小路中间，还是有点偏移。该算法的伪代码如下所示：

```
按下述规则设置偏移：
  如果当前偏移值为 0，则将它设为偏移距离；
    否则，将它设为 0。
```

绘制每一排砖

既然您已经有了绘制小路中砖的思路，下一步是指出怎样绘制每一排中的砖。在下图中包含了两种情况：一排砖相对于小路中心的偏移值为 0，另一排砖的偏移值则不为 0。先看一下该图，然后再仔细阅读随后的伪代码。




```
设置 StartPoint、angp90、angm90 等变量。
将变量 FirstCenterPoint 设为 StartPoint + offset（offset 可能为 0）。
将 TileCenterPt 的初始值设为 FirstCenterPoint。
```

（注释：首先将在 angp90 的方向上绘制圆。）
当从 StartPoint 到 TileCenterPt 的距离小于 HalfWidth 时：
　绘制一个圆（并将它添加到圆表中）。
　将 TileCenterPt 沿 angp90 方向移动一个砖间距的距离。
循环结束

将 TileCenterPoint 重新设为 FirstCenterPoint 沿 angm90 方向移动一个砖间距的距离。
当从 StartPoint 到 TileCenterPt 的距离小于 HalfWidth 时：
　绘制一个圆（并将它添加到圆表中）。
　将 TileCenterPt 沿 angp90 方向移动一个砖间距的距离。
循环结束

返回圆表

阅读代码

现在看一下 **gp:calculate-Draw-TileRow** 函数的代码：

```
(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)
    Tilespacing (+ (* 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")
      )
    )
   )
   ;; 绘制路中心左边的圆。
   (while (< (distance startPoint tileCenterPt) HalfWidth)
     ;; 将每块砖添加到要返回的表中。
    (setq tileList
         (cons
         (ObjectCreationFunction tileCenterPt TileRadius)
               tileList
         )
    )

    ;; 计算下一块砖的中心点位置。
    (setq tileCenterPt
      (polar tileCenterPt angp90 TileSpacing)
    )
   );_ 结束 while

   ;; 绘制路中心右边的圆。
   (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
           )
      )

    ;; 计算下一块砖的中心点位置。
     (setq tileCenterPt (polar tileCenterPt angm90 TileSpacing))
   );_ 结束 while

   ;; 返回砖表
   tileList
) ;_ 结束 defun
```

加上下述代码，就完成了由伪代码向 AutoLISP 代码的转换工作：

```
(setq 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 the developer for assistance."
        "\n        ObjectCreationStyle set to ACTIVEX"
       ) ;_ strcat
     ) ;_ 结束 alert
     (setq ObjectCreationStyle "ACTIVEX")
    )
  ) ;_ 结束 cond
) ;_ 结束 setq
```

在前面的说明中曾提到允许用户用三种方式绘制砖（圆）：使用 ActiveX、**entmake** 函数或 **command** 函数吗。根据 ObjectCreationStyle 参数（从 C:GPath 由 **gp:Calculate-and-Draw-Tiles** 传入）可将变量 ObjectCreationFunction 设置为三个函数之一。这三个函数在 gpdraw.lsp 中的定义如下所示：

```
(defun gp:Create_activeX_Circle (center radius)
  (vla-addCircle *ModelSpace*
    (vlax-3d-point center) ; 转换为和 ActiveX 兼容的三维点
    radius
  )
) ;_ 结束 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))
)
```

第一个函数用 ActiveX 函数绘制圆并返回 ActiveX 对象。

第二个函数用 **entmake** 函数绘制圆并将图元名转换为 ActiveX 对象后返回。

第三个函数则用 **command** 函数绘制圆，它也将图元名转换为 ActiveX 对象后返回。

测试代码

如果已完成了上述工作，那您现在可以对代码进行测试了。

测试代码的步骤

1. 关闭 VLISP 中的所有活动窗口，包括打开的任何工程窗口。
2. 将 Tutorial\VisualLISP\Lesson5 目录下的所有内容都复制到您的教程目录 MyPath 下。
3. 从 VLISP 菜单栏上选择“工程”➤“打开工程”，打开工程文件 gpath5.prj。
4. 加载工程源文件。
5. 切换到 AutoCAD 窗口并发出 **gpath** 命令，运行该程序。
6. 运行 **gpath** 绘制花园小路三次，每次用不同的图元创建方法。请注意用各种方法绘制小路时的速度差别。

第五课回顾

本课开始所讲述的 VLISP 编辑功能可帮您

- 在代码中进行括号匹配。
- 查找并完成函数名。
- 获取函数的在线帮助。

完成本课时，您已经获得了绘制花园小路中的砖的程序代码。现在程序已能达到本教程最开始提出的要求。

现在您获得的 VLISP 经验可能足以让您在 VLISP 中开发自己的程序了。不过在您从事开发时，还有一些内容值得参考——本教程的最后两课讲述的是如何使用反应器函数，以及 VLISP 环境的其他高级功能。

使用反应器

本课内容包括

6

- 反应器基础
- 设计花园小路应用程序的反应器
- 测试反应器
- 第六课回顾

在本课中，您将学习反应器，以及如何将它们附着到图形事件和图元上。反应器可以在特定事件发生时让 AutoCAD 通知您的应用程序。例如，如果用户移动附着了反应器的图元，您的应用程序将接到该图元已被移动的通知。这样您可以让程序触发另外的操作，如移动与用户所移动图元相关联的其他图元，或更新用于记录图形修改信息的文本标签等。实际上，它相当于给您的应用程序配一个传呼机，在特定事件发生时让 AutoCAD 通知您的应用程序。

反应器基础

反应器是附着到图形编辑器或图形中某指定图元的一个对象。如果从传呼机的比喻出发，反应器对象相当于一个自动拨号器，它知道在特定事件发生时呼叫您的传呼机。应用程序中的传呼机是该反应器调用的一个 AutoLISP 函数，此类函数称为回调函数。

注意 最后两课中应用程序代码的复杂程度和对编程技巧的要求要大大高于第一到第五课。由于需要讲解大量的信息，所以不能象以前的课程那么详细地给出解释。如果您是初学者，不必担心在第一次时无法完全理解，只是把它看成是对 VLISP 的一些非常强大但在技术上也非常复杂的功能的第一次尝试就可以了。

反应器类型

AutoCAD 的反应器有许多种类型。每种反应器对应于一个或多个 AutoCAD 事件。反应器可以分为如下几个大类：

编辑器反应器	在调用 AutoCAD 命令时通知应用程序
链接反应器	在加载或卸载 ObjectARX 应用程序时通知应用程序。
数据库反应器	用于与图形数据库中的特定图元或对象联系。
文档反应器	在 MDI 模式下改变当前图形文档时（如打开新图形文档、激活另外的文档窗口、改变文档的锁定状态等）通知应用程序。
对象反应器	在指定对象被修改、复制或删除时通知您的应用程序。

除了编辑器反应器之外，每大类反应器中只有一种反应器类型，而编辑器反应器中则包括多种反应器：例如，DXF™ 反应器在输入或输出 DXF 文件时将通知应用程序，而鼠标反应器将通知鼠标事件（如双击等）。

在各类反应器中，您可以给反应器附着多种事件。AutoCAD 允许用户完成多种不同的操作，但需要您自己决定对什么操作感兴趣。一旦决定了事件种类，您就可以将您的反应器（自动拨号器）附着到该事件上，然后再编写该事件发生时要触发的回调函数。

设计花园小路应用程序的反应器

为了实现花园小路应用程序的反应器功能，下面先从一小部分事件开始，而不是一下子就考虑所有可能的事件。

选择花园小路应用程序的反应器事件

本教程的目标如下：

- 用户重新设置小路边界的角点（多段线的顶点）时，重新绘制该小路，使其轮廓仍保持为矩形。另外，还得基于新尺寸和形状重新绘制小路上的砖。
- 用户删除花园小路边界时，同时删除小路上的砖。

规划回调函数

对每个反应器事件，必须规划在事件发生时要调用的回调函数。以下伪代码从逻辑上给出了当用户将多段线顶点拖到新位置时应发生的事件序列：

```
Defun gp:outline-changed
删除小路上的砖。
    确定应如何修改边界。
    使边界成为矩形。
    重新绘制小路上的砖。
函数结束
```

但事情还是有点复杂。当用户开始拖动轮廓多段线的顶点时，AutoCAD 发出 **:vlr-modified** 事件通知您的应用程序。但此时用户刚开始拖动多段线的一个顶点。如果立即调用 **gp:outline-changed** 函数，就会中断用户正在执行的操作，这样就无法知道新顶点应该在什么位置，因为用户还没有确定它的位置。最后，在用户拖动多段线时，AutoCAD 也不会允许函数修改该多段线对象。在用户修改多段线对象时，AutoCAD 将打开该多段线对象，直到用户重新指定对象的位置完成对对象的修改。

所以需要修改前面的思路，修改后它应如下所示：

```
当用户开始拖动多段线顶点时，
 调用 gp:outline-changed 函数
 Defun gp:outline-changed.
   设置一个全局变量保存指向用户修改的多段线的指针。
 函数结束
当命令完成时，
 调用 gp:command-ended 函数
 Defun gp:command-ended.
        删除小路上的砖。
        获取以前多段线顶点位置的信息。
        获取现在多段线顶点新的位置信息。
        重新定义该多段线（使它成为矩形）。
        重新绘制小路上的砖。
 函数结束
```

当用户完成对小路轮廓的修改时，如果您已设置了一个编辑器反应器，AutoCAD 将发出一个 **:vlr-commandEnded** 事件通知您的应用程序。

用一个全局变量来指明用户修改的多段线是必要的，因为在函数 **gp:outline-changed** 和 **gp:command-ended** 之间是没有连续性的。在应用程序中，这两个函数相互独立地被调用和运行。全局变量中保存的重要信息是在一个函数中设置的，而被另一个函数访问。

现在再考虑一下如果用户删除花园小路边界时应该怎么办。最终的结果应该是删除小路上所有的砖。下面的伪代码给出了解决问题的逻辑思路：

```
当用户开始删除边界时，
 调用 gp:outline-erased 函数
 Defun gp:outline-erased.
   设置一个全局变量保存指向用户删除的多段线所附着的反应器的指针。
 函数结束
当删除操作完成时，
 调用 gp:command-ended 函数
 Defun gp:command-ended.
    删除属于现在删除的多段线所对应小路中的所有砖。
 函数结束
```

规划多重反应器

用户可能在屏幕上绘制了好几条花园小路，因此可能一次删除多条花园小路。您需要考虑这种可能性。

和图元相关联的反应器是对象反应器。如果图形中有多个图元，其中也可能有多个对象反应器（每个图元一个反应器）。有些编辑事件（如删除命令），可能触发多个回调函数，这取决于有多少个所选图元附着有反应器。另一方面，编辑器反应器一般只有一个，所以您的应用程序只能附着一个 **:vlr-commandEnded** 事件反应器。

这意味着 **gp:command-ended** 函数要处理两种情况：一是改变顶点位置，一是删除整个多段线。您需要确定在每种情况下要执行哪些操作，如下伪代码给出了逻辑思路：

```
Defun gp:command-ended （版本 2）
 （从全局变量中）获取指向多段线的指针。
    分情况讨论：
      如果已修改多段线：
        删除小路上的砖。
        获取以前多段线顶点位置的信息。
        获取现在多段线顶点新的位置信息。
        重新定义该多段线（使它成为矩形）。
        重新绘制小路上的砖。
      条件表达式结束
      如果已删除多段线：
        删除小路上的砖。
      条件表达式结束
    分情况讨论结束
函数结束
```

附着反应器

规划基于反应器的应用程序的下一步是确定如何以及何时附着反应器。您需要给任何花园小路的多段线边界附着两个对象反应器（一个响应修改事件，另一个响应删除事件），还需要附着一个编辑器反应器，让它在用户完成对多段线的修改时通知您的应用程序。对象反应器是附着到图元上的，而编辑器反应器则应该在 AutoCAD 中注册。

还有一件事情需要考虑清楚。为了在用户修改多段线轮廓后重新计算它并使它仍保持矩形外观，您必须知道多段线被修改以前的顶点结构。在多段线被修改后，该信息就无法确定，因为那时您只能得到它的新顶点结构。那么怎样解决这个问题呢？您可以把该信息保存在全局变量中，但用这种方法会有很大一个问题——由于用户可以绘制许多条花园小路，那么每一条小路都将要求一个全局变量，这样程序就会变得很乱。

用反应器保存数据

可利用 VLISP 反应器的另一个功能来解决这个问题——反应器有保存数据的能力，所以可用它来保存多段线最初的顶点结构。当用户第一次绘制小路边界时，您将要保存的数据保存到反应器，然后再附着到该边界。根据这个思路，可对主程序函数 **C:GPath** 作如下修改：

```
Defun C:GPath
  完成花园小路程序以前完成的所有工作
  （不要中断任何已有工作）。
  用下述参数将对象反应器附着到多段线上：
    指向刚绘制的多段线的指针，
    您要反应器记录的数据表，
    要跟踪的特定多段线对象事件表，
    要调用的 LISP 回调函数。
  对象反应器设置结束。

  用如下参数将编辑器反应器附着到图形编辑器中：
    所有您要附着到反应器上的数据（在本例中没有这种数据）
    要跟踪的特定编辑器反应器事件表，
    要调用的 LISP 回调函数。
  编辑器反应器设置结束。
函数结束
```

更新 C:GPath 函数

更新 **C:GPath** 函数，加入反应器创建部分。

在 C:GPath 中增加反应器创建部分的步骤

1 用下述代码替换您的旧 gpmain.lsp 文件，可从 <AutoCAD 目录>\Tutorial\VisualLISP\Lesson6 下复制这些代码：

```
(defun C:GPath (/
       gp_PathData
       gp_dialogResults
       PolylineName
       tileList
         )
 (setvar "OSMODE" 0)             ;; 关掉对象捕捉功能。
 ;|
 ;; 第六课将把一个简单空命令反应器添加到 AutoCAD 中。
 ;; 然而，如果用户选择用 COMMAND 函数绘制圆砖，
 ;; 那么每画一块砖就会响应一次，这样会让人觉得不舒服。
 ;; 所以，如果存在 *commandReactor*，就先将它禁用。
 ;;|
```

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

;; 要求用户输入：先是小路的位置，然后是小路的
;; 参数。只有在获取了有效值后才继续执行，并将
;; 数据保存在 gp_PathData 中。
(if (setq gp_PathData (gp:getPointInput))
  (if (setq gp_dialogResults
        (gp:getDialogInput
      (cdr (assoc 40 gp_PathData))
        ) ;_ 结束 gp:getDialogInput
  ) ;_ 结束 setq

    (progn
  ;; 现在先取得 gp:getPointInput 的结果，然后将
  ;; gp:getDialogInput 提供的信息附加到该结果后面。

  (setq gp_PathData (append gp_PathData gp_DialogResults))

  ;; 现在您已取得所有用户输入信息。
  ;; 绘制小路轮廓，并将指向结果多段线的指针
  ;; 保存到名为 PolylineName 的变量中。

  (setq PolylineName (gp:drawOutline gp_PathData))

  ;; 下一步该绘制边界里的砖了。
  ;; gp_tileList 中包括指向这些砖的对象指针，
  ;; 通过计算指针的数目（用 length 函数），
  ;; 我们可以知道所画砖的数目。
  (princ "\nThe path required ")
  (princ
    (length
      (setq tileList (gp:Calculate-and-Draw-Tiles gp_PathData))
    ) ;_ 结束 length
  ) ;_ 结束 princ
  (princ " tiles.")

  ;; 将（由 gp:Calculate-and-Draw-Tiles 返回的）
  ;; 指向砖的指针列表添加到 gp_PathData 中。
  ;; 这些数据将保存在附着到边界多段线上的反应器中。
  ;; 有了这些数据，多段线就能“知道”哪些砖（圆）
  ;; 是属于它的。

  (setq gp_PathData
       (append (list (cons 100 tileList))
              ; 所有砖
          gp_PathData
       ) ;_ 结束 append
  ) ;_ 结束 setq
```

```
;; 在将反应器数据附着到对象之前，先看看
;; vlr-object-reactor 函数。
;; vlr-object-reactor 包括如下参数：
;;  (vlr-object-reactor owner 謂 data callbacks)
;;     callback 参数是如下形式条目的表：
;;     '( 事件名称 . 回调函数 ).
;;
;; 在本练习中我们将使用和 vlr-object-reactor
;; 相关联的所有参数。
;; 只有在修改或删除 PolylineName 中的多段线时，
;; 才会执行这些反应器函数。

(vlr-object-reactor

  ;; vlr-object-reactor 的第一个参数是 "Owner's List"，
  ;; 该参数指定和反应器相关联的对象所需放置的位置。
  ;; 本例中，它是保存在 PolylineName 中的 vlaObject。

  (list PolylineName)

  ;; 第二个参数包含小路的数据

  gp_PathData

  ;; 第三个参数是我们要用的特定反应器类型。
  '
   (
    ;; 当修改对象时要调用的反应器
    (:vlr-modified . gp:outline-changed)
    ;; 当删除对象时要调用的反应器
    (:vlr-erased . gp:outline-erased)
   )
) ;_ 结束 vlr-object-reactor

;; 下一步要注册命令反应器，
;; 在修改命令完成时调整多段线。
(if (not *commandReactor*)
  (setq *commandReactor*
    (VLR-Command-Reactor
      nil        ; 没有和命令反应器关联的数据
      '(
        (:vlr-commandWillStart . gp:command-will-start)
        (:vlr-commandEnded . gp:command-ended)
       )
    ) ;_ 结束 vlr-command-reactor
  )
)
```

```
;; 以下代码在关闭图形时删除所有的反应器，
;; 这点非常重要 !!!!!!!!!
;; 不这么做，AutoCAD 在退出时可能会崩溃！
(if (not *DrawingReactor*)
    (setq *DrawingReactor*
      (VLR-DWG-Reactor
        nil        ; 没有和图形反应器关联的数据
        '((:vlr-beginClose . gp:clean-all-reactors))
         )
      ) ;_ 结束 vlr-DWG-reactor
    )
)
    ) ;_ 结束 progn
    (princ "\nFunction cancelled.")
   ) ;_ 结束 if
   (princ "\nIncomplete information to draw a boundary.")
  ) ;_ 结束 if
  (princ)            ; 静默退出
) ;_ 结束 defun

;;; 显示一个消息，让用户知道命令名。
(princ "\nType GPATH to draw a garden path.")
(princ)
```

2 检查修改的代码以及说明新语句功能的注释。本教程把所有要修改的代码用黑体字显示。

添加反应器的回调函数

反应器回调函数给您的应用程序添加了不少代码，在 Lesson6 目录下提供了这些代码。

将反应器回调函数添加到程序中的步骤

1 将文件 gpreact.lsp 从 Tutorial\VisualLISP\Lesson6 目录复制到您的工作目录 MyPath 中。

2 打开 GPath 工程（如果尚未打开它的话），在 gpath 工程窗口中单击“工程特性”按钮。

3 将文件 gpreact.lsp 文件添加到工程中。

4 gpreact.lsp 文件在文件列表中的位置没有要求，但请记住 utils.lsp 文件必须是第一个，而 gpmain.lsp 文件必须是最后一个文件。如果必要，可移动这些文件，然后单击“确认”按钮。

5 在 gpath 工程窗口中双击文件名 gpreact.lsp 打开该文件。

请通读该文件中的注释，它可以帮您理解代码的含义。注意所有的回调函数都是简单空函数，它们完成的唯一功能是在它们被触发时显示警告信息。

该文件中的最后一个函数非常重要，所以它有一个自己的说明标题。

清除反应器

反应器是非常活跃的，所以在您设计依赖于反应器的程序时，它很可能经常会让您的程序甚至 AutoCAD 崩溃。VLISP 提供了一个工具可帮助您在必要时删除所有添加的反应器。

gpreact.lsp 文件包括一个名为 **gp:clean-all-reactors** 的函数，它除了调用函数 **CleanReactors** 外，没有做什么其他工作。请复制如下代码并将它粘贴到您的 utils.lsp 文件的最后：

```
;;;--------------------------------------------------------------
;;;    函数：CleanReactors
;;;--------------------------------------------------------------
;;;  说明：用于清除反应器的通用工具函数。
;;;          它可以在调试时使用，也能在关闭
;;;          图形以前清除任何打开的反应器。
;;;--------------------------------------------------------------
(defun CleanReactors ()
  (mapcar 'vlr-remove-all
         '(:VLR-AcDb-reactor
          :VLR-Editor-reactor
          :VLR-Linker-reactor
          :VLR-Object-reactor
         )
  )
)
```

测试反应器

现在，您有了使用反应器所需的所有代码，可对它们进行测试了。

测试反应器代码的步骤

1 从工程加载所有源代码。（只需在 gpath 工程窗口单击“加载源文件”按钮即可。）

2 运行 **C:GPath** 函数，并将它旋转。

   该程序将绘制一条花园小路，就象您在第五课看到的一样。开始时您不会看到新鲜的东西。

3 在您绘制了小路之后，可试着做如下操作：

   - 移动多段线的一个顶点。可拾取多段线，打开它的夹点功能，然后将它的一个顶点拖到新位置。
   - 拉伸多段线。

■ 移动多段线。
■ 删除多段线。

检查一下显示的消息。您是在观察该强大功能的幕后操作过程。

（如果您的应用程序运行不正确，而您现在又不想花太多时间进行调试，您可以直接运行 Tutorial\VisualLISP\Lesson6 目录下提供的代码样例。此时请使用该目录下的 Gpath6 工程。）

注意 由于使用了反应器，您可能会注意到在 AutoCAD 中测试完反应器后，按 ALT+TAB 不能回到 VLISP，单击 VLISP 窗口也无法激活它。如果出现这种情况，只需在 AutoCAD 命令提示处输入 **vlisp** 命令就可返回到 VLISP。

详细检查反应器的行为

要在应用程序中跟踪反应器事件，您需要一大堆草稿纸（因为要做的事太多了）。下面的例子说明了您应该跟踪哪些东西：

Command: erase Object: Polyline border
Reactor sequence
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

绘制十条花园小路，然后相继选取绘制的多段线，跟踪如下命令 / 对象操作：

■ Erase / 多段线边界（路 1）。
■ Erase / 多段线（路 2）中的某个圆。
■ Erase / 两条多段线（路 3 和路 4）。
■ Move / 多段线边界（路 5）。
■ Move / 多段线（路 6）中的某个圆。
■ Move / 两条多段线（路 7 和路 8）和几个圆。
■ （用夹点）移动多段线顶点 / 多段线边界（路 9）。
■ Stretch / 多段线边界（路 10）。

本练习可让您很好地理解在后台发生了什么事情。在学习第七课的任何部分时，如果您对反应器的功能有些混淆，请参考您记录的反应器跟踪结果。

第六课回顾

本课介绍了 AutoCAD 反应器以及如何用 VLISP 实现反应器。您就如何给花园小路应用程序添加反应器进行了规划，并往程序中添加了一些代码来实现该规划。

您现在已经了解了 AutoLISP 中一些非常新也很让人兴奋的功能。反应器可给应用程序增加许多功能，但请记住，您的应用程序越强大，它们崩溃起来就会越猛烈。

另一件要时刻记住的事情是设计应用程序的方法，反应器功能并不是永久的，它不能被保留到下一个绘图任务。如果您保存的图形中包含能激活反应器的花园小路，那么您下次打开该图形时，该反应器将不再存在。如果您想添加永久性反应器，请参见 Visual LISP 开发人员手册中的“临时反应器和永久反应器”和 AutoLISP 参考中的有关函数。

程序集成

本课内容包括

7

- 规划反应器整体过程
- 添加新的反应器功能
- 重新定义多段线边界
- 代码回顾
- 生成应用程序
- 教程回顾
- LISP 和 AutoLISP 参考书

在第六课，您学习了基于反应器的应用程序的基本机制。在第七课，您将利用这些知识给应用程序添加相应功能，使花园小路应用程序知道在何时该怎样修改自身。在测试了您的应用程序并确认其运行正确之后，您可以用 VLISP 工程创建一个 VLISP 应用程序。

本部分教程讲述的是 VLISP 中的高级主题，如果您是初学者，您可能无法一下子理解本章的所有 AutoLISP 代码。在本课的最后列出了一些书名，它们可提供和本课所讲述的 AutoLISP 高级概念有关的详细信息。

规划反应器整体过程

在本课中需要定义几个新函数。本课没有给出对新代码所有方面的详细介绍，而是在对代码进行整体介绍之后，指出代码所包含的概念。在本课最后，您将完成创建花园小路应用程序所需的所有源代码，这次创建的应用程序将和您在第一课中运行的样例程序完全相同。

注意 在开发和调试基于反应器的应用程序时，AutoCAD 将处于不稳定状态。这是由多种原因引起的，例如没有删除已经被删除的图元上附着的反应器等。所以建议您在开始学习第七课以前，先关闭 VLISP 并保存其中打开的所有文件，然后退出 AutoCAD，再重新启动这两个应用程序。

先请加载第六课最后的工程。

在花园小路应用程序中还有两个明显的工作没有完成：

- 编写对象反应器回调函数。
- 编写编辑器反应器回调函数。

您还需要考虑怎样处理程序中的全局变量。一般情况下，全局变量会在整个 AutoCAD 会话期间保持其值，但对反应器来说，情况并不是这样。为了说明这点，我们假设您花园小路应用程序的用户在某图形中绘制了几条花园小路，然后又将它们删除：先是第一次删除一条，然后一次又删除两条，如此下去，直到最后只剩下一条小路没被删除。

第五课引入了全局变量 *reactorsToRemove*，它负责保存指向要删除的多段线上的反应器的指针。*reactorsToRemove* 是在 **gp:outline-erased** 中声明的，也是该事件通知您多段线要被删除，但在 **gp:command-ended** 事件出现以前，该多段线并没有真正被删除。

用户第一次删除多段线时，程序的确如您所预想的那样运行。在 **gp:outline-erased** 中，您保存了指向反应器的指针。调用 **gp:command-ended** 时，您可以删除和反应器所附着的多段线相关联的砖，所有工作都正常。然后，用户决定删除两条小路。这时，应用程序将调用两次 **gp:outline-erased** 函数：每条被删除的小路都得调用一次本函数。现在您必须先考虑两个潜在的问题：

- 在用 **setq** 设置变量 *reactorsToRemove* 时，必须给全局变量添加一个指向反应器的指针，注意不要覆盖了已保存在那儿的任何数据。这意味着 *reactorsToRemove* 必须是一个表，您才能将反应器指针添加上去。这样不管用户在一个 erase 命令中删除几条小路，您都可以把它们关联的反应器指针全添加到 ***reactorsToRemove*** 中。
- 每次开始新命令序列调用 **gp:command-will-start** 时，您应该将变量 *reactorsToRemove* 初始化为 nil。您必须这样做，才不至于让该全局变量保存上一个 erase 命令的反应器指针。

  如果您不重新初始化该全局变量，或用的数据结构不正确（在本例中正确的数据结构是表），应用程序的执行结果将无法预料。在使用反应器时，无法预料的结果可能会给 AutoCAD 任务造成致命后果。

下面是用户用一个 erase 命令删除两条花园小路时应发生的事件列表，请注意对全局变量的处理：

- 初始化 **erase** 命令。这将触发 **gp:command-will-start** 函数，请将 *reactorsToRemove* 设为 nil。
- 选取两条多段线，您的应用程序现在还不会得到通知。
- 按 ENTER 键删除两条选定的多段线。
  应用程序将为其中一条多段线调用回调函数 **gp:outline-erased**，将其反应器指针添加到为空的全局变量 *reactorsToRemove* 中。
  然后，应用程序将为第二条多段线调用回调函数 **gp:outline-erased**，将其反应器指针添加到全局变量 *reactorsToRemove* 中，而该全局变量中已包含了第一个反应器指针。
- AutoCAD 删除多段线。
- 调用回调函数 **gp:command-ended**，删除与 *reactorsToRemove* 保存的反应器指针相关联的所有砖。

除了全局变量 *reactorsToRemove* 外，您的应用程序中还有一个全局变量 *polyToChange*，它保存着指向要修改的多段线的指针。在本课的后面还会给该应用程序引入另外两个全局变量。

响应更多的用户调用命令

编写基于反应器的应用程序时，您必须处理所有会影响到对象的命令。在规划程序时必须检查所有可能的 AutoCAD 编辑命令，并确定应用程序应该如何响应它们。在第六课结束时提到的“反应器跟踪结果”对此很有用。请调用您预期用户会使用的命令，并写下应用程序应该如何响应这些命令。规划时要考虑的其他问题有

- 确定用户发出 UNDO 和 REDO 命令时程序要如何响应。
- 确定用户在删除带有反应器的图元后调用 OOPS 命令时程序要如何响应。

为了避免使本已很复杂的问题变得更复杂，本教程并不打算考虑所有可能性，而只是实现了其中很少一部分功能。

尽管您不打算实现响应其他命令的功能，另外一小部分编辑命令也要求：

- 如果用户拉伸多段线边界（用 STRETCH 命令），他可以往任意方向拉伸该多段线，而不只限在小路方向或与小路方向垂直的方向。所以边界可能不再保持为矩形。另外，您还需要考虑拉伸多个顶点的情况，因为拉伸多段线的一个顶点和拉伸多段线的两个顶点的结果差别很大。而不论在何种情况下，都必须删除所有的砖，并在确定了怎样调整边界之后重新计算它们的新位置。
- 如果用户移动多段线边界，也应该删除所有的砖，然后再在新位置重新绘制这些砖。这个操作相当简单，因为并没有改变多段线边界的尺寸或形状。
- 如果用户缩放多段线边界，您就必须作出选择：是同时缩放砖，使小路里砖的数目不变；还是保持砖尺寸不变，应用程序根据多段线是放大还是缩小来添加或删除砖。
- 如果用户旋转多段线边界，您就必须删除所有的砖，然后在新的方向上重新绘制它们。

但开始时我们只做如下规划：

- 响应 command-start 事件，警告用户所选编辑命令（如拉伸、移动或旋转等）将对花园小路造成不良影响。
- 如果用户继续进行，删除相应砖但不重新绘制它们。
- 从小路轮廓上删除反应器。

注意 除了用户调用的 AutoCAD 命令之外，AutoLISP 或 ObjectARX 应用程序也可能修改图元。本花园小路教程提供的例子没有考虑其他程序对花园小路多段线边界的操作（如 (**entdel** <polyline entity>) 等操作）。在这种情况下，编辑器反应器事件 :vlr-commandWillStart 和 :vlr-commandEnded 将不会被触发。

将信息保存在反应器对象中

对应用程序，您需要考虑清楚的另一个重要方面是将哪些信息附着到每个多段线图元的对象反应器上。在第六课中，您添加的代码将 **gp_PathData** 所包含的内容（关联表）附着到反应器上。您扩展了 **gp_PathData** 中包含的数据，给关联表增加了一个新的关键字域 100，这个新的子表包含的是指向与该多段线边界相关联的所有圆图元（砖）的指针。

因为我们必须重新计算多段线的边界，所以需要给 **gp_pathData** 加上另外四个关键字域：

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

在用户拖动角点的夹点到新位置时，重新计算多段线边界必须要有这些有序顶点的信息。在 gpdraw.lsp 文件的 **gp:drawOutline** 函数中已存在这些信息。但看一下该函数的返回值就会发现——当前只返回了指向多段线对象的指针。所以您需要做以下三件事情：

- 按要求的格式组合四个角点。
- 修改该函数，让它返回角点表和指向多段线的指针。
- 修改 **C:GPath** 函数，让它能正确处理 **gp:drawOutline** 函数新的返回值格式。

将四个角点按要求的格式组合成表比较容易实现。请看 **gp:drawOutline** 中的代码，局部变量 p1 对应组码 12（所关联的值），p2 对应 13，p3 对应 14，而 p4 对应 15。可以添加如下函数调用来组合这些信息：

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

修改函数使它返回多段线的角点以及指向多段线的指针也比较容易。在 **gp:drawOutline** 函数的最后一个表达式中，将您要返回的两条信息组合成一个表就可以了。

```
(list pline polyPoints)
```

实现程序能保存多段线角点信息的步骤

1 按如下代码中的黑体字所示那样修改 **gp:drawOutline** 函数（请不要忽略 **defun** 语句中添加的局部变量 polyPoints）：

```
(defun gp:drawOutline (BoundaryData     /      PathAngle
          Width     HalfWidth  StartPt   PathLength
          angm90    angp90    p1     p2
          p3     p4       poly2Dpoints
          poly3Dpoints      plineStyle  pline
          polyPoints
          )
 ;; 从表 BoundaryData 中取出数据。
 (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))

  ;; 获取多段线类型。
```

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

    ;; 用 ActiveX Automation 将多段线添加到模型空间。
    pline       (if (= plineStyle "LIGHT")
              ;; 创建优化多段线。
              (vla-addLightweightPolyline
             *ModelSpace*   ; 模型空间的全局定义
             (gp:list->variantArray poly2Dpoints)
                  ; 数据转换
             ) ;_ 结束 vla-addLightweightPolyline
              ;; 或创建普通多段线。
              (vla-addPolyline
             *ModelSpace*
             (gp:list->variantArray poly3Dpoints)
                  ; 数据转换
             ) ;_ 结束 vla-addPolyline
              ) ;_ 结束 if
    polyPoints   (list
              (cons 12 p1)
              (cons 13 p2)
              (cons 14 p3)
              (cons 15 p4)
              )
    ) ;_ 结束 setq
  (vla-put-closed pline T)

  (list pline polyPoints)
) ;_ 结束 defun
```

2 修改 gpmain.lsp 文件中的 **C:GPath** 函数。请找到该函数中的下述代码行：

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

将它修改为：

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

现在变量 gp_PathData 中包括了反应器函数所需的所有信息。

3 将 PolylineList 添加到 **C:GPath** 函数定义的局部变量声明处。

添加新的反应器功能

在第六课中我们已经将回调函数 **gp:command-will-start** 和反应器事件 :vlr-commandWillStart 相关联。目前该函数可显示一些信息并将两个全局变量 *polyToChange* 和 *reactorsToRemove* 初始化为 nil。

给回调函数 gp:command-will-start 添加功能的步骤

1 打开 gpreact.lsp 文件。

2 在 **gp:command-will-start** 函数中给 **setq** 函数调用添加两个变量，将它修改为如下形式：

```
;; 将四个反应器全局变量全部重新设为 nil。
(setq   *lostAssociativity* nil
       *polyToChange* nil
       *reactorsToChange* nil
       *reactorsToRemove* nil)
```

3 用下述代码替换 **gp:command-will-start** 中的原有代码，直到最后一个 **princ** 函数调用：

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

该代码检查用户是否发出了一个命令打断砖和小路之间的联系。如果用户发出了这样一个命令，程序将设置全局变量 *lostAssociativity* 并警告用户。

在测试花园小路应用程序时，您可能会发现，还有一些编辑命令也能修改花园小路，并导致砖和小路失去关联。请将这些命令加到被引号引起来的列表中，这样用户就会得到警告，知道将会发生什么情况。当调用该函数时，用户已开始了某命令，但他还没有选取要修改的图元。用户仍能取消该命令，让图形保持不变。

实现对象反应器的回调函数

在第六课中，您为对象反应器事件注册了两个回调函数：与反应器事件 :vlr-erased 相关联的是 **gp:outline-erased** 函数，而与事件 :vlr-modified 相关联的是 **gp:outline-changed** 函数。您现在要让这些函数能具体实现其功能。

让对象反应器的回调函数能实现其功能的步骤

1 在 gpreact.lsp 文件中，修改 **gp:outline-erased** 函数，使它如下所示：

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

在这里只完成了一个操作：将附着到多段线的反应器保存到一个表中，该表包含了所有要删除的反应器。（记住：虽然反应器是附着到图元上的，但它们完全是单独的对象，我们必须与处理常规 AutoCAD 图元一样小心地处理它们和图元的关系。）

2 修改 **gp:outline-changed** 函数，使它如下所示：

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

共有两类函数能修改多段线轮廓。第一类包括那些能打断小路和砖的关联的命令，您已经在 **gp:command-will-start** 函数中对此进行检查，并根据检查结果相应设置全局变量 *lostAssociativity*。在这种情况下，需要删除小路的所有砖，然后该小路就完全由用户处理了。另一类是 STRETCH 命令的夹点模式，此时小路和砖的关联仍被保留，在用户将多段线的一个顶点拖到新位置后，必须使轮廓仍保持矩形外观。

polyToChange 变量保存指向多段线自身的 VLA 对象指针，在 **gp:command-ended** 函数重新计算多段线边界时要用到该变量。

设计回调函数 gp:command-ended

编辑器反应器回调函数 **gp:command-ended** 是最经常发生动作所在的函数。花园小路的边界多段线一直打开以供修改，直到调用该函数。也就是说，在调用该函数之前，用户可能还在 AutoCAD 中操作小路边界。在反应器序列中，您需要等待 AutoCAD 完成它的那部分工作，然后才能做您自己要做的工作。

下面的伪代码给出了函数 **gp:command-ended** 的思路：

```
对多段线的情况进行判断。
 第一种情况 - 多段线被删除（Erase 命令）
  删除小路的所有砖。
 第二种情况 - 丢失了多段线和砖之间的关联（Move、Rotate 等命令）
  删除小路的所有砖。
 第三种情况 - 通过夹点拉伸多段线
  删除小路的所有砖。
  从多段线获取当前边界数据。
  如果是优化多段线，
   按二维点处理边界数据
  否则
   按三维点处理边界数据
  IF 结束
 重新定义多段线边界（传入的参数包括多段线当前的和以前的结构）。
 获取新边界的信息，将它们组合成所要求的格式并以此设置多段线图元。
 重新生成多段线。
 将修改过的边界信息送回到名为 *reactorsToChange* 的反应器中。
函数结束
```

以上伪代码相对比较简单，但它隐藏了一些很重要的细节，而这些细节您现在可能还不太清楚。

处理多种图元类型

第一个细节是您的应用程序可能绘制了两种多段线：普通多段线和优化多段线。这两种不同的多段线返回的图元数据的格式也不同：普通多段线返回的表中包括 12 个双精度实数——四组 X、Y、Z 坐标；而优化多段线返回的表中只包括 8 个双精度实数——四组 X、Y 坐标。

在用户移动了多段线的某个顶点之后，需要做一些计算工作来修正多段线边界。当然，如果多段线数据的格式统一，计算会更加容易。

在第七课中 utils.lsp 文件包含的函数可完成下述必要的格式转换：**xyzList->ListOfPoints** 将三维点的表转换为以表为元素的表；而 **xyList->ListOfPoints** 则将二维点的表转换为以表为元素的表。

添加代码使多段线数据格式统一的步骤

1 如果您在 VLISP 文本编辑窗口中打开了 *utils.lsp* 文件，则应先将它关闭。

2 将 Tutorial\VisualLISP\Lesson7 目录下的 utils.lsp 文件复制到工作目录。

除了那两个重新设置多段线数据格式的函数之外，utils.lsp 文件还包括了另外一些工具函数，在处理用户修改花园小路时需要用到它们。

3 在 VLISP 文本编辑窗口中打开 utils.lsp 文件，查看新代码。

在反应器回调函数中使用 **ActiveX** 方法

伪代码中的第二个细节出现在末尾，重新绘制砖的地方。伪代码如下所示：

重新绘制砖（强制使用 ActiveX 绘图）

括号里的信息说明了这点：必须强制使用 ActiveX 绘图。为什么要这样要求呢？为什么应用程序不能用关联表子表中保存的对象创建方法来绘图呢？

这是因为在反应器回调函数中不能用 **command** 函数创建图元。这涉及到 AutoCAD 必须完成的一些内部工作。因此必须在绘制砖的子程序中强制使用 ActiveX，本课还会多次在这点上提醒您。

处理非线性的反应器序列

最后一个重要细节是：在用户通过特殊的 GRIP 命令修改多段线时，要处理 AutoCAD 中的命令 / 反应器序列。您在选取了对象的夹点后单击鼠标右键，就可访问这些命令（如 GRIP_MOVE 和 GRIP_ROTATE 等）。这时反应器序列不象简单的 MOVE 或 ERASE 命令那样是线性的。实际上，用户在一个命令执行当中可以又改为执行另一个命令。为了说明这点，可以加载第六课中跟踪反应器事件的代码，或简单地检查一下 VLISP 控制台窗口显示的如下注解，就知道发生了什么事情了：

```
;; 开始请用交叉选择框选定多段线和一些圆（砖），
;; AutoCAD 将会显示选择集中条目（所选多段线和圆）的夹点。
;; 要开始其序列，只需单击多段线的某个夹点：

(GP:COMMAND-WILL-START #<VLR-Command-reactor> (GRIP_STRETCH))

;; 现在将命令改为移动：单击鼠标右键
;; 并从弹出式菜单中选择 MOVE。注意为关闭
;; GRIP_STRETCH 命令而出现的 command-ended
;; 反应器，但它并没有触发对象反应器事件：

(GP:COMMAND-ENDED #<VLR-Command-reactor> (GRIP_STRETCH))
(GP:COMMAND-WILL-START #<VLR-Command-reactor> (GRIP_MOVE))

;; 现在将轮廓（和所选圆）拖到某新位置。

(GP:OUTLINE-CHANGED #<VLA-OBJECT IAcadLWPolyline 028f3188>
        #<VLR-Object-reactor> nil)
(GP:COMMAND-ENDED #<VLR-Command-reactor> (GRIP_MOVE))
```

这说明不能肯定在所有情况下都会调用您的对象反应器回调函数。

在该序列中还有一个有关的问题，就是即使在最后的 command-ended 回调函数中，我们也无法删除夹点选择集中的那部分圆，因为 AutoCAD 仍打开了这些圆。如果您试图在 command-ended 回调函数中删除它们，则会导致 AutoCAD 崩溃。为了避免出现这种情况，还需要使用另一个全局变量，它专门保存指向砖（圆）对象的指针所组成的表，直到能删除这些砖为止。

处理非线性反应器序列的步骤

1 将下述函数添加到 gpreact.lsp 文件：

```
(defun gp:erase-tiles (reactor / reactorData tiles tile)
  (if (setq reactorData (vlr-data reactor))
    (progn
      ;; 小路的砖的信息被保存在反应器的数据中。
      (setq tiles (cdr (assoc 100 reactorData)))
      ;; 删除小路中现有的所有砖。
      (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)
      )
    )
  )
```

在删除砖的子程序的开始部分要用到这个新函数。注意这时并没有真正删除砖：它们只是被设为不可见，并被添加到名为 *Safe-to-Delete* 的全局变量中。

2 将下述函数添加到文件 gpreact.lsp 中：

```
(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)
    )
  )
 )
```

没有执行 GRIP_MOVE 或 GRIP_STRETCH 命令时，可调用该函数。

编写 command-ended 函数

您已看到了伪代码，并知道如何处理其中的一些重要细节，现在您就可以用如下代码替换您以前的反应器回调函数 **gp:command-ended** 的简单空函数版本了：

```
(defun gp:command-ended (reactor      command-list
          /       objReactor
          reactorToChange  reactorData
          coordinateValues currentPoints
          newReactorData   newPts
          tileList
          )
 (cond

   ;; 第一种情况 - 多段线被删除（Erase 命令）
   ;; 如果正删除一个或多个多段线边界
   ;; （由 *reactorsToRemove* 变量标明，
   ;; 则删除边界内的砖，然后删除反应器。
   (*reactorsToRemove*
    (foreach objReactor *reactorsToRemove*
      (gp:erase-tiles objReactor)
     )
    (setq *reactorsToRemove* nil)
    )
```

```
;; 第二种情况 - 丢失了多段线和砖之间的关联（Move、Rotate 等命令）
;; 如果丢失了多段线和砖之间的关联（Move、Undo 等命令），
;; 则删除每个边界内的砖
;;
((and *lostassociativity* *reactorsToChange*)
 (foreach reactorToChange *reactorsToChange*
   (gp:erase-tiles reactorToChange)
   )
 (setq *reactorsToChange* nil)
 )

;; 第三种情况 - GRIP_STRETCH
;; 在这种情况下，砖和小路之间仍保持关联，
;; 但必须重新计算和绘制小路与砖。
;; GRIP_STRETCH 命令一次只能操作一条多段线。
((and (not *lostassociativity*)
  *polytochange*
  *reactorsToChange*
  (member "GRIP_STRETCH" command-list)
  ;; 对于一个 GRIP_STRETCH 命令，全局变量
  ;; *reactorsToChange* 中只有一个反应器。
  (setq reactorData
     (vlr-data (setq reactorToChange
            (car *reactorsToChange*)
           )
         )
   )
  )

 ;; 首先删除多段线边界里的砖。
 (gp:erase-tiles reactorToChange)
 ;; 接着获取多段线顶点的当前坐标值。
 (setq coordinateValues
   (vlax-safearray->list
     (vlax-variant-value
   (vla-get-coordinates *polyToChange*)
   )
     )
   )

 ;; 如果轮廓是优化多段线，您获取的是二维点，
 ;; 所以要用工具函数 xyList->ListOfPoints
 ;; 将坐标数据转换为由点组成的形如 ((x y) (x y) ...) 的表。
 ;; 对普通多段线，则用 xyzList->ListOfPoints 函数
 ;; 处理三维点，将坐标数据转换为由点组成的
 ;; 形如 ((x y z) (x y z) ... ) 的表。
 (setq CurrentPoints
   (if (= (vla-get-ObjectName *polytochange*) "AcDbPolyline")
     (xyList->ListOfPoints coordinateValues)
     (xyzList->ListOfPoints coordinateValues)
     )
   )
```

```
;; 将这些新信息传给 RedefinePolyBorder，
;;  它将返回新多段线边界
(setq NewReactorData
  (gp:RedefinePolyBorder CurrentPoints reactorData)
 )
;; 获取所有的边界点，并 ...
(setq newpts (list (cdr (assoc 12 NewReactorData))
      (cdr (assoc 13 NewReactorData))
      (cdr (assoc 14 NewReactorData))
      (cdr (assoc 15 NewReactorData))
      )
 )

;; ... 用上面算出的新坐标点更新多段线轮廓。
;; 如果处理的是优化多段线，则将这些点转换为
;; 二维点（因为 newpts 中的所有点都是三维点），
;; 如处理的是普通多段线，则不用处理这些点。
(if (= (cdr (assoc 4 NewReactorData)) "LIGHT")
 (setq newpts (mapcar '(lambda (point)
           (3dPoint->2dPoint Point)
           )
         newpts
         )
   )
 )

;; 现在用正确的坐标点更新多段线。
(vla-put-coordinates
 *polytochange*
;; 关于 list->variantArray 的说明，请参见 utils.lsp 文件。
 (gp:list->variantArray (apply 'append newpts))
)

;; 现在可使用 NewReactorData 的当前定义，
;; 它和花园小路的数据结构几乎是一样的，
;; 唯一的不同点是包含砖表的域 (100) 的值为 nil。
;; 不过这没有关系，因为 gp:Calculate-and-Draw-Tiles
;; 函数绘制砖时并不需要该数据。事实上，
;; 该函数将创建砖并返回所绘制砖的表。
(setq tileList (gp:Calculate-and-Draw-Tiles
        ;; 小路数据表，但不包括砖表
        NewReactorData
        ;; 对象创建函数
        ;; 由于是在反应器内，所以必须用 ActiveX 方法
        "ActiveX"
        )
 )
```

```
    ;; 既然您已经获取了所有绘制的砖，
    ;; 那么可以利用正确的 tileList 重新构造数据结构，
    ;; 并将正确的数据重新设置到反应器中，
    ;; 更新和多段线边界相关联的砖。
    (setq NewReactorData
      (subst (cons 100 tileList)
        (assoc 100 NewReactorData)
        NewReactorData
        )
     )

    ;; 现在您有了和多段线关联的新数据。
    ;; 剩下的工作就是用 vlr-data-set 函数
    ;; 将它和反应器相关联。
    (vlr-data-set (car *reactorsToChange*) NewReactorData)

    ;; 清除所有对临时变量 *polytochange* 和
    ;; *reactorsToChange* 的引用
    (setq *polytochange*     nil
     *reactorsToChange* nil
     )
    )
          )
  ;; 尽可能删除全局变量 *Safe-to-Delete* 中的所有条目！！
  (Gp:Safe-Delete (car command-list))
  (princ)
  )
```

更新 gp:Calculate-and-Draw-Tiles

在本课前面部分曾提到，当从反应器回调函数中调用 **gp:Calculate-and-Draw-Tiles** 函数时，您必须强制使用 ActiveX 来创建对象。这意味着，如果必要，您需要覆盖用户所选的对象创建方式（ActiveX、**entmake** 或 **command**）。刚才更新的 **gp:command-ended** 函数中的代码包括对绘制砖子程序的调用如下：

```
(setq tileList (gp:Calculate-and-Draw-Tiles
   ;; 小路数据表，但不包括砖表
   NewReactorData
   ;; 对象创建函数
   ;; 由于是在反应器内，所以必须用 ActiveX 方法
   "ActiveX"
  )
)
```

传给函数 **gp:Calculate-and-Draw-Tiles** 的参数有两个：NewReactorData（它包含的表的格式和最初 gp_PathData 中的关联表格式一样）和字符串 "ActiveX"（它将设置对象创建方式）。但请注意一下 **gp:Calculate-and-Draw-Tiles** 的当前定义（如果您忘了，可在 gpdraw.lsp 文件中找到该函数的定义），下面是该函数的参数和局部变量的声明：

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

请注意当前只定义了一个参数，而 ObjectCreationStyle 被声明为局部变量，看一下函数中的下述语句，看看它是怎样设置变量 ObjectCreationStyle 的：

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

ObjectCreationStyle 变量在函数内部被设置为由变量 BoundaryData（关联表）取出的数据，但现在必须能够覆盖该值。

修改 gp:Calculate-and-Draw-Tiles 让它接受对象创建方式参数的步骤

1 将变量 ObjectCreationStyle 添加到函数参数表中。

2 从局部变量表中删除 ObjectCreationStyle。

这时该函数的 **defun** 语句应如下所示：

```
(defun gp:Calculate-and-Draw-Tiles (BoundaryData
                ObjectCreationStyle
  / PathLength     TileSpace
    TileRadius     SpaceFilled
    SpaceToFile    RowSpacing
    offsetFromCenter   rowStartPoint
    pathWidth   pathAngle
    TileList)       ; 从局部变量表中删除 ObjectCreationStyle
```

请注意如果将一个变量既声明为参数（在斜线前面），又声明为局部变量（在斜线后面），那么 VLISP 会给您指出这点。例如，如果您把 ObjectCreationStyle 同时声明为参数和变量，然后用 VLISP 语法检查工具检查函数 **gp:Calculate-and-Draw-Tiles**，那么在“编译输出”窗口将出现如下消息：

```
; *** 警告：参数列表中 / 前后有相同符号 OBJECTCREATIONSTYLE
```

3 修改 **gp:Calculate-and-Draw-Tiles** 函数中的第一个 **setq** 表达式，使它如下所示（已用黑体字显示要作修改的部分）：

```
(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))
) ;_ 结束 setq
 (if (not ObjectCreationStyle)
  (setq ObjectCreationStyle (strcase (cdr (assoc 3 BoundaryData))))
)
```

这样就删除了最初给 ObjectCreationStyle 赋值的语句。该代码现在检查是否给 ObjectCreationStyle 提供了值，如果没提供（也就是说该值为 nil），那么函数将用变量 BoundaryData 中的相应值设置 ObjectCreationStyle。

现在您还需要对与 **gp:Calculate-and-Draw-Tiles** 相关的其他代码进行修改。

修改其他调用 gp:Calculate-and-Draw-Tiles 函数的代码

在反应器回调函数中调用 **gp:Calculate-and-Draw-Tiles** 函数时，传给参数 ObjectCreationStyle 的是字符串 "ActiveX"，那么在其他代码中要怎样调用函数 **gp:Calculate-and-Draw-Tiles** 呢？

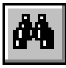
在第四课中我们曾提到，在修改简单空函数时，必须考虑如下问题：

- 是否修改了 **defun** 语句？也就是说，函数的参数数目是否仍相同？
- 函数的返回值是否有了不同？

在构建、完善和更新应用程序时，如果需要对函数进行重大修改，就应该考虑这两个问题。现在，您需要在工程中查找其他调用了 **gp:Calculate-and-Draw-Tiles** 的函数。VLISP 有个功能可帮您做到这点。

查找您工程中所有调用函数 gp:Calculate-and-Draw-Tiles 处的步骤

1 在 VLISP 文本编辑窗口中，双击 gpdraw.lsp 文件中的词 **gp:Calculate-and-Draw-Tiles**。



2 从 VLISP 菜单中选择“搜索”➢“查找”。

由于您预先选择了该函数名，因此它将自动出现在指定待查找字符串的文本框中。

3 在“查找”对话框的“搜索”一栏中选择“工程”按钮。

当您选择该选项时，“查找”对话框会展开其底部，让您选择要搜索的工程。

4 指定您当前的工程名，然后单击“查找”按钮。

VLISP 将在“查找输出”窗口显示搜索结果：

```
<查找输出>
C:/Program Files/ACAD2000/TUTORIAL/VisualLISP/lesson6/GPDRAW.lsp
        ;;;              drawing function gp:Calculate-and-Draw-Tiles.
        ;;;      Function: gp:Calculate-and-Draw-Tiles
        (defun gp:Calculate-and-Draw-Tiles (BoundaryData     /
C:/Program Files/ACAD2000/TUTORIAL/VisualLISP/lesson6/GPMAIN.lsp
                    (setq tileList (gp:Calculate-and-Draw-Tiles gp_PathData))
                ;; gp:Calculate-and-Draw-Tiles) to the gp_PathData variable.
5 个已发现
```

5 查看“查找输出”窗口中的搜索结果，确定您的代码是否还在其他位置调用了 **gp:Calculate-and-Draw-Tiles**。在本例中，应该只有一处（在 gpmain.lsp 文件中）调用了该函数。

6 在“查找输出”窗口中双击调用 **gp:Calculate-and-Draw-Tiles** 函数的代码行。

VLISP 将激活一个文本编辑窗口，并将您带到 gpmain.lsp 文件的这行代码处，这行代码现在如下所示：

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

7 将这行代码替换为：

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

为什么要加上 nil？再看一下伪代码就知道原因了：

如果 ObjectCreationStyle 为 nil，用变量 BoundaryData 中的相应值设置 ObjectCreationStyle。

把 nil 作为参数传给 **gp:Calculate-and-Draw-Tiles** 函数，将让该函数检查用户选择的绘制砖的方式（由对话框中的输入决定，它保存在变量 **gp_PathData** 中）。而在 command-ended 反应器回调函数中调用该函数时，将强制使用 ActiveX 方法，从而覆盖该选择。

现在可以向您表示祝贺了，因为您已实现了基本的反应器功能。如果您愿意，可将文件 gpmain.lsp 和 gpdraw.lsp 从 Tutorial\VisualLISP\Lesson7 目录复制到工作目录下，还可以测试一下这些完整的已调试过的代码。

然而，现在还只能是短暂庆祝一下，因为还有许多工作要做。这些都是由 **gp:Command-ended** 函数中的下述代码引起的：

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

重新定义多段线边界

学习到这儿时，您脑子里可能已经充满了新的概念、术语、命令等。所以，建议您复制本教程提供的代码样例，而不是自己键入这些代码。

复制用于重新定义多段线边界的代码的步骤

1. 将 Tutorial\VisualLISP\Lesson7 目录下的 gppoly.lsp 文件复制到工作目录下。
2. 在您工程的工程窗口中，单击“工程特性”按钮。
3. 将 gppoly.lsp 文件添加到工程中。
4. 单击“确定”按钮让工程包含该文件。
5. 在工程窗口中，双击 gppoly.lsp 文件以打开该文件。

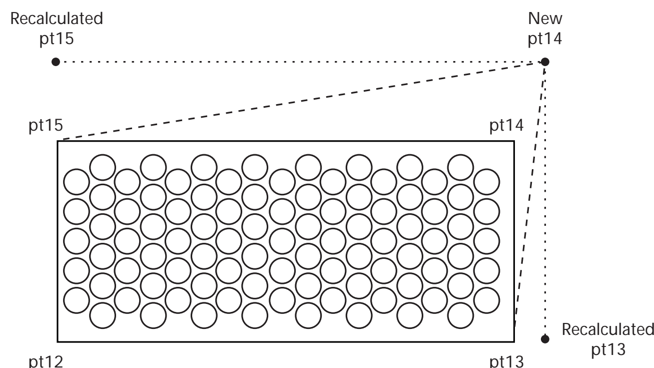
查看 **gppoly.lsp** 文件中的函数

gppoly.lsp 文件中包含了许多函数，在拉伸了多段线的一个夹点后使多段线重新恢复矩形外观时，需要用到这些函数。由于篇幅原因，本教程只对这些函数中的一部分进行详细解释。

注意 花园小路教程的这个部分包括了全课中最复杂的代码和概念。如果您是初学者，不妨先跳过本节，直接学习本章后面的“生成应用程序”一节。

您可能会发现，gppoly.lsp 文件组织函数的方式和其他 AutoLISP 源代码文件相同：高层函数（经常是主函数或 **C:** 函数，如本例中的 **gp:Redefine-PolyBorder** 函数）位于文件的尾部；而在主函数中调用的函数则定义在源文件的前面部分。这种习惯起源于过去，因为那时有些开发环境要求这样组织函数。在 VLISP 中，这只是个人风格问题，它对函数顺序没有任何要求。

在深入讨论细节之前，先回过头来看一下要怎样重新计算和绘制花园小路边界。下面的说明展示了一个花园小路的例子，并给出了反应器数据中保存的关联表中的点：



在本例中，组码 12 对应的点是左下角，13 对应的是右下角，由此类推。如果用户移动的是右上角（组码 14 对应的点），程序需要重新计算右下角 (13) 和左上角 (15) 两个点的位置。

理解 gp:RedefinePolyBorder 函数

以下伪代码给出了主函数 **gp:RedefinePolyBorder** 的逻辑思路：

```
函数 gp:RedefinePolyBorder
  取出以前的多段线角点坐标（与组码 12、13、14、15 对应的点）。
  比较当前多段线角点和以前多段线角点，找到移动的是哪个角点（不
    匹配的那个角点就是移动的那个角点）。
  根据移动的角点重新计算相应的那两个点的坐标。
  将反应器数据中的角点坐标更新（它们将被保存回修改过的多段线的
    反应器中）。
  更新反应器数据中的其他信息。（小路的起点、端点、宽度和长度都必
    须重新计算。）
```

理解 gp:FindMovedPoint 函数

gp:FindMovedPoint 函数中包含了一些功能非常强大的 LISP 表达式，它们用于处理表。实际上本函数完成的工作是比较（用户将某角点拖到新位置后的）当前多段线角点和以前多段线的角点，并以关联表形式 (13 <xvalue> <yvalue>) 返回被移动的角点：

理解该函数的最好办法是跟踪其代码并监视它处理的变量的值的变化。在该函数的第一个表达式处 (setq result . . .) 设置一个断点，然后在您单步执行该函数时监视如下变量：

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

mapcar 和 **lambda** 函数将在随后的章节中给予解释。现在请先阅读代码中的注释，看看能否理解该函数中代码的作用。

理解 gp:FindPointInList 函数

源代码中的函数头解释了 **gp:FindPointInList** 函数是如何转换它所处理的信息的。与前面所讲的 **Gp:FindMovedPoint** 函数类似，该函数也利用了 LISP 的表处理能力。在处理表时，您经常会看到一起使用的 mapcar 和 lambda 函数。您第一次看到它们时，会觉得这两个函数很奇怪，而且还容易混淆，因为它们的函数名并没有说明它们所实现的功能。然而，一旦掌握了这两个函数，您将会发现它们是 AutoLISP 指令系统中两个最有用的函数。下面是对 **mapcar** 和 **lambda** 函数的简单介绍。

mapcar 函数将表达式应用到表的每一个条目上。例如，假设给定的表中包括整数 1、2、3、4，**mapcar** 可将 **1+** 函数应用到表中每一个数字上，让它们各自加 1：

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

mapcar 函数最初的定义是将函数（第一个参数）应用到表（第二个参数）的每一个条目。**mapcar** 操作的返回值是应用了函数或表达式后转换成的表。（事实上，**mapcar** 能完成的功能比这多得多，但现在这个定义已经足够了。）

在所提供的例子中，**mapcar** 将表 '(1 2 3 4) 中的每个值传给了 **1+** 函数。事实上，**mapcar** 完成了如下操作，并将结果组合成一个表：

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

这里还有一个 **mapcar** 的例子，这次是用 **null** 函数测试表中的值是否为空（或非真）值：

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

该代码实际上进行了如下操作：

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

在 **mapcar** 中可以使用许多 AutoLISP 函数，也能使用您自己的函数。例如，假设您已创建了一个功能非常强大的函数 **equals2**：

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

当然，**equals2** 功能并不强大，但正是在这种情况下用 **lambda** 更方便：在您不想也不必小题大做定义一个函数时，可以使用 **lambda** 函数。有些时候，您可把 **lambda** 定义看成匿名函数。例如，您不用定义名为 **equals2** 的函数，而是直接用下述 **lambda** 表达式完成同样的操作（以省掉函数定义的麻烦）：

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

该代码完成的操作如下：

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

了解了这两个函数之后，再看看您能否看懂 **gp:FindPointInList** 函数，再次提醒您，要仔细看看源代码中的注释。

理解 gp:recalcPolyCorners 函数

理解 **gp:recalcPolyCorners** 函数代码的关键是要仔细体会图中所示组码 12 到 15 所对应的值表示的点：

在该图中，用户移动的角点与组码 14 相关联。这意味着需要重新计算与 13 和 15 相关联的两个角点。

必须沿着由点 12 和点 15 当前值定义的方向移动点 15，直到它满足如下条件：从点 12 到点 15 的矢量和从点 14 到点 15 的矢量相互垂直。重新计算点 13 的新位置时也要采用同样的方法。

现在再看看您是否能理解该函数的代码。

理解 gp:pointEqual、gp:rtos2 和 gp:zeroSmallNum 函数

这三个函数要求您了解 AutoCAD 系统中的一个编程技巧，那就是双精度实数的精度问题。有时候由于浮点数的四舍五入，代表几何位置的数字并不是非常精确。而您必须将一些点与另一些点作比较，所以您必须处理这种情况。

不知您是否已注意到，在列出 AutoCAD 图元的信息时，有时会出现形如 1.0e-017 的值。该数字和 0 非常接近，但如果您在 LISP 程序中将它与 0 作比较，一般都认为它们不相等。

在花园小路应用程序中，您比较数字时不必担心 1.0e-017 和 0 的差别。**gp:pointEqual**、**gp:rtos2** 和 **gp:zeroSmallNum** 函数在比较点表时可处理任何由四舍五入引起的误差。

这样您就已经基本理解了 gppoly.lsp 文件中的所有函数。

代码回顾

到现在为止，您在本课中已完成了下述工作：

- 修改了 **gp:drawOutline** 函数，使它在返回指向多段线的指针的同时还返回多段线的角点。您还将该信息添加到变量 gp_PathData 中，该变量被保存在附着到每条花园小路多段线的对象反应器的反应器数据中。
- 更新了 gpreact.lsp 文件中的反应器函数。
- 在 utils.lsp 文件中添加了 **xyzList->ListOfPoints**、**xyList->ListOfPoints** 以及其他工具函数。
- 更新了 **gp:Calculate-and-Draw-Tiles** 函数，使 ObjectCreationStyle 现在成为该函数的一个参数，而以前它是该函数的局部变量。
- 修改了 gpmain.lsp 文件的 **C:GPath** 函数中对 **gp:Calculate-and-Draw-Tiles** 的调用。
- 向工程中添加了 gppoly.lsp 文件，并查看了其中的函数。

现在您可以试着运行完整的应用程序。保存您的工作，然后加载工程源代码并运行 **Gpath** 函数，然后可以试着拉伸并移动花园小路的边界。记住，如果运行不正确而您又无法调试该问题，您可加载 Tutorial\VisualLISP\Lesson7 目录下的完整代码。

生成应用程序

本教程的最后一个工作是完成花园小路应用程序的所有代码，并将它们生成独立的应用程序。这样，就能将它作为一个单独的可执行文件发布给任何用户或顾客。幸运的是，最后这些任务可能是整个教程中最容易的部分，因为实际上 VLISP 替您做了所有的工作。

注意 建议您在确认代码能正确工作后再将它们生成应用程序。请您先用源代码文件测试应用程序，在对测试结果满意时再进行生成工作。

使用“生成应用程序”向导

为了帮您创建独立应用程序，VLISP 提供了“生成应用程序”向导。

运行“生成应用程序”向导的步骤

1. 从 VLISP 菜单中选择“文件”➢“生成应用程序”➢“新建应用程序向导”，启动该向导。
2. 选择“专家”模式，然后单击“下一步”。

   向导提示您指定创建应用程序所需的源文件所在的目录，并提示您为应用程序命名。“生成应用程序”向导将生成两个输出文件：一个 .vlx 文件包含您的可执行程序，一个 .prv 文件包含给“生成应用程序”向导指定的选项。该 .prv 文件也被称为*生成*文件。在需要时，您能利用生成文件重新生成应用程序。
3. 请将工作目录 Tutorial\VisualLISP\MyPath 指定为应用程序的位置，并将应用程序命名为 **gardenpath**。VLISP 将应用程序名称用作输出文件的名称（在本例中是 gardenpath.vlx 和 gardenpath.prv）。

   单击“下一步”继续。
4. 本教程不详细讲解应用程序选项，现在您只需接受缺省值，单击“下一步”按钮即可。（关于独立名称空间应用程序的详细信息，请参见 *Visual LISP 开发人员手册*中的“运行应用程序于自身的名称空间中”。）
5. 在本步中，向导提示您指定组成应用程序的所有 AutoLISP 源代码文件。您可以个别地选择 LISP 源文件，但还有一个更简便的方法：从文件类型列表框的下拉列表中选择“Visual LISP 工程文件”，然后单击“添加”按钮，选择 Gpath 工程文件后单击“打开”按钮。

注意 根据您学习教程的不同情况，可能会显示好几个 Gpath 工程文件，这种情况下您应该选择最近创建的那个文件。如果您从第七课复制了所有源代码文件，要选择的工程名应该是 Gpath7.prj。

选择了工程文件后，单击“下一步”以继续执行。

6 编译过的 VLX 应用程序的一个优点是能将对话框控制文件 (.dcl) 也编译到应用程序中，这将减少最终用户要处理的文件数目，并且不会发生加载 DCL 文件时可能出现的搜索路径问题。

从文件类型列表框的下拉列表中选择“DCL 文件”，然后单击“添加”按钮，选择 gpdialog.dcl 文件后单击“打开”按钮。

单击“下一步”以继续编译该应用程序。

7 本教程不详细讲解编译选项。现在您只需接受缺省值，单击“下一步”即可。（关于编译选项的详细信息，请参见 Visual LISP 开发人员手册中的“优化应用程序代码”。）

8 最后一步是检查您所做的选择。此时您可以选择“完成”。VLISP 将开始编译过程，并在“编译输出”窗口显示编译结果。这时将产生几个临时文件，您的每个源代码文件也将被编译为能链接到单个 VLX 应用程序中的格式。

完成这些操作后，您将得到一个名为 *gardenpath.vlx* 的可执行文件。您可进行如下操作以测试该应用程序：

- 从 AutoCAD 的“工具”菜单上，选择“加载应用程序”。
- 加载 Tutorial\VisualLISP\MyPath 目录下刚创建的 gardenpath.vlx 应用程序。
- 运行 **gpath** 命令。

教程回顾

您终于走到了“花园小路”教程的尽头！正如您所看到的，本教程包括了许多内容：既介绍了许多 AutoLISP 概念，也说明了许多 VLISP 操作。再次设计“花园小路”应用程序是为了给您提供有关许多主题和概念的实例。您可能还想进一步学习 AutoLISP 和 LISP，下面是一些常用的关于 LISP 和 AutoLISP 的书。

LISP 和 AutoLISP 参考书

AutoLISP 参考书

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.

普通 LISP 参考书

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.