

Curso Práctico de Personalización Y Programación Bajo AutoCAD



por Jonathan Préstamo Rodríguez
(Delineante Industrial y de Edificios y
Técnico Superior en CAD)

Índice

PARTE PRIMERA

MÓDULO UNO: Personalización de menús

UNO.1. INTRODUCCIÓN	0
UNO.2. EL ARCHIVO ACAD.MNU	0
UNO.2.1. Las secciones ***BUTTONS y ***AUX	1
UNO.2.2. Menús desplegables (secciones ***POP)	1
UNO.2.3. Las barras de herramientas (la sección ***TOOLBARS)	5
UNO.2.4. La sección ***IMAGE de menús de imágenes	8
UNO.2.5. El arcaico menú de pantalla de AutoCAD bajo la sección ***SCREEN	9
UNO.2.6. Configuración del tablero digitalizador bajo las secciones ***TABLET	11
UNO.2.7. ***HELPSTRINGS; las cadenas de ayuda	12
UNO.2.8. Teclas rápidas bajo ***ACCELERATORS	13
UNO.3. SUBMENÚS	14
UNO.3.1. Referencias a submenús	15
UNO.3.1.1. Llamadas a los submenús de las distintas secciones	16
UNO.4. CARACTERES ESPECIALES, DE CONTROL Y OTROS MECANISMOS	19
UNO.4.1. Caracteres especiales	19
UNO.4.2. Caracteres de control	21
UNO.4.3. Otros mecanismos y apuntes	22
UNO.4.3.1. Opciones de menú mediante DIESEL	22
UNO.4.3.2. Variable MENUCTL	23
UNO.4.3.3. Creación y uso de menús de macros	23
UNO.4.3.3.1. Funcionamiento de MC.EXE	26
UNO.4.4. Uso de la orden externa EDIT	26
UNO.5. CARGA y DESCARGA DE MENÚS EN AutoCAD	26
UNO.6. EJEMPLOS PRÁCTICOS DE MENÚS	29
UNO.6.1. Menú desplegable simple	29
UNO.6.2. Menús desplegables	30
UNO.6.3. Menú de imagen y desplegable	31
UNO.6.4. Menú completo de cartografía	33
UNO.FIN. EJERCICIOS PROPUESTOS	35

MÓDULO DOS: Personalización de barras de herramientas desde AutoCAD

DOS.1. INTRODUCCIÓN	37
DOS.2. EL PRIMER ACERCAMIENTO	37
DOS.3. NUESTRA BARRA DE HERRAMIENTAS	38
DOS.3.1. Añadiendo botones a la barra	39
DOS.3.2. Añadiendo un botón vacío	39
DOS.3.3. Editar el icono del botón	40
DOS.4. BOTONES DESPLEGABLES	41
DOS.5. COPIA Y DESPLAZAMIENTO DE BOTONES	42
DOS.6. COMPROBANDO EL .MNS	43
DOS.7. EJEMPLOS PRÁCTICOS DE BOTONES	44
DOS.7.1. Insertar DWG's en el 0,0	44
DOS.7.2. Matriz de pentágonos	44
DOS.7.3. Inserción de formatos desplegable	44
DOS.FIN. EJERCICIOS PROPUESTOS	45
EJERCICIOS RESUELTOS DEL MÓDULO UNO	46

MÓDULO TRES: Creación de tipos de línea

Curso Práctico de Personalización y Programación bajo AutoCAD
Índice

TRES.1. TIPOS DE LÍNEA EN AutoCAD	51
TRES.2. PODEMOS CREAR O PERSONALIZAR UN .LIN	51
TRES.2.1. Examinando el ACADISO.LIN	51
TRES.2.2. Sintaxis de personalización	52
TRES.2.2.1. Creación desde un editor ASCII	52
TRES.2.2.2. Tipos de línea complejos	54
TRES.2.2.3. Creación desde la línea de comandos	58
TRES.3. CARGAR TIPOS DE LÍNEA CREADOS	59
TRES.4. EJEMPLOS PRÁCTICOS DE TIPOS DE LÍNEA	60
TRES.4.1. Tipo simple 1	60
TRES.4.2. Tipo simple 2	60
TRES.4.3. Tipo complejo 1	60
TRES.4.4. Tipo complejo 2	60
TRES.4.5. Tipo complejo 3	61
TRES.FIN. EJERCICIOS PROPUESTOS	61
EJERCICIOS RESUELTOS DEL MÓDULO DOS	61

MÓDULO CUATRO: Creación de patrones de sombreado

CUATRO.1. PATRONES DE SOMBREADO	65
CUATRO.2. SINTAXIS DE DEFINICIÓN	65
CUATRO.3. TÉCNICA DE CREACIÓN	69
CUATRO.4. UTILIZANDO EL PATRÓN DEFINIDO	70
CUATRO.4.1. Iconos del menú de imagen	70
CUATRO.5. EJEMPLOS PRÁCTICOS DE PATRONES DE SOMBREADO	70
CUATRO.5.1. Patrón sólido	71
CUATRO.5.2. Patrón simple inclinado	71
CUATRO.5.3. Patrón de líneas cruzadas a 90 grados	71
CUATRO.5.4. Patrón de hexágonos	71
CUATRO.FIN. EJERCICIOS PROPUESTOS	71
EJERCICIOS RESUELTOS DEL MÓDULO TRES	72

MÓDULO CINCO: Definición de formas y tipos de letra

CINCO.1. INTRODUCCIÓN	73
CINCO.2. ARCHIVOS DE FORMAS PROPIOS	74
CINCO.2.1. Sintaxis de creación de formas	75
CINCO.2.2. Cómo cargar e insertar formas	77
CINCO.2.3. Compilando el fichero .SHP	78
CINCO.2.4. Códigos especiales	79
CINCO.3. ARCHIVOS DE TIPOS DE LETRA	84
CINCO.3.1. Utilizar los tipos de letra creados	87
CINCO.3.2. Tipos de letra <i>Unicode</i>	87
CINCO.3.3. Tipos de letra grande y grande extendido	88
CINCO.3.3.1. Utilizar estos tipos de letra grande	92
CINCO.3.4. Soporte <i>PostScript</i>	93
CINCO.4. EJEMPLOS PRÁCTICOS DE FORMAS Y TIPOS DE LETRA	94
CINCO.4.1. Cuadrado con diagonales	94
CINCO.4.2. Subforma anterior y triángulos	94
CINCO.4.3. Número ocho simple	94
CINCO.4.4. Letra G mayúscula románica	94
CINCO.4.5. Letra n minúscula gótica	94
CINCO.FIN. EJERCICIOS PROPUESTOS	94
EJERCICIOS RESUELTOS DEL MÓDULO CUATRO	95

MÓDULO SEIS: Creación de archivos de ayuda

SEIS.1. INTRODUCCIÓN A LOS ARCHIVOS DE AYUDA	97
SEIS.2. LA AYUDA DE AutoCAD. FORMATO .AHP	97
SEIS.2.1. Visualización del archivo en AutoCAD	98
SEIS.2.2. Introduciendo más temas	100
SEIS.2.3. Retornos suaves, tabulaciones y sangrías	103
SEIS.2.4. Vínculos de hipertexto	105

Curso Práctico de Personalización y Programación bajo AutoCAD
Índice

SEIS.2.5. Archivos de ayuda en directorios de sólo lectura	106
SEIS.3. FORMATO WINDOWS. ARCHIVOS .HLP	107
SEIS.3.1. Microsoft Help Workshop	108
SEIS.3.2. Añadiendo temas a la pestaña <i>Contenido</i>	108
SEIS.3.3. Añadiendo temas a <i>Índice</i> y <i>Buscar</i>	109
SEIS.3.4. Utilizar los archivos .HLP desde AutoCAD	110
SEIS.4. AYUDA EN FORMATO HTML	110
SEIS.4.1. Añadiendo temas HTML a <i>Contenido</i>	111
SEIS.5. EJEMPLOS PRÁCTICOS DE ARCHIVOS DE AYUDA	111
SEIS.5.1. Ayuda a nuevos comandos creados	111
SEIS.5.2. Documentación sobre un comando nuevo	112
SEIS.FIN. EJERCICIOS PROPUESTOS	113
EJERCICIOS RESUELTOS DEL MÓDULO CINCO	113

MÓDULO SIETE: Creación de órdenes externas, redefinición y abreviaturas a comandos

SIETE.1. INTRODUCCIÓN	116
SIETE.2. DEFINICIÓN DE COMANDOS EXTERNOS	116
SIETE.2.1. Comandos externos a nivel MS-DOS	117
SIETE.2.2. Reiniciar el archivo ACAD.PGP	119
SIETE.2.3. Comandos externos Windows	119
SIETE.2.4. Los comandos de Windows START y CMD	119
SIETE.2.5. Visto lo visto nada funciona	121
SIETE.3. ABREVIATURAS A COMANDOS	121
SIETE.4. REDEFINICIÓN DE COMANDOS DE AutoCAD	122
SIETE.5. EJEMPLOS PRÁCTICOS DE COMANDOS EXTERNOS Y ABREVIATURAS	123
SIETE.5.1. Comandos externos MS-DOS	123
SIETE.5.2. Comandos externos Windows	124
SIETE.FIN. EJERCICIOS PROPUESTOS	124
EJERCICIOS RESUELTOS DEL MÓDULO SEIS	124

MÓDULO OCHO: Fotos, fototecas y archivos de guión

OCHO.1. LAS FOTOS DE AutoCAD	128
OCHO.1.1. Fotos de mayor rendimiento	129
OCHO.2. FOTOTECAS O BIBLIOTECAS DE FOTOS	129
OCHO.3. UTILIZACIÓN DE FOTOS Y FOTOTECAS	131
OCHO.3.1. En línea de comandos de AutoCAD	131
OCHO.3.2. En macros	131
OCHO.3.3. En menús de imágenes	131
OCHO.3.4. En patrones de sombreado. El programa SlideManager	132
OCHO.4. ARCHIVOS DE GUIÓN	133
OCHO.4.1. Ejecutando archivos <i>scripts</i>	135
OCHO.4.2. Retardos con RETARDA	135
OCHO.4.3. Repeticiones con RSCRIPT	136
OCHO.4.4. Reanudar con REANUDA	136
OCHO.4.5. Carga de fotos antes de su visualización	137
OCHO.4.6. Otros archivos de guión	137
OCHO.4.7. Ejecución de guiones en el arranque	138
OCHO.5. EJEMPLOS PRÁCTICOS DE ARCHIVOS DE GUIÓN	139
OCHO.5.1. Ejemplo 1	139
OCHO.5.2. Ejemplo 2	139
OCHO.FIN. EJERCICIOS PROPUESTOS	139
EJERCICIOS RESUELTOS DEL MÓDULO SIETE	140

PARTE SEGUNDA

MÓDULO NUEVE: Lenguaje DIESEL y personalización de la línea de estado

NUEVE.1. INTRODUCCIÓN	141
NUEVE.2. LA VARIABLE <code>MODEMACRO</code>	142
NUEVE.3. EL LENGUAJE DIESEL	143
NUEVE.3.1. Catálogo de funciones DIESEL	143
NUEVE.3.2. DIESEL para la línea de estado	147
NUEVE.3.3. Expresiones DIESEL en menús	149
NUEVE.3.3.1. DIESEL entre corchetes	149
NUEVE.3.3.2. DIESEL en la macro	150
NUEVE.3.4. Expresiones DIESEL en botones	151
NUEVE.3.5. Expresiones DIESEL en archivos de guión	151
NUEVE.3.6. Variables <code>USERn1</code> a <code>USERn5</code> y <code>MACROTRACE</code>	151
NUEVE.4. EJEMPLOS PRÁCTICOS EN DIESEL	152
NUEVE.4.1. Línea de estado 1	152
NUEVE.4.2. Línea de estado 2	152
NUEVE.4.3. Línea de estado 3	152
NUEVE.4.4. Visibilidad de objetos <i>Proxy</i>	152
NUEVE.4.5. Orden de objetos	152
NUEVE.4.6. Ventanas en mosaico y flotantes	153
NUEVE.FIN. EJERCICIOS PROPUESTOS	153
EJERCICIOS RESUELTOS DEL MÓDULO OCHO	153

MÓDULO DIEZ: Lenguaje DCL; personalización y creación de cuadros de diálogo

DIEZ.1. LENGUAJE DCL	155
DIEZ.2. ESTRUCTURA JERARQUIZADA DE DISEÑO	155
DIEZ.3. TÉCNICA DE DISEÑO	156
DIEZ.4. LAS HERRAMIENTAS	157
DIEZ.4.1. Los <i>tiles</i> o elementos	158
DIEZ.4.2. Los atributos predefinidos	159
DIEZ.4.2.1. Atributos de título, clave y valor inicial	159
DIEZ.4.2.2. Atributos de tamaño	161
DIEZ.4.2.3. Atributos de limitaciones de uso	162
DIEZ.4.2.4. Atributos de funcionalidad	163
DIEZ.4.3. Los <i>tiles</i> y sus atributos	164
DIEZ.4.3.1. Grupos de componentes	164
DIEZ.4.3.2. Componentes individuales de acción	166
DIEZ.4.3.3. Componentes decorativos e informativos	168
DIEZ.4.3.4. Botones de salida y componente de error	169
DIEZ.4.4. Elementos predefinidos	170
DIEZ.5. PROGRAMANDO CUADROS DCL	171
DIEZ.5.1. Ejemplo sencillo: letrero informativo	171
DIEZ.5.1.1. Cómo cargar y visualizar el cuadro	173
DIEZ.5.2. Ejemplo con casillas de verificación	174
DIEZ.5.3. Letrero de control de variables de AutoCAD	176
DIEZ.5.4. Parámetros de control de una curva	179
DIEZ.FIN. EJERCICIOS PROPUESTOS	181
EJERCICIOS RESUELTOS DEL MÓDULO NUEVE	183

PARTE TERCERA

MÓDULO ONCE: Programación en AutoLISP

ONCE.1. INTRODUCCIÓN	185
ONCE.1.1. AutoLISP, ADS, ARX, VBA y Visual Lisp	185
ONCE.1.1.1. Entorno AutoLISP	185
ONCE.1.1.2. Entorno ADS	186
ONCE.1.1.3. Entorno ARX	186
ONCE.1.1.4. Entorno VBA	186
ONCE.1.1.5. Entorno Visual Lisp	187
ONCE.2. CARACTERÍSTICAS DE AutoLISP	187
ONCE.2.1. Tipos de objetos y datos en AutoLISP	188
ONCE.2.2. Procedimientos de evaluación	189
ONCE.2.3. Archivos fuente de programas	191
ONCE.2.4. Variables predefinidas	192
ONCE.3. PROGRAMANDO EN AutoLISP	193
ONCE.3.1. Convenciones de sintaxis	193
ONCE.4. OPERACIONES MATEMÁTICAS Y LÓGICAS	193
ONCE.4.1. Aritmética básica	193
ONCE.4.2. Matemática avanzada	197
ONCE.4.3. Operaciones relacionales	203
ONCE.4.4. Operaciones lógicas	206
ONCE.5. CREAR Y DECLARAR VARIABLES	209
ONCE.5.1. A vueltas con el apóstrofo (')	212
ONCE.6. PROGRAMANDO EN UN ARCHIVO ASCII	213
ONCE.7. CAPTURA Y MANEJO BÁSICO DE DATOS	216
ONCE.7.1. Aceptación de puntos	216
ONCE.7.2. Captura de datos numéricos	221
ONCE.7.3. Distancias y ángulos	222
ONCE.7.4. Solicitud de cadenas de texto	224
ONCE.7.5. Establecer modos para funciones <i>GET</i> ...	224
ONCE.7.5.1. Palabras clave	227
ONCE.8. ACCESO A VARIABLES DE AutoCAD	230
ONCE.9. ESTRUCTURAS BÁSICAS DE PROGRAMACIÓN	232
ONCE.10. MANEJO DE LISTAS	242
ONCE.11. FUNCIONES DE CONVERSIÓN DE DATOS	246
ONCE.11.1. Conversión de unidades	249
ONCE.11.1.1. Personalizar el archivo ACAD.UNT	250
ONCE.11.1.2. Ejemplos de CVUNIT	252
ONCE.12. MANIPULACIÓN DE CADENAS DE TEXTO	252
ONCE.13. ÁNGULOS Y DISTANCIAS	260
ONCE.14. RUTINAS DE CONTROL DE ERRORES	263
ONCE.14.1. Definir una función de error	264
ONCE.14.2. Otras características del control de errores	268
ONCE.15. CARGA y DESCARGA DE APLICACIONES	270
ONCE.15.1. ACADR14.LSP, ACAD.LSP y *.MNL	273
ONCE.15.1.1. Configuraciones múltiples	274
ONCE.15.1.2. Definir función como S::STARTUP	274
ONCE.15.2. Aplicaciones ADS	275
ONCE.15.3. Aplicaciones ARX	276
ONCE.15.4. Acceso a comandos externos	276
ONCE.15.4.1. Comandos programados en AutoLISP	277
ONCE.15.4.2. Comandos de transformaciones 3D	278
ONCE.15.4.3. Calculadora de geometrías	278
ONCE.15.4.4. Intercambios en formato <i>PostScript</i>	279
ONCE.15.4.5. Proyección de sólidos en ventanas	279
ONCE.15.4.6. Comandos de <i>Render</i>	279
ONCE.15.4.7. Intercambio con 3D Studio	289
ONCE.15.4.8. Comandos de ASE	290
ONCE.15.5. Inicio de aplicaciones Windows	290

Curso Práctico de Personalización y Programación bajo AutoCAD
Índice

ONCE.16. INTERACCIÓN CON LETREROS EN DCL	290
ONCE.16.1. Carga, muestra, inicio, fin y descarga	291
ONCE.16.2. Gestión de elementos del letrero	294
ONCE.16.3. Gestión de componentes de imagen	303
ONCE.16.4. Gestión de casillas de listas y listas desplegables	315
ONCE.17. OTRAS FUNCIONES DE MANEJO DE LISTAS	321
ONCE.18. MISCELÁNEA DE FUNCIONES ÚTILES	329
ONCE.18.1. Asegurándonos de ciertos datos	330
ONCE.18.2. Acceso a pantalla gráfica	331
ONCE.18.3. Lectura de dispositivos de entrada	333
ONCE.18.4. Atribuir expresión a símbolo literal	334
ONCE.19. ACCESO A OTRAS CARACTERÍSTICAS	335
ONCE.19.1. Modos de referencia	336
ONCE.19.2. El redibujado	338
ONCE.19.3. Transformación entre Sistemas de Coordenadas	339
ONCE.19.4. Ventanas y vistas	340
ONCE.19.5. Calibración de tablero digitalizador	341
ONCE.19.6. Control de elementos de menú	341
ONCE.19.7. Letrero de selección de color	342
ONCE.19.8. Funciones de manejo de ayuda	342
ONCE.19.9. Expresiones DIESEL en programas AutoLISP	344
ONCE.19.10. Macros AutoLISP en menús y botones	346
ONCE.19.11. Macros AutoLISP en archivos de guión	346
ONCE.19.12. Variables de entorno	346
ONCE.20. ACCESO A LA BASE DE DATOS DE AutoCAD	347
ONCE.20.1. Organización de la Base de Datos	347
ONCE.20.1.1. Introducción	347
ONCE.20.1.2. Estructura para entidades simples	347
ONCE.20.1.3. Estructura para entidades compuestas	350
ONCE.20.1.3.1. Polilíneas no optimizadas	350
ONCE.20.1.3.2. Inserciones de bloque con atributos	352
ONCE.20.1.4. Estructura para objetos no gráficos	354
ONCE.20.1.4.1. Capa	355
ONCE.20.1.4.2. Estilo de texto	355
ONCE.20.1.4.3. Tipo de línea	356
ONCE.20.1.4.4. Definición de bloque	356
ONCE.20.1.5. Códigos de acceso a Base de Datos	356
ONCE.20.2. Funciones de gestión de la Base de Datos	377
ONCE.20.2.1. Crear un conjunto de selección	377
ONCE.20.2.2. Obtener el nombre de una entidad	382
ONCE.20.2.3. Extraer la lista de una entidad	382
ONCE.20.2.4. Actualizar lista y Base de Datos	384
ONCE.20.2.5. Nombre de entidad por punto	386
ONCE.20.2.6. Añadir, eliminar y localizar entidades	388
ONCE.20.2.7. Aplicar y determinar pinzamientos	389
ONCE.20.2.8. Obtener nombre con modo de selección	389
ONCE.20.2.9. Otras formas de obtener nombres	391
ONCE.20.2.10. Borrar/recuperar entidades	393
ONCE.20.2.11. Obtener rectángulo de texto	394
ONCE.20.2.12. Construcción de una entidad	394
ONCE.20.2.13. Manejo de tablas de símbolos	395
ONCE.20.2.14. Funciones relativas a datos extendidos	399
ONCE.21. ACCESO A ARCHIVOS	415
ONCE.21.1. Fundamento teórico somero sobre el acceso a archivos	415
ONCE.21.2. Funciones para el manejo de archivos	416
ONCE.22. FUNCIONES DE CHEQUEO	425
ONCE.22.1. Rastreo	429
ONCE.23. OPERACIONES BINARIAS LÓGICAS	430
ONCE.24. GESTIÓN DE LA MEMORIA	432
ONCE.25. CÓDIGOS Y MENSAJES DE ERROR	435
ONCE.25.1. Códigos de error	435
ONCE.25.2. Mensajes de error	437
ONCE.FIN. EJERCICIOS PROPUESTOS	441

EJERCICIOS RESUELTOS DEL MÓDULO DIEZ	445
 MÓDULO DOCE: Programación en Visual Basic orientada a AutoCAD (VBA)	
DOCE.1. INTRODUCCIÓN	449
DOCE.2. Visual Basic Y ActiveX Automation	449
DOCE.2.1. La línea de productos de Visual Basic	450
DOCE.3. EL MÓDULO VBA DE AutoCAD	451
DOCE.4. COMENZANDO CON VBA	452
DOCE.4.1. La plantilla de objetos	452
DOCE.4.2. Comenzar un programa	454
DOCE.5. DIBUJO Y REPRESENTACIÓN DE ENTIDADES	455
DOCE.5.1. Líneas	455
DOCE.5.2. Círculos	468
DOCE.5.3. Elipses	470
DOCE.5.4. Arcos	472
DOCE.5.5. Puntos	473
DOCE.5.6. Texto en una línea	474
DOCE.5.7. Objetos de polilínea	479
DOCE.5.7.1. Polilíneas de antigua definición	479
DOCE.5.7.2. Polilíneas optimizadas	481
DOCE.5.8. Polilíneas 3D	482
DOCE.5.9. Rayos	482
DOCE.5.10. Líneas auxiliares	483
DOCE.5.11. Trazos	483
DOCE.5.12. Splines	484
DOCE.5.13. Texto múltiple	488
DOCE.5.14. Regiones	489
DOCE.5.15. Sólidos 3D	491
DOCE.5.15.1. Prisma rectangular	491
DOCE.5.15.2. Cono	492
DOCE.5.15.3. Cilindro	492
DOCE.5.15.4. Cono elíptico	492
DOCE.5.15.5. Cilindro elíptico	492
DOCE.5.15.6. Esfera	492
DOCE.5.15.7. Toroide	493
DOCE.5.15.8. Cuña	493
DOCE.5.15.9. Extrusión	493
DOCE.5.15.10. Extrusión con camino	493
DOCE.5.15.11. Revolución	493
DOCE.5.15.12. Propiedades y métodos de los sólidos 3D	494
DOCE.5.16. Caras 3D	495
DOCE.5.17. Mallas poligonales	496
DOCE.5.18. Imágenes de trama	498
DOCE.5.19. Sólidos 2D	501
DOCE.5.20. Formas	501
DOCE.5.21. Acotación, directrices y tolerancias	502
DOCE.5.21.1. Cotas alineadas	502
DOCE.5.21.2. Cotas angulares	504
DOCE.5.21.3. Cotas diamétricas	505
DOCE.5.21.4. Cotas por coordenadas	506
DOCE.5.21.5. Cotas radiales	507
DOCE.5.21.6. Cotas giradas	507
DOCE.5.21.7. Directrices	508
DOCE.5.21.8. Tolerancias	509
DOCE.5.22. Sombreado	510
DOCE.5.23. Referencias a bloques	515
DOCE.5.24. Atributos de bloques	516
DOCE.5.24.1. Referencias de atributos	516
DOCE.5.24.2. Objeto de atributo	518
DOCE.6. LA APLICACIÓN AutoCAD	519
DOCE.7. EL DOCUMENTO ACTUAL ACTIVO	523
DOCE.8. LAS COLECCIONES Y SUS OBJETOS	535

Curso Práctico de Personalización y Programación bajo AutoCAD
Índice

DOCE.8.1. Colección de objetos de Espacio Modelo	535
DOCE.8.2. Colección de objetos de Espacio Papel	537
DOCE.8.3. Colección de bloques y el objeto bloque	539
DOCE.8.4. Colección de diccionarios y el objeto diccionario	540
DOCE.8.5. Colección de estilos de acotación y el objeto estilo de acotación	542
DOCE.8.6. Colección de grupos y el objeto grupo	542
DOCE.8.7. Colección de capas y el objeto capa	543
DOCE.8.8. Colección de tipos de línea y el objeto tipo de línea	547
DOCE.8.9. Colección de aplicaciones registradas y el objeto aplicación registrada	548
DOCE.8.10. Colección de conjuntos de selección y el objeto conjunto de selección	549
DOCE.8.11. Colección de estilos de texto y el objeto estilo de texto	552
DOCE.8.12. Colección de SCPs y el objeto SCP	554
DOCE.8.13. Colección de vistas y el objeto vista	555
DOCE.8.14. Colección de ventanas y el objeto ventana	556
DOCE.8.14.1. Ventanas del Espacio Papel	561
DOCE.9. UTILIDADES VARIAS (EL OBJETO <i>Utility</i>)	563
DOCE.10. EL TRAZADO	585
DOCE.11. EL OBJETO DE PREFERENCIAS	592
DOCE.11.1. Preferencias de archivos	593
DOCE.11.2. Preferencias de rendimiento	598
DOCE.11.3. Preferencias de compatibilidad	601
DOCE.11.4. Preferencias generales	603
DOCE.11.5. Preferencias de visualización	605
DOCE.11.6. Preferencia de dispositivo	610
DOCE.11.7. Preferencia de perfil	610
DOCE.11.8. Métodos del objeto de preferencias	610
DOCE.12. ALGUNOS TRUCOS <i>ActiveX Automation</i> PARA AutoCAD	611
DOCE.12.1. Compilación de programas con un compilador de Visual Basic externo	612
DOCE.12.1. Compilación de programas con un compilador de Visual Basic externo	612
DOCE.12.1.1. Objeto de aplicación en programas compilados	614
DOCE.12.2. Ejecución de programas VBA desde AutoLISP y en macros	615
DOCE.12.3. Enviar cadenas a la línea de comandos desde VBA	616
DOCE.13. COMO APUNTE FINAL	619
DOCE.FIN. EJERCICIOS PROPUESTOS	619
EJERCICIOS RESUELTOS DEL MÓDULO ONCE	620

MÓDULO TRECE: Entorno de programación Visual Lisp

TRECE.1. Visual Lisp ES...	685
TRECE.2. PROCESO DE CREACIÓN DE UN PROGRAMA	685
TRECE.3. INSTALACIÓN E INICIACIÓN	685
TRECE.3.1. Carga y ejecución de programas	686
TRECE.4. ESCRITURA DEL CÓDIGO FUENTE	687
TRECE.4.1. Ventana de Consola	687
TRECE.4.2. Editor de texto	688
TRECE.4.2.1. La herramienta <i>Apropos</i>	689
TRECE.4.2.2. Utilidades de gestión de texto	691
TRECE.4.2.3. Formateo del código fuente	694
TRECE.4.2.4. Chequeo de errores de sintaxis	695
TRECE.5. DEPURACIÓN DE PROGRAMAS	696
TRECE.5.1. Modo de depuración <i>Break Loop</i>	698
TRECE.5.2. Modo de depuración <i>Trace</i>	699
TRECE.5.3. Ventana de seguimiento <i>Watch</i>	700
TRECE.5.4. Cuadro de diálogo de servicio de símbolos <i>Symbol Service</i>	701
TRECE.5.5. Ventana de inspección de objetos <i>Inspect</i>	701
TRECE.6. CONSTRUCCIÓN Y GESTIÓN DE APLICACIONES	702
TRECE.6.1. Compilación de archivos de programa	703
TRECE.6.2. Creación de módulos de aplicación	704
TRECE.6.3. Gestión de proyectos	704
TRECE.7. UTILIZACIÓN DE OBJETOS <i>ActiveX</i>	705
TRECE.7.1. Funciones Visual Lisp	705
EJERCICIOS RESUELTOS DEL MÓDULO DOCE	710

APÉNDICES

APÉNDICE A: Comandos y abreviaturas de AutoCAD

A.1. COMANDOS DE AutoCAD CON SU CORRESPONDENCIA EN INGLÉS	723
-----------------------------------------------------------	-----

APÉNDICE B: Variables de sistema y acotación

B.1. VARIABLES DE SISTEMA Y ACOTACIÓN	730
---------------------------------------	-----

APÉNDICE C: Bibliotecas suministradas

C.1. TIPOS DE LÍNEA ESTÁNDAR	756
C.2. TIPOS DE LÍNEA COMPLEJOS	756
C.3. PATRONES DE SOMBREADO	756
C.4. PATRONES DE RELLENO <i>PostScript</i>	758
C.5. TIPOS DE LETRA BASADOS EN DEFINICIÓN DE FORMAS	758
C.6. FUENTES <i>True Type</i>	759
C.7. SÍMBOLOS DE TOLERANCIAS GEOMÉTRICAS	760

PARTE PRIMERA

MÓDULO UNO

Personalización de menús

UNO.1. INTRODUCCIÓN

Un menú de **AutoCAD** es una secuencia de órdenes del programa agrupadas en un archivo de texto que podemos visualizar con cualquier editor ASCII. Los archivos de menú, en principio, tienen la extensión `.MNU`, además existen otros que ya se comentarán más adelante. El archivo de menú que proporciona **AutoCAD** es el llamado `ACAD.MNU`, que se encuentra en el directorio `\SUPPORT\` del programa. En él residen todas las definiciones necesarias para el funcionamiento de los menús desplegables de **AutoCAD**, de las barras de herramientas, el menú de pantalla, los botones del ratón (o los del dispositivo señalador correspondiente), menús de imágenes, textos auxiliares de ayuda y algunas teclas rápidas. Este menú es susceptible de ser editado y alterado al gusto, así como también tenemos la posibilidad de crear nuestros propios menús personalizados para **AutoCAD**.

La mejor forma de aprender cómo funcionan estos menús es recurriendo al ya proporcionado por Autodesk es su programa estrella. Para ello, únicamente debemos abrir `ACAD.MNU`, como ya se ha dicho con cualquier editor ASCII. Recordemos que se encuentra en el directorio `\SUPPORT\` de **AutoCAD**.

UNO.2. EL ARCHIVO `ACAD.MNU`

Como podemos apreciar, lo primero que nos encontramos en este archivo de menú, es una serie de líneas en las que Autodesk explica la forma de proveer este archivo y que es posible modificarlo a nuestro gusto. Generalmente, este texto estará escrito en inglés. Pero fijémonos en los dos primeros caracteres impresos en cada línea (`//`). Estos dos caracteres de

barra seguidos indican que lo que viene a continuación es un texto explicativo o una aclaración que no debe ser procesada por **AutoCAD**. Todo lo que se escriba tras // será ignorado por el programa. Además, también podemos introducir líneas blancas completas para separar sin que **AutoCAD** interprete nada en ellas. Pero ojo, no deberemos abusar de ello ya que, en estas explicaciones entre secciones no significan nada, pero más adelante, bajo cada sección, pueden significar mucho. Es decir, abusar si se quiere, pero con control de dónde. Deberemos tener también en cuenta que un alto contenido de explicaciones o líneas blancas aumentará el tamaño del fichero y, por lo tanto, el tiempo empleado por **AutoCAD** para procesarlo.

La primera línea que, podríamos decir, tiene sentido para **AutoCAD** es la que dice *****MENUGROUP=ACAD**. Los caracteres ******* son indicativo de categoría sección. Un archivo .MNU de **AutoCAD** puede tener hasta 31 secciones distintas. Los 31 nombres que adoptan esta serie de secciones son normalizados y no pueden alterarse. Cada uno de ellos hace referencia a un dispositivo y debe ocupar una sola línea en el archivo.

Concretamente, este *****MENUGROUP=** especifica el nombre de grupo de archivos de menú, en este caso **ACAD** (nombre del archivo). Este nombre es una cadena de 32 caracteres como máximo y que no puede contener ni espacios ni signos de puntuación. Su nombre no ha de coincidir obligatoriamente con el nombre del archivo que lo contiene, pero es conveniente para evitar fallos o equivocaciones por nuestra parte.

UNO.2.1. Las secciones ***BUTTONS** y *****AUX****

Las cuatro secciones siguientes, desde *****BUTTONS1** hasta *****BUTTONS4**, definen la actuación de los pulsadores del dispositivo señalador de **AutoCAD** (ratón, lápiz óptico o digitalizador de tableta). Concretamente *****BUTTONS1** define el modo de actuar de todos los botones del dispositivo. Bajo esta sección se escribe una línea por cada botón configurado, además de la acción que debe realizar al ser pulsado. Esto a partir del segundo pulsador, ya que el primero es reservado para la entrada de datos y elección de órdenes y es el pulsador principal por defecto del sistema. Esto nos lleva a pensar que, en el caso de un dispositivo tipo lápiz digitalizador, el cual sólo posee un pulsador, todas las demás definiciones de botones serán ignoradas.

Lo demás que encontramos bajo esta sección son las referencias a submenús o a alias necesarias para que los pulsadores funcionen. Estas referencias serán explicadas más adelante. Por lo general, esta sección de *****BUTTONS1** en concreto, y también las siguientes secciones *****BUTTONS**, no han de ser modificadas por el usuario, ya que puede ser molesto tener que acostumbrarse a un nuevo juego con los pulsadores diferente al actual.

*****BUTTONS2** a *****BUTTONS4**, por su lado, especifican la acción combinada de ciertas teclas con los pulsadores del dispositivo. Concretamente sus definiciones son las siguientes:

***BUTTONS2	SHIFT + botón
***BUTTONS3	CTRL + botón
***BUTTONS4	CTRL + SHIFT + botón

Las secciones siguientes que nos encontramos en **ACAD.MNU** son las cuatro que van desde *****AUX1** hasta *****AUX4**. El funcionamiento de ellas es exactamente el mismo que el de las secciones *****BUTTONS**, pero con la particularidad de que están orientadas a los dispositivos señaladores de entornos Macintosh y estaciones de trabajo.

UNO.2.2. Menús desplegados (secciones ***POP**)**

A continuación topamos con las secciones *****POP**. Las secciones *****POP** definen la apariencia de los menús desplegables y de cascada de **AutoCAD**, así como las órdenes que se ejecutarán al hacer clic en cada elemento.

Las secciones *****POP1** a *****POP16** guardan las definiciones de los menús desplegables de la barra de menús de **AutoCAD** (Archivo, Edición, Ver,...,?). En el archivo **ACAD.MNU** están definidas de la *****POP1** a la *****POP10**, correspondiéndose con cada uno de los elementos que nos encontramos en dicha barra de menús. Podemos definir, entonces, hasta un máximo de *****POP16**. Además de esto, disponemos de otras dos secciones especiales: *****POP0**, que define el menú de cursor de referencia a objetos (el que aparece al pulsar el botón central del ratón o **SHIFT** + botón derecho o **CTRL** + botón derecho) y *****POP17**, que define el menú de cursor contextual de pinzamientos (el aparecido al pulsar botón derecho tras mostrar los puntos de pinzamiento).

Después del indicativo de sección, podemos apreciar otro que comienza con dos asteriscos (**). El indicativo ****** es categoría de submenú, pero en este caso, en la sección *****POP**, es lo que se denomina alias. Estos alias definen un nombre para el menú desplegable según su función (no son obligatorios). Después, podremos referenciar dicho menú desplegable por su identificador de sección o por su alias (ya se verá más adelante).

Lo siguiente que vemos, la primera línea tras el indicativo de sección —o tras el de sección y alias si lo hubiera— es el título del menú desplegable. Para este título se permiten 14 caracteres como máximo, pero es conveniente limitar este número debido a la posterior alineación en la barra de menús de **AutoCAD**. Si esta línea en la que se indica el título no existiera, el menú no funcionaría. Además, ha de ser la inmediatamente posterior al indicativo de sección o alias, esto es, no puede haber un espacio en blanco. Decir también que, este título, no puede contener comandos asignados, esto es, no se puede utilizar como una opción —lo que se explica a continuación—.

Tras el título del desplegable se describen las diferentes opciones que se desplegarán. Como vemos, el título de cada una de ellas, lo que se verá en pantalla, va encerrado entre corchetes, aunque esto no es estrictamente necesario. La diferencia estriba en que la colocación de corchetes permite la introducción de un máximo de 78 caracteres para el primer menú (el situado más a la izquierda). Al desplegarse los nombres aparecen alineados por la izquierda. Si no se escribieran los corchetes, **AutoCAD** truncaría todos los nombres a 8 caracteres. Por supuesto, la opción más larga determina el ancho de persiana desplegada.

Los títulos de opciones pueden contener una letra subrayada, la cual será el acceso a la opción por medio del teclado. Así mismo, el nombre de título puede contener también una letra subrayada para acceder a él mediante **ALT** + *letra*. Este carácter subrayado se consigue anteponiendo el símbolo **&** (*ampersand*) al carácter en cuestión. Así:

[&Archivo]	<u>A</u> rchivo
[&Nuevo]	<u>N</u> uevo
[C&oordenada]	C <u>o</u> ordenada
[A<altura]	A <u>l</u> tura

NOTA: Existía antiguamente otra forma de indicar el subrayado de una de las letras de la cadena, y era especificar cuál de ellas iba a ser la subrayada, de la siguiente forma: [/AArchivo]. Cuidado, porque esto ya no funciona (o no debería).

Debemos tener muy en cuenta no repetir dentro de un mismo menú desplegable la misma letra subrayada para dos opciones diferentes, ya que, en el peor de los casos, sólo funcionaría la opción que antes se encuentre, y eso no nos interesa. Lo mismo ha de comprobarse con los desplegable dentro de un mismo archivo de menú, e incluso en diversos archivos de menú parciales cargados al mismo tiempo.

La sintaxis de estas opciones no es complicada. Veamos un ejemplo:

```
ID_Line      [&Línea]^C^C_line
```

Lo situado más a la izquierda (ID_Line) es una simple etiqueta que después utilizaremos para referenciar determinados textos de ayuda rápida que aparecen en la barra o línea de estado (en la sección *****HELPSTRING**) y para referenciar teclas rápidas (en la sección *****ACCELERATORS**). No es necesario incluir esta etiqueta, pero puede servir como veremos.

Tras ello, y luego de un espacio o tabulador (da igual el número de espacios, todos se interpretan como uno), aparece la definición textual (entre corchetes) que será la visualizada en pantalla ([&Línea]). Como podemos apreciar, aparecerá con el primer carácter subrayado.

Por último, se escribe la orden en cuestión de **AutoCAD** que será ejecutada al pinchar con el cursor del dispositivo señalador en la opción correspondiente. Además, suelen incluirse dos caracteres CTRL+C seguidos (^C^C) para anular cualquier orden anterior en proceso no terminada (tecla ESC en Windows), excepto en comandos transparentes.

La orden que se ejecutará podrá indicarse en castellano —si trabajamos con la edición española de **AutoCAD**— o con su equivalente inglesa antecedita por un guión de subrayado (_) —tanto si trabajamos con la versión española como con la inglesa—. Téngase en cuenta que lo que se indica tras el último corchete de cierre es lo mismo que si se escribiera en la línea de comandos de **AutoCAD**. Así, las siguientes órdenes son análogas:

[&Círculo]^C^Ccírculo	(sólo versión castellana)
[&Círculo]^C^C_circle	(versiones castellana e inglesa)
[&Círculo]^C^Ccircle	(sólo versión inglesa)

Existe la posibilidad de presentar alguna de estas opciones, en algún momento, con un tono apagado (gris claro), con el fin de indicar que, en ese momento, la opción no está disponible. Para realizar esto deberemos escribir una tilde (~) antes del nombre propio de la opción. Por ejemplo:

```
[~A&tributos nuevos]
```

Recordemos que el carácter tilde se corresponde con el código ASCII 126 y se escribe manteniendo pulsada la tecla ALT y tecleando 126 en el teclado numérico.

Este tipo de opciones apagadas han de ir envueltas bajo una condición, es decir, la opción estará apagada siempre y cuando se cumpla determinada condición y si, por el contrario, no se cumple, se encenderá —o viceversa—. Este tipo de ordenes aprenderemos a realizarlas más adelante.

También podemos introducir una línea separadora entre grupos de comandos. Esta línea podemos conseguirla añadiendo una opción de menú que sea exclusivamente dos guiones entre corchetes, esto es [--]. Una igual la podemos observar en ACAD.MNU, en la sección *****POP1**, entre la opción [&Abrir] y [&Guardar]. Esta línea, luego en pantalla, tendrá una longitud igual a la opción más larga, siempre que ésta no sobrepase los 39 caracteres.

A veces, es conveniente indicar al lado del nombre de opción, y con un espacio tabulado, la combinación de teclas (si las hubiere) rápidas para acceder a dicha opción de una forma acelerada. Esto se consigue con el mecanismo \t de la forma que vemos a continuación como ejemplo del archivo que estamos estudiando:

```
[&Nuevo...\tCtrl+N]^C^C_new
```

De esta manera (sin incluir ningún espacio) se indica que la combinación CTRL+N también accede al cuadro de diálogo *Nuevo*. Esta combinación de teclas rápidas puede ser definida en la última sección de un archivo de menús, llamada ***ACCELERATORS (explicada más adelante).

NOTA: Apréciase que, en los menús desplegables, a las opciones que abren cuadros de diálogo se les suelen añadir tres puntos suspensivos detrás del nombre. Esto no es obligatorio, pero se ha convertido en un estándar de Windows para discriminar este tipo de opciones de las que se ejecutan nada más seleccionarse. En **AutoCAD** tampoco es así exactamente, ya la inmensa mayoría de las órdenes no se ejecutan directamente y piden opciones en la línea de comandos, pero la técnica se utiliza de todos modos para las que abren cuadros de diálogo.

En este archivo ACAD.MNU también podemos apreciar otro juego de caracteres propios de las secciones ***POP de los archivos de menú de **AutoCAD**. Estos caracteres son: ->, que indica el principio de un menú de cascada y <-, que indica el final del menú de cascada. Es el caso siguiente del ACAD.MNU:

```
[->Filtros para puntos]
  [.X].X
  [.Y].Y
  [.Z].Z
  [--]
  [.XY].XY
  [.XZ].XZ
  [<-.YZ].YZ
  ...
```

NOTA: Como podemos comprobar en este ejemplo, las órdenes invocadas (.x, .y...) no son anteceditas por caracteres ^C. Esto se debe a que son comandos transparentes de **AutoCAD** y no sería conveniente cerrar el proceso de la orden en curso para ejecutarlos, sino todo lo contrario. Cuidado con los comandos transparentes que son precedidos de apóstrofo (') y su correspondiente no transparente no lo lleva; hay que utilizarlos tal y como se utilizarían en la línea de comandos.

Los dos caracteres de apertura -> (guión y mayor que) indican el título de una opción de menú que se desplegará en menú de cascada. Al representarse en los menús de **AutoCAD**, aparecerá una pequeña flecha negra, indicando hacia la derecha, que muestra la presencia de un menú de cascada posterior y que se despliega a partir de ahí. Los dos caracteres de final de menú de cascada <- (guión y menor que) han de colocarse en la última opción de dicho submenú de cascada e indican la vuelta al menú desplegable.

Se pueden anidar (meter unos dentro de otros) diversos menús de cascada, pero con la particularidad de que, al final, deberán aparecer tanto caracteres <- como niveles de anidamiento se hayan producido. Veamos otro ejemplo del archivo por defecto de menús de **AutoCAD**:

```
[->Ayúdas al dibujo]
  [&Revisar]^C^C_audit
  [R&ecuperar]^C^C_recover
  [--]
  [->L&impiar]^C^C_purge
    [&Todo]^C^C_purge _a
    [--]
    [&Capas]^C^C_purge _la
    [Tipos &línea]^C^C_purge _lt
    [&Estilos de texto]^C^C_purge _st
    [Estilos de &acotación]^C^C_purge _d
    [Estilos línea &múltiple]^C^C_purge _m
```

```
[&Bloques]^C^C_purge _b
[<-<-&Formas]^C^C_purge _sh
...
```

Como podemos apreciar, al existir un doble anidamiento, hemos de cerrarlo al final con dos grupos de caracteres de cierre (<-<-). En conclusión, deberá haber tantos <- como -> haya. Además conviene terminar cualquier menú o submenú con una línea en blanco, de esta forma, cualquier menú que se reference (ya lo veremos) se superpondrá totalmente al anterior.

NOTA: Nótese que, a partir de la siguiente línea al segundo anidamiento, cada comando `_purge` está separado de su parámetro u opción (`_a`, `_lt`, `_m`...) por un espacio blanco. Como ya se dijo, escribir las órdenes aquí es como hacerlo en la línea de comandos, y en ella, escribiríamos el comando, pulsaríamos INTRO (= Barra Espaciadora o carácter *espacio*) y escribiríamos la abreviatura de la opción correspondiente. Tras esto, volveríamos a pulsar INTRO. Este último INTRO se corresponde con el último carácter de *retorno de carro* de cada línea, es decir, al acabar de escribir cada una de las líneas hay que pulsar ENTER (también en la última del archivo si fuera una instrucción u orden).

En conclusión, podríamos decir que las diversas sintaxis que se inscriben bajo esta sección se pueden generalizar como las que siguen:

```
etiqueta    [nombre_del_desplegable]
etiqueta    [opción_de_menú]^C^Corden_de_AutoCAD
etiqueta    [->entrada_a_menú_de_cascada]
etiqueta    [<-salida_de_menú_de_cascada]
[--]
```

UNO.2.3. Las barras de herramientas (la sección ***TOOLBARS)

La sección *****TOOLBARS** describe el aspecto y función de todas las barras de herramientas incluidas en `ACAD.MNU`. Nosotros podremos modificarlas o crear nuevas barras, ya sea dentro de este archivo de menús o dentro de uno propio. Cada barra de herramientas se define como un submenú de la sección *****TOOLBARS**, es decir, con el indicativo de submenú (**).

Es posible especificar cinco tipos distintos de elementos en la creación de barras de herramientas. La sintaxis general de dichos tipos es que sigue (lo indicado en letra *itálica* se corresponde con texto variable que se sustituirá por valores o palabras claves):

```
etiqueta    [_Toolbar("nombre_barra", _orient, _visible, valx, valy,
                    filas)]
etiqueta    [_Button("nombre_botón", id_pequeño, id_grande)]macro
etiqueta    [_Flyout("nombre_botón_desplegable", id_pequeño, id_grande,
                    _icono, alias)]macro
etiqueta    [_Control(_elemento)]
[--]
```

La *etiqueta* realiza la misma función explicada en las secciones *****POP** de menús desplegables, es decir, es un identificador para referenciar pequeños textos de ayuda que aparecen en la línea de estado de **AutoCAD** y que se definen en los archivos de menús en la sección *****HELPSTRINGS**, que veremos posteriormente. También se utiliza en la sección *****ACCELERATORS**. Esta etiqueta de referencia puede incluirse o no.

Tras la etiqueta aparece el tipo de elemento, con sus modificadores o parámetros entre paréntesis, entre corchetes. Vamos a explicar cada uno de ellos por separado y en el orden indicado.

El primero, `_Toolbar`, establece las características de la definición de barra de herramientas. Sus opciones tienen el siguiente significado:

- `"nombre_barra"`. Cadena alfanumérica que define el nombre de la barra de herramientas. Se permiten espacios y caracteres de guión (-) y guión de subrayado (_) como únicos símbolos de puntuación. Debe ir encerrada entre comillas dobles (" ").
- `_orient`. Es una palabra clave de orientación. Puede ser `_Floating`, `_Top`, `_Bottom`, `_Left` o `_Right`. Indica dónde aparece la barra de herramientas al arrancar **AutoCAD**: flotando sobre el área de dibujo, en la parte superior, en la inferior, a la izquierda o a la derecha. Aunque todo depende de la siguiente opción. Puede ser escrito en mayúsculas o minúsculas.
- `_visible`. Debe ser una palabra clave de visibilidad. Puede ser `_Show` o `_Hide`, según se quiera visible u oculta. Mayúsculas o minúsculas.
- `valx`. Es un valor numérico que especifica la coordenada X (en píxeles) desde el lado izquierdo de la pantalla hasta el lado izquierdo de la barra de herramientas.
- `valy`. Es otro valor numérico que indica la coordenada Y (en píxeles) desde el lado superior de la pantalla hasta la parte superior de la barra.
- `filas`. Un valor que indica el número de filas de la barra de herramientas.

Unos ejemplos de `ACAD.MNU` son los siguientes:

```
ID_TbDimensi [_Toolbar("Acotar", _Floating, _Hide, 100, 130, 1)]
ID_TbDraw    [_Toolbar("Dibujo", _Left, _Show, 0, 0, 1)]
ID_TbModifII [_Toolbar("Modificar II", _Floating, _Hide, 100, 270, 1)]
ID_TbModify  [_Toolbar("Modificar", _Left, _Show, 1, 0, 1)]
```

NOTA: No olvidar los espacios tras las comas.

El segundo elemento es `_Button`. Éste define un botón de la barra de herramientas. Sus opciones tienen el siguiente significado:

- `"nombre_botón"`. Es una cadena alfanumérica que define el nombre del botón; se permiten el guión (-) y el guión de subrayado (_) como únicos caracteres de puntuación. Esta cadena es la que se muestra como pista o *tip* amarillo cuando el cursor se sitúa encima del botón. Entre comillas.
- `id_pequeño`. Es una cadena alfanumérica que define el recurso de imagen pequeña (mapa de bits de 16 × 15), esto es, cuál es el icono que se representará en el botón. Podemos crear iconos nuevos en formato `.BMP` y referenciarlo, pero esto ya lo veremos más adelante, desde el propio **AutoCAD**. La cadena en cuestión puede contener los caracteres - y _.
- `id_grande`. Cadena que define el recurso de imagen grande (24 × 22). Por lo demás, igual al anterior.
- `macro`. Es la macroinstrucción o, instrucción simple, que ejecuta los comandos, o el comando, en cuestión de **AutoCAD** asignada al botón.

Los ejemplos siguientes son definiciones de botones en `ACAD.MNU`:

```
ID_Line      [_Button("Línea", ICON_16_LINE, ICON_24_LINE)]^C^C_line
ID_Arc       [_Button("Arco", ICON_16_ARC3PT, ICON_24_ARC3PT)]^C^C_arc
ID_Image     [_Button("Imagen", ICON_16_IMAGE, ICON_24_IMAGE)]^C^C_image
ID_Xref      [_Button("RefX", ICON_16_XREATT, ICON_24_XREATT)]^C^C_xref
```

NOTA: No olvidar los espacios tras las comas.

El tercer elemento que vamos a explicar es `_Flyout`, que define un botón desplegable en una barra de herramientas. Las opciones y su significado son las que siguen:

- *"nombre_botón_desplegable"*. Cadena alfanumérica que define el nombre del botón desplegable; puede contener como únicos caracteres de puntuación el guión normal (-) y el guión de subrayado (_). Este nombre es el que aparece como pista o *tip* amarillo al situar el cursor encima del botón. Entre comillas.
- *id_pequeño*. Es una cadena alfanumérica que define el recurso de imagen pequeña (mapa de bits de 16 × 15), esto es, cuál es el icono que se representará en el botón. Podemos crear iconos nuevos en formato `.BMP` y referenciarlo, pero esto ya lo veremos más adelante, desde el propio **AutoCAD**. La cadena en cuestión puede contener los caracteres - y _.
- *id_grande*. Cadena que define el recurso de imagen grande (24 × 22). Por lo demás, igual al anterior.
- *_icono*. Debe ser una palabra clave que controla si en el botón principal (el que siempre está a la vista) debe visualizarse el icono propio o el último seleccionado. Acepta sólo `_OwnIcon` (icono propio) u `_OtherIcon` (otro icono), ya sea en mayúsculas o minúsculas.
- *alias*. Hace referencia a la barra de herramientas que debe mostrarse tras el desplegado. El alias referencia a un submenú de barra de herramientas definido con la sintaxis estándar ***nombre_submenú*. Este submenú, evidentemente, puede ser uno de **AutoCAD** o uno propio creado por el usuario. El nombre del alias estará formado por el nombre del grupo de menús, seguido de un punto y del propio nombre del submenú. Por ejemplo, `ACAD.TB_ZOOM`.
- *macro*. Cadena de comando. No es imprescindible si en el submenú referenciado están todos los botones definidos. De todas formas, aún así, puede interesar ejecutar uno o varios comandos.

Ejemplos del tipo `_Flyout` son (de `ACAD.MNU`):

```
ID_TbZoom    [_Flyout("Zoom", ICON_16_ZOOM, ICON_24_ZOOM, _OtherIcon,
                      ACAD.TB_ZOOM)]
ID_TbInsert  [_Flyout("Bloque", ICON_16_BLOCK, ICON_24_BLOCK, _OtherIcon,
                      ACAD.TB_INSERT)]
ID_TbUcs     [_Flyout("SCP", ICON_16_UCS, ICON_24_UCS, _OtherIcon,
                      ACAD.TB_UCS)]
ID_TbInquiry [_Flyout("Consultar", ICON_16_LIST, ICON_24_LIST, _OtherIcon,
                      ACAD.TB_INQUIRY)]
```

NOTA: No olvidar los espacios tras las comas.

Como cuarto elemento en la creación de barras de herramientas tenemos `_Control`. `_Control` define un elemento de control especial. El único parámetro modificador es:

- *_elemento*. Puede tener tres valores, ya sea en mayúscula o en minúscula: *_Layer*, *_Linetype* o *_Color*.

_Layer especifica el elemento de control de capas. Este elemento es una lista desplegable desde la cual se controlan las capas actuales del dibujo. *_Linetype*, por su lado, especifica el elemento de control de tipo de línea, que es una lista desplegable que controla los tipos de línea. Y *_Color* es el elemento de control de color. Despliega una lista desde la que se controlan los colores de los objetos (ejecuta el comando de **AutoCAD** DDCOLOR).

Estos tres elementos son los típicos que aparecen en la barra de herramientas de *Propiedades de objetos*. Sus definiciones en *ACAD.MNU* son las siguientes:

```
ID_CtrlLayer [_Control(_Layer)]
ID_CtrlColor [_Control(_Color)]
ID_CtrlLinetype [_Control(_Linetype)]
```

El quinto y último elemento es *[--]*. Al igual que en los menús desplegables, lo que hace este elemento es separar grupos, en este caso, de botones. Ahora, en lugar de ser una línea divisoria, es un pequeño intersticio o espacio vacío entre botones de iconos —ya sea en horizontal o vertical, dependiendo de la posición que adopte la barra de herramientas—.

NOTA: Podemos utilizar bibliotecas *.DLL* de recursos de mapas de bits para almacenar los mapas de bits utilizados para las barras de herramientas. El nombre de la biblioteca *.DLL* debe ser el mismo que el nombre del archivo de menú asociado; los recursos deben nombrarse sin número de índice y el archivo *.DLL* debe ubicarse en el mismo directorio que el archivo de menús que lo utiliza. Para utilizar estos recursos en el menú, utilizaremos los nombres de recursos adecuados en los parámetros *id_pequeño* e *id_grande* para los botones de barra de herramientas

UNO.2.4. La sección *****IMAGE** de menús de imágenes

Los menús de imágenes de **AutoCAD** son aquellos que se nos presentan en una ventana en la que podemos elegir una imagen, o icono, o su correspondiente nombre. Un ejemplo, de los tres que incluye **AutoCAD** en *ACAD.MNU*, es el cuadro de objetos 3D predefinidos (cubo, cono, cuña, etc.) y que podemos abrir bajo *Dibujo>Superficies>Superficies 3D...*

Las especificaciones necesarias para la creación de este tipo de menús se encuentran recogidas en la sección *****IMAGE**. La manera general de crear este tipo de menús es similar a la que se ha estudiado en la sección de desplegables *****POP**, la diferencia reside en la presentación de las opciones, ya que en estos menús el proceso que realizará una opción se representa mediante un elemento gráfico. Dicho elemento no es otro que un archivo de foto de **AutoCAD** o un elemento de una fototeca (la creación de bibliotecas de fotos se explica en el **MÓDULO OCHO** de este curso).

Estos menús de iconos son visualizados en una ventana dividida en dos partes en la que se muestra, por un lado y a la izquierda, una lista de los términos correspondientes a los iconos y, por otro lado y a la derecha, los iconos propiamente dichos. Esta parte derecha la componen cinco filas de cuatro iconos cada una. Evidentemente, si incluimos más iconos tenemos la posibilidad de visualizar otras ventanas más navegando con los botones creados a tal efecto.

Veamos un ejemplo del *ACAD.MNU*:

```
[acad(Box3d,Prisma_rectang.)]^C^Cai_box
```

El texto que se presenta entre corchetes ([]) corresponde a la llamada a la foto —en este caso dentro de una fototeca— que, además, permite incluir una cadena que será la que se presente en la zona izquierda del menú de imagen. Lo que sigue a todo ello, como sabemos ya, es la orden o comando de **AutoCAD** que debe ejecutarse. Así, las opciones de sintaxis, en cuanto al texto entre corchetes, que podemos utilizar bajo esta sección y su resultado en pantalla son las siguientes:

Sintaxis	Área de iconos	Área de texto
[<i>nombre_de_foto</i>]	foto	nombre del fichero
[<i>nombre_de_foto,texto</i>]	foto	texto
[<i>fototeca(foto)</i>]	foto	nombre de la foto
[<i>fototeca(foto,texto)</i>]	foto	texto
[<i>espacio_blanco</i>]	<i>vacío</i>	espacio blanco
[<i>texto</i>]	<i>vacío</i>	texto

El caso anterior expuesto se correspondería con la cuarta sintaxis de la lista.

El título del menú será visualizado en la barra de título de la ventana de menú con un máximo de 40 caracteres. Este título se corresponde con la primera línea de la sección o submenú (entre corchetes) tras el propio indicador de sección o submenú. Ejemplo:

```
**image_poly  
[Definición de variables Spline]  
[acad(pm-quad,Malla cuadrática)]'_surftype 5  
[acad(pm-cubic,Malla cúbica)]'_surftype 6  
[acad(pm-bezr,Malla Bézier)]'_surftype 8  
[acad(pl-quad,Polilínea cuadrática)]'_splinetype 5  
[acad(pl-cubic,Polilínea cúbica)]'_splinetype 6
```

El texto [Definición de variable Spline] es el título de cuadro de menú.

UNO.2.5. El arcaico menú de pantalla de AutoCAD bajo la sección ***SCREEN

En los “tiempos heroicos” de **AutoCAD**, el único acceso a los comandos del programa, aparte de la línea de comandos, era un menú formado por una columna de órdenes, que se iban superponiendo mientras las ejecutábamos, y sito en la parte derecha del área de dibujo del programa. Hoy en día, para nostálgicos de aquellas versiones, seguimos disponiendo de una parte configurable en archivos de menú con el fin de utilizar éste de pantalla. Por defecto, al correr **AutoCAD**, no se muestra dicho menú, sin embargo, podemos visualizarlo desde Herr.>Preferencias..., en la pestaña *Visual.*, activando la opción Mostrar menú de pantalla de AutoCAD en la ventana de dibujo.

La sección ***SCREEN, podríamos decir que es la opción “por defecto” en un archivo de menús. Esto significa que, si al principio de un fichero .MNU no aparece un indicativo de sección, todas las opciones sin especificador de sección (hasta que aparezca uno) son asignadas al identificador de pantalla ***SCREEN.

La sintaxis es muy parecida a la de los menús desplegables, obviando la etiqueta que aquí no se puede utilizar:

```
[texto]comando
```

El primer submenú que se presenta bajo el identificador de sección es el que aparecerá primero al cargar el archivo de menú. Obviamente, éste ha de ser el que contenga las referencias más generales, si es que hay otros submenús. Así, el primer submenú que incorpora ACAD.MNU es:

```
**S
[AutoCAD ]^C^C^P(ai_rootmenus) ^P
[* * * * ]$$=ACAD.OSNAP
[ARCHIVO ]$$=ACAD.01_FILE
[EDICIÓN ]$$=ACAD.02_EDIT
[VER 1   ]$$=ACAD.03_VIEW1
[VER 2   ]$$=ACAD.04_VIEW2
[INSERTAR]$$=ACAD.05_INSERT
[FORMATO ]$$=ACAD.06_FORMAT
[HERRAM 1]$$=ACAD.07_TOOLS1
[HERRAM 2]$$=ACAD.08_TOOLS2
[DIBUJO 1]$$=ACAD.09_DRAW1
[DIBUJO 2]$$=ACAD.10_DRAW2
[ACOTAR  ]$$=ACAD.11_DIMENSION
[MODIF 1 ]$$=ACAD.12_MODIFY1
[MODIF 2 ]$$=ACAD.13_MODIFY2

[AYUDA   ]$$=ACAD.14_HELP
```

Desde él se hacen referencia a los otros submenús incluidos (veremos esto al hablar de submenús). Los demás submenús mantienen la sintaxis indicada, por ejemplo:

```
[Nuevo   ]^C^C_new
[Abrir   ]^C^C_open

[Guardar ]^C^C_qsave
[Guarcomo]^C^C_saveas
[Exportar]^C^C_export

[Config  ]^C^C_config
...
```

El texto entre corchetes es el que se presentará en pantalla. Este texto está limitado, bajo esta sección de menú, a ocho caracteres.

Como hemos dicho, los comandos se colocan en formato de columna. El número de líneas permitidas en dicha columna dependerá del monitor de vídeo y, evidentemente, de la tarjeta gráfica instalada en el equipo. De esta forma, el fichero suministrado por Autodesk con **AutoCAD** prevé un máximo de 26 líneas. Cuando no se puedan introducir todas las opciones necesitadas en una sola columna, se introducen referencias a diferentes páginas del mismo submenú para poder visualizar todo.

La siguiente tabla muestra una relación entre las tarjetas de vídeo y el número de líneas máximo permitido:

Tarjeta	Líneas
CGA color	21
CGA monocromo	22
EGA	21
VGA	26

Los sistemas SuperVGA y otros de alta resolución permite mayor número de líneas en pantalla.

Un menú de pantalla se despliega sobre el anterior visualizado. En principio, lo hace desde la primera línea útil, superponiendo sus opciones a todas las anteriores. Si deseamos que esta superposición comience desde otra línea, para dejar opciones generales al descubierto por ejemplo, únicamente deberemos indicar tras el submenú, y luego de un espacio en blanco, el número de línea desde el que queremos que empiece a desplegarse. Este es el caso de todos los submenús incluidos en ACAD.MNU (excepto el primero) que comienzan su visualización en la línea 3. Veamos algún ejemplo:

```
**ASSIST 3
**06_FORMAT 3
**101_SOLIDS 3
**AREA 3
```

De esta forma se impide sobrecribir las dos primeras líneas del menú general: AutoCAD y * * * *, que hacen referencia al propio submenú principal o general y al submenú de modos de referencia a objetos, respectivamente.

Repasando el archivo ACAD.MNU, podemos apreciar también unos grandes espacios en blanco entre un submenú y otro. El objeto de este espaciado se corresponde con la necesidad de tapar las opciones de un submenú anterior cuando otro, más corto, se superpone. Como norma general, se ampliarán los submenús con tantas líneas blancas como sean necesarias para completar el cómputo total de líneas. De esta manera, nos aseguramos de que ninguna opción de ningún submenú permanezca visible al llamar a otro submenú.

Apreciamos también en este archivo, que todos los submenús se completan con líneas blancas, pero no hasta la última (26) sino hasta la antepenúltima (24). La misión de esta táctica consiste en que las dos últimas opciones del submenú general (ASISTIR y ÚLTIMO) queden, también, siempre visibles.

De la misma forma, se hace conveniente rellenar con espacios blancos todos los nombres de opciones dentro de los corchetes has ocho caracteres. Así evitamos la desagradable visión de comandos “montados” sobre otros, y todas las opciones taparán a las anteriores perfectamente:

```
[VER 1    ]
[AYUDA    ]
[DESDE    ]
[Redibuj  ]
[RefX     ]
```

Por último, decir que toda sección ***SCREEN ha de terminar con un indicador de submenú **ENDSCREEN para que todo funcione correctamente y el último submenú real no tenga problemas en saber dónde termina.

UNO.2.6. Configuración del tablero digitalizador bajo las secciones *TABLET**

Bajo las cuatro secciones posibles del menú de tableta, ***TABLET1 a ***TABLET4, se encuentran las definiciones necesarias para el funcionamiento de la tableta o tablero digitalizador.

Las casillas o celdas de la plantilla suministrada con **AutoCAD** deberán coincidir con las especificaciones de estas secciones. Esta plantilla la podemos encontrar, preparada para imprimir o trazar a escala natural y pegar o fijar a la tableta, bajo el nombre de archivo

TABLET14.DWG en el directorio \SAMPLE\ de **AutoCAD**. Aún así, podemos modificar los términos de funcionamiento del tablero.

NOTA: Modificar estas secciones, así como las que hacían referencia a los pulsadores del dispositivo señalador, no es buena idea, ya que cambiaría la forma y el hábito adquirido de trabajar con **AutoCAD**. En el caso de la sección que nos ocupa, habría que variar hasta los iconos de sitio en la plantilla —por ello nos la entregan como fichero de dibujo— o modificarla por completo. No tiene mucho sentido, a no ser que sean cambios justificados y que reviertan comodidad y productividad en el dibujo y diseño.

En las áreas del tablero digitalizador, la configuración de cada una de ellas (comando TABLERO de **AutoCAD**, opción CFG) determina el número de casillas en que se encuentra dividida (número de filas × número de columnas). Internamente, estas casillas se ordenan de izquierda a derecha y de arriba abajo. El orden en que se encuentran las opciones en las secciones *****TABLET** en el archivo de menú es lo que determina a qué casilla concreta del tablero se asocia. Basta con pinchar en dicha casilla para que se ejecute el contenido de la línea del archivo de menú asociado.

Bajo *****TABLET1** se encuentran las definiciones del primer menú de tableta, esto es, el correspondiente al área superior de la plantilla; concretamente englobado por 25 columnas (1 a 25) y 9 filas (A a I). Bajo *****TABLET2**, las definiciones del menú segundo, el del extremo izquierdo de 11 columnas (1 a 11) y 9 filas (J a R). Bajo *****TABLET3**, las de menú tercero; las que se corresponden con 3 columnas (23 a 25) y 9 filas (J a R). Lo que queda en la plantilla corresponde al área gráfica de dibujo.

Por lo demás, decir que su funcionamiento es igual al de los demás menús, con la particularidad de que, en éste, los textos entre corchetes no tienen ningún efecto visual en pantalla, por lo que sólo se escriben —sin ser necesario— como referencias o aclaraciones de posición.

Hoy día, con el uso del ratón y la versatilidad de los menús desplegables y la barras de herramientas, el uso de las tabletas digitalizadoras se reduce escasamente a la digitalización o al “calcado” de planos y poco más.

UNO.2.7. *HELPSTRINGS; las cadenas de ayuda**

Si seguimos examinando detenidamente ACAD.MNU, nos encontramos, casi al final del mismo, con la sección *****HELPSTRINGS**. Esta sección es la encargada de recoger todos aquellos pequeños textos de ayuda a comandos que son identificados por su etiqueta ID. Esta línea de ayuda es la que se presenta en la línea de estado de **AutoCAD** cuando nos posicionamos con el cursor sobre un botón de barra de herramientas o sobre una opción de menú desplegable. La etiqueta ID es la que se ha comentado al hablar de la sección *****POP** y de la sección *****TOOLBARS**. Estas secciones, además de *****ACCELERATORS**, *****SCREEN** y *****TABLET**, son las únicas donde tienen sentido las etiquetas identificadoras.

La sintaxis bajo la sección *****HELPSTRINGS** es asaz sencilla:

etiqueta [*texto_de_ayuda*]

Así, los siguientes ejemplos son lo suficientemente demostrativos:

ID_Line	[Crea segmentos de línea recta: linea]
ID_Pan	[Desplaza la vista del dibujo en la ventana gráfica actual: encuadre]
ID_Replay	[Visualiza una imagen BMP, TGA o TIFF: reproducir]
ID_Rotate	[Desplaza los objetos alrededor de un punto base: gira]

La etiqueta `ID` ha de coincidir, evidentemente, con la que señalemos antes de los comandos correspondientes tanto en menús desplegables como en barras de herramientas. De este modo, la etiqueta `ID_Line` (*helpstring* de la orden de dibujo de líneas simples) del primer ejemplo, coincide exactamente con las etiquetas que se corresponden con la orden `LINEA` de la sección `***POP: ID_Line` `[&Línea]^C^C_line` y con el botón análogo de la sección `***TOOLBARS: ID_Line` `[_Button("Línea", ICON_16_LINE, ICON_24_LINE)]^C^C_line`.

Como vemos en estos casos del archivo de menús por defecto de **AutoCAD**, la cadena de ayuda está formada por un pequeño texto aclaratorio de la función del comando, además del correspondiente comando textual que habríamos de introducir por teclado para obtener el mismo efecto. Nosotros podemos escribir el texto que deseemos, siempre teniendo en cuenta no sobrepasar el ancho de la pantalla. Además, decir también que no es obligatorio que las etiquetas comiencen por `ID_`, pero que aclaran mucho la interpretación general de un archivo de menús si así lo hacen, ya que podríamos confundir comandos con líneas y etcétera.

UNO.2.8. Teclas rápidas bajo *****ACCELERATORS**

Bajo esta última sección del `ACAD.MNU` se encuentra la definición de teclas rápidas o aceleradoras Windows. El uso de estas combinaciones —o no— de teclas proporcionan al usuario un rápido acceso a los comandos más utilizados de **AutoCAD**.

Como sabemos, por ejemplo, la combinación `CTRL+A` produce los mismos efectos que el comando `ABRE` en la línea de comandos. También podemos acceder a este comando mediante el menú desplegable `Archivo>Abrir...` o a través del botón correspondiente en la barra de herramientas *Estándar* o en la tableta (casilla *T-25*).

La manera en que está especificada esta combinación de teclas rápidas en la sección `***ACCELERATORS` del archivo de menús `ACAD.MNU` es la siguiente:

```
ID_Open      [CONTROL+"A"]
```

Hay dos modos de definir teclas aceleradoras. La primera consiste en utilizar un identificador de etiqueta ya existente seguido de un título, entre corchetes, formado por uno o varios modificadores (separados mediante el símbolo de adición `+`) un símbolo `+` y carácter único o una cadena de tecla virtual especial entre comillas dobles. Es decir, con las siguientes sintaxis:

```
etiqueta     [modif+modif...+ "carácter"]  
etiqueta     [modif+modif...+ "tecla_virtual"]
```

Algunos ejemplos son los siguientes:

```
ID_Line      [CONTROL+"L"]  
ID_Circle     [CONTROL+SHIFT+"C"]  
ID_Cancel     ["ESCAPE"]  
ID_Ellipse    [SHIFT+"F11"]
```

La segunda forma consiste en utilizar un título con un modificador y una cadena de tecla seguido de una secuencia de comandos. Podemos definir así, combinaciones de teclas que no tienen referencia de etiqueta. Sus sintaxis generales podrían ser las siguientes:

```
[modif+modif...+"carácter"]^C^Ccomando_de_AutoCAD  
[modif+modif...+"tecla_virtual"]^C^Ccomando_de_AutoCAD
```

Algunos ejemplos:


```
[CONTROL+"Q"]^C^C_quit  
[SHIFT+CONTROL+"HOME"]^C^C_move  
["ESCAPE"]^C^C_quit
```

De esta forma indicamos el comando que queremos que se ejecute tras la pulsación de la combinación de teclas.

Los modificadores aceptados son los que se ven en la tabla siguiente:

Cadena	Descripción
CONTROL	Tecla Control de la mayoría de los teclados
SHIFT	Tecla Mayúsculas, izquierda o derecha
COMMAND	Tecla APPLE de los ordenadores Macintosh
META	Tecla META en teclado UNIX

Y las teclas virtuales especiales son las siguientes:

Cadena	Descripción
F1	Tecla F1
F2	Tecla F2
F3	Tecla F3
F4	Tecla F4
F5	Tecla F5
F6	Tecla F6
F7	Tecla F7
F8	Tecla F8
F9	Tecla F9
F10	Tecla F10
F11	Tecla F11
F12	Tecla F12
INSERT	Tecla Insert
DELETE	Tecla Supr
NUMPAD0	Tecla 0 del teclado numérico
NUMPAD1	Tecla 1 del teclado numérico
NUMPAD2	Tecla 2 del teclado numérico
NUMPAD3	Tecla 3 del teclado numérico
NUMPAD4	Tecla 4 del teclado numérico
NUMPAD5	Tecla 5 del teclado numérico
NUMPAD6	Tecla 6 del teclado numérico
NUMPAD7	Tecla 7 del teclado numérico
NUMPAD8	Tecla 8 del teclado numérico
NUMPAD9	Tecla 9 del teclado numérico
ESCAPE	Tecla Esc

NOTA: Como veremos mucho más adelante, las etiquetas o identificadores ID también sirven para poder cambiar el estado de una opción de menú (activado/desactivado o marcado/no marcado) desde macros de menús o mediante AutoLISP. También para su utilización como argumento en el acceso a archivos de ayuda personalizados (también se verá).

UNO.3. SUBMENÚS

Un submenú, como ya se ha comentado, es aquel que, debajo de una sección, define un nuevo menú o *subsección*. Estos submenús son identificables por comenzar con dos asteriscos (**) en lugar de tres (***), que era categoría de sección. A este identificador le sigue un nombre que identifica al submenú. Este nombre puede contener hasta 33 caracteres alfanuméricos, además de algunos especiales como \$ (símbolo de dólar), - (guión) o _ (guión de subrayado). El identificador debe estar en una sola línea del archivo de menú y puede contener espacios en blanco.

Todas las opciones que se escriban tras el identificador de submenú —hasta el siguiente identificador de submenú o de sección— pertenecen a dicho submenú. Los nombres de submenú habrán de ser únicos; no puede haber dos repetidos. Es posible, sin embargo, dar varios nombres a un mismo submenú.

En el archivo ACAD.MNU podemos identificar muchos submenús como los siguientes:

```
**SNAP
**INSERT
**TB_INQUIRY
**image_poly
**01_FILE
```

Todos ellos se encuentran en diferentes secciones de menú y tienen la propiedad de dividir la información en grupos de características homólogas. Así, bajo la sección *****TOOLBARS**, cada barra de herramientas definida se encuentra bajo un submenú.

Esto, en principio, no es obligatorio, pero es que la característica más importante de la división en submenús, reside en la posibilidad de hacer referencia a ellos desde cualquier punto del archivo de menús. Esto varía un poco en la secciones *****POP**, *****AUX**, *****BUTTONS** y *****TABLET**, ya que ellas no admiten submenús; los títulos que comienzan con ****** bajo estas secciones no son submenús, sino una especie de nombres mnemotécnicos para su mejor comprensión llamados, como ya se comentó, alias. Sin embargo pueden ser referenciadas, desde la versión 14 de **AutoCAD**, las propias secciones o, incluso, el nombre de alias.

UNO.3.1. Referencias a submenús

Como hemos apuntado, se puede llamar (activar) a un submenú desde cualquier punto del archivo de menú. La forma de hacerlo responde a la siguientes sintaxis:

\$inicial=nombre_grupo_menús.nombre_submenú

Esto es, primero el símbolo de dólar (\$) seguido de la inicial del identificador de sección al que pertenece el submenú, el símbolo de igual (=), el nombre del grupo de menús (identificador *****MENUGROUP=**), un punto (.) y, por último, el nombre completo del submenú referenciado.

Las iniciales de los identificadores de sección son las siguientes:

Iniciales	Sección
B1 – B4	***BUTTONS1 a ***BUTTONS4 (menús de pulsadores)
A1 – A4	***AUX1 a ***AUX4 (menús auxiliares)
P0	***POP0 (menú de cursor)
P1 – P16	***POP1 a ***POP16 (menús desplegables)
S	***SCREEN (menú de pantalla)

I	***IMAGE (menús de imágenes)
Iniciales	Sección
<hr/>	
T1 – T4	***TABLET1 a ***TABLET4 (menús de tableta)

Así, si quisiéramos realizar una llamada al submenú `**Bloques_Sanitarios` situado bajo la sección `***IMAGE` de un archivo del menú con nombre de grupo `***MENUGROUP=BLOQUES`, escribiríamos lo siguiente:

```
$I=BLOQUES.Bloques_Sanitarios
```

Las mayúsculas y minúsculas son indiferentes en todo el archivo `.MNU`, ya que **AutoCAD** convierte todo a mayúsculas —excepto contenidos textuales que serán mostrados como tales— al compilar el archivo (se explicará más adelante), pero es conveniente realizar distinciones entre secciones y submenús únicamente para obtener mayor claridad.

NOTA: Antiguamente no era necesario indicar el nombre de grupo en una llamada a un submenú, hoy sí. La razón estriba en que, como ya veremos, **AutoCAD** puede tener más de un archivo de menú cargado y, al referenciar un submenú, deberemos indicarle en qué archivo de menú de los cargados se encuentra. Si únicamente existiera un menú cargado, no haría falta indicar este nombre de grupo, pero siempre conviene hacerlo por los futuros menús parciales que se puedan cargar; evitando así errores venideros.

Cada vez que se activa un submenú (sólo los de pantalla), **AutoCAD** guarda en una pila el menú o submenú desde el cual se ha llamado al nuevo. Si se desea salir del nuevo submenú para volver al anterior (proceso muy frecuente), basta introducir en el archivo una llamada sin identificador de submenú. De esta forma, si se encontrara activado un submenú y se produjera una llamada a un nuevo submenú así:

```
$S=COTAS_HOR
```

el nuevo submenú queda activado y el anterior se almacena en pila. Si entre las opciones de este nuevo submenú se encuentra una llamada del tipo:

```
$S=
```

en el momento en que esa llamada es leída por el programa (al ejecutar la opción en que se encuentre), se abandona el submenú correspondiente mostrándose el anterior.

UNO.3.1.1. Llamadas a los submenús de las distintas secciones

Las referencias a los submenús se realizan de forma muy parecida en cada una de las secciones de un archivo de menú. Lo único que diferencia los procesos son pequeñas variaciones que vamos a comentar ahora.

Quando se invoca a un submenú que se encuentra bajo la sección `***SCREEN` de menú de pantalla, se da lugar a un despliegue en persiana que superpone al menú anterior. Por defecto, este despliegue se realiza desde la primera línea válida del menú, visualizándose los textos de las opciones, uno debajo de otro, en el orden en que fueron escritos. Si se quiere que esto no suceda así —esto ya se comentó— se puede especificar el número de línea donde queremos que empiece, después del identificador de submenú y tras un espacio:

```
**Ins_Bloq 3
```

De esta manera, si quisiéramos llamar a este submenú desde otro punto del archivo .MNU, normalmente desde una opción del propio menú de pantalla, podríamos hacer lo siguiente:

```
[Insertar]$S=BLOQUES.Ins_Bloq
```

Según este método, y siempre que el nombre de grupo del archivo de menús que contiene dicho submenú tuviera por nombre BLOQUES, al seleccionar (pinchar) la opción Insertar se activaría (se mostraría) el submenú ****Ins_Bloq** a partir de la tercera línea.

Podemos también realizar varias llamadas que, en el caso del menú de pantalla, habrán de superponerse sin taparse —añadiendo las líneas en blanco pertinentes— y, a la vez, ejecutar un comando:

```
[MATRIZ:]$S=X $S=ARRAY ^C^C_ARRAY
```

Las llamadas a submenús en el menú de tablero, secciones *****TABLET**, se utilizan de igual modo, pero hay existe una pequeña puntualización que se explica en la sección **UNO.4.3.3..**

Por su parte, los menús desplegables bajo las secciones *****POP**, tienen un tratamiento especial. No se admiten submenús en secciones *****POP**, pero se puede hacer referencia a la sección propiamente dicha, de la siguiente forma:

```
$Pn=*
```

siendo *n* el número de la sección *****POP**. Un ejemplo puede ser el siguiente:

```
[Desplegar Menú Herramientas]$P6=*
```

De esta forma, al seleccionar esta opción, se desplegará el menú correspondiente y se esconderá el actual (si se llama desde otra parte de una sección *****POP**).

NOTA: También podemos referenciar el alias de la sección.

Una línea en blanco después de cada submenú anula todas las opciones que puedan quedar del menú anterior. Para que todo menú referenciado anule completamente al anterior es buena costumbre incluir, al final de cada sección o submenú, la citada línea en blanco. Ejemplo:

```
***MENUGROUP=BLOQUES

***POP1
**Princ
[BLOQUES]
[Mecánica]$P2=*
[Electricidad]$P3=*
[Piping]$P4=*

***POP2
**Meca
[MECÁNICA]
[Tornillo]^C^C_insert Tornillo
[Tuerca]^C^C_insert Tuerca
[Arandela]^C^C_insert Arandela

***POP3
**Elec
[ELECTRICIDAD]
```

```
[Trafo]^C^C_insert Trafo
[Diodo]^C^C_insert Diodo
[Conmutador]^C^C_insert Conmut

***POP4
**Pip
[PIPING]
[Válvula]^C^C_insert Valvul
[Soplador]^C^C_insert Soplador
[Tubería]^C^C_insert Tubo
```

NOTA: Recordar dos cosas ya explicadas. La primera que es conveniente teclear las órdenes en inglés con el carácter de subrayado delante; de esta forma podrá ser interpretado por cualquier versión de **AutoCAD** cualquiera que sea su idioma. Y la segunda, distinguir secciones de submenús, o caracteres de control, de comandos, etcétera, con mayúsculas y minúsculas a nuestro gusto, por dar mayor claridad y comprensión. Por último, decir que ejemplos como el anterior no son muy utilizados, pero son perfectamente lícitos.

En los menús de imágenes bajo la sección *****IMAGE** las referencias a submenús se realizan de la forma siguiente. Para llamar a un submenú de imagen desde otro cualquiera (imagen, desplegable, pantalla...) hay que referenciarlo y después activarlo para que se visualice. Lo más lógico parece ser referenciar un menú de imagen desde un desplegable o un botón de barra de herramientas; de la siguiente manera, por ejemplo:

```
***MENUGROUP=ROCA

***POP1
**Sanitarios
[Sanitarios de Roca]
[Bloques]$I=ROCA.Bloques $I=ROCA.*

***TOOLBARS
**TB_Sanitarios
[_Button("Bloques", "ICON1.BMP", "ICON2.BMP")]$I=ROCA.Bloques $I=ROCA.*

***IMAGE
**Bloques
[Rocafot(foto-1,Lavabo)]^C^C_insert lavabo
[Rocafot(foto-2,Bañera)]^C^C_insert bañera
[Rocafot(foto-3,Bidé)]^C^C_insert bide
[Rocafot(foto-4,Inodoro)]^C^C_insert inodoro
```

Bajo las secciones *****BUTTONS** (menú de pulsadores) las llamadas a submenús se utilizan de igual modo, teniendo en cuenta, que si llamamos a un alias o una sección que esté bajo *****POP** o a un submenú que esté bajo *****IMAGE**, habremos de activarlos con los métodos explicados. En estas secciones de *****BUTTONS** no se admiten submenús.

Una clara referencia a submenús en el menú de pulsadores lo observamos en el archivo **ACAD.MNU** (el siguiente ejemplo puede variar en dicho archivo, ya que para el botón derecho se establece una condicional en **DIESEL**):

```
***BUTTONS1
;
$P0=*
```

Como ya hemos comentado, bajo *****BUTTONS1** se establecen las funciones de todos los pulsadores accionados de manera sencilla —comenzando por el segundo; el primero es para aceptar datos y no se puede redefinir—. En este caso del ratón del sistema, el segundo pulsador (botón derecho) realiza un **INTRO** —carácter punto y coma (;)— y, el tercero (botón

central) llama a la sección *****POP0** (con alias ****SNAP**); menú de cursor de modos de referencia a objetos.

Como vemos, en *****BUTTONS2** y *****BUTTONS3** se hace referencia al mismo submenú en la primera, y única, probablemente, línea; ciertamente, al pulsar **SHIFT + botón derecho** o **CTRL + botón derecho** se activa el menú de cursor. *****BUTTONS4**, normalmente, estará vacío.

Además, en los menús de pulsadores podemos utilizar la contrabarra para introducir las coordenadas del cursor automáticamente. Por ejemplo:

```
***BUTTONS2
_line \
```

La forma de actuar en menús bajo *****AUX** es la misma.

UNO.4. CARACTERES ESPECIALES, DE CONTROL Y OTROS MECANISMOS

A parte de todo lo estudiado, existen otro tipo de caracteres especiales, caracteres de control y otros mecanismos de edición de menús que se indican a continuación.

UNO.4.1. Caracteres especiales

En todas las secciones de un archivo de menú podemos incluir diversos caracteres especiales que **AutoCAD** reconoce como tales. Estos caracteres especiales son:

;	igual a un INTRO o a un espacio blanco
+	determina la continuidad en otra línea del archivo ASCII de una opción
\	interrumpe la opción para que el usuario introduzca un dato
-	antepuesto a un comando de doble formato (cuadro de diálogo y línea de comandos) hace que se ejecute el de la línea de comandos
_	antepuesto a un comando en inglés, hace que se admita el comando en cualquier versión idiomática de AutoCAD
<<	ángulos en grados sexagesimales
*	repetición de opciones completas de menú

AutoCAD añade automáticamente a cada opción de menú un espacio en blanco al final. Esto lo realiza para aceptar la orden como si hubiera sido introducida por teclado. Los espacios en blancos pueden ser sustituidos por caracteres punto y coma (;), siendo a veces necesaria esta elección. Por ejemplo, la siguiente opción de menú escribe un texto en pantalla:

```
[ACABADO]^C^C_text;80,20;3.5;0;Exento de rebabas;;;en cara vista;
```

El texto en cuestión comienza con el comando **_text**. Tras él, escribimos un **;** para aceptar el comando (=INTRO), luego las coordenadas del punto de inicio y otro **;** para entrarlas. Después, la altura del texto y la rotación con sus correspondientes **;** detrás de cada uno para aceptar. Más tarde, el texto en cuestión. Los tres caracteres **;** que le siguen son tres **INTROS** para, primero, entrar el texto; segundo, repetir la orden **TEXT0** y, tercero, situar el punto de

inicio bajo el anterior (formato párrafo alineado a la izquierda). Introducimos el final del texto y, luego, un ; para acabar la orden.

De esta forma, vemos que se hace necesario la introducción de un carácter ; al final de la opción, ya que, al estar escribiendo texto, AutoCAD interpretaría un espacio en blanco como lo que es: un espacio en blanco. Lo mismo ocurre con los tres ; seguidos.

NOTA: Como vemos, estos caracteres nos permiten crear verdaderas macroinstrucciones en opciones de menú. Es muy importante conocer bien las órdenes de **AutoCAD** y qué es exactamente lo que solicita el programa, así como el orden en el cual lo solicita, en cada momento. Por supuesto, es necesario dominar los comandos de teclado del programa, esto es, aquellos que sólo pueden ser introducidos por teclado o aquellos que, teniendo varios formatos, tienen un modo de teclado en el que se nos solicitan todos los datos en la línea de comandos.

Los caracteres ; del ejemplo anterior podían haber sido sustituidos por espacios blancos (excepto el último y los tres seguidos). Lo que ocurre es que se suelen alternar ambos métodos para dar mayor claridad a la macro.

Por su parte, el carácter +, se utiliza para separar macros u opciones de menú complicadas en varias líneas. Así, el ejemplo anterior podría haberse escrito:

```
[ACABADO]^C^C_text;80,20;3.5;0;+
Exento de rebabas;+
;;en cara vista;
```

El carácter \ detiene la ejecución de la macro y solicita un dato al usuario en un momento dado. Si en el ejemplo anterior queremos dejar a elección del usuario la entrada de un valor para las coordenadas de inicio del texto y otro para el ángulo de rotación, la macro habría quedado de la siguiente forma:

```
[ACABADO]^C^C_text;\3.5;\Exento de rebabas;;;en cara vista;
```

De esta manera, y tras ejecutarse la orden **TEXTO**, **AutoCAD** se detendría a preguntarnos por el dato donde esté el carácter contrabarra (\), es decir, las coordenadas de inicio. Nosotros, introduciríamos las coordenadas y pulsaríamos **ENTER**; la macro continúa y da el valor de 3.5 a la altura de texto, lo introduce o acepta (;) y vuelve a detenerse para preguntarnos el ángulo de rotación. Lo introducimos y pulsamos **INTRO** y, la macroinstrucción, continúa hasta el final como anteriormente.

Por último, decir que la introducción de los caracteres <<, precediendo al valor de un ángulo, hace que éste se tome en grados sexagesimales, con el origen y sentido normales de la trigonometría plana, independientemente de lo establecido con el comando **UNIDADES** de **AutoCAD**. Por ejemplo:

```
[ACABADO]^C^C_text 80,20 3.5 <<135 Exento de rebabas;;;en cara vista;
```

También existe el carácter * (asterisco) que repite la opción completa de un menú de forma automática —hasta que se pulsa **ESC**—. Es muy poco utilizado; la forma de manejarlo es la siguiente:

```
[Polilínea]^C^Cpol
```

NOTA: Del carácter de subrayado (_) ya se ha hablado ampliamente, por ello, no se explicará de nuevo aquí.

NOTA: En la líneas del archivo que terminan con un carácter de control (que veremos ahora), una contrabarra (\), un signo más (+) o un punto y coma (;), **AutoCAD** no añade espacios en blanco.

UNO.4.2. Caracteres de control

Además de los caracteres especiales, podemos incluir en archivos de menús diversos caracteres de control ASCII. Estos caracteres se introducen mediante el acento circunflejo —o signo de intercalación para **AutoCAD**— ^, seguido del carácter correspondiente. El carácter ^ representa la tecla CTRL, por ello se llaman caracteres de control a los formados por dicho símbolo y un carácter.

Nosotros ya conocemos uno, que es ^C. ^C representa un ESC o cancelación (viene de los tiempos de MS-DOS). Esta combinación anula órdenes en curso; algunas órdenes necesitan de dos cancelaciones para ser anuladas, por eso es conveniente, como ya hemos explicado, la introducción de ^C^C en todas las opciones de menú para que, de este modo, anulen por completo cualquier comando anterior, sea cual sea. Otros comandos, los cuales permiten una introducción transparente (ZOOM, ENCUADRE, CAL...) no necesitarán de estos caracteres y sí del oportuno de transparencia '.

Existen otros caracteres de control, como ^H que representa un retroceso de cursor, esto es, borra el último carácter introducido. Puede ser necesario en opciones que no deseemos que **AutoCAD** coloque un espacio blanco al final de la línea, y también en otros casos. Seguidamente se muestra una lista con todos los caracteres de control posibles:

^B	activa/desactiva el Forzcursor
^C	cancela el comando en curso (ESC)
^D	activa/desactiva coordenadas (CTRL+D)
^E	define el siguiente plano isométrico (CTRL+E)
^F	activa/desactiva la referencia a objetos (CTRL+F)
^G	activa/desactiva el modo Rejilla.
^H	ejecuta retroceso
^I	ejecuta un tabulado (TAB)
^M	ejecuta entrada de datos (INTRO)
^O	activa/desactiva el modo Orto (CTRL+O)
^P	activa/desactiva el eco de mensajes (MENUECHO)
^Q	transmite todos los mensajes, listas de estado y datos de entrada a la impresora (CTRL+Q)
^T	activa/desactiva la tableta (CTRL+T)
^V	cambia la venta gráfica actual (CTRL+V)
^Z	carácter nulo que suprime la adición automática de un espacio en blanco al final de un elemento de menú

Existe otro modo de activar o desactivar modos si no conocemos su carácter de control, y es introduciendo el comando en forma de macro (quizá no lo podamos hacer en una sola opción). Es decir, los dos grupos de opciones siguientes realizan la misma acción:

```
[Cambia Orto]^O  
  
[Cambia Orto ON]_ortho _on  
[Cambia Orto OFF]_ortho _off
```

NOTA: Como veremos, tenemos otros métodos condicionales para poder realizar lo mismo en una sola línea.

UNO.4.3. Otros mecanismos y apuntes

UNO.4.3.1. Opciones de menú mediante DIESEL

Sobre el lenguaje DIESEL hablaremos ampliamente en el **MÓDULO NUEVE** del presente curso, sin embargo, existe una forma de definir opciones de menú —casi siempre mediante métodos condicionales— por medio de este lenguaje. Estudiemos la siguiente proposición condicional:

```
[$(if,$(getvar,tilemode),~) Esp. Papel]espaciop
```

La opción propiamente dicha podría reducirse a: [Esp. Papel]espaciop, es decir, al pinchar sobre ella cambiaríamos al Espacio Papel de **AutoCAD**. El problema es que, como deberíamos saber, la posibilidad de cambiar de modo de espacio depende del valor de la variable TILEMODE (mantenida en esta versión del programa por razones de compatibilidad con versiones antiguas).

Lo que nos dice la condición es que, si TILEMODE tiene valor 1, se cumple y, entonces, se incluye el carácter tilde (~) que, como estudiamos, hace que la opción quede inutilizada y la escribe en un tono gris “apagado”.

NOTA: No debemos ahora preocuparnos por la sintaxis de la línea, ya que se estudiará a fondo en su parte correspondiente.

Existe la posibilidad de visualizar en las opciones de un menú una señal o marca indicativa (típica de Windows) de que está activada esa opción (✓). Para ello deberemos incluir en dicha opción el carácter de cerrar exclamación (!) seguido de un punto (.), de la siguiente manera:

```
[!.texto_de_la_opción]
```

En dispositivos gráficos que no admiten la señal o marca mencionada, se puede incluir otro carácter, por ejemplo:

```
[!ctexto_de_la_opción]
```

que visualizará una señal con forma de letra “c”.

La forma de utilizar esta señal consiste en incluirla en una condicional DIESEL que establezca o no su visualización. Por ejemplo:

```
[$(if,$(getvar,gridmode),!.)Rejilla]^G  
[$(if,$(getvar,orthomode),!.)Orto]^O  
[$(if,$(getvar,snapmode),!.)Forzcursor]^B
```

que visualizará una señal de activación en cada modo cuando se seleccione o esté activado.

Se puede combinar la utilización de !. y ~ para mostrar señales en opciones no disponibles de la siguiente forma:

```
[~!.texto_de_la_opción]
```

La visualización de opciones no disponibles y marcas aparecerá siempre que se despliegue el menú que las contiene. Cuando interesa que esa visualización se haga desde una llamada determinada a menú, y no en las demás, se debe indicar en la línea de llamada de la siguiente forma:

`$Pn.i=opciones`

`$Pn` es la abreviatura del desplegable; `i` es el número de orden de la opción cuya visualización se va a cambiar; `opciones` es el tipo de visualización: `~` o `!` o ambas. El número de orden (`i`) empieza a contar desde el nombre del submenú, el cual tendrá el número 1, hacia abajo, siendo las opciones consiguientes 2, 3, 4... En el siguiente submenú, por ejemplo, el número de orden de la opción [TC] es el 5.

```
**Bloques  
[Ladrillo]  
[Roblón]  
[TAR]  
[TC]  
[Arandela]  
...
```

Por ejemplo, una llamada `$P4.6=~!` visualiza como no disponible la opción numerada con el 6 del desplegable `***POP4` y le añade una señal. Para eliminar marcas y visualizaciones como no disponible, se incluirán llamadas de la forma, siguiendo con el anterior ejemplo:

`$P4.6=`

Si en vez de una llamada a un submenú concreto se desea llamar al último submenú o a la última opción utilizada, se puede emplear el carácter arroba (@) así:

`$P@.=opciones`

que se refiere a la última opción de menú desplegable utilizada, o así:

`$P@.i=opciones`

que se refiere a la opción de número `i` en el último menú desplegable utilizado (aunque no sea la última opción usada).

UNO.4.3.2. Variable `MENUCTL`

Si la variable de sistema `MENUCTL` tiene valor 1, se establece que, cuando se utilice una orden de **AutoCAD**, automáticamente en el menú de pantalla aparezca el submenú que corresponde a dicha orden.

UNO.4.3.3. Creación y uso de menús de macros

Ocurre con frecuencia que, en un archivo de menú, se repiten constantemente determinadas cadenas de texto a lo largo del mismo. El empleo de macros de texto permite abreviarlos definiendo una especie de abreviatura para que luego, cuando aparezca dicho texto en el menú, se sustituya por el nombre de la macro.

El archivo de menú que contiene macros es esencialmente igual que el `.MNU`, pero debe ser forzosamente guardado con la extensión `.MND`. Después habremos de compilarlo con la utilidad `MC.EXE` que proporciona **AutoCAD** y que trocará dicho archivo en otro del mismo nombre pero con extensión `.MNU`, esto es, apto para ser manejado por **AutoCAD**.

NOTA: `MC.EXE` se encuentra en el directorio `\SAMPLE\` de **AutoCAD**.

La manera de definir una macro de texto es la siguiente:

```
{nombre_macro}=texto
```

Las definiciones deben ocupar una sola línea y comenzar por el primer carácter de la primera línea del archivo. El nombre de la macro debe ir encerrado entre llaves ({}) y puede contener de uno a 31 caracteres en mayúsculas o minúsculas. *texto* es la cadena de caracteres que la macro sustituye.

La macro puede albergarse en cualquier punto del archivo de menús. En dicho punto, al ser compilado el archivo, se sustituirá por el texto completo con el que se ha definido. Un ejemplo de definición puede ser:

```
{imp}=capa;def;0;co;porcapa;tl;porcapa;
```

Se pueden definir, las macros, de forma recursiva, es decir, anidar unas dentro de otras, de la siguiente forma:

```
{rep}=zoom;e;{imp}
```

De esta forma, todas las macros anidadas se sustituirán por sus textos equivalentes, formando una macroinstrucción (no confundir con la macro de texto) de longitud variable.

Existe otra característica del compilador de menús muy importante, además del uso de las macros. Es la posibilidad de indicar el número de línea o líneas del archivo de menú en que se desea colocar una opción concreta. Esto es particularmente útil en las secciones de menú de tablero *****TABLET**. Como se ha explicado ya, una vez configurada cada área de tablero en un determinado número de casillas, será el orden en que aparecen las opciones en el archivo lo que determina a qué casilla concreta corresponde cada opción. Esto implica que si una orden debe ocupar más de una casilla en el tablero, toda la instrucción de menú se deba repetir en las líneas del archivo que correspondan.

También se puede evitar el tener que contar las líneas del archivo para saber a qué casilla corresponden. Para ello, basta con indicar al principio de cada opción la línea o líneas en que se desea situarla, de la forma siguiente:

```
<número,número...>instrucción
```

Por ejemplo, si se desea situar la orden **REDIBUJA** en las casilla 20, 21, 22, 98, 99 y 100 de un área de menú de tableta, se puede poner en cualquier lugar dentro de la sección *****TABLET**:

```
<20,21,22,98,99,100>redibuja
```

```
o
```

```
<20,21,22,98,99,100>_redraw
```

Al compilar el archivo **.MND**, el comando en cuestión se situará automáticamente en todas y cada una de las líneas especificadas por sus números de orden.

El archivo de menú suministrado por **AutoCAD** contiene un submenú alternativo llamado *****TABLET1ALT** perteneciente al área primera del tablero (*****TABLET1**) sin aplicaciones, totalmente reservado para opciones de menú que sean creadas por el propio usuario. La plantilla está pensada en principio para una configuración de 25 columnas por 9 filas, y una utilización prevista de 200 casillas.

El usuario puede personalizar esta área de menú editando el archivo ACAD.MND y modificando las líneas del submenú correspondiente. También se podría editar el ACAD.MNU, pero es preferible el .MND por la posibilidad comentada de referir casillas por sus números de orden.

El submenú **TABLET1ALT presenta en principio el siguiente aspecto:

```
<1>[T1-1]
<2>[T1-2]
<3>[T1-3]
<4>[T1-4]
...
<200>[T1-200]
```

Para incluir opciones de menú propias, basta con introducirlas a continuación de los números de casilla que interesen. Por ejemplo, si se desea que la casilla 176 contenga la orden REGEN, entonces la línea correspondiente del archivo quedará como sigue:

```
<176>[T1-176]^C^Cregen
```

Esto puede hacerse con todas las casillas necesarias. Una vez modificado el .MND, será preciso compilarlo de nuevo.

Por otro lado, el empleo de submenús de tablero es una característica que brinda grandes posibilidades a la hora de personalizar el tablero digitalizador para aplicaciones específicas. Es habitual encontrar un campo del diseño en el que se requiera el empleo de amplias bibliotecas de bloques para ser insertados. Es el caso, por ejemplo, de dibujos de electrónica, electricidad, *pipng*, etc..., y, en general, todos los dibujos de circuitería. En estos casos resulta particularmente útil el uso de submenús de tablero. Con ellos se puede habilitar por ejemplo el área de tableta primera, citada anteriormente, para diversos submenús.

Supongamos, por ejemplo, que se va a utilizar la primera área de tablero del menú suministrado por **AutoCAD**, con vistas a su empleo en diseños electrónicos. En primer lugar, se utilizarán las 200 casillas previstas para las opciones y procedimientos generales (control específico de capas, instrucciones espaciales de dibujo, de edición, de visualización, utilidades de AutoLISP, etcétera).

Una plantilla colocada encima de esa área informará con texto o símbolos de la opción contenida en cada casilla. En el momento en que se precise insertar un bloque predefinido de una biblioteca de bloques (un transistor, un diodo, una puerta lógica, etc.), se puede llamar a un submenú, dentro de esa misma sección, que habilite otras 200 casillas para nuevas opciones de inserción de bloques.

Supongamos que el nombre del submenú es BL-ELEC y la opción que permite llamarlo se sitúa en la casilla 200. Habrá que escribir en esa línea del submenú **TABLET1ALT lo siguiente:

```
<200>[T1-200]$T1=bl-elec
```

Cada vez que sea seleccionada la casilla 200, se activará el submenú BL-ELEC que, en este caso, habilitará las casillas para insertar bloques de la biblioteca. Bastará entonces colocar una nueva plantilla sobre el área de menú, con los dibujos o los nombres de los bloques impresos en ella, para que se sepa en qué casilla se inserta cada uno.

El submenú creado BL-ELEC se situará en el archivo de menú al final de la sección ***TABLET1 y contendrá todas las opciones de inserción de bloques. Por ejemplo:

```
**BL-ELEC
<1,2,26,27>^C^C_insert;trans \l;i;0
<3,4,28,28>^C^C_insert;diodo;\l 0
<5,6,7,8,30,31,32,33>^C^C_insert nano \l;i;0
...
```

Lógicamente, cada submenú llamado debe contener opciones que permitan pasar de un submenú a otro y, también, al principal. Si se utilizan las últimas casillas, éstas podrían contener, por ejemplo:

```
<199>$T1=
<200>$T1=princ
```

de forma que, pinchando sobre la casilla 199 se volverá al submenú precedente y, sobre la 200, al principal. Para ello es preciso añadir inmediatamente después del identificador de sección *****TABLET1** la línea ****PRINC**.

Sería útil que en la pantalla se visualizara un mensaje que indicara cuándo se ha entrado en un nuevo submenú de tablero, como advertencia para cambiar de plantilla. Esto lo podemos realizar con una instrucción de AutoLISP (**ALERT**) que ya se estudiará en el momento adecuado.

UNO.4.3.3.1. Funcionamiento de MC.EXE

El compilador **MC.EXE** (en **\SAMPLE**) se emplea desde **MS-DOS**. Su sintaxis es la siguiente:

```
MC [opciones] nombre_archivo
```

nombre_archivo es el nombre del fichero **.MND** que contiene las macros de texto (no hay que especificar la extensión). Las opciones del comando no son obligatorias y son las que siguen:

```
-D      Lista definiciones de macros
-I      Lista texto de entrada
-M      Desactiva característica de macro
```

Indicar solamente **MC** da como resultado un texto de ayuda explicativo.

Ejemplos:

```
MC TOPOGRAF
MC C:\ACAD\MIPRO\MEN\BLOQUES
MC -D PIPING
MC -D -M A:\MENÚS\CARTOG
```

UNO.4.4. Uso de la orden externa EDIT

Esta orden permite el uso del editor del sistema operativo **MS-DOS** (**Edit**) sin interrumpir la edición del dibujo en curso. Es muy útil para la edición de menús de forma ágil.

UNO.5. CARGA Y DESCARGA DE MENÚS EN AutoCAD

Como hemos aprendido, un archivo de menú de **AutoCAD** consiste en una serie de secuencias de comandos que se ejecutan según determinadas condiciones. Existen diversos archivos de menú, algunos ASCII que podemos modificar y otros no. Los diversos archivos son los siguientes:

Extensión	Descripción
.MNU	Llamado menú de plantilla por ser el archivo de menús base. Es un archivo ASCII y podemos generarlo o modificarlo a nuestro gusto pero, al tenerlo totalmente terminado, es preferible no volver a tocarlo por si deshacemos algo importante, y realizar las subsiguientes pruebas en el .MNS.
.MNS	Archivo de menú fuente. Generado por AutoCAD con el mismo nombre que el .MNU y muy parecido a éste. Lo único que varía es la falta de la mayoría de los comentarios y de las secciones no utilizadas. Es el archivo que debemos modificar al realizar las pruebas y el que modifica AutoCAD al, por ejemplo, crear una barra de herramientas o un botón desde el propio programa.
.MNC	Archivo compilado (por lo tanto no es ASCII) directamente del .MNU, que crea AutoCAD al cargar un .MNU para manejarlo más rápidamente.
.MNR	Archivo llamado de recursos de menú. En él se albergan todos los mapas de bits que utiliza el menú en cuestión, por ejemplo, los de los iconos creados para botones de barras de menús.
.MNL	Archivo ASCII de menú con rutinas en AutoLISP que necesita el archivo de menú en cuestión creado.
.MND	Archivo ASCII de definición de menús. Contiene macros para utilizar con el menú creado. Debe ser compilado externamente mediante la utilidad MC.EXE.

La secuencia de carga y obtención de los diferentes archivos de menú por parte de **AutoCAD**, al indicarle un nombre de archivo, es la siguiente:

1. **AutoCAD** empieza buscando el archivo fuente .MNS. En caso de localizarse, busca a continuación el archivo de menú compilado .MNC con el mismo nombre. Si su fecha y hora es posterior al .MNS, se carga. En caso de no encontrarse, o de ser su fecha y su hora anterior, se compila automáticamente y se obtiene un menú .MNC actualizado, que es el que carga.
2. Si **AutoCAD** no ha podido encontrar el nombre de menú .MNS especificado, intenta localizar un archivo .MNC con el mismo nombre y lo carga.
3. Si **AutoCAD** no encuentra ningún menú .MNS ni .MNC con el nombre especificado, busca un archivo de plantilla .MNU con el mismo nombre. Si lo encuentra, crea un .MNS y compila un .MNC, cargando este último.
4. Una vez cargado un archivo .MNC con cualquiera de los procedimientos anteriores, **AutoCAD** genera el archivo de recursos .MNR y busca un archivo .MNL con el mismo nombre (si se necesita). Si lo encuentra evalúa las expresiones AutoLISP contenidas en él.
5. Si **AutoCAD** no encuentra ningún archivo .MNS, .MNC o .MNU con el nombre especificado muestra un mensaje de error y solicita otro nombre de menú para cargar.

Tenemos dos métodos para cargar un menú: realizar una carga completa o una carga parcial. La carga completa se realiza con el comando MENU de **AutoCAD** desde la línea de comando. Por defecto **AutoCAD** buscará archivos .MNS o .MNC. Podemos indicarle el tipo .MNU en la casilla del cuadro de diálogo *Archivos del tipo:*. Al cargar un archivo de menú con esta

orden, el nuevo menú sustituirá por completo al actual. Si cargamos un archivo de menús de plantilla (.MNU), **AutoCAD** muestra una advertencia sobre la necesidad de sobrescribir y redefinir el archivo fuente .MNS. Si aceptamos, este archivo se renovará y perderemos los cambios realizados en él.

Esto nos lleva a recordar que el archivo .MNS es en el que podemos realizar todos los cambios y modificaciones que deseemos en un menú. Si perdemos este fichero o lo modificamos de forma que no es de nuestro agrado, siempre tendremos en archivo de plantilla .MNU para recuperar el original. Debemos advertir también que, las modificaciones de barras de herramientas realizadas desde **AutoCAD** (que ya veremos) se guardan, como ya explicamos, en el archivo .MNS, por ello, al cargar el .MNU se perderán. Si queremos conservarlas deberemos escribirlas (copiar/cortar y pegar) en el archivo de plantilla .MNU.

Por su lado, el método de carga parcial de menús proporciona un mayor control a la hora de cargar un archivo. Este método se realiza con el comando CARGARMENU (MENULOAD en inglés) desde la línea de comando, desde Herr.>Personalizar menús... o pinchando en la casilla Y-9 del tablero digitalizador.

De esta forma se carga un menú parcial que se añade al menú actual base (ACAD.MNU u otro cargado mediante la orden MENU). Las secciones de estos menús (***POP, ***TOOLBARS, ***IMAGE o ***HELPSTRINGS) permiten añadir o reorganizar los menús del menú base sin modificar el resto de secciones de éste.

El comando visualiza un cuadro de diálogo con dos pestañas que permiten gestionar los menús que serán cargados. El procedimiento que debemos seguir para cargar un menú es el siguiente:

1. En la pestaña Grupos de menús pinchar en Examinar... y buscar el archivo que queramos cargar. Por defecto se buscan archivos .MNS y .MNC. Podemos cambiar esto, con las mismas consecuencias explicadas en la carga de menús base.
2. Una vez elegido el fichero, pinchar en Cargar para cargar el menú. Debe aparecer en el cuadro superior que lleva por título Grupos de menús: (con el nombre del ***MENUGROUP=).
3. Pasar a la siguiente pestaña (Barra de menús) y elegir, si no estuviera ya, el menú cargado elegido en la lista desplegable Grupo de menús:.
4. Bajo el cuadro Menús: escoger el/los que deseamos incluir en la barra de menús.
5. Bajo el cuadro Barra de menús: escoger el menú de la barra actual delante del cual queremos insertar el elegido en el punto anterior (normalmente irá antes del último, el de ayuda, ?).
6. Pinchar en Insertar>>.
7. Pinchar en Cerrar.

Tras todo este proceso, el menú estará cargado y operativo.

La manera de descargar un menú es análoga. El comando por teclado de DESCARGARMENU (MENUUNLOAD en inglés). Desde la barra de menús y demás se accede de igual forma que para cargar un menú parcial.

Para descargar un menú deberemos designarlo bajo Grupo de menús: en la pestaña Grupo de menús y pinchar en Descargar. Podemos también suprimir ciertas porciones de menús desde la otra pestaña pinchando en <<Suprimir.

La finalidad de que existan dos comandos por teclado para el mismo cuadro de diálogo es para su utilización en línea de comandos (por el usuario, macros, rutinas AutoLISP...).

NOTA: Si queremos cargar un menú que ya está cargado, hemos de descargarlo antes.

NOTA: Podemos especificar o cambiar los caminos de búsqueda o rutas de acceso a archivos de menú desde Herr.>Preferencias..., en la pestaña *Archivos* y bajo *Archivos de menú, ayuda, registro y otros>Archivo de menús*.

UNO.6. EJEMPLOS PRÁCTICOS DE MENÚS

UNO.6.1. Menú desplegable simple

```
***MENUGROUP=SIMPLE

***POP1
[Me&nú Simple]
[->&Dibujo]
    [&Línea]^C^C_line
    [&Polilínea]^C^C_pol
    [&Círculo]^C^C_circle
    [&Arco]^C^C_arc
    [P&olígono]^C^C_polygon
    [&Elipse]^C^C_ellipse
    [&Spline]^C^C_spline
    [Pu&nto]^C^C_point
    [<-&Texto]^C^C_mtext
    [--]
    [->&Edición]
    [&Borra]^C^C_erase
    [Co&pia]^C^C_copy
    [&Simetría]^C^C_mirror
    [&Equidistancia]^C^C_offset
    [&Matriz]^C^C_array
    [&Desplaza]^C^C_move
    [&Gira]^C^C_rotate
    [&Recorta]^C^C_trim
    [&Alarga]^C^C_extend
    [C&haflán]^C^C_chamfer
    [E&mpalme]^C^C_fillet
    [<-Desco&mponer]^C^C_explode
    [--]
[->&Visualización]
    [->&Zoom]
        [&Ventana]'_zoom _w
        [&Dinámico]'_zoom _d
        [&Todo]'_zoom _a
        [&Extensión]'_zoom _e
        [<-Tiempo &Real]'_zoom ;
        [--]
    [<-&Encuadre Tiempo Real]'_pan
```

NOTAS INTERESANTES:

1. Nótese que no importa que el desplegable se llame ***POP1 aunque exista un ***POP1 en el menú base y éste lo carguemos como parcial. Cada ***POP1 corresponde a su menú y no habrá conflictos.
2. La opción *Tiempo real* del comando *ZOOM* se ejecuta con el propio comando (*_zoom*), un espacio para aceptar el comando y un *;* (que podría ser otro espacio) como *INTRO* para aceptar la opción por defecto de *ZOOM*, que es *Tiempo real*.
3. Pruébese a cambiar la opción *[<-Tiempo &Real]* por *[<-<-Tiempo &Real]* para diferenciar el efecto y comprender el mecanismo perfectamente.

4. Si no se incluye un *****MENUGROUP=** —aunque sugerimos incluirlo siempre, y a veces es necesario— se tomará, como nombre de grupo, la ruta y nombre de archivo de menú.

UNO.6.2. Menús desplegables

```
***MENUGROUP=DESPLEG

***POP1
**Bloques
    [&Bloques]
    [->&Puertas]
ID_phj1      [&1 Hoja]^C^C_insert hj1.dwg
ID_phj2      [<-&2 Hojas]^C^C_insert hj2.dwg
    [->&Ventanas]
ID_vhj1      [&1 Hoja]^C^C_insert hj1.dwg
ID_vhj2      [<-&2 Hoja]^C^C_insert hj2.dwg
    [->&Mobiliario]
ID_silla     [&Silla]^C^C_insert silla.dwg
ID_mesa      [&Mesa]^C^C_insert mesa.dwg
ID_camilla   [Mesa &Camilla]^C^C_insert camilla.dwg
ID_sofa      [So&fá]^C^C_insert sofa.dwg
ID_armar     [<-&Armario]^C^C_insert armar.dwg

***POP2
**Modificar
    [Herramientas B&loques]
ID_crea      [&Crear bloque...]^C^C_bmake
ID_limpia    [&Limpiar bloque]^C^C_purge b \_n;
    [--]
    [->&Otras]
ID_Edatr     [Definir a&tributos...]^C^C_ddattdef
    [->&Base]
ID_base00    [Base en el &0,0]^C^C_base 0,0;
ID_baseus    [<-&Base de &usuario]^C^C_base
    [--]
ID_borrtd    [&Borrar todo]^C^C_erase _all;;

***HELPSTRINGS
ID_phj1      [Inserta una puerta de 1 hoja.]
ID_phj2      [Inserta una puerta de 2 hojas.]
ID_vhj1      [Inserta una venta de 1 hoja.]
ID_vhj2      [Inserta una venta de 2 hojas.]
ID_silla     [Inserta una silla.]
ID_mesa      [Inserta una mesa cuadrada.]
ID_camilla   [Inserta una mesa camilla.]
ID_sofa      [Inserta un sofá.]
ID_armar     [Inserta un armario empotrado.]
ID_crea      [Define un bloque.]
ID_limpia    [Limpia el bloque o los bloques especificados.]
ID_Edatr     [Define atributos para un bloque.]
ID_base00    [Sitúa el punto base de inserción en el 0,0,0.]
ID_baseus    [Sitúa el punto base de inserción en el indicado por el usuario.]
ID_borrtd    [Borra todos lo objetos del dibujo actual.]
```

NOTAS INTERESANTES:

1. Evidentemente, los bloques habrán de estar en alguno de los directorios de soporte de **AutoCAD** o en algún otro especificado en alguna de las rutas de búsqueda bajo *Herr.>Preferencias...*, en la pestaña de *Archivos*. Se puede añadir un directorio a estas rutas de búsqueda (normalmente a los archivos de soporte).
2. Nótese que se pueden ordenar las diferentes opciones de menú mediante tabuladores y/o espaciados, pero las etiquetas siempre han de estar a partir del primer carácter de línea de menú.
3. No es necesario que las etiquetas de un menú comiencen por ID_ (de Identificador), pero parece una forma lógica y simple de distinguirlas a simple vista y, así, llevar una mejor organización del archivo de menú.
4. Revísense y asimílense las tres pequeñas macroinstrucciones (que casi ni lo son) de este menú. Para realizarlas hay que seguir estrictamente la secuencia que se reproduciría al introducir las órdenes y opciones por teclado (INTROS incluidos).

UNO.6.3. Menú de imagen y desplegable

```
//Aquí comienza el archivo de menús.
```

```
***MENUGROUP=MOBIL
```

```
//Ahora sigue el menú desplegable.
```

```
***POP12
```

```
[&Mobiliario]
```

```
[->&Cocina]
```

```
  [&Mesas]$I=MOBIL.CocinaMesa $I=MOBIL.*
```

```
  [&Sillas]$I=MOBIL.CocinaSilla $I=MOBIL.*
```

```
  [<-M&uebles]$I=MOBIL.CocinaMueble $I=MOBIL.*
```

```
[->&Salón]
```

```
  [&Sofás]$I=MOBIL.SalónSofá $I=MOBIL.*
```

```
  [&Mesas] $I=MOBIL.SalónMesa $I=MOBIL.*
```

```
  [<-M&uebles]$I=MOBIL.SalónMueble $I=MOBIL.*
```

```
[->&Baño]
```

```
  [&Lavabos]$I=MOBIL.BañoLavabo $I=MOBIL.*
```

```
  [&Bañeras]$I=MOBIL.BañoBañera $I=MOBIL.*
```

```
  [Bi&dés]$I=MOBIL.BañoBidé $I=MOBIL.*
```

```
  [&Inodoros]$I=MOBIL.BañoInodoro $I=MOBIL.*
```

```
  [<-&Muebles]$I=MOBIL.BañoMueble $I=MOBIL.*
```

```
[->Habitación]
```

```
  [&Mesillas]$I=MOBIL.HabitaciónMesilla $I=MOBIL.*
```

```
  [&Camas]$I=MOBIL.HabitaciónCama $I=MOBIL.*
```

```
  [<-<-M&uebles]$I=MOBIL.HabitaciónMueble $I=MOBIL.*
```

```
//Lo que viene ahora es el menú de imagen.
```

```
***IMAGE
```

```
**CocinaMesa
```

```
[Mesas de cocina]
```

```
[mobfot(mesc01,Mesa cuadrada)]^C^C_insert C:/BIBLIO/BLOQ/COCINA/MESA/mesc01.dwg
```

```
[mobfot(mesc02,Mesa rectag.)]^C^C_insert C:/BIBLIO/BLOQ/COCINA/MESA/mesc02.dwg
```

```
[mobfot(mesc03,Mesa circular)]^C^C_insert C:/BIBLIO/BLOQ/COCINA/MESA/mesc03.dwg
```

```
[mobfot(mesc04,Mesa extens.)]^C^C_insert C:/BIBLIO/BLOQ/COCINA/MESA/mesc04.dwg
```

```
**CocinaSilla
```

```
[Sillas de cocina]
```

```
[mobfot(silc01,Silla 1)]^C^C_insert C:/BIBLIO/BLOQ/COCINA/SILLA/silc01.dwg
```

```
[mobfot(silc02,Silla 2)]^C^C_insert C:/BIBLIO/BLOQ/COCINA/SILLA/silc02.dwg
```

Curso Práctico de Personalización y Programación bajo AutoCAD

Personalización de menús

```
[mobfot(silc03,Silla 3)]^C^C_insert C:/BIBLIO/BLOQ/COCINA/SILLA/silc03.dwg

**CocinaMueble
[Muebles de cocina]
[mobfot(mobc01,Mueble 1)]^C^C_insert C:/BIBLIO/BLOQ/COCINA/MUEBL/mobc01.dwg
[mobfot(mobc02,Mueble 2)]^C^C_insert C:/BIBLIO/BLOQ/COCINA/MUEBL/mobc02.dwg

**SalónSofá
[Sofás]
[mobfot(sofs01,Sofá dos piezas)]^C^C_insert c:/BIBLIO/BLOQ/SAL/SOF/sofs01.dwg
[mobfot(sofs02,Sofá una pieza)]^C^C_insert c:/BIBLIO/BLOQ/SAL/SOF/sofs02.dwg
[mobfot(sofs03,Sofá en ángulo)]^C^C_insert c:/BIBLIO/BLOQ/SAL/SOF/sofs03.dwg
[mobfot(sofs04,Esquinero)]^C^C_insert c:/BIBLIO/BLOQ/COCINA/SOF/sofs04.dwg

**SalónMesa
[Mesas de salón]
[mobfot(mess01,Mesa ovalada)]^C^C_insert c:/BIBLIO/BLOQ/SAL/MESA/mess01.dwg
[mobfot(mess02,Mesa rectang.)]^C^C_insert c:/BIBLIO/BLOQ/SAL/MESA/mess02.dwg

**SalónMueble
[Muebles de salón]
[mobfot(mobs01,Mueble telev.)]^C^C_insert c:/BIBLIO/BLOQ/SAL/MUEBL/mobs01.dwg
[mobfot(mobs02,Mueble 3 modul.)]^C^C_insert c:/BIBLIO/BLOQ/SAL/MUEBL/mobs02.dwg

**BañoLavabo
[Lavabos]
[mobfot(lavb01,ROCA-1)]^C^C_insert c:/BIBLIO/BLOQ/BAÑO/LAV/lavb01.dwg
[mobfot(lavb02,ROCA-2)]^C^C_insert c:/BIBLIO/BLOQ/BAÑO/LAV/lavb02.dwg
[mobfot(lavb03,ROCA-3)]^C^C_insert c:/BIBLIO/BLOQ/BAÑO/LAV/lavb03.dwg

**BañoBañera
[Bañeras]
[mobfot(bañb01,ROCA-1)]^C^C_insert c:/BIBLIO/BLOQ/BAÑO/BAÑERA/bañb01.dwg
[mobfot(bañb02,ROCA-2)]^C^C_insert c:/BIBLIO/BLOQ/BAÑO/BAÑERA/bañb02.dwg
[mobfot(bañb03,ROCA-3)]^C^C_insert c:/BIBLIO/BLOQ/BAÑO/BAÑERA/bañb01.dwg

**BañoBidé
[Bidés]
[mobfot(bidb01,ROCA-1)]^C^C_insert c:/BIBLIO/BLOQ/BAÑO/BID/bidb01.dwg
[mobfot(bidb02,ROCA-2)]^C^C_insert c:/BIBLIO/BLOQ/BAÑO/BID/bidb02.dwg
[mobfot(bidb03,ROCA-3)]^C^C_insert c:/BIBLIO/BLOQ/BAÑO/BID/bidb03.dwg

**BañoInodoro
[Inodoros]
[mobfot(inob01,ROCA-1)]^C^C_insert c:/BIBLIO/BLOQ/BAÑO/INOD/inob01.dwg
[mobfot(inob02,ROCA-2)]^C^C_insert c:/BIBLIO/BLOQ/BAÑO/INOD/inob02.dwg
[mobfot(inob03,ROCA-3)]^C^C_insert c:/BIBLIO/BLOQ/BAÑO/INOD/inob03.dwg

**BañoMueble
[Muebles de baño]
[mobfot(mobb01,Mueble 1)]^C^C_insert c:/BIBLIO/BLOQ/BAÑO/MUEBL/mobb01.dwg
[mobfot(mobb02,Mueble 2)]^C^C_insert c:/BIBLIO/BLOQ/BAÑO/MUEBL/mobb02.dwg

**HabitaciónMesilla
[Mesillas de noche]
[mobfot(mesh01,Mesilla cuadrada)]^C^C_insert c:/BIBLIO/BLOQ/HAB/MESA/mesh01.dwg
[mobfot(mesh02,Mesilla circular)]^C^C_insert c:/BIBLIO/BLOQ/HAB/MESA/mesh02.dwg

**HabitaciónCama
[Camas]
[mobfot(camh01,Cama 1.20)]^C^C_insert c:/BIBLIO/BLOQ/HAB/CAMA/camh01.dwg
```

```
[mobfot(camh02,Cama 1.60)]^C^C_insert c:/BIBLIO/BLOQ/HAB/CAMA/camh02.dwg
[mobfot(camh03,Cama 1.90)]^C^C_insert c:/BIBLIO/BLOQ/HAB/CAMA/camh03.dwg
[mobfot(camh04,Cama 2.10)]^C^C_insert c:/BIBLIO/BLOQ/HAB/CAMA/camh04.dwg

**HabitaciónMueble
[Muebles de habitación]
[mobfot(mobh01,Mueble 1)]^C^C_insert c:/BIBLIO/BLOQ/HAB/MUEBL/mobh01.dwg
[mobfot(mobh02,Mueble 2)]^C^C_insert c:/BIBLIO/BLOQ/HAB/MUEBL/mobh02.dwg

//Fin del archivo de menús.
```

NOTAS INTERESANTES:

1. En este ejemplo vemos claramente la utilización de llamadas a submenús de imágenes, con su correspondiente activación. Pruébese a quitar dicha activación de alguna de las llamadas para comprobar el resultado.
2. Las llamadas a los diferentes submenús de imágenes se realizan desde un menú desplegable. Es posible, también, llamar a un menú de imágenes desde otro menú de imágenes. Para hacer esto, en lugar de asociar a cada imagen un comando de inserción de bloque, como en este ejemplo, debemos asociarle una referencia a otro submenú; sencillo.
3. Los comentarios con las dos barras inclinadas (//) pueden incluirse en cualquier punto del archivo de menú, como podemos apreciar.
4. En el menú de imagen y según lo estudiado, mobfoto es la librería de fotos, o fototeca, donde se encuentran todas las fotos guardadas; la siguiente cadena (tras la apertura de paréntesis) es el nombre de la foto incluida en la fototeca (mesc01, mess02, mobs02, camh04...); la cadena siguiente (tras la coma) es el nombre que veremos en la lista de la izquierda del menú de imagen. Después de todo esto, viene el comando de inserción de bloques con la ruta de acceso y el bloque propiamente dicho que se insertará. En este caso, se han hecho las fotos con el mismo nombre que los bloques a los que corresponden, pero, evidentemente, con extensión .SLD. Véase el **MÓDULO OCHO**, el cual se refiere a la obtención de fotos y a la creación de bibliotecas de fotos.
5. Apréciase que los caminos o rutas de acceso a archivos deben introducirse con barras inclinadas normales (/), y no con barras inversas o contrabarras (\) como es habitual en sistemas MS-DOS y Windows. Esta notación —que recuerda a UNIX y, por ende, a la manera de indicar URLs en los navegadores Web, direcciones FTP, etcétera— se debe aquí a la necesidad de la contrabarra o barra inversa (\) como símbolo especial para entrada de datos por parte del usuario en una macroinstrucción (ya estudiado).
6. Evidentemente, tanto archivos .DWG como la fototeca .SLB deberán estar en los directorios indicados, en el directorio actual o en algún directorio o ruta de soporte de **AutoCAD**.

UNO.6.4. Menú completo de cartografía

```
//
// AutoCAD archivo de menús - C:\MENÚS\CARTOGR.MNU
//
// NOTA IMPORTANTE:
// Para garantizar el acceso a los archivos de soporte de este menú,
// por medio del comando PREFERENCIAS del menú HERR, en la pestaña
// ARCHIVOS se debe añadir como CAMINO DE BÚSQUEDA DE ARCHIVO DE SOPORTE
// el siguiente:
// C:\MENÚS
***MENUGROUP=CARTOGR
```

Curso Práctico de Personalización y Programación bajo AutoCAD

Personalización de menús

```
***POP1
[Utils-Carto]
ID_Pol_Grosor [Polilínea grosor 0]^C^C_pline \_w 0 0
ID_Pol_Curl [Pol curvas1]^C^C_layer _set curvas1;;_pline \_w 0 0
ID_Pol_Cur10 [Pol curvas10]^C^C_layer _set curvas10;;_pline \_w .5 .5
[--]
ID_Borrar1 [Borrar 1]^C^C_erase \;
ID_Editpol1 [Editpol]^C^C_pedit $s=editpol
ID_Curva_B [Curva B]^C^C_pedit \s;;
ID_Copias [Copias]$S=X $s=designa ^C^C_select \_copy _p;i_m
ID_Partel [Parte @]^C^C_break \_f \@
[->Cambiar]
ID_Cambiar [Cambiar]$S=X $s=designa ^C^C_change
ID_Cambiar_col [Cambiar color]$S=X $s=designa _select \_change _p;i_p _col;\;
ID_Cambiar_txt [Cambiar texto]^C^C_change;\;i;i;i;i;i
ID_Cambiar_capa [C<-Cambiar capa]$S=X $s=designa ^C^C_select \_change _p;i_p _layer;\;
[--]
ID_ayuda_cart [Ayuda]^C^C(help "c:/menús/cartograf.ahp")

***POP2
[Bloques-Carto]
ID_Insertar [INSERTAR]^C^C_DDINSERT
[ banco]^C^C_insert banco
[ farola]^C^C_insert farola
[ quiosco]^C^C_insert kiosko
[ pozo]^C^C_insert pozo
[ árbol]^C^C_insert arbol
[ cabina]^C^C_insert cabina
[ fuente]^C^C_insert fuente
[ caseta]^C^C_insert caseta
[ torre]^C^C_insert torre
[ poste]^C^C_insert poste
[ ermita]^C^C_insert ermita

***POP3
[Capas-Carto]
ID_Capas [Gestión de Capas]^C^C_DDLMODES
[ Edificios]^C^C_layer _set edificios;;
[ Medianerías]^C^C_layer _set medianerías;;
[ Aceras]^C^C_layer _set aceras;;
[ Curvas1]^C^C_layer _set curvas1;;
[ Curvas10]^C^C_layer _set curvas10;;
[ Carreteras ]^C^C_layer _set carreteras;;
[ Caminos]^C^C_layer _set caminos;;
[ Ferrocarril]^C^C_layer _set ferrocarril;;
[ Ríos]^C^C_layer _set rios;;
[ Arroyos]^C^C_layer _set arroyos;;
[ Arbolado]^C^C_layer _set arbolado;;
[ Cultivos]^C^C_layer _set cultivos;;
[ T. eléctrico]^C^C_layer _set t_electrico;;

***TOOLBARS
**CARTOGRAFIA
**CARTOGRAFÍA
ID_Cartografía [_Toolbar("Cartografía", _Floating, _Show, 238, 156, 1)]
ID_Línea_1 [_Button("Línea", "ICON_16_LINE", "ICON_24_LINE")]^C^C_line
ID_Centro_Radio_1 [_Button("Centro Radio", "ICON_16_CIRRAD", "ICON_24_CIRRAD")]^C^C_circle
ID_Polilínea_1 [_Button("Polilínea", "ICON_16_PLINE", "ICON_24_PLINE")]^C^C_pline
[--]
ID_Borrar_1 [_Button("Borrar", "ICON_16_ERASE", "ICON_24_ERASE")]^C^C_erase
ID_Editar_polilínea_1 [_Button("Editar polilínea", "ICON_16_PEDIT", "ICON_24_PEDIT")]^C^C_pedit
ID_Equidistancia_1 [_Button("Equidistancia", "ICON_16_OFFSET", "ICON_24_OFFSET")]^C^C_offset
[--]
ID_inser_cart [_Button("Símbolos", "ICON5705.bmp", "ICON_24_BLANK")]^C^C$I=cartogr.Simb_carto $I=*
[--]
ID_ayuda_cart [_Button("Ayuda", "ICON4464.bmp", "ICON_24_BLANK")]^C^C(help "c:/menús/cartograf.ahp")
[--]
ID_Ctrl_Lyr_1 [_Control(_Layer)]

***IMAGE
```

```
**SIMB_CARTO
[Bloques de Cartografía]
[simbcart(banco,Banco)]^C^C_insert banco
[simbcart(farola,Farola)]^C^C_insert farola
[simbcart(quiosco,Quiosco)]^C^C_insert kiosko
[simbcart(pozo,Pozo)]^C^C_insert pozo
[simbcart(arbol,Árbol)]^C^C_insert arbol
[simbcart(cabina,Cabina)]^C^C_insert cabina
[simbcart(fuente,Fuente)]^C^C_insert fuente
[simbcart(caseta,Caseta)]^C^C_insert caseta
[simbcart(torre,Torre)]^C^C_insert torre
[simbcart(poste,Poste)]^C^C_insert poste
[simbcart(ermita,Ermita)]^C^C_insert ermita
***HELPSTRINGS
ID_POL_GROSOR [Dibuja una polilínea de grosor 0]
ID_POL_CUR1 [Dibuja una polilínea en la capa Curvas1 con grosor 0]
ID_POL_CUR10 [Dibuja una polilínea en la capa Curvas10 con grosor 0.5]
ID_BORRAR1 [Suprime un único objeto del dibujo cada vez]
ID_EDITPOL1 [Edita polilíneas y mallas poligonales tridimensionales: editpol]
ID_CURVA_B [Convierte una polilínea en curva B]
ID_COPIAS [Realiza varias copias de los objetos designados]
ID_PARTE1 [Parte un objeto por un sólo punto]
ID_CAMBIAR [Comando Cambia]
ID_CAMBIAR_COL [Cambia el color de los objetos designados]
ID_CAMBIAR_TXT [Modifica un texto insertado en el dibujo]
ID_CAMBIAR_CAPA [Cambia la capa de los objetos designados]
ID_INSERTAR [Permite la inserción de bloques en el dibujo]
ID_CAPAS [Accede al cuadro de diálogo para la gestión de capas: Ddlmodes]
ID_CARTOGRAFÍA [Menú específico para la aplicación de cartografía]
ID_LÍNEA_1 [Crea segmentos de línea recta: línea]
ID_CENTRO_RADIO_1 [Crea círculos mediante el centro y el radio: círculo]
ID_POLILÍNEA_1 [Crea polilíneas bidimensionales: pol]
ID_BORRAR_1 [Suprime objetos de un dibujo: borra]
ID_EDITAR_POLILÍNEA_1 [Edita polilíneas y mallas poligonales tridimensionales: editpol]
ID_EQUIDISTANCIA_1 [Crea círculos concéntricos y líneas y curvas paralelas: eqdist]
ID_INSERT_CART [Inserción de simbología mediante un menú de iconos]
ID_AYUDA_CART [Ayuda sobre la aplicación de cartografía]
ID_CTRL_LYR_1 [Gestión de capas mediante botón desplegable especial]

//
//      Final de AutoCAD archivo de menús - C:\MENÚS\CARTOGR.MNU
//
```

NOTAS INTERESANTES:

1. Este es un ejemplo muy jugoso de menú parcial para **AutoCAD**. En él están incluidos tres menús desplegables, un menú de imágenes o iconos, la definición de una barra de herramientas y una sección *****HELPSTRINGS**.
2. Nótese que todos los elementos de este menú deberán estar almacenados en disco, para su correcto funcionamiento, y en la carpeta actual o en alguna de soporte, ya que no se indican rutas de acceso a archivos en casi ningún punto del archivo.
3. Algunos de los conceptos aún no han sido estudiados, como la personalización y creación de archivos de ayuda tipo Windows, o ciertas notaciones que podemos encontrar en el ejemplo. No hay que preocuparse por esto, ya se verá en su momento todo.

UNO.FIN. EJERCICIOS PROPUESTOS

1. Revísese el último ejemplo explicado (menú de cartografía) y analicéense ciertos espacios en blanco incluidos en algunas líneas. ¿Que ocurriría si no existieran dichos espacios?

- II. Crear un archivo de menú de **AutoCAD** que, mediante menús desplegables y de persiana, permita la inserción de bloques —cualesquiera— en un dibujo, con la escala predeterminada para todos ellos de 1:5.
- III. Realizar un archivo de menús en el que, desde un menú desplegable, se llame a un menú de imagen y, desde éste, a otros menús de imagen (más de cuatro). Contemplar la posibilidad de retornar al menú principal desde cada uno de ellos. NOTA: Al no haber sido estudiado el apartado de fotos y fototecas, los menús de imagen pueden quedar vacíos, pero sólo en sus cuadros de iconos.
- IV. Modificar el menú de pantalla de **AutoCAD** añadiéndole nuevas opciones que permitan el cambio interactivo de los modos ortogonal (Orto), forzado de cursor (Forzcursor) y la rejilla (Rejilla) del programa, así como el cambio de valores de la variable TILEMODE y la permutación entre Espacio Modelo Mosaico y Espacio Papel/Modelo Flotante de **AutoCAD**. Permitir la vuelta al menú anterior.
- V. Modificar las definiciones de las secciones *****BUTTONS** (o *****AUX**) del archivo de menús de **AutoCAD** para permutar o variar al gusto las funciones de los pulsadores del dispositivo señalador actual del sistema.
- VI. Crear un archivo de menú con las definiciones de una barra de herramientas que contenga cuatro botones, los cuales realizarán las siguientes funciones respectivamente: dibujar polilíneas (por el usuario) con grosor 3; dibujar rectángulos de tamaño variable y achaflanados en sus cuatro esquinas; limpiar bloques sin pedir confirmación; dibujar una polilínea (por el usuario; 2 tramos), copiar el resultado múltiples veces (dos veces) a los puntos indicados (por el usuario) y realizar una regeneración del dibujo.
- VII. Crear un botón desplegable con la barra de herramientas anterior.
- VIII. Crear una serie de teclas rápidas que ejecuten diversos comandos de **AutoCAD** que carezcan de ellas.
- IX. Modificar la sección *****TABLET** del archivo de menús de **AutoCAD** asignando la inserción de bloques a diferentes casillas.
- X. Crear un archivo de menú completo con todas las definiciones necesarias para un oficina técnica de calderería y *piping*.

MÓDULO DOS

Personalización de barras de herramientas desde AutoCAD

DOS.1. INTRODUCCIÓN

En el **MÓDULO UNO** vimos la posibilidad de introducir en los archivos de menús las definiciones necesarias para el funcionamiento de las barras de herramientas de **AutoCAD**. Si recordamos, estas definiciones eran difíciles de comprender y llevar a la práctica por el simple hecho de introducirlas como líneas de código puro. Por ello, y para una realización más intuitiva de estas barras de herramientas, **AutoCAD** provee al usuario de una interfaz sencilla en su manejo que nos proporciona un total dominio sobre su creación.

DOS.2. EL PRIMER ACERCAMIENTO

Bajo Ver>*Barras de herramientas...* entramos en la interfaz dicha. De esta forma se abre el cuadro de diálogo *Barras de herramientas*. En este cuadro tenemos varios elementos que pasamos a comentar a continuación.

En el cuadro principal *Barras de herramientas*: podemos visualizar todas la barras incluidas en el grupo de menús indicado bajo dicho cuadro (*Grupo de menús:*). Si cambiamos el archivo de menú en esta lista desplegable, evidentemente los nombres de barras de herramientas variarán, visualizándose las correspondientes al archivo elegido. Las barras de menú se presentan identificadas por su nombre y con una casilla de verificación a su izquierda. Si dicha casilla está activada (☒) la barra será visible en pantalla; si no lo está (☐) la barra permanecerá oculta. Desde este cuadro activaremos (visualizaremos) o no las diversas barras de herramientas de **AutoCAD**.

En la parte inferior del cuadro de diálogo aparecen dos casillas de verificación. *Botones grandes*, muestra los botones de barras de herramientas en su formato grande (refiérase al **MÓDULO** anterior para comprender esto) en monitores de pantalla grande y/o configurados en alta resolución. *Mostrar pistas* activa o desactiva la pista o *tip* amarillo que aparece al situar el puntero del ratón sobre cualquiera de los botones de una barra de herramientas.

A la derecha del cuadro aparecen unos botones que se explican a continuación. *Cerrar* cierra el cuadro de diálogo actual. Con *Nueva...* empezaremos con la creación de una nueva barra de herramientas; lo veremos más adelante. *Suprimir* elimina la barra designada en el cuadro de la izquierda (*Barras de herramientas:*). Con *Personalizar...* (que ya veremos) añadimos o quitamos botones a nuestra barra de herramientas o a una ya creada. El botón *Propiedades...* muestra la propiedades de la barra designada. Estas propiedades son las que siguen:

Propiedad	Explicación
<u>N</u> ombre	Es el nombre de la barra y el que aparece en la lista <i>Barras de herramientas...</i> del cuadro principal <i>Barras de herramientas</i> .
A <u>y</u> uda	Es el texto que aparece en la línea de estado de AutoCAD al situar el puntero del ratón sobre una barra de

Propiedad	Explicación
<i>Alias:</i>	herramientas, esto es, sobre un intersticio o pequeño espacio vacío, y no sobre un botón. Es el nombre interno que utiliza AutoCAD para la barra de herramientas. Este nombre está formado por el nombre del archivo de menú donde se encuentra definida la barra, un punto y el nombre de alias dado a la propia barra en el archivo de menú. Por ejemplo, ACAD.TB_DIMENSION es el alias para la barra de herramientas de acotación, incluida en el ACAD.MNU.

Por último, el botón *Ayuda* del cuadro *Barras de herramientas* muestra la ayuda correspondiente a esta cuadro del fichero de ayuda de **AutoCAD**.

NOTA: El cuadro de diálogo de manejo de barras de herramientas también puede ser arrancado mediante el comando BARRAHERR (abreviado BH), su correspondiente equivalencia sajona TOOLBAR (abreviado TB) para versiones inglesas (_TOOLBARS para las demás versiones idiomáticas del programa) o mediante la casilla X-7 de la plantilla de **AutoCAD** para el tablero digitalizador.

DOS.3. NUESTRA BARRA DE HERRAMIENTAS

Para la creación de nuestra primera barra de herramientas vamos a exponer un ejemplo práctico, de esta manera conseguiremos un total comprendimiento del proceso que debemos seguir. Así pues, vamos a crear un barra de herramientas que contenga un solo botón que cree polilíneas con grosor 3.

Para todo ello vamos a crear un archivo de menú donde se guardarán las definiciones de la barra. Podríamos crearla directamente en el grupo de menús ACAD, pero conviene no mezclar nuestra propias barras con las del programa para que cada cosa esté en su sitio. Dicho archivo de menú se llamará NUEVO.MNU y únicamente tendrá escrita la línea siguiente:

```
***MENUGROUP=NUEVO
```

De esta forma, creamos un archivo de menú vacío y preparado. Lo guardaremos y lo cargaremos en **AutoCAD** como aprendimos en el **MÓDULO UNO**.

Accedemos ahora al cuadro *Barras de herramientas* y pulsamos el botón *Nueva....* Se abre entonces un nuevo cuadro denominado *Nueva barra de herramientas*. En este cuadro le daremos un nombre a nuestra nueva barra (POLGROSOR, por ejemplo) en la casilla *Nombre de la barra de herramientas*; y elegiremos el grupo donde queramos introducir sus definiciones en la lista desplegable *Grupo de menús*. Es importante asegurarse de que estamos en el grupo de menús adecuado. Pulsamos el botón *Aceptar*. Nuestra barra ha sido ya creada, la veremos superpuesta a la barra de herramientas de **AutoCAD** *Propiedades de objetos*, aún sin botones, sobre el cuadro de diálogo actual.

NOTA: Si al entrar en el cuadro *Nueva barra de herramientas*, en el cuadro *Barra de herramientas* teníamos elegido un grupo de menús que no es el que nos interesa para nuestra barra, al volver atrás pulsado el botón *Aceptar* podemos ver incluida nuestra barra en el grupo de menús que no es, o sea, en el que estábamos antes de pulsar *Nueva....* Esto es sólo un efecto visual (pequeño *bug* de **AutoCAD** quizás), ya que si elegimos ahora el grupo de menús adecuado podremos ver como nuestra barra se ha incluido donde debe; si volvemos al grupo anterior apreciaremos que ya no se encuentra allí.

DOS.3.1. Añadiendo botones a la barra

El siguiente paso consiste en agregar botones (uno en nuestro caso) a la recién creada barra de herramientas. Para ello deberemos pulsar el botón *Personalizar...* del *cuadro Barras de herramientas* teniendo designada nuestra barra. Al pulsar dicho botón se abre un nuevo cuadro llamado *Personalizar barras de herramientas*. En él podemos apreciar varias zonas: la lista desplegable *Categorías:*, donde se encuentran todas las barras de **AutoCAD** por categorías y algunas extra que ahora explicaremos; un cuadro bajo la lista anterior que muestra todos los botones incluidos en la categoría elegida; y un cuadro de descripción donde podemos leer un pequeño texto explicativo de cada botón al pulsarlo.

Desde este cuadro podemos añadir botones a nuestra barra de herramientas personalizada. La forma de hacerlo es, una vez escogida la categoría y el botón que vamos a añadir, arrastrarlo (pinchar, mantener, mover y soltar) a la nueva barra. Así, podemos crear una barra personalizada con los comandos que más utilicemos de **AutoCAD**. Para hacer una prueba podemos arrastrar el botón del comando que dibuja líneas (*LINEA*). Una vez colocado en su sitio, pulsemos sobre él con el botón derecho del ratón. Al hacer esto aparece un nuevo cuadro de diálogo denominado *Propiedades del botón*. Estas propiedades son similares a las de la barra de herramienta antes explicadas, y son las siguientes:

Propiedad	Explicación
<i>Nombre:</i>	Es el nombre que le damos al botón. Es el que aparece en forma de pequeña pista o <i>tip</i> amarillo al posicionar el cursor sobre el botón (si estas pistas están activadas, como hemos visto).
<i>Ayuda:</i>	Es el texto que aparece en la línea de estado de AutoCAD al situar el puntero del ratón sobre un botón cualquiera.
<i>Macro:</i>	Es la macroinstrucción, o instrucción simple, que determina el funcionamiento del botón, esto es, la acción o serie de acciones que realizará al ser pulsado.
<i>Icono del botón</i>	Es el icono que irá representado sobre el botón. Puede ser uno cualquiera de AutoCAD o uno propio personalizado. Esto último lo conseguimos pulsando el botón <i>Editar...</i> del cuadro de diálogo (se explica más adelante).

DOS.3.2. Añadiendo un botón vacío

Pero como nosotros queremos añadir un botón propio sin ninguna definición, deberemos recurrir a otro método.

NOTA: No deberemos cambiar las propiedades de un botón de **AutoCAD**, a no ser que lo hagamos con conocimiento de causa, ya que no se pueden “guardar como” y lo que haríamos sería cambiar las definiciones originales del botón.

Lo primero que haremos será eliminar el botón de la orden *LINEA* que habíamos agregado a nuestra barra de herramientas. Para ello, y teniendo en pantalla el cuadro de diálogo *Personalizar barras de herramientas* —de otra forma no funciona—, arrastraremos dicho botón desde la barra a la pantalla gráfica de **AutoCAD**, esto es, como dejándolo “caer al vacío”; el botón desaparecerá (internamente se eliminarán su definiciones del archivo de menú correspondiente). Esta es la forma de eliminar botones de una barra de herramientas de **AutoCAD**.

NOTA: No lo “dejemos caer” sobre otra barra de herramientas, ya que de esa manera de añadirá a dicha barra.

Para agregar ahora un botón vacío a nuestra barra, dentro del cuadro *Personalizar barras de herramientas*, elegiremos la categoría *Personalizar*. Dentro de esta categoría existen dos tipos de botón: uno simple y otro desplegable. De los botones desplegables hablaremos más adelante; añadamos ahora —según el método explicado— un botón simple a la barra.

Como hemos expuesto anteriormente, hacemos clic sobre este nuevo botón con el botón derecho del ratón; aparecerá el cuadro *Propiedades del botón* explicado. Como nombre le damos, por ejemplo, POLGR 3. Esta casilla deberá estar obligatoriamente rellena, si no **AutoCAD** da un mensaje de error. En la casilla de *Ayuda:* escribimos, por ejemplo, *Dibuja polilíneas con grosor 3*. Y la macroinstrucción sería la siguiente:

```
^C^C_pol \_w 3 3
```

La explicación es bien sencilla: ^C^C para anular cualquier otro comando en curso (lo pone **AutoCAD** por defecto); _pol y un espacio (o punto y coma) para escribir y aceptar la orden de dibujo de polilíneas; \ para dejar introducir al usuario el primer punto de la polilínea; _w y espacio (o punto y coma) para escribir y aceptar la opción de grosor; 3 y espacio (o punto y coma) y otro 3 para escribir y aceptar el grosor inicial y final (en este caso de 3 unidades de dibujo). El resto ya corre a cargo del usuario, pues deberá ir introduciendo diferentes puntos (hasta acabar con INTRO) como en el comando original.

NOTA: Todos los caracteres especiales introducibles en macroinstrucciones están explicados en el **MÓDULO UNO** de este curso. Como vimos en él, podemos introducir macros en menús de **AutoCAD**, pero donde realmente cobran sentido estos pequeños “programas” es en la definición de botones.

NOTA: Como sabemos, los caracteres ^C^C pueden eliminarse de estas macros (simplemente borrándolos) si no nos interesan.

DOS.3.3. Editar el icono del botón

El último paso en la creación de un botón consiste en darle una figura o icono representativo del mismo. Tenemos varias opciones: dejarlo vacío (no es conveniente), asignarle un icono de **AutoCAD** (tampoco es muy conveniente por razones obvias), editar o modificar un icono de **AutoCAD**, crear uno propio mediante el editor incluido o asignarle uno creado en otro editor de mapas de bits (MS Paint por ejemplo). Todas estas operaciones las elegimos desde el área *Icono del botón* del cuadro de diálogo *Propiedades del botón*.

En la lista de iconos de esta área podemos escoger uno de los iconos de **AutoCAD** o, incluso, uno vacío. El método más recomendable será elegir uno existente o uno vacío y editarlo (modificar o crear desde cero) por medio del editor de iconos incluido en el programa. Este editor se arranca, una vez elegido el icono, con el botón *Editar...* del cuadro de diálogo.

El *Editor de botones* es una interfaz sencilla e intuitiva para la creación de iconos. Pasamos ahora a describir sus áreas.

En la parte de la derecha tenemos el área de la paleta de colores. En ella existen 16 cuadros con igual número de colores que podemos aplicar a nuestro icono. Una de estas casillas deberá ser designada antes o después de cualquiera de las cuatro siguientes herramientas de dibujo.

Las herramientas de dibujo, en la parte superior del cuadro, son, de izquierda a derecha, la de dibujo pixel a pixel, dibujo de líneas, dibujo de círculos y borrado pixel a pixel.

En la zona izquierda tenemos, arriba del todo, una presentación preliminar de la figura del icono en pequeño y, debajo de ésta, una casilla de verificación llamada Rejilla. Esta casilla, cuando está activada, despliega un cuadrículado en la figura del icono (situada en medio y en formato ampliado) que facilita las operaciones de modificación y dibujo del mismo. Bajo esta casilla existen tres botones: Borrar, que borra la figura del icono; Abrir..., que permite cargar un icono existente en un archivo de extensión .BMP (mapa de bits); y Deshacer, que elimina la última operación realizada.

NOTA: Al editar un icono para **AutoCAD** en un editor de mapas de bits habrá de tenerse muy en cuenta el formato de dichos iconos, es decir, el tamaño en pixeles de 16 de ancho por 15 de alto (icono pequeño estándar). Con el editor incluido en el programa no se pueden crear iconos de formato grande (24 de ancho por 22 de alto), habremos de crearlos, si se necesitan, con un editor externo e incluir la referencia al .BMP en el archivo de menú.

Por último, en el área inferior del cuadro de diálogo, nos encontramos cuatro botones que, de izquierda a derecha, son: Guardar como..., que guarda el icono con un nombre y en un directorio elegido por nosotros y con la extensión .BMP; Guardar, que simplemente guarda el icono, lo hace el propio **AutoCAD** con un nombre interno y, en principio, en el directorio donde se encuentre el archivo de menú que encierra las definiciones de la barra de herramientas creada; Cerrar, que cierra el cuadro preguntando si no se han guardado los cambios; y Ayuda, que arranca la ayuda en línea de **AutoCAD** en la sección correspondiente al *Editor de botones*.

NOTA: Si se deja un icono vacío o si **AutoCAD** no encuentra el archivo .BMP al iniciarse, el botón recibirá por defecto un simpático icono en forma de pequeño *smiley*, o cara sonriente, con gafas de sol.

NOTA: A veces **AutoCAD** guarda, junto al icono creado, otro archivo .BMP con un icono vacío; no debemos preocuparnos de ello ya que son acciones internas del programa.

Y lo que queda ahora es la propia pericia del usuario dibujando iconos. Recomendamos examinar los propios iconos de **AutoCAD** para aprehender las técnicas de suavizado (con colores más oscuros adyacentes a los principales), relieve, dimensión, etcétera.

Al acabar de dibujar el icono, guardamos los cambios, cerramos el cuadro, pulsamos Aplicar en el cuadro *Propiedades del botón*, lo cerramos (con la x de la ventana) y pulsamos Cerrar en el cuadro *Barras de herramientas*. El proceso habrá finalizado, **AutoCAD** recargará los menús y podremos utilizar nuestro nuevo botón.

NOTA: Para acceder rápidamente a la propiedades de un botón creado, o de uno de los de **AutoCAD**, únicamente deberemos realizar doble clic con el botón derecho del ratón en dicho botón.

DOS.4. BOTONES DESPLEGABLES

Como sabemos, en **AutoCAD**, además de los botones simples existen una serie de botones desplegables. Estos, como por ejemplo el de *Vistas con nombre* o el del SCP, son botones que, al ser pulsados y mantenidos, despliegan una lista de botones (barra de herramientas) que lleva implícita el que cada uno de los botones que incluye suele tener similitud con el resto, en cuestión de acción o ejecución de comandos. Estos botones también pueden ser creados y/o modificados en **AutoCAD**. La manera la explicaremos en seguida, pero antes, vamos a ver la propiedades de un botón desplegable cualquiera —haciendo doble

clic en él con el botón derecho del ratón—. Estas propiedades también son similares a las de las barras de herramientas y a las de los botones simples; son las que siguen:

Propiedad	Explicación
<u>N</u> ombre:	Es el nombre que le damos al botón. Aparecerá como pista o <i>tip</i> amarillo dependiendo de la casilla de verificación <u>M</u> ostrar el icono de este botón.
Ayuda:	Es el texto que aparece en la línea de estado de AutoCAD y se corresponde con el de la barra de herramientas asociada si está activada la casilla comentada.
<u>B</u> arra de herramientas asociada:	Es la barra de herramientas que desplegará el botón al ser pulsado. Viene identificada por el nombre de grupo de menús (**MENUGROUP= en el archivo), un punto (.) y el nombre en sí de la barra. Aparecerán en la lista todas las barras de todos los archivos de menú cargados.
<u>I</u> cono del botón:	Al igual que en el cuadro de propiedades de botones simples, desde aquí podemos elegir o editar un icono para nuestro botón desplegable.
<u>M</u> ostrar el icono de este botón:	Casilla de verificación que, activada hace que el icono actual visualizado sea el propio del botón; desactivada hace que se visualice en el botón desplegable la última herramienta utilizada de la barra asociada (típico de los botones desplegables de AutoCAD).

La manera de crear un botón desplegable es asaz similar a la creación de un botón simple. Únicamente habremos de elegir el icono de botón desplegable de la categoría *Personalizar* del cuadro *Personalizar barras de herramientas* y arrastrarlo a una barra creada. Después, pulsaremos con el botón derecho del ratón sobre él para acceder al cuadro comentado *Propiedades de los iconos desplegables*. Rellenaremos las casillas (la del nombre obligatoriamente), designaremos qué barra queremos asociar a su despliegue y editaremos un botón o elegiremos activar la casilla inferior para que el icono varíe según qué herramienta se use. Aceptamos, cerramos todo y listo.

DOS.5. COPIA Y DESPLAZAMIENTO DE BOTONES

En la creación de barras de herramientas desde **AutoCAD** podemos evitarnos la ardua tarea de tener que editar muchos botones que contengan casi la misma definición de macro y sólo cambie una pequeña porción. Y es que podemos copiar botones dentro de una misma barra de herramientas o de una barra a otra. La manera es sencilla y conocida.

Cuando queremos copiar un archivo —arrastrándolo— de una carpeta a otra dentro del mismo disco duro y en Windows, al estar realizándose la operación dentro de la misma unidad de disco, por defecto el archivo tenderá a moverse, no a copiarse (así como tiende a copiarse, y no a moverse, cuando se realiza la operación entre unidades de disco diferentes, ya sean locales o remotas). Y, ¿qué es lo que hacemos para decirle al sistema que queremos copiar y no mover? Pues pulsar la tecla **CTRL** a la vez que arrastramos y soltamos.

En la copia de botones ocurre lo mismo. Si queremos copiar un botón de una barra de herramientas a otra, o dentro de la misma barra, solamente deberemos tener pulsada la tecla **CTRL** mientras arrastramos y soltamos. De esta forma, el botón será copiado, así como todas sus propiedades internas. Esto es un buen método, como decíamos, para no andar creando botones a diestro y siniestro, que tengan todos ellos unas macros de veinte líneas y en las que

únicamente varíe un par de caracteres o un valor. Copiaríamos el botón las veces que haga falta y sólo habríamos de cambiar lo indispensable de la macro, así como, probablemente, su icono —seguramente no mucho—.

Pero, como las aplicaciones de software no son perfectas, **AutoCAD** no va a ser menos que las demás y tiene un pequeño error de programación (subsancable).

El *bug* (si lo es) es el siguiente. Si en una barra de herramientas tengo un solo botón y deseo copiarlo al lado de sí mismo, seré incapaz. Y es que **AutoCAD** cree que lo queremos es moverlo hacia un lado y, al no haber más que un botón, no puede desplazarse hacia ninguno de los lados. El truco que utilizaremos es sencillo. Sólo tenemos que introducir otro botón (cualquiera de cualquier categoría) al lado del nuestro, copiar éste al otro lado del botón introducido —ahora sí podremos— y eliminar el botón del medio. Sencillo.

NOTA: Esto con el cuadro *Personalizar barras de herramientas* abierto, si no, no funcionará.

Tras esta operación, y si queremos copiar más veces el botón, ya podremos hacerlo sin problemas. Pero ojo, siempre copiando el primero de los botones, porque si intentamos copiar el segundo al lado de sí mismo (o el tercero, cuarto...), por mucho que pulsemos la tecla CTRL, el botón sólo se moverá. Frustrante.

Si lo que queremos es copiar otro botón, ya no tendremos ningún problema porque hay más en la barra. Pero recordando que no podremos copiarlo al lado de sí mismo, sino a otro sitio para luego moverlo al lado.

Como habremos deducido, la operación de mover botones dentro de una barra de herramientas consiste en arrastrarlos hacia uno de los lados para que queden separados por un pequeño hueco libre con respecto a los demás. Esto únicamente se realiza por estética o agrupación de funciones. Recordemos que se codificará en el archivo de menú como [--].

DOS.6. COMPROBANDO EL .MNS

Como sabemos, o deberíamos saber ya, las especificaciones de control de barras de herramientas se guardan en el archivo .MNS del grupo de menús, es decir, en el archivo fuente. En estos archivos es donde debemos realizar pruebas, sin miedo a cometer errores graves, ya que, si esto ocurriera, para ello tendríamos el archivo de plantilla de menú .MNU como salvaguarda. Sólo cuando sepamos de buena tinta que queremos conservar una barra de herramientas con un menú, para su distribución o para lo que sea, deberemos copiarla al .MNU. Recordar que cuando carguemos el .MNU, al crearse un nuevo .MNS se eliminarán todas las configuraciones de barras de herramientas existentes en él.

Pues bien, tras crear nuestra propia barra de herramientas —la que hemos hecho antes con un botón, por ejemplo— podemos comprobar las definiciones que se ha incluido en el .MNS, en nuestro caso NUEVO.MNS. Éstas habrán sido escritas bajo la sección *****TOOLBARS** y, si existen ayudas, bajo la sección *****HELPSTRINGS**. Serán algo así:

```
***TOOLBARS
**POLGROSOR
ID_Polgrosor_0    [_Toolbar("Polgrosor", _Floating, _Show, 400, 50, 0)]
ID__0    [_Button("Polgr 3", "ICON.bmp", "ICON_24_BLANK")]^C^C_pol /_w 3 3

***HELPSTRINGS
ID_POLGROSOR_0    [Barra de polilíneas con grosor]
ID__0             [Polilíneas con grosor 3]
```

DOS.7. EJEMPLOS PRÁCTICOS DE BOTONES

DOS.7.1. Insertar DWG's en el 0,0

Nombre del botón: InsertDWG

Ayuda: Inserta un DWG en el 0,0

Macro: ^C^C_insert \0,0;;;;

NOTAS INTERESANTES:

1. Este es un ejemplo sencillo con una pequeña macro comprensible. Se ejecuta el comando INSERT y se pregunta por el nombre de archivo .DWG (o bloque) que será insertado. Luego se introduce el 0,0 como punto de inserción y se le da un INTRO para cada una de las siguientes preguntas del comando.

DOS.7.2. Matriz de pentágonos

Nombre del botón: Matriz de pentágonos

Ayuda: Crea una matriz de pentágonos circunscritos en un círculo de radio 10

Macro: ^C^C_polygon 5 _c 10;_array _l _r 10 10 25 25

NOTAS INTERESANTES:

1. Nótese que se utiliza un espacio (o varios) para un producir un INTRO (o varios) entre opciones del mismo comando. Los puntos y coma (;) se reservan para producir un INTRO cuando empieza otro comando, para separar, o para el final de la macro. Todo esto se realiza por claridad a la vista, ya que es lo mismo un espacio que un punto y coma.
2. Entre la opción _l (*last* = último) y la opción _r (rectangular) hay dos espacios. El primer INTRO es para aceptar la designación del pentágono (último objeto dibujado) y el segundo para dejar de designar objetos.
3. La macro debemos escribirla de forma continua, sin retornos y saltos de carro (INTRO). En el momento en que lleguemos al final del cuadro, el cursor saltará automáticamente a la línea siguiente.

DOS.7.3. Inserción de formatos desplegable

Botón 1

Nombre del botón: DIN A1

Ayuda: Inserta una lámina formato A1

Macro: ^C^C_insert c:/diseño/formatos/dina1 0,0;;;;

Botón 2

Nombre del botón: DIN A2

Ayuda: Inserta una lámina formato A2

Macro: ^C^C_insert c:/diseño/formatos/dina2 0,0;;;

Botón 3

Nombre del botón: DIN A3

Ayuda: Inserta una lámina formato A3

Macro: ^C^C_insert c:/diseño/formatos/dina3 0,0;;;

Botón 4

Nombre del botón: DIN A4

Ayuda: Inserta una lámina formato A4

Macro: ^C^C_insert c:/diseño/formatos/dina4 0,0;;;

NOTAS INTERESANTES:

1. Nótese la necesidad de incluir las rutas de acceso con la barra normal (/) estilo UNIX, ya que la contrabarra o barra inversa (\) está reservada. Esto ya se comentó en el **MÓDULO** anterior.
2. Este último ejemplo es práctico sobremanera. Precisamente, la necesidad de los botones radica en la utilidad que se les dé, es decir, conviene utilizarlos para ahorrarnos trabajo a la hora de realizar tareas repetitivas, o largas, o para comandos de **AutoCAD** a los que no se pueda acceder más que desde la línea de comandos.
3. No olvidarnos de que, antes de crear un botón, habrá que crear la barra de herramientas donde será incluido o que, en su defecto, habrá que incluirlo en una barra ya creada dentro de un grupo de menús.

DOS.FIN. EJERCICIOS PROPUESTOS

- I. Crear una barra de herramientas con 3 botones que permitan conmutar entre los estados activado y desactivado del modo Orto, activado y desactivado del modo Forzcursor y activado y desactivado de la Rejilla.
- II. Diseñar un botón que abra el navegador de Internet configurado por defecto en el sistema y acceda a "La Web del Programador". (No tiene mucho sentido porque se puede configurar **AutoCAD** para que realice eso con su propio botón del *browser*, pero bueno, es un modo de practicar).
- III. Crear una barra de herramientas con 3 botones que, respectivamente, impriman un giro en torno a los ejes X, Y y Z del SCP de la ventana actual. Se requerirá la entrada del ángulo al usuario.
- IV. Créese un botón desplegable con los botones simples del ejemplo anterior.

- V. Partiendo de una configuración de tres ventanas en Espacio Modelo Mosaico, diseñese una pareja de botones que, el primero amplíe a pantalla gráfica completa la ventana actual y, el segundo retorne a la configuración inicial de tres ventanas con las vistas primitivas guardadas.
- VI. Pártase de una configuración de cuatro ventanas en Espacio Modelo Mosaico como la siguiente: ventana inferior izquierda es la vista superior (planta); ventana superior izquierda es la vista frontal (alzado); ventana superior derecha es la vista izquierda (perfil); ventana inferior derecha es perspectiva isométrica SE. Crear un botón que realice un ZOOM EXTENSIÓN en todas las ventanas gráficas y, además, en todas ellas, a excepción de la perspectiva, se “aleje” un poco para que no quede el dibujo muy pegado a los bordes.
- VII. De todos es sabido que el comando ESCALA de **AutoCAD** no permite escalar los objetos independientemente en X y en Y. Se trata de diseñar un botón que tenga esta característica, es decir, que escale un objeto con factores diferentes respecto al eje X y al eje Y. (Este ejercicio es para pensar un poco).
- VIII. Crear una juego completo de barras de herramientas para su utilización en un estudio de arquitectura.

EJERCICIOS RESUELTOS DEL MÓDULO UNO

EJERCICIO I

Analicemos la línea siguiente:

```
[ farola]^C^C_insert farola
```

Los espacios que se encuentran antes de farola simplemente están para sangrar las líneas de estas opciones dentro del desplegable; cuestión de estética.

EJERCICIO II

```
***MENUGROUP=BLOQUES
```

```
***POP1
```

```
[Blo&ques]  
[->&Mecánica]  
[&Tuerca]^C^C_insert c:/bloques/tuerca.dwg \1.5;;;  
[&Tornillo]^C^C_insert c:/bloques/tornillo.dwg \1.5;;;  
[&Arandela]^C^C_insert c:/bloques/arandela.dwg \1.5;;;  
[&-&Junta]^C^C_insert c:/bloques/junta.dwg \1.5;;;  
[->&Electrónica]  
[&Diodo]^C^C_insert c:/bloques/diodo.dwg \1.5;;;  
[&Resistencia]^C^C_insert c:/bloques/resist.dwg \1.5;;;  
[&-&Condensador]^C^C_insert c:/bloques/cond.dwg \1.5;;;  
[->&Piping]  
[&Tubo]^C^C_insert c:/bloques/tubo.dwg \1.5;;;  
[&Codo]^C^C_insert c:/bloques/codo.dwg \1.5;;;  
[&-&Válvula]^C^C_insert c:/bloques/valvul.dwg \1.5;;;  
[->&Topografía]  
[&Vértice]^C^C_insert c:/bloques/vertice.dwg \1.5;;;  
[&Árbol]^C^C_insert c:/bloques/arbol.dwg \1.5;;;  
[&Casa]^C^C_insert c:/bloques/casa.dwg \1.5;;;  
[&-&-&Estación]^C^C_insert c:/bloques/estacion.dwg \1.5;;;
```

EJERCICIO III

```
***MENUGROUP=IMAGEN

***POP1
[&Bloques]
  [&Insertar]$I=IMAGEN.Insertblq $I=IMAGEN.*

***IMAGE
**Insertblq
[Grupos]
[Grupo 1]$I=IMAGEN.Gr1 $I=IMAGEN.*
[Grupo 2]$I=IMAGEN.Gr2 $I=IMAGEN.*
[Grupo 3]$I=IMAGEN.Gr3 $I=IMAGEN.*
[Grupo 4]$I=IMAGEN.Gr4 $I=IMAGEN.*
[Grupo 5]$I=IMAGEN.Gr5 $I=IMAGEN.*

**Gr1
[Bloques Grupo 1]
[Bloque 11]^C^C_insert bl11
[Bloque 21]^C^C_insert bl21
[Bloque 31]^C^C_insert bl31
[Volver]$I=IMAGEN.Insertblq $I=IMAGEN.*

**Gr2
[Bloques Grupo 2]
[Bloque 12]^C^C_insert bl12
[Bloque 22]^C^C_insert bl22
[Bloque 32]^C^C_insert bl32
[Bloque 42]^C^C_insert bl42
[Volver]$I=IMAGEN.Insertblq $I=IMAGEN.*

**Gr3
[Bloques Grupo 3]
[Bloque 13]^C^C_insert bl13
[Bloque 23]^C^C_insert bl23
[Volver]$I=IMAGEN.Insertblq $I=IMAGEN.*

**Gr4
[Bloques Grupo 4]
[Bloque 14]^C^C_insert bl14
[Bloque 24]^C^C_insert bl24
[Bloque 34]^C^C_insert bl34
[Bloque 44]^C^C_insert bl44
[Bloque 54]^C^C_insert bl54
[Volver]$I=IMAGEN.Insertblq $I=IMAGEN.*

**Gr5
[Bloques Grupo 5]
[Bloque 15]^C^C_insert bl15
[Bloque 25]^C^C_insert bl25
[Bloque 35]^C^C_insert bl35
[Bloque 45]^C^C_insert bl45
[Volver]$I=IMAGEN.Insertblq $I=IMAGEN.*
```

NOTA: Recuérdese introducir un `INTRO` al final de la última línea de los archivos de menú, de otra forma no funcionarán correctamente.

EJERCICIO IV

Bajo *SCREEN y **S**

```
[MODOS      ]$S=ACAD.Modos
```

Después

```
**Modos 3  
[Orto      ]^O  
[Forzcoor]^B  
[Rejilla  ]^G  
  
[TILE 1    ]TILEMODE 1  
[TILE 0    ]TILEMODE 0  
  
[EspPAPEL]EP  
[EspMODEL]EM
```

```
[VOLVER]$I=ACAD.
```

NOTA: Recuérdese la importancia de los amplios espaciados.

EJERCICIO V

Bajo *BUTTONS1 o ***AUX1**

```
$P0=SNAP $p0=*  
^C^C_line
```

Bajo *BUTTONS2 o ***AUX2**

```
^O  
^B
```

Bajo *BUTTONS3 o ***AUX3**

(vacío)

Bajo *BUTTONS4 o ***AUX4**

```
^C^C_pol \_w 3 3
^C^C_purge
```

EJERCICIO VI

```
***MENUGROUP=Barra
```

```
***TOOLBARS
```

```
**Varios
```

```
ID_Varios [_Toolbar("Varios", _Floating, _Show, 400, 50, 0)]
ID_PG3 [_Button("Polgrosor 3", "ICPolP.BMP", "ICPolG.BMP")]^C^C_pol \_w 3 3
ID_Rect [_Button("Rectchaflán", "ICRecP.BMP", "ICRecG.BMP")]^C^C_rectang c 5 5
ID_PC [_Button("PolyCopy", "ICPCP.bmp", "ICPCG.BMP")]^C^C_pol \\\ copia u m @ \\\ regen
ID_LB [_Button("LimpiaBloque", "ICLBP.BMP ", "ICLBG.BMP")]^C^C_purge b;n
```

```
***HELPSTRINGS
```

```
ID_Varios      [Barra de varias herramientas]
ID_PG3         [Dibuja polilíneas de grosor 3]
ID_Rect        [Dibuja rectángulos achaflanados]
ID_PC          [Dibuja y copia polilíneas]
ID_LB          [Limpia bloques sin pedir confirmación]
```

EJERCICIO VII

```
***MENUGROUP=Barra
```

```
***TOOLBARS
```

```
**DESPLEG
```

```
ID_DV [_Flyout("Desplegable Varios", VP.BMP, VG.BMP, _OtherIcon, BARRA.Varios)]
```

```
***HELPSTRINGS
```

```
ID_DV          [Botón desplegable de la barra VARIOS]
```

EJERCICIO VIII

```
***ACCELERATORS
```

```
ID_Line        [SHIFT+CONTROL+"F3"]
ID_Quit        ["ESCAPE"]
ID_Circle       [SHIFT+"INSERT"]
ID_Purge       [CONTROL+"F12"]
```

```
[SHIFT+"DOWN"]^C^C_scp _w
["NUMPAD9"]^C^C_tabsurf
```

EJERCICIO IX

```
***TABLET1
```

```
...
```

```
**TABLET1ALT
```

```
**BL1
```

```
<1>[T1-1]^C^C_insert c:/bloques/bl0001.dwg
<2>[T1-2]^C^C_insert c:/bloques/bl0002.dwg
<4>[T1-4]^C^C_insert c:/bloques/bl0004.dwg
```

```
<6>[T1-6]^C^C_insert c:/bloques/bl0006.dwg  
<34>[T1-34]^C^C_insert c:/bloques/bl0034.dwg  
<35>[T1-35]^C^C_insert c:/bloques/bl0035.dwg  
<70>[T1-70]^C^C_insert c:/bloques/bl0070.dwg  
<175>[T1-175]^C^C_insert c:/bloques/bl0175.dwg  
<182>[T1-182]^C^C_insert c:/bloques/bloq0182.dwg  
<183>[T1-183]^C^C_insert c:/bloques/bloq0183.dwg  
<184>[T1-184]^C^C_insert c:/bloques/bloq0184.dwg  
<185>[T1-185]^C^C_insert c:/bloques/bloq0185.dwg  
<199>[T1-199]^C^C_insert c:/bloques/bloq0199.dwg  
<200>[T1-200]$T1=ACAD.B2
```

EJERCICIO X

(Ejercicio completo para resolver por técnicos y/o especialistas).

MÓDULO TRES

Creación de tipos de línea

TRES.1. TIPOS DE LÍNEA EN AutoCAD

Los tipos de línea de **AutoCAD** no son otra cosa que las definiciones de cada una de las líneas, en un archivo ASCII con extensión `.LIN`. Estas bibliotecas serán leídas por el programa en el momento que lo necesite.

Los archivos de tipos de línea suministrados por **AutoCAD** se encuentran en el directorio `\SUPPORT\` del programa y son dos: `ACAD.LIN` y `ACADISO.LIN`. Ambos contienen las mismas definiciones de líneas, la diferencia estriba en que, en `ACAD.LIN` estas definiciones están en pulgadas y, en `ACADISO.LIN` están adaptados, los tipos de línea, a unidades métricas decimales —será el que utilicemos habitualmente—. La distinción, por parte de **AutoCAD**, a la hora de cargar uno u otro como predeterminado, se realiza al inicio de una sesión (si está así configurado) o al comenzar un dibujo nuevo en los cuadros de diálogo *Inicio* y *Crear nuevo dibujo*, respectivamente. Si no se permite abrir o se cancela alguno de estos cuadros, **AutoCAD** arrancará con la última configuración de tipos de línea, o sea, con el último archivo de tipos de línea (`ACAD.LIN` o `ACADISO.LIN`) que se utilizó antes de cerrar la última sesión. De todas formas, en cualquier momento podemos cargar cualquier tipo de línea de cualquiera de los archivos, pero teniendo en cuenta sus unidades de creación y escalado.

Estos archivos del programa contienen ocho tipos de líneas básicos en tres versiones para cada uno de ellos: escala normal, escala 0,5X (la mitad) y escala 2X (el doble); un total de 24. Además, 14 tipos de línea bajo norma ISO 128 (ISO/DIS 12011) y siete tipos más de líneas complejas (ahora incluidos en estos archivos; en la versión 13 se encontraban en el archivo `LTPESH.P.LIN`). Todos ellos hacen un total de 45 tipos de línea.

TRES.2. PODEMOS CREAR O PERSONALIZAR UN `.LIN`

En cualquier caso, si no se amoldan a nuestras apetencias o necesidades todos estos tipos de línea, o si nos vemos obligados a dibujar ciertos objetos con ciertos tipos de línea que no contiene **AutoCAD**, siempre podemos personalizar una de las bibliotecas `.LIN` del programa o crear nuestros propios archivos aparte.

Y es que **AutoCAD** permite una completa personalización de los tipos de línea, en tanto en cuanto nos deja variar las definiciones de sus propias líneas o crear otras nuevas que se adapten a nuestro gusto, trabajo o necesidad.

TRES.2.1. Examinando el `ACADISO.LIN`

Para empezar a ver cómo se crean los tipos de línea, vamos a abrir con cualquier editor ASCII el archivo `ACADISO.LIN`.

Lo primero que vamos a apreciar es lo siguiente:

```
;;  
;; Archivo de definición de tipos de línea de AutoCAD
```

```
;; Versión 2.0
;; Copyright 1991, 1992, 1993, 1994, 1996 por Autodesk, Inc.
;;
```

Es la serie de comentarios a los que nos tienen acostumbrados los archivos ASCII de **AutoCAD**. Si recordamos, en los archivos de menú los comentarios los introducíamos con una doble barra (//), pues aquí, en archivos de definición de tipos de línea, se introducen con un punto y coma (;). Pero cuidado, no nos confundamos; con un solo punto y coma es suficiente, lo que ocurre es que los diseñadores de Autodesk parece ser que, por mayor claridad, han decidido incluir dos caracteres seguidos. De esta forma, al primer golpe de vista podemos distinguir donde están los comentarios. Pero repetimos, con uno es suficiente, en el ejemplo anterior el que vale es el primero.

Podemos poner tantos comentarios como queramos y en la parte del archivo que deseemos, pero tengamos en cuenta lo de siempre: a mayor número de líneas, mayor tiempo de proceso.

Lo siguiente que os encontramos ya es una definición de tipo de línea:

```
*MORSE_G,Morse G _ _ . _ _ . _ _ . _ _ . _ _ .
A, 12.7, -6.35, 12.7, -6.35, 0, -6.35
```

Con dos renglones se define un tipo de línea. No hacen falta más —ni está permitido— ni se pueden utilizar menos. Pero, ¿qué significado tienen estas dos líneas?

TRES.2.2. Sintaxis de personalización

Existen dos maneras de crear tipos de líneas, una desde un editor ASCII externo a **AutoCAD** y, la otra, desde la propia línea de comandos de **AutoCAD**, con el comando **TIPOLIN**. Asimismo, existen dos clases de tipos de línea generalizados, los tipos de línea sencillos y los complejos.

Primero vamos a ver la creación de tipos sencillos y complejos desde un editor ASCII; después nos acercaremos al comando **TIPOLIN**, con su opción *Crea*, para estudiar cómo crear los tipos de línea sencillos desde el propio **AutoCAD**.

NOTA: Desde **AutoCAD**, y por medio de **TIPOLIN**, no se pueden crear tipos de línea complejos.

TRES.2.2.1. Creación desde un editor ASCII

La manera de crear tipos de línea desde un editor ASCII, como ya hemos visto, es con dos líneas para cada definición. El archivo donde lo vayamos a guardar puede contener varias definiciones, puede llamarse con cualquier nombre y debe tener la extensión **.LIN** obligatoriamente.

La primera línea de la definición es una línea de encabezamiento. Su sintaxis es la que sigue (los corchetes en *italica* indican la no obligatoriedad del parámetro):

```
*nombre_tipo_línea[,descripción]
```

El asterisco (*) siempre ha de preceder a este encabezamiento de definición de tipo de línea. *nombre_tipo_línea* es un nombre obligatorio para el tipo de línea que vamos a crear; es el nombre que aparecerá en **AutoCAD** al cargar o elegir el tipo de línea. *descripción* es

una descripción, que puede ser textual y/o gráfica (por medio de caracteres ASCII) del tipo de línea; esta descripción es opcional y, si se introduce, debe ir separada del nombre del tipo de línea por una coma (,) y ocupar como máximo 47 caracteres. Dentro de esta descripción podemos introducir espacios blancos en cualquier posición. Esta descripción aparecerá al cargar el tipo de línea y en el cuadro de propiedades de los tipos de línea.

NOTA: En *nombre_tipo_línea* es conveniente no escribir espacios blancos, así como procurar reducir a ocho los caracteres del nombre. Todo esto puede parecer arcaico y, no es que no se pueda, pero se debe tener en cuenta esta serie de recomendaciones. **AutoCAD** es un programa perfectamente integrado en Windows pero, como sabemos, los nombres de bloque, por ejemplo, tienen ciertos problemas a la hora de tratar nombres largos o con caracteres no permitidos. Como veremos en el siguiente **MÓDULO**, a la hora de crear patrones de sombreado tendremos muy en cuenta sus nombres, ya que utilizaremos una pequeña aplicación basada en MS-DOS para introducir las nuevas fotos en la fototeca de **AutoCAD** para patrones de sombreado. Por todo ello, es conveniente acostumbrarnos a utilizar —aunque a veces no haga falta— sintaxis MS-DOS para todo tipos de nombres en archivos ASCII de personalización, ya sean menús, definiciones de tipos de línea, de patrones de sombreado, aplicaciones AutoLISP, etcétera.

Así por ejemplo, el encabezado de una definición de tipo de línea de trazos, podría ser de cualquiera de la siguientes maneras (o de otras también):

```
*TRAZOS, Línea de trazos -- -- -- -- -- -- -- --  
*TRAZOS, Línea de trazos _ _ _ _ _ _ _ _  
*TRAZOS, Línea de trazos  
*TRAZOS
```

La segunda línea de la definición es la que realmente define cómo será el tipo de línea creado. Su sintaxis es la siguiente:

```
A, def_traz, def_traz, def_traz, ...
```

El carácter A del inicio es el tipo de alineamiento o alineación. Este carácter únicamente puede ser uno, A. No se admite cualquier otro carácter de alineación en esta posición; **AutoCAD** sólo reconoce éste. Esta alineación hace referencia a la propiedad de **AutoCAD** de alinear el patrón de tipo de línea con los puntos extremos de líneas, círculos y arcos individuales, esto es, cuando dibujamos una línea de un punto a otro con un tipo de línea de trazo y punto, por ejemplo, en los dos puntos extremos siempre se fuerza un trazo continuo; nunca quedará en un extremo un espacio vacío.

Los sucesivos *def_traz* son la propia definición de las dimensiones de trazos, espacios y puntos del patrón. Han de introducirse en el mismo orden en que van a aparecer en la línea. Estas dimensiones están en unidades de dibujo y han de aproximarse a las medidas de trazos y espacios del archivo ACADISO.LIN (o ACAD.LIN, dependiendo de cuál utilicemos) para que se guarde una proporción con ellos al luego aplicárseles un factor de escala global a todos (en el cuadro de las propiedades de los tipos de línea o con el comando **ESCALATL**, **LTSCALE** en inglés).

Estos guarismos defintorios van separados por comas y pueden incluirse entre ellas y los números que las siguen espacios blancos para darle claridad a la definición. Los valores de estas definiciones se entenderán de la siguiente forma:

Valor	Explicación
<i>Positivo</i>	Longitud de un trazo.
<i>Negativo</i>	Longitud de un espacio en blanco.
<i>Cero (0)</i>	Posición de un punto.

Por ejemplo, y siguiendo con la anterior línea de trazos, su definición completa sería:

```
*TRAZOS, Línea de trazos  _ _ _ _ _  
A, 10, -5
```

Esto quiere decir, un trazo de 10 unidades de dibujo y un espacio de 5 unidades de dibujo de longitud. Como ya hemos dicho, el alineamiento siempre A.

Como se puede apreciar, el patrón de tipo de línea hace referencia únicamente al mínimo conjunto de trazos, espacios y/o puntos cuya repetición sucesiva va a generar el tipo de línea. Se pueden especificar hasta un máximo de doce valores separados por comas, siempre que quepan en una línea de texto de 80 caracteres.

Así, en esta definición anterior, con un trazo y un espacio es más que suficiente; el resto es repetición de lo mismo continuamente.

NOTA: El tipo de alineamiento A requiere que el primer valor del patrón sea positivo (lo que corresponde a un trazo) o 0 (correspondiente a un punto). El segundo valor debe ser negativo (espacio en blanco). Se deben especificar, como mínimo, dos valores en el patrón de tipo de línea.

Veamos otros dos ejemplos. Analicemos el siguiente:

```
*Trazo_Puntos, Trazos y puntos  _ . . _ . . _ . . _ . . _  
A, 10, -5, 0, -5, 0, -5
```

O sea, un trazo de 10 unidades de dibujo (10), un espacio de 5 (-5), un punto (0), otro espacio, otro punto y un último espacio. A partir de ahí se repite de nuevo comenzando por el trazo.

Veamos el segundo:

```
*Nuevo_Tipo,  _ _ _ _ _ . _ _ _ _ _ _ _ _ _ _  
A, 20, -5, 10, -5, 0, -5, 10, -5
```

Esto significa, trazo grande de 20, espacio de 5, trazo pequeño de 10, espacio de 5, el punto, espacio de 5, trazo de 10, espacio de 5 y vuelta a empezar.

NOTA: Conviene hacer un pequeño boceto del tipo de línea en un papel, o servirnos, siempre que podamos, del esquema incluido en la descripción de la primera línea, para ir desgranando trazos, huecos y puntos uno por uno.

TRES.2.2.2. Tipos de línea complejos

Un tipo de línea complejo es una línea sencilla (como las estudiadas) que contiene símbolos intercalados. Estos símbolos pueden ser textos o formas de **AutoCAD**. Vamos a ver primero los tipos de líneas de texto intercalado.

Para introducir un texto en la definición de un línea hay que utilizar la siguiente sintaxis, dentro de la propia de la línea:

```
... ["cadena",estilo,S=factor_escala,R|A=ang_rotación,X=despl_X,Y=despl_Y] ...
```

Toda la definición aparecerá encerrada entre corchetes ([]); los diferentes parámetros separados por comas (,) y sin espacios blancos.

- *cadena*. Es el texto que se incluirá intercalado en la línea. Debe ir entre comillas.
- *estilo*. Se corresponde con el estilo de texto con el que se dibujará el mismo. Generalmente se utilizará el estilo **STANDARD** de **AutoCAD**, ya que es el definido por defecto y el que siempre existe. Utilizando otro estilo no tenemos la completa seguridad de que esté creado, a no ser que lo hayamos creado con alguna rutina de AutoLISP o macroinstrucción de menú y el tipo de línea forme parte de una distribución completa de personalización.
- *S=factor_escala*. Es el factor de escala que se le aplicará al texto.
- *R|A=ang_rotación*. Ángulo de rotación que será imprimido al texto en cuestión. No se utilizan ambos modificadores a la vez (R y A) sino uno u otro. R se refiere a la rotación relativa del texto y A a la rotación absoluta.
- *x=despl_X*. Es el desplazamiento en X medido en el sentido de la línea. Lo utilizaremos para centrar el texto en un hueco (ya se verá).
- *Y=despl_Y*. Es el desplazamiento en Y medido en sentido perpendicular a la línea. Lo utilizaremos también para centrar en texto perpendicularmente a la línea (también se verá).

NOTA: No es necesario incluir todos los modificadores en una definición, sino que pueden existir algunos y otros no. Eso sí, la cadena de texto y el estilo son obligatorios.

Veamos un ejemplo:

```
*Agua_Caliente, __ HW __ HW __ HW __  
A, .5, -.2, ["HW", STANDARD, S=.1, R=0, X=-0.1, Y=-.05], -.2
```

Lo primero que tenemos es un trazo de 0,5 y un hueco de 0,2. Lo siguiente es la definición del texto: **HW** como cadena literal (lo que aparecerá), en estilo **STANDARD**, a escala 0,1 (10 veces más pequeño), con un ángulo de rotación relativo de 0, un desplazamiento en X de 0,1 hacia "atrás" (negativo) y un desplazamiento en Y de 0,05 hacia "abajo" (negativo).

Los huecos en la líneas de **AutoCAD** no son realmente objetos dentro del propio objeto de la línea, esto es, es un espacio en el que no hay nada (por ello a veces no se puede designar un línea por un hueco o no se designa correctamente una intersección entre dos líneas si existe un hueco o más en la misma, aunque **AutoCAD** recalcula la línea para que esto no suceda). Los textos en tipos de línea complejos no tiene reserva de hueco, por lo que hay que reservárselo premeditadamente. Así, si definimos una línea de la siguiente manera:

```
*Trazo_Texto, Trazos y texto  
A, 1, ["Texto", STANDARD, S=1]
```

el resultado será el siguiente:

~~Texto~~Texto~~Texto~~Texto

Por eso, en el ejemplo del agua caliente, se reserva un hueco de 0,2 + 0,2 (al principio y al final) para el texto. Aún así, si no definiéramos un desplazamiento en X negativo, el resultado produciría el texto superpuesto en el principio del segundo hueco (aunque más bien es este hueco el que se superpone al texto), ya que este va justo detrás del hueco anterior y, como hemos dicho, no se reserva hueco para el texto. Por ello, hemos de centrarlo en el hueco, concretamente 0,1 unidades de dibujo hacia "atrás" en el sentido de la línea para dejar un espacio de 0,1 por cada lado. Al estar el texto a escala 0,1 (mide 0,1 del alto y 0,1 de ancho, ya que el texto base es de 1×1) se consigue perfectamente este efecto. Si no se tiene

esto en cuenta, el texto se colocará justo detrás del primer espacio (en su posición), pero el segundo se hueco superpondrá al texto al no haber reserva, como hemos dicho.

Con respecto a la rotación, y continuando con este ejemplo del agua caliente, se le da una rotación relativa de 0 grados. La diferencia entre la rotación relativa y la absoluta es que, la relativa rota el texto con respecto a la posición actual de la línea y la absoluta le imprime un ángulo fijo de rotación. Así pues, al haberle dado un ángulo relativo de 0 grados, el texto siempre se fijará a 0 grados con respecto a la línea; si dibujáramos una línea a 90 grados con este patrón, el texto estaría rotado también 90 grados. Si le hubiésemos puesto rotación absoluta (A) de 0 grados, el texto siempre se dibujaría con una inclinación de 0 grados, fuese cual fuese la inclinación de la línea trazada.

De esta manera, un texto con rotación absoluta de 45 grados, en una línea perpendicular al eje X (90 grados) estaría 45 grados inclinado. Un texto con una rotación relativa de 45 grados, en la misma línea estaría 135 grados inclinado.

Y respecto a la escala, decir que es muy importante elegir un factor adecuado, ya que con una escala muy pequeña puede no llegar a verse el texto en pantalla.

Vamos a tratar ahora los tipos de línea con formas intercaladas. Las formas son pequeños objetos de **AutoCAD** que se definen en un archivo de extensión .SHP, el cual se compila luego para obtener un .SHX manejable por el programa. Para visualizar las formas de un archivo de formas deberemos cargarlo primero con el comando CARGA (LOAD) y luego insertarlas con FORMA (SHAPE). Podemos ver sus nombres con la opción ?, y luego *, de este último comando.

NOTA: En el **MÓDULO CINCO** se tratará por completo la creación y personalización de archivos de formas.

AutoCAD provee un archivo con formas creadas. Este archivo se llama LTYPESHP.SHX y podemos acceder también al archivo fuente LTYPESHP.SHP. Ambos se encuentran en el directorio \SUPPORT\ del programa y han sido creados precisamente para que sus formas sean incluidas en patrones de tipos de línea.

Nosotros podemos intercalar en los tipos de línea de **AutoCAD** formas, al igual que textos. La sintaxis, dentro de la propia de la línea, para realizar esto es la que se muestra a continuación:

```
... [nombre_forma,nombre_archivo,S=factor_escala,R|A=ang_rotación,X=despl_X,  
Y=despl_Y] ...
```

Toda la definición aparecerá encerrada entre corchetes ([]); los diferentes parámetros separados por comas (,) y sin espacios blancos.

Los parámetros de escala (S), rotación (R o A), desplazamiento en X (x) y desplazamiento en Y (Y) siguen siendo los mismos, y con la misma función, que en los tipos de línea complejos con texto intercalado. Desaparece, evidentemente, el estilo de texto y aparecen dos nuevas entradas.

- *nombre_forma*. Es el nombre de la forma que se encuentra dentro del archivo de definición de formas.
- *nombre_archivo*. Es el nombre del archivo de formas donde se encuentra dicha forma. Debe ser el .SHX, no el .SHP.

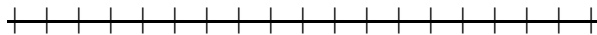
NOTA: No es necesario incluir todos los modificadores en una definición, sino que pueden existir algunos y otros no, al igual que con los textos. Eso sí, el nombre de la forma y el archivo donde se encuentra incluida son obligatorios.

NOTA: Las formas del archivo LTYPEHP.SHX son: BAT (∩, con los tramos verticales rectos), CIRC1 (O), ZIG (^), TRACK1 (|) y BOX (□).

Vamos a ver tres ejemplos:

```
*VIAS, _ _ | _ _ | _ _ | _ _ | _ _  
A, .15,[TRACK1,ltypeshp.shx,S=.25], .15
```

El resultado sería algo como lo que sigue:



Otro ejemplo:

```
*CIRC, --o-o-o-o-o-o--  
A, 1,[CIRC1,ltypeshp.shx,S=.1],-.2
```

Al igual que en los tipos de línea con textos, el espacio para las formas no se reserva. Es por ello que debemos indicárselo nosotros explícitamente. Hay veces en los que dicho espacio o hueco no nos interesará, como en el primer ejemplo (el de las vías), pero otras —el segundo ejemplo— sí se necesitará. En este último ejemplo, primero se dibuja un trazo de 1 y, después se intercala la forma. Dicha forma es un círculo unitario (radio 1) que, al aplicársele un factor de escala de 0,1 unidades, se convierte en un círculo de 0,1 de radio, esto es, diámetro 0,2. Por eso, hay que indicar al final un hueco de 0,2 unidades de dibujo para que el siguiente trazo se comience a dibujar tras la forma.

NOTA: Como veremos en su momento, todas las formas han de ser unitarias, o sea, en este caso, el círculo es de radio 1, el cuadrado tiene un semilado de 1, la barra vertical tiene de semilongitud 1, etcétera.

Último ejemplo:

```
*Lim_Terr  
A, 1,[LIMIT,topograf.shx,S=.1,R=45,X=-.25,Y=-.35], -1.5
```

Como último apunte diremos que, en la creación de tipos de línea complejos es posible intercalar más de un texto o más de una forma en cada línea. El método es idéntico. Por ejemplo:

```
*CUARCIR  
A,1,[CIRC1,ltypeshp.shx,S=.1],-.2,1,[BOX,ltypeshp.shx,S=.1],-.2  
  
*ELEC  
A,1,-.3,["ELEC",STANDARD,S=.1,A=45,X=-.17,Y=-.17],-.3,.5,[  
"1",STANDARD,S=.1],-.5
```

O incluso, textos y formas combinados:

```
*ELEC2  
A,1,-.3,["ELEC",STANDARD,S=.1,A=45,X=-.17,Y=-.17],-.3,.5,[BOX,  
ltypeshp.shx,S=.1],-.2
```

NOTA: No se puede comenzar una definición de tipos de línea con carácter complejo (sea texto o forma) o con hueco. Debemos de empezar con trazo o con punto. Tampoco es recomendable terminar con carácter complejo (aunque se puede).

NOTA: Es irrelevante la utilización de mayúsculas o minúsculas en cualquiera de las dos líneas del patrón.

NOTA: La coma anterior al corchete de apertura, en tipos de línea complejos, y la posterior al corchete de cierre han de estar pegadas a ellos, esto es, sin espacios. Por lo demás, sabemos que podemos introducir espacios aclaratorios (siempre después de comas; nunca entre número y coma).

TRES.2.2.3. Creación desde la línea de comandos

Los tipos de línea simples también podemos crearlos desde la propia línea de comandos de **AutoCAD**, no así los tipos complejos. La forma de crear tipos de línea simples así es mediante el comando **TIPOLIN** (**LINETYPE** en inglés). Para utilizarlo desde la línea de comandos deberemos introducirlo precedido de un guión: **-TIPOLIN (-LINETYPE)**, ya que de otra forma, lo que hará es mostrar el cuadro de diálogo *Propiedades de las capas y de los tipos de línea*, en su pestaña *Tipo línea*.

NOTA: Esta característica del guión se vio en el **MÓDULO UNO**. Existen muchos comandos de **AutoCAD** que tienen un doble formato de uso, uno con cuadro de diálogo y otro desde la línea de comandos. En estos casos (cuando el nombre del comando es el mismo para los dos), se antepone un guión (-) al comando para que se ejecute su versión de línea de comandos. Estos comandos suelen utilizarse casi exclusivamente para macros de menús, botones de barras de herramientas y demás.

Pues bien, tras ejecutar el comando de la forma indicada en la línea de comandos aparece:

?/cRear/Cargar/Def:

La opción **?** lista todos los tipos de línea, con sus correspondientes comentarios si los hay, contenidos en un archivo que hay que indicar. La opción **Cargar** carga el tipo o tipos de línea indicados en un archivo que hay que especificar. La opción **Def** establece un tipo de línea cargado como actual. Pero, la que nos interesa es la opción **cRear** que va a servirnos para la creación de estos tipos de línea simples.

Pues bien, una vez introducida la **R** para activar la opción aparecerá lo siguiente:

Nombre del tipo de línea a crear:

donde introduciremos el nombre en sí de la línea que se pretende crear (lo que en los archivos de definición estudiados se encuentra inmediatamente después del asterisco (*)).

Una vez hecho esto se muestra el cuadro de *diálogo Crear o añadir archivo de tipo de línea*, donde elegiremos un archivo para añadirle la nueva definición o crearemos uno nuevo, simplemente eligiendo ubicación y dándole un nombre no existente. Si creamos uno nuevo, el programa indica en línea de comandos **Creando archivo nuevo**, si anexamos las definiciones a uno que ya existe se indica **Espere**, comprobando si el tipo de línea está ya definido.... Si el tipo de línea ya existe en el archivo especificado, **AutoCAD** mostrará su descripción preguntando si se desea modificar. Ante una respuesta negativa, el programa solicitará otro nombre de tipo de línea.

El siguiente paso es indicarle el texto opcional de descripción (revisemos las definiciones estudiadas) tras el mensaje:

Texto de descripción:

Podremos darle INTRO para no escribir ninguno y, si no, deberemos acordarnos de no sobrepasar los 47 caracteres.

Por último se nos pregunta por el patrón:

Introducir patrón (en la línea siguiente):
A,

Como vemos, **AutoCAD** escribe por defecto el alineamiento y una coma (A,) lo demás corre de nuestro cargo. Sólo hemos de seguir las mismas explicaciones que al crear un patrón de tipo de línea desde un editor ASCII.

NOTA: Podemos introducir aquí también espacios blancos a modo de ordenamiento visual. Si **AutoCAD** detecta algún error en el tipo de línea, el proceso se aborta y habrá que empezar desde el principio.

TRES.3. CARGAR TIPOS DE LÍNEA CREADOS

La manera de cargar uno o varios tipos de línea creados es conocida por todos, ya que no difiere apenas de la forma de cargar los tipos de línea que trae **AutoCAD**.

Desde *E formato > Tipo de línea...*, desde el botón *Tipo de línea* de la barra de herramientas de **AutoCAD** *Propiedades de objetos*, desde la línea de comandos mediante TIPOLIN (en inglés LINETYPE) o, incluso, desde el botón de control de capas y luego cambiando de pestaña, accedemos al cuadro *Propiedades de las capas y los tipos de línea*, en su pestaña *Tipo línea*.

En este cuadro existe, a la derecha, un botón llamado *Cargar...*. Pulsándolo accedemos a un nuevo cuadro de diálogo, denominado éste *Cargar o volver a cargar tipos de línea*. En el cuadro bajo *Tipos de línea disponibles* se encuentran todos los tipos de línea cargados correctamente (los erróneos no aparecerán) que están definidos en el archivo de definición de tipos de línea indicado en la casilla superior, a la derecha del botón *Archivo...*. Además, las barras superiores *Tipo línea* y *Descripción* permiten ordenar A-Z y Z-A —típico en entorno Windows— tanto los nombres de las líneas como sus descripciones.

Si pulsamos el botón *Archivos...* mencionado podremos elegir el archivo de definición del que queremos cargar tipos de línea. Es el cuadro estándar de elección para apertura de archivos de Windows con el nombre *Seleccionar archivo de tipo de línea*. Una vez buscado y elegido, pulsamos *Abrir* y volveremos al cuadro anterior donde se mostrarán los tipos de línea que contiene el archivo.

NOTA: Si existe algún error en alguna línea del archivo de definición, el tipo de línea en concreto no se cargará (**AutoCAD** mostrará un mensaje), pero los demás, si están correctos, sí.

Ahora únicamente debemos elegir el tipo o los tipos de línea que queremos cargar de dicho archivo. Podemos utilizar la tecla CTRL y la tecla SHIFT para elegir archivos al estilo Windows. Pulsamos *Aceptar* y ya están los nuevos tipos cargados.

NOTA: Si intentamos cargar un tipo de línea ya cargado, **AutoCAD** muestra un mensaje diciendo que ya está cargado y si queremos recargarlo. Esto es útil cuando se trabaja interactivamente con **AutoCAD** y el editor en que se tiene abierto el archivo .LIN. Podemos editar nuestro tipo de línea, volver a **AutoCAD** y recargarlo, sin la necesidad de borrarlo primero con el botón **Borrar** del cuadro de diálogo principal. Un último apunte, este botón Borrar no borra las definiciones del archivo, sino que descarga los tipos de línea cargados y seleccionados.

Tras cargarlo, sólo debemos probarlo y utilizarlo de la manera más sencilla: trazando líneas. Antes deberemos haber escogido el tipo de línea de la lista *desplegable Control de tipos de línea* de la barra de herramientas de **AutoCAD Propiedades de objetos**, o habérselo asignado a una capa establecida como actual y con el tipo de línea PORCAPA, etcétera.

Otra manera de cargar tipos de línea es con la opción Cargar del comando -TIPOLIN (desde la línea de comandos) antes visto. Se elige primero el tipo o tipos que serán cargados y luego el archivo .LIN.

NOTA: Desde *Herr.>Preferencias...*, en la pestaña *Archivos* del cuadro *Preferencias*, en la opción *Camino de búsqueda de archivos de soporte*, podemos especificar la ruta de búsqueda a archivos de definición de tipos de línea que no se encuentren en el directorio actual.

TRES.4. EJEMPLOS PRÁCTICOS DE TIPOS DE LÍNEA

TRES.4.1. Tipo simple 1

```
*Línea_1, _ . . _ . . _  
A, 10,-5, 0,-5,0,-5, 5,-5, 0,-5,0,-5
```

NOTAS INTERESANTES:

1. Nótese la separación puramente decorativa entre diversos grupos de trazos, huecos o puntos dentro del mismo patrón de tipo de línea. Su misión es únicamente dar claridad.
2. Al final de un archivo completo de definición de tipos de línea (tenga las definiciones que tenga), y como ocurría con los archivos de menú, es necesario introducir un INTRO —al final del último carácter de la última línea— para que todo funcione correctamente.

TRES.4.2. Tipo simple 2

```
*Línea_2, _ . . . . _ . . . . _  
A, 5,-5, 0,-5,0,-5,0,-5,0,-5
```

TRES.4.3. Tipo complejo 1

```
*AGUA_FRIA,Circuito de agua fría _ AF _ AF _ AF _  
A,.5,-.2,["AF",STANDARD, S=.1,R=0,X=-.1,Y=-.05],-.2
```

TRES.4.4. Tipo complejo 2

```
*ELEC, _ ELEC _ ELEC _ (inclinado)
```

A,1,-.3,["ELEC",STANDARD,S=.1,A=45,X=-.17,Y=-.17],-.3

TRES.4.5. Tipo complejo 3

*MIXTO, Cuadrados y círculos

A,1,[CIRC1,ltypeshp.shx,s=.1],-.2,1,[BOX,ltypeshp.shx,s=.1],-.2

TRES.FIN. EJERCICIOS PROPUESTOS

- I. Crear un tipo de línea simple compuesta por los siguientes elementos y en el siguiente orden: trazo largo, hueco, punto, hueco, trazo corto, hueco y vuelta a empezar. Las dimensiones como se elijan. (___ . _ __ . _ __ . _ __).
- II. Diseñese un tipo de línea con un texto justificado a la izquierda dentro de un hueco flanqueado por trazos de longitud cualquiera. (___Texto ___Texto ___Texto ___).
- III. Crear un tipo de línea que incluya un texto a 90 grados con respecto a la línea (ángulo relativo). El texto habrá de estar perfectamente centrado en un hueco, en cuyos extremos habrá sendos puntos. El resto de la línea a gusto del creador. El estilo y la escala también a gusto del creador. (___ . Texto . ___, con el texto girado 90 grados).
- IV. Diseñar un tipo de línea que incluya diversas formas perfectamente alineadas y pegadas a la línea. (--□--○--^--□--○--^--□--○--^--).
- V. Crear un tipo de línea que incluya una forma y un texto, éste último, girado 30 grados de manera absoluta y perfectamente centrado en su hueco. (--○--Texto--○--Texto--, el texto girado lo estipulado).
- VI. Desarrollar un juego completo de patrones de tipos de línea para su manejo por profesionales de la topografía.

EJERCICIOS RESUELTOS DEL MÓDULO DOS

EJERCICIO I

Botón 1

Nombre del botón: Orto

Ayuda: ACT/DES el modo Orto

Macro: ^O

Botón 2

Nombre del botón: Forzcursor

Ayuda: ACT/DES el modo Forzcursor

Macro: ^B

Botón 3

Nombre del botón: Rejilla

Ayuda: ACT/DES la Rejilla

Macro: ^G

NOTA: Cuidado al escribir ^O. Si tecleamos el carácter ^ y luego el carácter O aparecerá Ô; habremos de teclear ^, un espacio y, luego, O.

EJERCICIO II

Nombre del botón: Web del programador

Ayuda: Abre La Web del Programador en el navegador por defecto

Macro: ^C^C_browser <http://www.casarramona.com/mt/programador/index.htm>

EJERCICIO III

Botón 1

Nombre del botón: SCP X

Ayuda: Gira el SCP sobre X

Macro: ^C^C_ucs x \

Botón 2

Nombre del botón: SCP Y

Ayuda: Gira el SCP sobre Y

Macro: ^C^C_ucs y \

Botón 3

Nombre del botón: SCP Z

Ayuda: Gira el SCP sobre Z

Macro: ^C^C_ucs z \

EJERCICIO IV

Nombre del botón: SCP X, Y y Z

Ayuda: Botón de giros del SCP actual

Barra de herramientas asociada: CURSO.SCPGiros

NOTA: Este ejemplo supone un nombre de SCPGiros para la barra anterior y un nombre de CURSO para el grupo de menú (**MENUGROUP) del archivo donde está guardada.

EJERCICIO V

Botón 1

Nombre del botón: Amplía

Ayuda: Establece la ventana actual como ventana única

Macro: ^C^C_vports _s 3d _y;vports _si

Botón 2

Nombre del botón: Restituye

Ayuda: Establece la antigua configuración de 3 ventanas

Macro: ^C^C_vports _r 3d

NOTA: Este ejemplo supone ya guardada la configuración de nuestras 3 ventanas con el nombre 3d. Si no realizamos este paso previo los botones no funcionarán correctamente. Una vez guardada dicha configuración podemos utilizarlos tranquilamente. Sería útil guardar la configuración de 3 ventanas (si las usamos a menudo) en un archivo de plantilla .DWT, de esta manera se cargará al cargar la plantilla y podremos utilizar los botones del ejemplo.

EJERCICIO VI

Nombre del botón: ZoomExtensión4V

Ayuda: Realiza Zoom Extensión en 4 ventanas y se aleja

Macro: ^C^C_cvport 3 _z _e _z .9x;_cvport 2 _z _e;_cvport 4 _z _e _z .9x;_cvport 4 _z _e _z .9x

NOTA: La variable de sistema CVPORT almacena el número de la ventana actual. Podemos acceder a las distintas ventanas, además de por los métodos conocidos, indicando su número de CVPORT (desde la línea de comandos). **AutoCAD** no da un orden lógico de numeración a las ventanas, comienza por el número 2 (el 1 lo reserva para Espacio Papel) y le da números consecutivos a ventanas alternas, sin orden lógico aparente. Únicamente deberemos teclear CVPORT en la ventana actual para saber qué número tiene asignado (o con el comando VENTANAS, VPORT en inglés, y la opción ?) y personalizar este ejemplo para saber cuál es nuestra ventana en isométrico y que no se aleje en ella (CVPORT 2 aquí). De todas formas, si no se guarda la configuración de las ventanas —en una plantilla, por ejemplo—, perderemos la funcionalidad de este botón, ya que al crear otras nuevas en otra sesión de dibujo, los identificadores CVPORT variarán. Veremos mucho más adelante que con AutoLISP podemos obtener un mayor control en esta faceta.

EJERCICIO VII

Nombre del botón: EscalaXY

Ayuda: Escala objetos independientemente en X e Y

Macro: ^C^C_select _block obj _p ;_insert obj @ \\\0;_explode _l;_purge _b obj
_n

NOTA: Este ejemplo ofrece mucho juego a la hora de escalar objetos porque, como decíamos al proponer el ejercicio en el **MÓDULO** anterior, permite escalar en X y en Y de una manera independiente; cosa que el comando ESCALA (SCALE) de **AutoCAD** no hace. La mecánica es bien simple: sabemos que al insertar un bloque sí se nos permite escalar de modo diferente en X que en Y (e incluso en Z). Pues sólo tenemos que crear un bloque con el objeto que se quiere escalar e insertarlo. La macro pide primero un conjunto de selección (comando DESIGNA, SELECT en inglés; muy utilizado en las macros y en la programación en general). Permite designar una serie de objetos a los que luego se accederá con el modo de designación de objetos PREVIO (PREVIOUS en inglés), o su abreviatura P. Crea un bloque con los objetos al que llama OBJ, lo inserta y da la posibilidad de escalar en X y en Y de forma independiente. Tras esto, descompone el bloque para recuperar los objetos simples y lo limpia, indicando que no se pregunte para verificar. Lo dicho en el anterior **MÓDULO**: para pensar un poco.

EJERCICIO VIII

(Ejercicio completo para resolver por técnicos y/o especialistas).

MÓDULO CUATRO

Creación de patrones de sombreado

CUATRO.1. PATRONES DE SOMBREADO

Al igual que los tipos de línea, en **AutoCAD** es posible definir nuestros propios patrones de sombreado. Además, la técnica es muy parecida a la creación de líneas y no será difícil asimilarla si se adquirieron los conocimientos suficientes en el **MÓDULO** anterior.

Un patrón de sombreado de **AutoCAD** es un conjunto de caracteres ASCII que se guarda en un archivo de extensión `.PAT` específico. Contrariamente a lo que ocurría con los tipos de línea, las definiciones de los patrones de sombreado no podemos guardarlas en cualquier archivo y darle la extensión `.PAT`, sino que deberemos guardarlas en uno de los archivos que **AutoCAD** proporciona con los patrones de sombreado predefinidos. Estos archivos son dos y, al igual que los tipos de línea, uno dice referencia a los patrones en pulgadas (`ACAD.PAT`) y, el otro, a los mismos patrones en unidades métricas a escala según normas ISO (`ACADISO.PAT`). Ambos ficheros se encuentran recogidos en el directorio `\SUPPORT\` del programa.

AutoCAD provee al usuario de 54 patrones básicos de sombreado, además de 14 relacionados con tipos de línea según norma ISO/DIS 12011. Esto hace un total de 68 patrones de sombreado distintos.

NOTA: Como habremos podido comprobar, en el cuadro de diálogo para la elección de patrones de sombreado no existe una opción que nos lleve a una búsqueda de archivos propios de definición. Esta es la razón por la que habremos de incluir nuestros patrones propios en uno de los dos archivos de soporte correspondientes de **AutoCAD**.

CUATRO.2. SINTAXIS DE LA DEFINICIÓN

Como hemos dicho, habiendo comprendido completamente la definición de tipos de línea, tenemos medio camino recorrido a la hora de definir patrones de sombreado. Si examinamos atentamente el archivo `ACADISO.PAT`, podremos comprobar la semejanza que tiene con `ACADISO.LIN`.

El carácter punto y coma (;) se utiliza aquí también para la escritura de comentarios. Como se explicó en el **MÓDULO TRES**, un solo carácter ; es suficiente para definir una línea de comentarios. Lo que ocurre es que, a veces, se incluyen dos para dar mayor claridad al archivo. Podemos incluir tantos comentarios como queramos, teniendo en cuenta el tiempo de proceso que se invertirá si existen demasiados. Estas líneas serán directamente ignoradas por **AutoCAD**.

Aparte de los comentarios, y de líneas blancas de separación por claridad, el resto de renglones en un archivo de definición de patrones de sombreado se refieren a la propia definición del patrón.

Un patrón de sombreado se define con dos o más líneas dentro de un archivo. La primera línea obligatoria tiene la sintaxis siguiente (los corchetes en *itálica* indican la no obligatoriedad del parámetro):

**nombre_patrón_sombreado[, descripción]*

El carácter asterisco (*) es obligatorio en esta primera línea —como ocurría con los tipos de línea—. Este asterisco le dice al programa que lo que viene a continuación es una definición de un patrón de sombreado. *nombre_patrón_sombreado* se refiere al nombre que le damos al patrón en cuestión. Este nombre es el que aparece en la lista izquierda del menú de imagen de elección de patrones de sombreado (al pulsar en el botón *Patrón...* en el cuadro principal de sombreado *Sombreado por contornos*). Asimismo, aparece también en la lista desplegable *Patrón:* del cuadro *Sombreado por contornos*.

NOTA: Recordar que el cuadro *Sombreados por contornos* es el que aparece al elegir *Dibujo>Sombreado...*, al pulsar sobre el icono correspondiente en la barra de herramientas *Dibujo*, con los comandos de línea *SOMBCONT* (SB o SBC) o *BHATCH* (BH o H, en inglés) o desde la casilla *P-9* de la plantilla original de **AutoCAD** para tableta digitalizadora. Es el cuadro principal para la edición de sombreados.

Por último, y en esta primera línea, se puede incluir una descripción —tras una coma— del patrón de sombreado. A diferencia de los tipos de línea, esta descripción no suele ser gráfica en modo ASCII, ya que resulta poco menos que imposible dibujar de este modo algo que se asemeje a un patrón de sombreado. Dicha descripción se limita a un texto que explique el sombreado; no es obligatoria. Este texto aparecerá al ejecutar desde la línea de comandos la orden *SOMBREA* (*HATCH*) y elegir la opción ?.

Las siguientes líneas son las que definen propiamente el patrón. Su sintaxis es la que sigue:

ángulo,origen_X,origen_Y,desfase,distancia[, tipo_línea]

De esta forma quedan declarados todos los parámetros necesarios para la definición de cada línea de rayado o sombreado. Para conseguir comprender como se genera un sombreado es óbice explicar el concepto de *barrido*.

Un sombreado se compone de uno o varios barridos de líneas. Un barrido es un conjunto de líneas paralelas entre sí y, una línea de cada barrido es suficiente para definir éste. Por lo tanto, para definir un patrón de sombreado bastará definir cada una de las líneas que genera cada uno de los barridos. Esto puede parecer un poco lioso pero, en el momento en que se entienda bien resulta lógico y sencillo. Vamos a intentar explicarlo con un ejemplo.

Supongamos el ejemplo siguiente. Un tipo de sombreado que represente una sombra de triángulos equiláteros. Considerando un triángulo equilátero podemos dividirlo en sus tres lados, cada uno es un segmento de una determinada longitud y todos ellos de la misma. Dibujaremos en la imaginación uno de los segmentos de dicho triángulo en un papel de transparencias. Otro de los segmentos en otro papel igual, de forma que, al superponerlos, coincidan perfectamente en un punto y formen dos de los lados del triángulo. En un tercer papel transparente del mismo formato dibujaremos el tercer lado, cuidando dibujarlo en el lugar adecuado también para que, al superponer las tres transparencias se pueda ver el triángulo representado de manera perfecta.

Si ahora, en cada transparencia dibujáramos líneas paralelas a las existentes en cada una de ellas, todas a la misma distancia, con las mismas inclinaciones y con las mismas distancia que en cada una de ellas, al superponer todas las hojas obtendríamos un tramado de triángulos, esto es, un sombreado.

Pues bien, cada una de las hojas transparentes (con sus correspondientes líneas paralelas) es lo que asemejaremos con un barrido. Todos los barridos juntos formarán el sombreado. Ésta es la explicación.

Lo que ocurre es que no será necesario dibujar todas y cada una de las líneas de cada barrido, ya que son paralelas e iguales, sino únicamente las suficientes para la definición, además de la distancia que habrá hasta sus semejantes y otros pocos datos.

Todo esto se realiza con los parámetros de la sintaxis antes expuesta. Cada uno de ellos se explica por separado a continuación.

- *ángulo*. Es el ángulo formado por la línea de rayado con la referencia de 0 grados. Los signos son los trigonométricos, es decir, el antihorario es el positivo.
- *origen_X* y *origen_Y*. Son las coordenadas del punto de origen de la línea de rayado. Se explicará más detalladamente un poco más adelante.
- *desfase*. Dice relación al desplazamiento que tendrá cada línea de rayado con la siguiente del conjunto que forma el barrido. Sólo tiene sentido en líneas de trazo discontinuo y es medido en el sentido de la propia línea de rayado. Lo veremos enseguida.
- *distancia*. La separación entre cada línea que forma el barrido.
- *tipo_línea*. Descripción del patrón de tipo de línea en cuestión. Esta definición se construye de exactamente la misma manera que lo estudiado en el **MÓDULO TRES**. Si no se incluye, ya que es optativo, se supone una línea continua.

Tras ver todo esto, vamos a intentar construir un sombreado de cuadrados; dejaremos el expuesto anteriormente de los triángulos por ser más complejo para empezar. Siempre que vayamos a definir un patrón de sombreado habremos de tener muy en cuenta el número de barridos que lo forman, es decir, deberemos descomponer dicho sombreado en un número de barridos válido. A veces, un tipo de rayado o sombreado se puede construir con distinto número de barridos (como en este caso). En estos caso, parece lógico utilizar el menor número de líneas de definición para el archivo .PAT, esto es, el menor número de barridos.

Lo primero de todo, la línea de encabezado y descripción. Por ejemplo:

```
*LOSAS, Sombreado de cuadrados de lado 1
```

NOTA: Tendremos las mismas consideraciones, respecto a nombres y caracteres, para esta línea que las explicadas para los encabezados de los tipos de línea en el **MÓDULO** anterior.

Una trama de cuadrados la podemos definir con cuatro barridos (uno para cada lado del cuadrado), sin embargo, parece más lógico utilizar únicamente dos, ya que los lados de un cuadrado son todos iguales y paralelos dos a dos. Nuestro sombreado, pues, estará formado por cuadrados de lado 1 y con distancia de uno a otro de 1 unidad de dibujo también. Las dos líneas definidoras de sendos barridos serían:

```
0, 0,0, 0, 1, 1, -1  
90, 0,0, 0, 1, 1, -1
```

Por lo que la definición completa quedaría así:

```
*LOSAS, Sombreado de cuadrados de lado 1  
0, 0,0, 0, 1, 1, -1  
90, 0,0, 0, 1, 1, -1
```

NOTA: Con la definición de uno de los cuadrados es suficiente, ya que luego se repiten. Esto mismo ocurriría con los trazos de los patrones de tipos de línea.

Vamos a explicar todos los pasos. El primero de los renglones propios de la definición (tras el encabezado) se refiere a todas las líneas horizontales de los cuadrados. Es por ello que le damos un ángulo de 0 grados.

Después se indica el punto de partida de dichas líneas; el origen X y el origen Y. Este origen se hace necesario a la hora de definir sombreados compuestos de varios barridos. Supongamos que dibujamos en una hoja el rayado que queremos conseguir, acción muy aconsejable, por cierto. Al punto inferior izquierdo de nuestro cuadrado le vamos a dar el valor teórico de 0,0. Desde ese punto partirá la línea. Podemos colocar el 0,0 en cualquier otro punto, teniéndolo luego en cuenta a la hora de dibujar los barridos restantes.

Tras el origen se indica un desfase de 0 unidades de dibujo. Un desfase hace que cada línea paralela del mismo barrido se incremente una distancia con respecto al punto de origen, o sea, que comience en otro punto. Líneas sin desfase serían las siguientes:

```
-- -- -- -- -- -- -- -- -- --
-- -- -- -- -- -- -- -- -- --
-- -- -- -- -- -- -- -- -- --
-- -- -- -- -- -- -- -- -- --
```

Y líneas desfasadas estas otras:

```
-- -- -- -- -- -- -- -- -- --
  -- -- -- -- -- -- -- -- -- --
-- -- -- -- -- -- -- -- -- --
  -- -- -- -- -- -- -- -- -- --
```

NOTA: El desfase, evidentemente, sólo tiene sentido en tipos de línea discontinuos.

Volviendo a nuestro ejemplo, no indicamos desfase porque nos interesa que cada cuadrado tenga todos sus lados coincidentes.

La distancia de 1 unidad de dibujo, siguiente parámetro, se refiere a la existente entre cada línea paralela del mismo barrido.

Y, por último, sólo queda indicar el patrón de tipo de línea, en este caso una línea discontinua de un trazo de 1 y un hueco de 1.

NOTA: Revísese si es necesario el **MÓDULO TRES** sobre creación de tipos de línea.

Toda esta primera línea de definición, por lo tanto, produce un barrido tal que así:

```
-- -- -- -- -- -- -- -- -- --
-- -- -- -- -- -- -- -- -- --
...

```

La segunda línea, por su lado, es de idéntica definición. Únicamente varía el ángulo de la línea, que es de 90 grados. De esta manera, el barrido producido por esta segunda línea sería el siguiente:

```
| | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | |
...

```

Ambos barridos juntos completarían el sombreado de cuadrados.

NOTA: Es muy importante tener en cuenta el punto de origen de cada barrido. Como punto de partida se suele utilizar siempre el 0,0. Esto significa que el punto que nosotros escojamos como punto de partida, y si se hace un sombreado en las inmediaciones del 0,0 de **AutoCAD** lo podemos comprobar, estará en esa coordenada. Aunque el sombreado no ronde este origen seguirá dibujándose correctamente, únicamente se da esta referencia para que todos los barridos coincidan y el dibujo final sea el esperado.

Una vez construido este sombreado, no nos será difícil realizar y entender otros como el que sigue:

```
*DOBLE, Líneas cruzadas
45, 0,0, 0, 1
135, 0,0, 0, 1
```

NOTA: Nótese un par de aspectos. El primero es la separación mediante espacios blancos de grupos de números de definición. Esto se hace únicamente para aclarar el sentido de la línea. Pueden separarse los grupos (ángulo, coordenadas, distancia...) para darle más claridad a la definición (se pueden introducir tantos espacios como se quiera). Como segundo aspecto, notar la ausencia en este último ejemplo de definición del patrón de tipo de línea. Esto es posible hacerlo y significa, como hemos comentado al principio, que es una línea continua.

Este patrón muestra un entramado de líneas que se cortan a 90 grados. Cada barrido está inclinado 45 y 135 grados respectivamente.

Como conclusión, recordar que el patrón de sombreado tendrá tantos renglones como barridos formen el sombreado. Cada renglón de definición tendrá como máximo seis combinaciones de trazos, espacios y puntos (seis valores numéricos en la descripción del archivo de texto).

NOTA: Al igual que en los tipos de línea y en los archivos ASCII de definición de menús, en el archivo de patrones de sombreado se hace necesario un INTRO al final de la última línea para que el último patrón funcione perfectamente.

CUATRO.3. TÉCNICA DE CREACIÓN

La técnica que debemos seguir a la hora de crear un patrón de sombreado, sobre todo al principio, ha de ser metódica para no tener demasiados problemas.

Tras decidir cómo queremos que sea el rayado o trama que vamos a definir, habremos de dibujarlo en un papel para su mejor comprensión. En dicho dibujo decidiremos el punto de origen del patrón, así como las medidas, según la escala elegida, de los trazos y huecos (si los hubiera), la situación de los puntos (si los hubiera, también), la distancia entre barridos, el ángulo y el desfase.

Una vez hecho esto, iremos definiendo el patrón línea por línea —si es muy complejo— en un editor ASCII y probándolo en **AutoCAD**, pero sin aceptar del todo su dibujado, es decir, realizando una previsualización y cancelando. El problema, a la hora de aceptar por completo el dibujado de un patrón de sombreado, es que **AutoCAD** se quedará utilizando el archivo correspondiente (ACAD.PAT o ACADISO.PAT) y, aunque lo borremos, no podremos guardar más cambios en el archivo ASCII si no salimos de **AutoCAD**. Esto ocurre también cuando se nos produce un error de definición a la hora de utilizar un sombreado, así que hay que tener mucho cuidado. Si, aún así, ocurriera, sólo deberíamos salir de **AutoCAD** para poder guardar los cambios en el archivo y volver a correr el programa.

CUATRO.4. UTILIZANDO EL PATRÓN DEFINIDO

Para utilizar un patrón de sombreado recién creado, sólo debemos escogerlo como el resto de los patrones inherentes al programa. Esto lo podemos realizar de, al menos, dos maneras diferentes: desde la línea de comandos y desde el cuadro de diálogo preparado a tal efecto. Lo más usual es que utilicemos esta última opción, desde Dibujo>Sombreado... (comando SOMBCONT).

Desde este cuadro de diálogo elegiremos nuestro patrón en la lista desplegable llamada *Patrón:*, o accediendo al menú de imagen *Paleta de patrones de sombreado* pinchando en el botón *Patrón...* del mismo cuadro.

NOTA: En este cuadro, como sabemos, tenemos que elegir una escala adecuada al patrón que se va a dibujar. De todas maneras, y al igual que en los tipos de línea, es lógico indicar las medidas de la definición en el archivo .PAT similares a las de los patrones incluidos en el mismo.

CUATRO.4.1. Iconos del menú de imagen

El cuadro de elección de patrones de sombreado *Paleta de patrones de sombreado* no es otra cosa que un menú de imagen propio de **AutoCAD** que contiene todos los patrones de sombreados definidos por defecto. Al incluir un patrón nuevo en el archivo ACADISO.PAT (o en ACAD.PAT), la lista se actualiza con el nombre del nuevo patrón incluido. En dicha lista, aparecerá el nuevo patrón en el lugar donde haya sido introducido por el creador.

NOTA: Si modificamos alguna definición en el fichero .PAT sin salir del cuadro de diálogo de sombreado (SOMBCONT), dicho cambio no surtirá efecto hasta que salgamos y volvamos a entrar en ese cuadro.

Pero, como habremos percibido, la pequeña imagen relacionada con nuestro patrón, y que aparece a la derecha del menú de imagen comentado, no existe. Esta imagen es una foto de **AutoCAD** que deberemos sacar y deberemos incluir en la fototeca correspondiente (ACAD.SLB). Como se trata de añadir una imagen a una biblioteca de fotos, no podremos hacerlo correctamente con el programa de manejo de fototecas que proporciona **AutoCAD**, esto es SLIDELIB, sino que tendremos que utilizar otro.

Pero como todo lo referente a fotos y fototecas será comentado en el **MÓDULO OCHO** de este curso, vamos a dejar este pequeño aspecto para dicha parte posterior. Por ahora, y como hicimos con los menús de imagen del **MÓDULO UNO**, conformémonos con el texto de la lista izquierda y sin foto.

Debido a esta característica de utilizar programas basados en MS-DOS para el manejo de fototecas, es conveniente que demos nombres de menos de ocho caracteres —y con símbolos permitidos— a los patrones de sombreado. Como se ha comentado más de una vez, estos nombres basados en MS-DOS deberán ser los típicos que utilicemos en cualquier archivo personalizable de **AutoCAD**.

CUATRO.5. EJEMPLOS PRÁCTICOS DE PATRONES DE SOMBREADO

CUATRO.5.1. Patrón sólido

*SOLIDO, Sombreado sólido
0, 0,0, 0, .1

NOTAS INTERESANTES:

1. Nótese que no se puede definir un sombreado completamente sólido (con distancia igual a 0). **AutoCAD** devolvería siempre un error de sombreado demasiado denso, fuese cual fuese la escala. Hay que simularlo con distancias muy pequeñas entre líneas.
2. Como se ve, no es necesario incluir definición de tipo de línea si ésta es continua. Las demás definiciones son todas obligatorias.

CUATRO.5.2. Patrón simple inclinado

*ANG, Sombreado inclinado a 30 grados
30, 0,0, 0, 1

CUATRO.5.3. Patrón de líneas cruzadas a 90 grados

*Cruz, Cruzadas a 90 grados
0, 0,0, 0, 1
90, 0,0, 0, 1

CUATRO.5.4. Patrón de hexágonos

*HEX, hexágonos
0, 0,0, 0, 5.49926, 3.175,-6.35
120, 0,0, 0, 5.49926, 3.175,-6.35
60, 3.175,0, 0, 5.49926, 3.175,-6.35

CUATRO.FIN. EJERCICIOS PROPUESTOS

- I. Crear un patrón de sombreado compuesto por líneas de un trazo y dos puntos, inclinadas 45 grados y desfasadas una cantidad de unidades de dibujo entre sí.
- II. Crear un patrón formado por 3 líneas continuas que se crucen.
- III. Diseñese un patrón de sombreado compuesto por líneas continuas a 90 grados agrupadas de dos en dos. Existirá un espacio X entre dos líneas y un espacio mayor que X desde cada grupo de dos líneas al siguiente.
- IV. Diseñar un patrón de sombreado que dibuje cuadrados de lado 1 y que posea un intersticio de 0,5 unidades de dibujo entre cuadrado y cuadrado.
- V. Créese un patrón que represente una trama de ladrillos. Dimensiones a gusto del diseñador.
- VI. Diseñar un patrón de sombreado que represente estrellas de seis puntas.
- VII. Crear el conjunto de patrones normalizados de sombreado necesario para su utilización en los rayados de los diferentes materiales y elementos del mundo de la construcción.

EJERCICIOS RESUELTOS DEL MÓDULO TRES

EJERCICIO I

*Mult, __ . __ __ . __ __ . __
A, 2,-1,0,-1,1,-1

EJERCICIO II

*Justif, __Texto __Texto __
A, 3,-4,["TEXTO",STANDARD,S=1,R=0,X=-2.5,Y=-.5], -4

EJERCICIO III

*Relatv
A, 2, -1,0,-3.5,["TEXTO",STANDARD,S=1,R=90,X=.5,Y=-.5],-3.5, 0,-1

EJERCICIO IV

*Formas
A, 2,[BOX,ltypeshp.shx,S=1], -2,2,[CIRC1,ltypeshp.shx,S=1], -2,2,[ZIG,
ltypeshp.shx,S=1], -2

EJERCICIO V

*ForTex, Combinación de forma y texto
A, 1,[CIRC1,ltypeshp.shx,S=1],-2, 2,["TEXTO",STANDARD,S=1,A=30,X=0.5,Y=-1.5],-
4,1

EJERCICIO VI

(Ejercicio completo para resolver por técnicos y/o especialistas).

MÓDULO CINCO

Definición de formas y tipos de letra

CINCO.1. INTRODUCCIÓN

Después de vista la creación y personalización de menús de todo tipo para **AutoCAD** (desplegables, de imagen, etcétera), de barras de herramientas y botones, de tipos de línea y de patrones de sombreado, parece lógico pensar que tenemos la posibilidad de crear archivos de formas y de tipos de letras de **AutoCAD**, así como de personalizar los ya existentes proporcionados por el programa. Pues así es, esa posibilidad es tangible y es precisamente lo que vamos a estudiar en este **MÓDULO CINCO**.

Los archivos de formas de **AutoCAD**, de los cuales hablamos por encima al introducir formas en determinados tipos de línea (véase el **MÓDULO TRES**), son ficheros de texto ASCII con extensión **.SHP** que, posteriormente, habremos de compilar desde **AutoCAD** con el comando **COMPILA** (en inglés **COMPILE**) para producir los correspondientes archivos de extensión **.SHX**, que son con los que el programa trabaja.

Por otro lado, y refiriéndonos ahora a los archivos de fuentes de tipos de letra, decir que existen tres tipos integrados en **AutoCAD**: los archivos de fuentes escalables **WYSIWYG** (*What You See Is What You Get*, lo que ves es lo que obtienes), esto es, las fuentes **True Type** con extensión **.TTF** —gracias a la integración en Windows—, los archivos **.SHX** que reciben el mismo tratamiento que los archivos de formas, en cuanto a creación desde un **.SHP** y posterior compilación, y los archivos de fuentes con tecnología **PostScript**, de extensión **.PFB** y también personalizables para **AutoCAD**. Los que vamos a tratar en este **MÓDULO** son los archivos **.SHX** exclusivamente.

Los dos tipos principales, sin meternos en la tecnología **PostScript**, son las **fuentes True Type** y los archivos **.SHX**. La diferencia entre ambos tipos de fuentes de letras es evidente. La fuentes **True Type**, como hemos dicho, son fuentes escalables, es decir, podemos aumentar su tamaño sin que disminuya la resolución, ya sea en pantalla o a la hora de imprimir. Esto se consigue gracias a la tecnología **WYSIWYG**, que produce una visualización en pantalla perfectamente similar a lo que después obtendremos por una impresora o trazador. La programación de fuentes **True Type** para Windows requiere conocimientos amplios que escapan a los objetivos de este curso, así como la tecnología **PostScript** y el diseño de fuentes que la utilicen.

Las fuentes **.SHX** de **AutoCAD**, son tipos de letra vectorizados los cuales, al aumentar de escala pierden definición y terminan por verse todos los trazos o vectores rectos que las forman. Estos tipos de letra, por el contrario, son los más recomendados a las hora de materializar un texto en un plano o dibujo. La razón es que su regeneración y procesado no implica tanta carga de memoria y de tiempo al sistema. Además, para una letra de tamaño pequeño, como por ejemplo números de cota en un plano mecánico, industrial, de construcción y demás, o textos de un cajetín o lista de materiales, u otros, producen un efecto muy deseado y no se aprecia su vectorización. Por otro lado, hemos de pensar que muchos de estos tipos de letra que contiene **AutoCAD** siguen rigurosamente normas ISO de rotulación necesarias para muchos tipos de planos.

En resumen, utilizaremos textos **.SHX** casi siempre en la rotulación de planos técnicos y dejaremos la fuentes **True Type** para títulos, cajetines, planos no normalizados y textos en general que han de producir una presentación visual impactante. Recordemos que **AutoCAD**

es un programa de dibujo estándar; es factible de ser utilizado por un ingeniero técnico en topografía o por un publicista.

La tecnología *PostScript* sólo podremos utilizarla con medios de impresión o trazado que la admitan.

Comencemos pues aprendiendo a diseñar ahora nuestros propios archivos fuente de formas.

CINCO.2. ARCHIVOS DE FORMAS PROPIOS

Las formas de **AutoCAD** son esos pequeños dibujos que se utilizan, más que nada, a la hora de diseñar tipos de línea. También podemos insertarlos como leyenda de planos o como simbología esquemática de otros, por ejemplo de calderería o electrónica (válvulas, condensadores...). También pueden ser utilizados como símbolos propios de alguna característica de representación, como tolerancias geométricas.

Los archivos de formas, como ya hemos explicado, son archivos ASCII que contiene las definición de una o varias formas que, luego, podremos utilizar a la hora de definir, por ejemplo, tipos de líneas o insertarlas como tales. El archivo fuente ASCII ha de tener la extensión **.SHP**. Veamos, desde un editor de texto, las definiciones contenidas en el archivo **LTPESH.SHP** que se proporciona junto a **AutoCAD** y preparado para la inclusión de sus formas en patrones de tipos de línea. Este archivo es más o menos de la siguiente forma:

```
;;;
;;; ltypeshp.SHP - shapes for complex linetypes
;;;

*130,6,TRACK1
014,002,01C,001,01C,0

*131,3,ZIG
012,01E,0

*132,6,BOX
014,020,02C,028,014,0

*133,4,CIRCL
10,1,-040,0

*134,6,BAT
025,10,2,-044,02B,0
```

Como viene siendo habitual en los archivos que estudiamos últimamente, el carácter punto y coma (;) establece la situación de un comentario que será ignorado por **AutoCAD**. Al igual que en los archivos de tipos de línea y de patrones de sombreado, con un sólo carácter ; es suficiente para definir una línea de comentario. La inclusión de más de uno solamente produce comodidad a la hora de examinar el fichero en sí. Además, los espaciados interlineales pueden ser usados para dar claridad al texto general y separar las formas por grupos o entre sí.

Nosotros podemos incluir nuevas formas en este archivo o, por el contrario y siendo una técnica más lógica, crear nuestro propio fichero **.SHP** de formas desde un editor ASCII. Para ello habremos de asimilar la sintaxis de creación.

CINCO.2.1. Sintaxis de creación de formas

Una forma se define en dos o más líneas dentro del archivo de definición. En realidad, con dos líneas es suficiente. Lo que ocurre es que, a veces, por la excesiva extensión de la segunda de ellas —que es la que realmente define la forma— se suele separar en más de una por comodidad.

La primera de las líneas es el encabezado y tiene la sintaxis:

**número_forma,octetos_definición,nombre_forma*

El asterisco (*) es obligatorio para indicar a **AutoCAD** que lo siguiente es la definición de una forma. A continuación, *número_forma* es un número entre 1 y 255 que identifica a cada forma del archivo. Podemos escoger cualquiera dentro del rango, pero no podemos repetir cualquiera de ellos dentro el mismo archivo de definición de formas; evidente. Así, podemos deducir fácilmente cuántas formas se nos permite incluir, en principio, en un solo archivo: doscientas cincuenta y seis. Si se duplica el número de una forma, **AutoCAD** proporcionará un mensaje de error al respecto a la hora de compilar el archivo (que ya veremos).

octetos_definición es el número de octetos necesarios para definir la forma, es decir, el número de octetos que se utilizan en la segunda línea de definición. Y *nombre_forma* es el nombre que le damos a la forma creada, el que la identificará. Con este nombre cargaremos después la forma desde el programa; ha de ser obligatoriamente un nombre en mayúsculas para que, posteriormente, **AutoCAD** lo almacene en memoria. Si no es así, no se producirá ningún error en la compilación, pero **AutoCAD** no reconocerá la forma al cargar el archivo e intentar insertarla.

Todos los parámetros han de ir separados por comas (,). Se pueden incluir espacios o tabuladores entre los diversos parámetros, tanto en esta línea como en las demás, excepto inmediatamente antes del nombre de la forma, ya que no se reconocerá después, e intentar incluir un espacio antes del nombre al llamar a la forma desde el editor de dibujo producirá un INTRO no deseado.

NOTA: Aunque los nombres pueden ser largos, recomendamos, como hemos dicho muchas veces, utilizar nombres en formato MS-DOS para todos los nombres en archivos de personalización de **AutoCAD**.

Veamos entonces uno de los ejemplos, el tercero, del archivo expuesto anteriormente:

```
*132,6,BOX  
014,020,02C,028,014,0
```

Como vemos, tras el asterisco obligatorio, se indica el número que se le asigna (132), el número de octetos de definición de la segunda línea, esto es 6 (014,020,02C,028,014,0) y, por último, el nombre de la forma (BOX). Pero, ¿qué definen estos octetos y cómo lo hacen? Vamos a pasar a explicarlo seguidamente.

Los octetos que definen la forma en sí contienen la dirección y longitud de cada vector (línea o arco) de la misma. Los octetos han de ir todos ellos separados por comas y, como ya se ha dicho, pueden existir espaciados o tabulaciones de separación.

Si el primer dígito de un octeto es 0, como en el ejemplo anterior 014, 020, 02C..., se entiende que está expresado en notación hexadecimal (que es lo más habitual); en caso contrario estaría en decimal.

La primera parte del octeto, tras el 0 hexadecimal, es la longitud del vector. Dicha longitud será generalmente unitaria, es decir, igual a 1, debido a que a ella se le aplicará posteriormente el factor de escala de la forma. Como se utiliza la primera parte del octeto para la longitud, ésta será como mucho igual a 15 (0F en hexadecimal). En el ejemplo, las longitudes son los segundos dígitos indicados (después del 0).

La segunda mitad del octeto es la dirección del vector en cuestión. Esta dirección se indica por un dígito hexadecimal, correspondiente a 16 direcciones predefinidas, como se muestra en la tabla siguiente:

NOTA: En la siguiente tabla se expresa la dirección teniendo en cuenta que el ángulo de 0 grados sexagesimales está en la dirección Este (las 3 en la esfera de un reloj) y que el sentido positivo es el trigonométrico (el antihorario). De todas formas, se aclara con la indicación de una dirección conocida; las direcciones de las bisectrices se refieren a la parte que tiene el sentido que escapa del centro de coordenadas.

Dígito	Dirección
0	0 grados (dirección de la parte positiva del eje X)
1	22,5 grados (dirección de la bisectriz al primer octante)
2	45 grados (dirección de la bisectriz al primer cuadrante)
3	67,5 grados (dirección de la bisectriz al segundo octante)
4	90 grados (dirección de la parte positiva del eje Y)
5	112,5 grados (dirección de la bisectriz al tercer octante)
6	135 grados (dirección de la bisectriz al segundo cuadrante)
7	157,5 grados (dirección de la bisectriz al cuarto octante)
8	180 grados (dirección de la parte negativa del eje X)
9	202,5 grados (dirección de la bisectriz al quinto octante)
A	225 grados (dirección de la bisectriz al tercer cuadrante)
B	247,5 grados (dirección de la bisectriz al sexto octante)
C	270 grados (dirección de la parte negativa del eje Y)
D	292,5 grados (dirección de la bisectriz al séptimo octante)
E	315 grados (dirección de la bisectriz al cuarto cuadrante)
F	337,5 grados (dirección de la bisectriz al octavo octante)

NOTA: Los vectores diagonales se consideran siempre de la misma longitud que los que están en las direcciones X e Y.

Volvamos sobre el ejemplo anterior. La explicación al primer octeto (014), una vez vista la sintaxis correspondiente, es: un 0 que indica notación hexadecimal, una longitud 1 y una dirección 4 (90 grados). De esta manera se dibuja un trazo unitario perpendicular al eje X y hacia arriba.

El siguiente octeto (@20) es: 0 hexadecimal, trazo de longitud 2 y dirección 0 (0 grados); dibuja un trazo recto —partiendo del punto final del anterior— horizontal hacia la derecha y de dos unidades de dibujo de longitud.

Tras estos dos octetos, aparecen otros tres que, si los estudiamos detenidamente de la misma forma anterior, podemos ver que van dibujando un cuadrado cuyo semilado es igual a 1 (?). Por último, un octeto final que sólo contiene 0, acaba la definición. Este último octeto debe estar presente al final de todas las definiciones de formas; es un código especial que indica el final de la definición. Un poco más adelante hablaremos de estos códigos especiales.

Vamos a ver otro ejemplo del archivo LTPESH.P.SHP:

```
*131,3,ZIG  
012,01E,0
```

La primera línea indica el número de forma, el número de octetos de definición (2 más el 0 final) y el nombre asignado, además del asterisco inicial. La segunda línea, en su primer octeto, especifica un trazo unitario en la dirección 2 (45 grados) y, en su segundo octeto, un trazo unitario en la dirección E (315 grados). Al final se encuentra el 0 de fin de definición. Esta forma representa un símbolo similar al del acento circunflejo (^).

CINCO.2.2. Cómo cargar e insertar formas

Para poder insertar una forma en el dibujo actual, debemos cargar antes el archivo que contiene dicha forma. Así pues, para poder acceder a las formas contenidas en el archivo que hemos puesto de ejemplo en la sección anterior, deberemos cargarlo antes en memoria.

Para cargar un archivo de formas se utiliza el comando **CARGA** (o **LOAD** en versiones inglesas) en línea de comandos. Este comando llama a un cuadro de diálogo desde el que se elige la forma que deseamos cargar. Pero atención, únicamente podemos cargar archivos de extensión **.SHX**, es decir, los archivos de formas compilados, y no los fuente **.SHP**.

NOTA: Posteriormente veremos la manera de compilar nuestros **.SHP**.

Una vez cargado el archivo, y si no se produce ningún error de carga, se procederá a insertar las formas con el comando **FORMA** (en inglés **SHAPE**) desde la línea de comandos también. Este comando proporciona la posibilidad de escribir el nombre de una forma (el último parámetro de la primera línea de definición) o de listar los nombres de todas o algunas de las formas cargadas en memoria.

NOTA: Puede haber más de un archivo de formas cargado en memoria.

Al insertar la forma se nos pedirá el punto de inserción, la altura de la forma y ángulo de rotación. En este punto, debemos recordar la necesidad de diseñar formas unitarias para luego proporcionarles una altura o factor de escala en esta entrada. El punto de donde viene "enganchada" la forma a la hora de insertarla, es el primer punto de definición de la misma, es decir, desde donde se comenzó a dibujar en el primer octeto.

Vamos entonces a crear nuestro primer archivo de formas. Para este ejercicio diseñaremos una forma que representará un cuadrado con una de sus diagonales dibujada.

Tras abrir un editor de texto ASCII procederemos a escribir lo siguiente:

```
*1,6,CUADRADO  
014,010,01C,018, 012, 0
```

La explicación es bien sencilla. Le proporcionamos un número 1 a la forma y le indicamos que va a ser descrita con 6 octetos (este número lo sabremos siempre al final de escribir la forma, lógicamente) y que se llamará **CUADRADO**. Comenzando desde la esquina inferior izquierda, trazamos (siempre trazos unitarios) un vector vertical hacia arriba, otro horizontal hacia la derecha, otro vertical hacia abajo, otro horizontal hacia la izquierda y un último vector inclinado (la diagonal) desde el punto inicial del cuadrado (donde ha acabado también el último vector) hasta el final del segundo vector (a 45 grados). Recordemos que los vectores diagonales han de considerarse todos de la misma longitud que los que no son diagonales, por ello tienen todos los vectores la misma medida en este ejemplo.

NOTA: Conviene hacer un pequeño boceto o esquema de la forma que queremos diseñar, anotando un punto de origen y la dirección y longitud de cada vector, para guiarnos.

NOTA: Al final de un archivo de definición de formas, como en todos los archivos vistos hasta ahora en este curso, deberemos introducir un `INTRO` (tras el último carácter de la última línea) para que el archivo funcione correctamente. De no ser así, no será reconocido como archivo de formas válido.

Guardaremos nuestro archivo (como archivo de texto si estamos en un editor que admite texto con formato; esto debemos tenerlo en cuenta, como deberíamos saber ya, en todos los archivos de personalización y programación de **AutoCAD**) con el nombre que deseemos y extensión `.SHP`.

CINCO.2.3. Compilando el fichero `.SHP`

Para poder insertar nuestra forma recién creada debemos cargar el archivo de definición, como hemos explicado. Pero para ello, habremos de compilarlo previamente para convertirlo en un archivo `.SHX` y que sea válido para **AutoCAD**.

Los archivos de formas fuente `.SHP` se compilan en **AutoCAD** desde la línea de comandos con la orden `COMPILA` (`COMPILE` en inglés). Este comando abre un cuadro Windows de búsqueda de archivos que permite localizar el `.SHP` fuente para compilarlo. En la casilla desplegable *Archivos de tipo:* debe especificar `*.shp`, de los otros archivos se habla más adelante en este mismo **MÓDULO**.

Una vez escogido, pulsamos el botón Abrir y el archivo, si no encuentra ningún error el programa, será compilado con éxito. A partir de aquí, únicamente deberemos cargarlo e insertar la forma creada como se ha indicado anteriormente. El archivo se compilará con el mismo nombre que tenía el fuente, pero con extensión `.SHX`.

Al compilar un archivo de definición de formas se nos indicará en línea de comandos el nombre del archivo de salida (compilado) y su ubicación, así como el número de bytes que contiene u ocupa. Si existe algún error, además del correspondiente mensaje se nos proporcionará el número de la línea en el archivo ASCII que produce el fallo.

NOTA: Si intentamos cargar un archivo `.SHX` y nos aparece un error de **AutoCAD** que dice algo así como: *Ruta_y_nombre_de_archivo* es un archivo de fuente de texto normal, no un archivo de formas, quiere decir que estamos intentado cargar como archivo de formas un archivo de tipos de letra (que veremos después). Estos archivos no se pueden cargar de esta manera.

Ahora, vamos a ver algún ejemplo práctico más. La siguiente forma dibuja un símbolo parecido a una señal de tráfico de las que indican la dirección a algún sitio dentro de una población (mirando a la derecha):

```
*12,6,DIRECC  
014,020,01E,01A,028,014, 0
```

La siguiente definición dibuja una forma que representa una flecha mirando a la derecha:

```
*56,6,FLECHA  
020,014,02F,029,014, 0
```

NOTA: Podemos cargar archivos de formas que contengan números iguales asignados a las definiciones, pero no puede haber números idénticos dentro del mismo archivo —como ya se ha comentado—. Si se cargan archivos que contengan formas con nombres idénticos, solamente se podrá insertar la forma que resida en el primer archivo cargado. Si se carga un archivo con dos formas que tengan el mismo nombre, sólo se podrá insertar la que esté definida antes.

NOTA: Al editar un archivo de definición de formas y cambiar sus líneas, recordemos volver a compilarlo y volver a cargarlo para comprobar sus variaciones. Pero ojo, para volver a cargarlo deberemos entrar en un dibujo nuevo para que **AutoCAD** descargue de memoria las antiguas definiciones, de otra manera, no veríamos cambios en nuestras formas.

CINCO.2.4. Códigos especiales

Aparte de las orientaciones que definen los vectores, se puede indicar a modo de octetos una serie de códigos especiales que realizan algunas operaciones adicionales para facilitarnos el dibujado de formas. Estos códigos comienzan por el 0 hexadecimal (que ya hemos visto) y llegan al 0F hexadecimal. En decimal van, evidentemente, del 0 al 14. Suele ser norma lógica especificar estos códigos en decimal para distinguirlos de los de dirección básica de vectores en hexadecimal.

Los códigos especiales y su explicación vamos a ir viéndolos a continuación uno por uno.

- Código 0 (0 hexadecimal). Termina la definición de una forma. Ya lo hemos visto antes y, como sabemos, ha de acabar siempre una definición.
- Código 1 (01 hexadecimal). Activa el modo de dibujo. Los vectores que se indiquen en los octetos que le sigan serán dibujados normalmente. Actúa como si se bajara una pluma conectada al cursor que traza la forma y sólo tiene sentido después del siguiente código.
- Código 2 (02 hexadecimal). Desactiva el modo de dibujo. Los vectores que se indiquen a continuación de este código no se dibujarán; se recorrerá la distancia indicada pero no se trazará ninguna línea o arco. Actúa como si se levantará una pluma conectada al cursor que traza la forma; para volver a activar el modo de dibujo se recurre al código anterior.

De esta forma —con estos códigos 01 y 02—, podemos dibujar trazos un poco más complejos que los estudiados hasta ahora, ya que no hace falta que pasemos varias veces por un mismo sitio para llegar a un punto. Por ejemplo, la siguiente forma se corresponde con el símbolo de una cruz:

```
*10,11,CRUZ  
010,014,002,01C,001,01C,002,014,001,010,0
```

Expliquémosla un poco; es muy simple. Dibujamos un trazo de una unidad de dibujo en el sentido horizontal hacia la derecha (010) y otro trazo unitario hacia arriba (014). En este momento, y para seguir dibujando la cruz por el medio, desactivamos el modo de dibujo (002). Debemos indicar el primer 0 porque es notación hexadecimal, como se explicó. Ahora, recorreremos el espacio inverso al último tramo dibujado pero sin trazar (01C), volvemos a activar el modo de dibujo (001) y trazamos el tramo inferior vertical (01C). A continuación, desactivamos el dibujo de nuevo (002) para volver al centro (014), lo volvemos a activar (001) y realizamos el último tramo horizontal hacia la derecha (010). Acabamos con el 0 de rigor.

Si hubiéramos indicado los códigos especiales en decimal habría sido así:

```
*11,11,CRUZ  
010,014,2,01C,1,01C,2,014,1,010,0
```

- Código 3 (03 hexadecimal). Divide la longitud de los vectores a partir de él por el factor indicado en el octeto siguiente.
- Código 4 (04 hexadecimal). Multiplica la longitud de los vectores a partir de él por el factor indicado en el octeto siguiente.

El efecto de estos dos códigos es acumulativo si se utilizan repetidas veces. Para anular el efecto de uno de ellos se utiliza el otro, es decir, si establecemos que todas las longitudes siguientes se dividan entre dos, por ejemplo, para paliar este efecto y volver a las medidas originales en otros vectores, debemos multiplicar por dos su longitud. Veamos un ejemplo:

```
*1,13,CUA2  
014,3,2,010,01C,4,2,018,3,2,01C,010,0
```

El primer trazo es unitario y vertical hacia arriba (014), luego se define que todos los trazos siguientes habrán de ser divididos (3) entre dos (2). Ahora se indica un trazo de 1 horizontal hacia la derecha (010), pero se dibujará de 0,5 unidades de dibujo —porque será dividido entre dos: $1 / 2 = 0,5$ —. A continuación, un trazo unitario vertical hacia abajo (01C), pero será también de 0,5. Luego se especifica que todos los trazos siguientes se multipliquen (4) por dos (2), esto es, se deshace la operación anterior, por lo que ahora los trazos medirán la longitud indicada. Se traza un trazo de 1 horizontal hacia la izquierda (018) y se vuelve a establecer la especificación de división (3) entre dos (2). Se termina con dos trazos de 0,5 hacia abajo y hacia la derecha (01C y 010), y con el 0 final. El resultado son dos cuadrados unidos por sus vértices inferior izquierdo y superior derecho, respectivamente.

- Código 5 (05 hexadecimal). Memoriza la posición actual del cursor de dibujo en un acumulador o pila. Este acumulador puede almacenar hasta cuatro posiciones. Todos los valores almacenados en la pila deben después ser extraídos con el código siguiente y no debe quedar ninguno almacenado al terminar la definición de la forma.
- Código 6 (06 hexadecimal). Toma la posición actual del acumulador. Ésta debe haber sido previamente almacenada con el código anterior. Vamos a ver un ejemplo:

```
*10,9,CRUZ  
010,5,014,6,5,01C,6,010,0
```

Este ejemplo se corresponde con el anterior estudiado de la cruz, pero realizado aquí de una manera más simple. Tras realizar el primer trazo hacia la derecha (010) se guarda la posición del cursor en el acumulador —esta posición será la del final del trazo, es decir, el centro de la cruz—. Dibuja el trazo vertical hacia arriba (014) y vuelve a la posición anterior guardada en el acumulador (6). Antes de dibujar nada guarda esta posición de nuevo (5), dibuja el trazo vertical inferior de la cruz (01C) y vuelve a la posición guardada (6). Finaliza dibujando el último trazo horizontal derecho y con el 0 último.

NOTA: Nótese que hemos de guardar una posición para poder luego recuperarla, es decir, con guardar una sola vez no podemos recuperar luego varias, aunque sea la misma posición como en este caso anterior. Cada vez que se llama al acumulador se pierde la posición. En resumen, habrá tanto códigos 6 como 5 haya; ni más 6 que 5, ni más 5 que 6.

NOTA: Las posiciones pueden estar anidadas: podemos llamar tres veces y luego ir recuperando los valores uno a uno y empezando por el último guardado hacia el primero.

Como máximo puede haber cuatro niveles de anidación (los cuatro valores que guarda el acumulador como mucho).

- Código 7 (07 hexadecimal). Llama a una subrutina consistente en una forma ya definida, o sea, una subforma de la forma actual. El octeto siguiente al de este código debe ser el número asignado (entre 1 y 255) a la subforma en el mismo archivo de definición. Una vez dibujada la subforma, se reanuda el dibujo del resto de octetos de la forma actual en curso. El siguiente es un archivo de ejemplo (con dos formas) en el que se usa este código:

; Ejemplo de forma con una subforma incluida

*134,11,CRUZ

5,014,5,014,6,5,018,6,010,6,0

*201,17,CUACRUZ

024,020,02C,028, 2,010,1, 7,134, 01C,3,2,5,018,6,010,0

Aquí (CUACRUZ) se dibuja un cuadrado de lado igual a 2 y se inserta (7) la forma CRUZ (134) en el medio (subforma), dividiéndolo en cuatro cuadrantes. Luego se dibuja una especie de pie al conjunto. El resto de la definición tiene que comprenderse ya perfectamente.

NOTA: Nótese en el ejemplo anterior que la subforma se dibujará comenzando en el punto en el que acabe la forma principal y desde el punto de inicio de dibujo de la subforma. La forma principal se reiniciará en el punto en el que acabe la subforma. Por eso hay que tener muy presente los puntos de inicio y final de las subformas, así como los puntos intermedios donde se van a insertar dentro de las formas principales. Y es por ello que la cruz de este ejemplo no se dibuja de la misma manera que la de ejemplos anteriores.

NOTA: Nótese también la técnica de anidamiento de memorizaciones y llamadas al acumulador en la subforma CRUZ.

- Código 8 (08 hexadecimal). Permite dibujar un vector en cualquier dirección y con cualquier longitud. De esta forma, evitamos las limitaciones impuestas por los octetos básicos de definición que, como sabemos, sólo nos permiten una longitud máxima de 15 en 16 orientaciones predeterminadas. Los dos octetos que siguen a este código se toman como incrementos en X e Y del vector que va ser definido. Por ejemplo:

*1,6,HOLA

024,8,-7,18,020,0

Esta forma define un trazo vertical hacia arriba de dos unidades y, posteriormente, indica un vector (8) con sendos incrementos X (-7) e Y (18). Después acaba con un trazo horizontal hacia la derecha de dos unidades de dibujo, y el 0 final.

Estos vectores, como vemos, están definidos por coordenadas cartesianas: desde el último punto trazado, una distancia en X y otra distancia en Y. Trazando una recta paralela imaginaria al eje Y que pase por dicho punto en X, y otra paralela imaginaria al eje X que pase por el punto en Y, donde se corten ambas rectas se producirá un punto que, unido con el inicio (el final del vector anterior), dará la dirección del vector actual propuesto, y tendrá como sentido el tomado desde el inicio hacia el final.

Para mayor claridad de los datos, se admiten paréntesis en la definición —entre ambos valores de X e Y—. Así el ejemplo anterior podría haberse escrito:

*1,6,EJEMPLO1

014,8,(-7,18),010,0

NOTA: Apréciase que, aunque vayan los vectores entre paréntesis, el número de octetos sigue siendo el mismo, esto es, el número de valores entre comas; o el número de comas más uno.

- Código 9 (09 hexadecimal). Permite indicar varios valores seguidos de vectores personales con incrementos X e Y, tal como se ha explicado en el código anterior. De esta forma, nos ahorramos el tener que especificar continuamente este código anterior, ya que todos los pares de valores que sigue al código 9 se toman como incrementos en X y en Y (respectivamente) hasta un par 0,0 que debe terminar siempre la secuencia. Se admite también la inclusión de paréntesis. Por ejemplo:

```
*2,9,EJEMPLO2  
9,(7,-5),(16,-8),(-2,21),(0,0)
```

- Código 10 (0A hexadecimal). Permite definir un arco octante mediante los dos octetos siguientes. Un arco octante abarca un ángulo múltiplo de 45 grados y empieza y termina siempre en los límites de uno de los octantes de la circunferencia. Esto no quiere decir que solamente pueda ocupar un octante, sino que pueden ser varios los que abarque. El primer octeto que sigue a este código especifica el radio del arco y, el segundo, contiene dos datos: la primera mitad del octeto es el octante inicial y, la segunda mitad, el número de octantes cubiertos de 0 a 7; 0 significa 8 octantes, es decir, la circunferencia completa. Si el arco se genera en sentido antihorario o trigonométrico no deberemos indicar signo. Por el contrario, si se genera en sentido horario habrá que indicar signo negativo (-). Se admiten paréntesis por claridad. Veamos un ejemplo:

```
*10,6,EMPALME  
034,10,(2,-043),02E,0
```

Este ejemplo indica lo siguiente: un primer trazo vertical hacia arriba de tres unidades de dibujo (034), ahora comienza la definición de un arco de octantes (10) de radio dos unidades (2), que se dibujará en sentido horario (signo -), que comenzará en el octante cuarto, es decir en 180 grados, y abarcará tres octantes, es decir 135 grados (043). Después se traza un tramo vertical hacia abajo (02E) y se acaba con 0. Esta forma asimila a dos líneas unidas con un empalme o enlazadas tangentemente.

El siguiente ejemplo traza un círculo completo:

```
*1,4,CIRCUL  
10,(2,010),0
```

NOTA: Indiquemos siempre el segundo octeto de los paréntesis en formato hexadecimal, es decir, comenzando con el 0. Lo mismo para el código siguiente.

- Código 11 (0B hexadecimal). Permite definir arcos que no comienzan ni terminan en octantes. Los cinco octetos siguientes a la especificación del código son utilizados para definir el arco. Los parámetros de dichos cinco octetos son:

- desplazamiento inicial: la distancia a la que empieza el arco desde el último octante completo que abarca. Se indica en fracción de octeto, multiplicado por 256 para que dé un valor de 0 a 255.
- desplazamiento final: la distancia a la que termina el arco desde el último octante completo que abarca. Se indica igual que el anterior.
- radio mayor: el octeto de mayor valor del radio. Será 0 salvo en los casos en que el radio exceda de 255.
- radio menor: el radio del arco, tal como se ha explicado en el código anterior (10).
- octante inicial y número de octantes: el mismo significado que con el código 10.

Como vemos, con este código se indica también un octante de arranque y un número de octantes cubierto, así como el radio del círculo. La única diferencia es que hay que especificar los desplazamientos necesarios con respecto a los límites de los octantes del círculo. De ahí que se pueda trazar cualquier arco. El siguiente ejemplo muestra un arco construido de esta manera:

```
*255,7,ARCO  
11,(85,171,0,5,012),0
```

Veamos la explicación, ya que este código resulta un poco árido al principio. Lo primero que se indica es el código (11) para trazar el arco. Después, y como hemos expuesto antes, el desplazamiento inicial. El arco comienza en un ángulo de $60 - 45 = 15$ grados del primer octante. Esto supone una fracción de $15 / 45 = 1 / 3$ de octante. Expresado en un octeto de definición se obtiene una cifra de $0,3333 * 256 = 85$ (85).

El desplazamiento final es el siguiente octeto. Se obtiene de igual modo que el desplazamiento inicial, esto es, el arco termina en $120 - 90 = 30$ grados del segundo octante. Luego el valor es $(30 / 45) * 256 = 171$ (171).

Como el radio es inferior a 255, el valor del radio mayor es 0 (0). El radio del arco, que se representa como radio menor es 5 (5) y, por último, se indica el octante inicial que es 1 (1), ya que empieza una fracción después del primer octante, y el octante final que es 2 (2), ya que empieza una fracción después del segundo octante.

La manera de trazar un círculo completo con este código sería la que sigue:

```
*1,7,CIR  
11,(0,0,0,10,010),0
```

- Código 12 (0C hexadecimal). Permite definir arcos mediante incrementos en X e Y, aplicando un factor de curvatura (tres octetos). Los incrementos indican el punto final del arco y, el factor de curvatura es el resultado de $2 * H / D$, siendo H la flecha del arco y D la cuerda. Como el factor se indica en un octeto con signo (+ si es antihorario y - si es horario), la curvatura debe multiplicarse por 127. Los incrementos también se indican en un octeto con signo, por lo tanto van de -127 a 127. Por ejemplo:

```
*1,5,SEMICIR  
12,(0,5,127),0
```

Este ejemplo dibuja una forma que es un semicírculo. Los incrementos, 0 en X y 5 en Y, se refieren, como hemos dicho, al punto final del arco, por lo que, en este caso, será un arco de diámetro igual a 5. Al ser un semicírculo, la flecha es igual al radio, por lo que $2 * 2,5 / 5 = 1$; o sea, $1 * 127 = 127$ (curvatura).

- Código 13 (0D hexadecimal). Al igual que el código 9 permitía indicar varios vectores seguidos para no tener que especificar continuamente el código 8, este código 13 permite definir varios arcos seguidos de la forma explicada en el código anterior. De esta manera no habremos de repetir el código 12 de forma continua cuando se quiera definir más de un arco seguido. De la misma forma que en aquel caso, deberemos acabar la secuencia con el par 0,0. Veamos un ejemplo que dibuja una forma que es similar a una letra S con los extremos alargados en dos segmentos:

```
*120,12,ESE  
020,13,(0,5,127),(0,5,-127),(0,0),020,0
```

- Código 14 (0E hexadecimal). Es una señal de procesamiento vertical. Se utiliza sólo en descripciones de tipos de letra —que enseguida veremos— que pueden tener

generación vertical. Establece una alternativa, de forma que el código siguiente se procesa o se salta, según se esté dibujando el texto con generación vertical o no.

NOTA: Los paréntesis pueden ser usados como aclaración en cualquier parte de la definición, no exclusivamente donde se ha indicado aquí; **AutoCAD** no los procesará.

NOTA: **AutoCAD** proporciona otros archivos de formas adicionales (aparte del `LTYPE$HP.SHX`) que pueden encontrarse en el directorio `\ACAD\BONUS\FONTS\` del CD-ROM de instalación (probablemente no habrán sido copiados a disco duro). Esos archivos de ejemplo son `ES.SHX` y `PC.SHX` (con sus correspondientes `.SHP`). Podemos revisar dichos archivos para comprender mejor el proceso de creación de una forma.

CINCO.3. ARCHIVOS DE TIPOS DE LETRA

Los tipos de letra `.SHX` son archivos de formas de **AutoCAD** que se definen siguiendo el mismo método explicado hasta este punto. Únicamente hay que tener una serie de consideraciones que vamos a indicar a continuación.

En archivos fuente de tipos de letra, el parámetro *número_forma* que se especifica en la sintaxis no puede ser cualquier valor, como ocurría con las formas, sino el código ASCII correspondiente al carácter que se defina. Este código lo podemos introducir tanto en decimal como en hexadecimal. Así por ejemplo, 70 (046 hexadecimal) se correspondería con la letra `F` (mayúscula), 106 (06A hexadecimal) con la `j` (minúscula) y 91 (05B hexadecimal) con el carácter corchete de apertura (`[`).

NOTA: Recordemos que los códigos en hexadecimal han de llevar un 0 delante, como mínimo.

Los códigos imprimibles comienzan desde el 32 (020 hexadecimal), que se corresponde con el espacio. Los anteriores son códigos de control que no son utilizados en la creación de tipos de letra, excepto el 10 (00A hexadecimal) que representa una interlínea o retorno de carro (pasa a la siguiente línea sin dibujar). Es el que se conoce en el mundo de la informática como *If*; es un `INTRO` en la escritura.

NOTA: Al final de este curso, en el **APÉNDICE F**, se proporciona una lista completa de códigos ASCII y sus correspondencias. De todas formas, podemos revisar los archivos `.SHP` proporcionados por **AutoCAD** de los tipos de letra suministrados para revisar las definiciones y ver los códigos ASCII de cada carácter. Estos archivos fuente no se copian al disco duro durante el proceso de instalación y han de buscarse en el directorio `\ACAD\BONUS\FONTS\` del CD-ROM de instalación.

Además los tipos de letra han de incluir una definición de forma especial al comienzo del archivo, la cual contiene información general sobre la fuente creada. Esta forma especial tiene número 0 (*nul*) y su sintaxis ha de ser la siguiente:

```
*0,4,nombre_tipo_letra  
arriba,abajo,modos,0
```

Como vemos, el asterisco (*), el número de forma 0 y el 4 (4 octetos) siempre se escriben, son obligatorios. *nombre_tipo_letra* es el nombre que le damos a nuestro tipo de letra, y ha de ser el mismo que el que tiene el archivo `.SHX` correspondiente.

Por su lado, *arriba* indica el número de longitudes de vector que ocupan las mayúsculas por encima de la línea base. *abajo* indica el número de longitud de vectores que

las “astas” verticales de la minúsculas descienden por debajo de la línea base (en las letras *q*, *y*, *p* y *j*). Esta línea base se entiende como la línea imaginaria donde se apoya el texto en general.

Y *modos* debe ser 0 para los tipos de letra de generación normal y 2 si admite también generación vertical (de arriba a abajo o de abajo arriba). En este punto entra en juego el código 14 explicado en la sección anterior. Se dijo que únicamente se utilizaba como señal de procesamiento vertical, y aquí lo podremos comprobar. Este código sólo se toma en consideración cuando *modos* está definido como 2.

Por último, decir que el nombre que asignemos a cada forma que defina una letra del conjunto habrá de escribirse, por norma general, en minúsculas, no como en el caso de las formas. Esto se hace así para que **AutoCAD** no guarde los nombres en memoria (recordar que en las formas se escribían en mayúsculas precisamente para que se guardaran en la memoria) y no ocupen espacio. Los nombres en minúscula no se guardan en memoria y **AutoCAD** dibuja los caracteres de texto según sus códigos ASCII (números de forma) y no según sus nombres, por lo que funcionarán perfectamente.

Todas las demás consideraciones han de tomarse idénticas a las de la creación de formas, en cuestión de comentarios, definición unitaria, etcétera.

La siguiente definición de un carácter para un tipo de letra se corresponde con una “D” mayúscula:

```
*68,13,dmayus  
1,030,012,044,016,038,2,010,1,06C,2,050,0
```

Los octetos de definición se interpretan de la misma manera que al dibujar formas. Primero se activa el modo de dibujo —por si estuviera desactivado— (1). Se comienza a dibujar con un trazo de tres unidades horizontal a la derecha (030), luego un trazo unitario a 45 grados hacia arriba a la derecha (012), un trazo vertical hacia arriba de cuatro unidades de dibujo (044), otro unitario a 45 grados hacia arriba a la izquierda (016) y uno de tres unidades horizontal a la izquierda (038). Después se desactiva el modo de dibujo (2) y se recorre el camino inverso al último tramo en una unidad y sin dibujar (010). Se vuelve a activar el modo de dibujo (1) y se dibuja la línea vertical hacia abajo que cierra la letra (06C). Por último, se vuelve a desactivar el modo de dibujo (2) y se recorren cinco unidades horizontalmente a la derecha (050). Esto último se hace para situar el siguiente carácter que se escriba, controlando así el espaciado entre letras. Se acaba con el 0 de final de definición.

En cuanto a la primera línea, se establece que se va a diseñar una letra “D” mayúscula mediante su código ASCII en decimal (68) y tras el asterisco. Después se indica que se dibujará en trece octetos (13) y se le da un nombre (*dmayus*) significativo.

Otro ejemplo. La siguiente serie de definiciones corresponde a fragmentos de un archivo de tipos de letra. En él se utilizan números hexadecimales para los códigos ASCII. Veámoslo:

```
; Mis propias fuentes en AutoCAD  
  
*0,4,MiFuente  
1,2,0,0  
  
*0A,7,lf  
2,0AC,14,8,(9,10),0  
  
*020,7,spc  
2,060,14,8,(-6,-8),0
```



```
...

*030,34,num0
2,14,3,2,14,8,(-3,-12),14,4,2,010,1,016,044,012,010,01E,04C,01A,
018,2,040,14,3,2,14,8,(-7,-6),14,4,2,0

*031,18,num1
2,14,8,(-1,-6),054,1,012,06C,2,018,1,020,2,020,14,03A,0

*032,23,num2
2,14,8,(-2,-6),054,1,012,020,01E,01C,01A,028,01A,02C,040,2,020,
14,8,(-4,-3),0
...

*041,21,amayus
2,14,8,(-2,-6),1,024,043,04D,02C,2,047,1,040,2,02E,14,8,(-4,-3),0

*042,29,bmayus
2,14,8,(-2,-6),1,030,012,014,016,028,2,020,1,012,014,016,038,2,
010,1,06C,2,050,14,8,(-4,-3),0
...

*061,24,aminus
2,14,04B,020,1,018,016,024,012,010,01E,02C,01A,2,012,1,01E,2,020,
14,8,(-4,-3),0

*062,25,bminus
2,14,8,(-2,-6),1,064,2,04C,1,022,010,01E,02C,01A,018,026,2,02C,
060,14,8,(-4,-3),0
...
```

Evidentemente faltarían muchos caracteres por definir aquí. La primera definición es el encabezado del archivo con la información general ya explicada. Después se definen el retorno de carro con salto de línea y el espacio. Luego los caracteres *0*, *1* y *2*. Por último, los caracteres *A* y *B* (mayúsculas) y los caracteres *a* y *b* (minúsculas). Como decimos, faltarían todos los demás: resto de números y letras —mayúsculas y minúsculas— y resto de diversos caracteres —signos de puntuación, tanto por ciento, dólar...—.

Si se define un archivo de tipo de letra en el que faltan caracteres, estos no podrán ser mostrados en pantalla al teclearlos mediante algún comando de dibujo de textos de **AutoCAD**, si se tiene elegido dicho tipo de letra en el estilo de texto actual. Además, en la línea de comandos aparecerá un mensaje de error advirtiéndolo que no se encuentra la definición del carácter en cuestión.

Por ello, es lógico definir todos los caracteres imprimibles en un archivo de tipos de letra.

NOTA: Estúdiense, en el último ejemplo, más a fondo las definiciones de salto de línea con retorno de carro y del espaciado. Nótese que lo primero que hacen es desactivar el modo de dibujo (con el código 2) porque no interesa trazar y, a continuación, describen el vector de desplazamiento. Revísese también la manera de emplear el código 14 de procesamiento vertical.

La forma de compilar estos archivos es exactamente la misma que la utilizada para los archivos de formas.

CINCO.3.1. Utilizar los tipos de letra creados

Así como se compilan de la misma manera que los archivos de formas, los archivos de tipos de letra no se pueden cargar como se hacía con aquellos. La manera de utilizarlos es simplemente eligiendo el tipo de letra creado (una vez compilado) desde *Formato>Estilo de texto...*, en el área *Tipo de letra* en la casilla desplegable *Nombre del tipo de letra*. Desde aquí se establecen las características generales de la fuente asociada a un estilo de texto. Luego, únicamente habremos de escoger dicho estilo para escribir con cualquiera de los comandos de dibujo de textos (TEXT, TEXTODIN o TEXTOM, cuyas correspondencias inglesas son TEXT, MTEXT y DTEXT, respectivamente).

Para que nuestro archivo de texto aparezca en la lista indicada, dicho archivo habrá de encontrarse en uno de los directorios de soporte del programa. Recordamos que desde *Herr.>Preferencias...*, en la pestaña *Archivos* y en la carpeta *Camino de búsqueda de archivos de soporte*, se pueden añadir carpetas o directorios de soporte que el programa reconocerá como tales.

El tipo de letra también se puede escoger con el comando ESTILO (STYLE en inglés) tecleado en la línea de comandos —aparecerá el mismo cuadro expuesto— o con su equivalente del mismo nombre para manejo desde línea de comandos, -ESTILO (-STYLE).

CINCO.3.2. Tipos de letra Unicode

Existe un juego de caracteres básico comunes a todos los idiomas, los cuales se introducen directamente desde cualquier teclado y de la misma forma. Pero también existen otros caracteres que son específicos de cada idioma, como letras acentuadas (Á, é, è, ù...) o con diéresis (ö, ë, ä, Ë...) u otros. Estos caracteres cambian la funcionalidad de algunas teclas, según en el idioma en que se esté trabajando.

Internamente, los tipos de letra estándar de **AutoCAD** corresponden a la asignación de caracteres usada por el sistema operativo base. Esto se debe a que los caracteres se almacenan directamente en la Base de Datos con el formato adquirido en el teclado. Se utilizan los mismos códigos de carácter para generar los tipos de letra. Esto puede convertirse en un problema si se emplean, por ejemplo, caracteres acentuados (de 8 bits) para los que existen muchas normas de codificación.

Las limitaciones de la asignación de caracteres han obligado a incluir este juego de tipos de letra para las diferentes páginas de códigos usadas por **AutoCAD**. Aunque en esencia estos tipos de letra son iguales, algunos de sus caracteres se encuentran en posiciones distintas, según la página de códigos para la que están definidos. Si la codificación de tipo de letra no se corresponde con la del texto del dibujo, es posible que se dibujen caracteres erróneos.

Con los tipos de letra *Unicode*, las cadenas de texto se convierten a este código antes de dibujarse. Esto garantiza una correcta generación de textos procedentes de otros idiomas, sin que aparezcan caracteres extraños o no reconocidos. De esta forma ya no se necesitan tipos de letra adicionales para otros idiomas o plataformas. Gracias a su amplio juego de caracteres, un solo tipo de letra *Unicode* permite utilizar cualquier idioma y plataforma. Para el usuario esta función resulta transparente porque, en caso necesario (debido a las distintas páginas de códigos), los dibujos se convierten a la página de códigos de sistema de **AutoCAD** durante el proceso de carga. Los dibujos siempre se guardan en la página de códigos de sistema de **AutoCAD**.

NOTA: Dado que *Unicode* no permite usar todos los idiomas asiáticos, algunas o todas las versiones asiáticas tendrán que utilizar tipos de letra grandes, los cuales serán explicados más adelante.

La sintaxis de los archivos de definición de tipos de letra *Unicode* es prácticamente idéntica a la de los archivos de definición de tipos habituales ya explicados. La principal diferencia se encuentra en la sintaxis del encabezamiento de tipo de letra, como lo indica este código:

```
*UNIFONT,6,nombre_tipo_letra  
arriba,abajo,modos,codificación,tipo,0
```

Los parámetros *nombre_tipo_letra*, *arriba*, *abajo* y *modos* son iguales a los de los tipos de letra habituales. Por su lado, *codificación* es un valor que indica la codificación del tipo de letra según la tabla siguiente:

Valor	Codificación
0	<i>Unicode</i>
1	Multibyte comprimido 1
2	Archivo de forma

Y *tipo* es la información de incrustación de tipo de letra. Este parámetro especifica si el tipo de letra tiene licencia. Los tipos de letra con licencia no pueden modificarse ni intercambiarse. Los valores según la tabla que sigue:

Valor	Información de incrustación
0	Puede incrustarse
1	No puede incrustarse
2	Incrustación de sólo lectura

Otra diferencia adicional entre estas definiciones y las habituales son los números de forma. Las definiciones con *Unicode* utilizan siempre números de forma hexadecimales en lugar de valores decimales. Cada tipo de letra se especifica mediante un código de la forma:

`\U+nnnn`

donde *nnnn* es el número de código hexadecimal. Esto no es necesario, pero conveniente para establecer correspondencia entre los números de forma y los valores de carácter de control `\U+`. Lo que sí es necesario es establecer los números de forma en hexadecimal. Las dos líneas siguientes son equivalentes:

```
*00031,18,num1  
*\U+031,18,num1
```

CINCO.3.3. Tipos de letra grande y grande extendido

Algunos idiomas, como el japonés, utilizan tipos de letra de texto con miles de caracteres que no son ASCII. Con **AutoCAD** es posible crear dibujos con este tipo de texto mediante un archivo de definición de forma especial denominado archivo de tipo de letra grande.

Un tipo de letra con tantos caracteres como el mencionado, debe manipularse de manera diferente a nuestros tipos de letra basados en un código ASCII de 256 caracteres (de

0 a 127 básicos y de 128 a 255 extendidos). Además de utilizar técnicas de búsqueda de archivos más complejas, **AutoCAD** necesita un método para representar caracteres con códigos de dos bytes como de uno. Ambos aspectos se resuelven colocando códigos especiales al principio de un archivo de definición de tipo de letra, convirtiéndolo en un archivo de definición de tipo de letra grande.

La primera línea pues de un archivo de este tipo habrá de tener la sintaxis siguiente:

```
*BIGFONT número_caracteres,número_rangos,i1,f1,i2,f2,...
```

número_caracteres es el número aproximado de definiciones de caracteres del juego. Si este número se rebasa en —aproximadamente— más de un diez por ciento, la velocidad o el tamaño del archivo pueden verse perjudicados.

Podemos utilizar el resto de la línea para especificar códigos de carácter especiales (códigos de escape) que indiquen el inicio de un código de dos bytes. Por ejemplo, en los ordenadores japoneses, los caracteres *Kanyī* comienzan con códigos hexadecimales en el rango 90 a AF o E0 a FF. Cuando el sistema operativo detecta uno de estos códigos, lee el byte siguiente y combina los dos bytes en un código para un carácter *Kanyī*.

número_rangos indica cuántos rangos contiguos de números se utilizan como códigos de escape, y *i1, f1, i2, f2*, etcétera, definen los códigos inicial y final de cada rango. En consecuencia, el encabezamiento de un archivo de tipo de letra grande japonés puede tener el formato siguiente:

```
*BIGFONT 4000,2,090,0AF,0E0,0FF
```

Después de esta línea, la definición de tipo de letra es idéntica a la de un tipo de letra de texto de **AutoCAD** estándar, con la diferencia que los códigos de carácter (números de forma) pueden tener valores de hasta 65535.

Para reducir el tamaño de los caracteres *Kanyī* compuestos, se puede definir un archivo de tipo de letra grande extendido. Los tipos de letra grandes extendidos utilizan el código de subforma, seguido inmediatamente de un 0.

La primera línea de un archivo de tipo de letra grande extendido es idéntica a la de un archivo de tipo de letra grande estándar. Éste es el formato del resto de las líneas del archivo:

```
*0,5,nombre_letra
altura_carácter,0,modos,anchura_carácter,0
...
*número_forma,bytes_def,nombre_forma
...

código,0,primitiva#,punto_base_X,punto_base_Y,anchura,altura,
...

código,0,primitiva#,punto_base_X,punto_base_Y,anchura,altura,
...

terminador
```

A continuación se explican todos los parámetros.

- *altura_carácter*. Se utiliza con la anchura de carácter (*anchura_carácter*) para indicar el número de unidades que definen los caracteres del tipo de letra.

- *anchura_carácter*. Utilizada con la altura de carácter (*altura_carácter*) para indicar el número de unidades que definen los caracteres del tipo de letra. Los valores de *altura_carácter* y *anchura_carácter* se usan para escalar las primitivas del tipo de letra. En este contexto, las primitivas son los puntos, líneas, polígonos o cadenas de caracteres del tipo de letra orientado geométricamente en un espacio bidimensional. Un carácter *Kanyi* está formado por varias primitivas usadas de forma repetida en diferentes escalas y combinaciones.
- *modos*. Igual que para los tipos de letra de texto estándar.
- *número_forma*. Código de carácter.
- *bytes_def*. Tamaño en bytes. Son siempre 2 bytes, que constituyen un código hexadecimal o una combinación de códigos decimales y hexadecimales.
- *nombre_forma*. Nombre del carácter.
- *código*. Código especial de descripción de la forma. Este código siempre se define en 7 para que pueda usar la función de subforma.
- *primitiva#*. Referencia al número de subforma. Esta referencia siempre se define como 2.
- *punto_base_X*. Origen X de la primitiva.
- *punto_base_Y*. Origen Y de la primitiva.
- *anchura*. Escala de la anchura de la primitiva.
- *altura*. Escala de la altura de la primitiva.
- *terminador*. Indicador de fin de archivo de la definición de forma; siempre es 0.

Para llegar al factor de escala, **AutoCAD** reduce la escala de la primitiva a una unidad cuadrada y después la multiplica por la altura y la anchura para obtener la forma del carácter. Los códigos de carácter (números de forma) del archivo de definición de formas de tipo de letra grande pueden tener valores hasta 65535. En la tabla siguiente se describen los campos del archivo de tipo de letra grande extendido:

Variable	Valor	Tamaño en bytes	Descripción
<i>número_forma</i>	XXXX	2 bytes	Código de caracteres.
<i>código</i>	7	2 bytes	Definición de tipo de letra extendido.
<i>primitiva#</i>	XXXX	2 bytes	Referencia al número de subforma.
<i>punto_base_X</i>		1 byte	Origen X de la primitiva.
<i>punto_base_Y</i>		1 byte	Origen Y de la primitiva.
<i>anchura</i>		1 byte	Escala de la anchura de la primitiva.
<i>altura</i>		1 byte	Escala de la altura de la primitiva.
<i>terminador</i>	0	1 byte	Final de definición de forma.

Curso Práctico de Personalización y Programación bajo AutoCAD
Definición de formas y tipos de letra

NOTA: No todos los tipos de letra se definen en matrices cuadradas, algunos se definen en matrices rectangulares.

A continuación se muestra un ejemplo de un archivo de definición de forma para un tipo de letra grande extendido.

```
*BIGFONT 50,1,080,09e
```

```
*0,5,Tipo de letra extendido  
15,0,2,15,0
```

```
*08D91,31,sin especificar  
2,0e,8,-7,-15,7,0,08cfb,0,0,16,16,7,0,08bca,2,3,12,9,2,8,18,0,2,0e,8,  
-11,-3,0
```

```
*08CD8,31,sin especificar  
2,0e,8,-7,-15,7,0,08be0,0,0,8,16,7,0,08cc3,8,0,8,16,2,8,18,0,2,0e,8,  
-11,-3,0
```

```
*08ADF,31,sin especificar  
2,0e,8,-7,-15,7,0,089a4,0,0,8,16,7,0,08cb3,8,0,8,16,2,8,18,0,2,0e,8,  
-11,-3,0
```

```
*08CE8,39,sin especificar  
2,0e,8,-7,-15,7,0,089a4,0,1,5,14,7,0,08cc3,5,2,5,14,7,0,08c8e,9,0,7,  
16,2,8,18,0,2,0e,8,-11,-3,0
```

```
*089A4,39,primitiva  
2,0e,8,-7,-15,2,8,1,14,1,0c0,2,8,-11,-6,1,0a0,2,8,-12,-7,1,0e0,  
2,8,7,13,1,0dc,2,8,11,-1,2,0e,8,-11,-3,0
```

```
*08BCA,41,primitiva  
2,0e,8,-7,-15,2,8,1,14,1,0c0,2,8,-11,-6,1,0a0,2,8,-12,-8,1,0e0,  
2,0e5,1,0ec,2,063,1,8,2,-3,2,06f,2,0e,8,-11,-3,0
```

```
*08BE0,81,primitiva  
2,0e,8,-7,-15,2,8,3,9,1,080,2,8,-10,-4,1,0c0,2,8,-13,-5,1,0e0,2,8,  
-7,9,1,09c,2,8,-1,14,1,8,-6,-5,2,8,8,5,1,8,6,-5,2,8,-11,-6,1,8,1,  
-3,2,8,7,3,1,8,-1,-3,2,8,-3,15,1,01a,2,012,1,01e,2,8,10,-14,2,0e,8,  
-11,-3,0
```

```
*08C8E,44,primitiva  
2,0e,8,-7,-15,2,8,3,15,1,090,0fc,038,2,8,-6,11,1,090,2,8,-9,-5,1,  
090,2,096,1,0ac,8,-1,-3,01a,01a,2,8,18,0,2,0e,8,-11,-3,0
```

```
*08CB3,61,primitiva  
2,0e,8,-7,-15,2,042,1,02b,02a,018,2,0d0,1,012,034,2,069,1,01e,040,2,8,  
-8,6,1,02b,2,8,4,5,1,08c,2,8,-3,8,1,03c,2,8,-5,3,1,0e0,2,8,-12,5,1,  
0a0,2,8,6,-14,2,0e,8,-11,-3,0
```

```
*08CC3,34,primitiva  
2,0e,8,-7,-15,2,0c1,1,06c,0a8,064,0a0,2,8,-5,9,1,09c,2,8,-7,5,1,0e0,2,8,4,  
-11,2,0e,8,-11,-3,0
```

```
*08CFB,22,primitiva  
2,0e,8,-7,-15,2,0d2,1,0cc,0c8,0c4,0c0,2,8,5,-13,2,0e,8,-11,-3,0
```

NOTA: Nótese que las letras de los números en hexadecimal pueden escribirse en mayúscula, como hacíamos antes, o en minúscula, como en este caso.

En algunas disciplinas de esbozo, muchos símbolos especiales pueden aparecer en cadenas de texto. Es posible extender los tipos de letra de texto estándar de **AutoCAD** para incluir dichos símbolos, si bien esta operación de extensión tiene algunos límites.

Cada archivo de tipo de letra puede contener 255 formas como máximo. El juego de caracteres estándar utiliza casi la mitad de los números de forma disponibles. Sólo es posible usar los códigos 1-9, 11-31 y 130-255. Para utilizar varios tipos de letra de texto, es necesario duplicar las definiciones de símbolo de cada tipo de letra. Para usar un símbolo especial, es necesario indicar %%*nnn*, donde *nnn* representa el número de forma del símbolo.

Con el tipo de letra grande se evitan estos problemas. Podemos seleccionar uno o varios caracteres poco usados —como la tilde (~) o la barra vertical (|)— como códigos de escape y utilizar el carácter siguiente para escoger el símbolo especial apropiado. Por ejemplo, se puede emplear el siguiente archivo de tipo de letra grande para dibujar letras griegas indicando una barra vertical (código ASCII 124) seguida de la letra latina equivalente. Dado que el primer byte de cada carácter es 124, los códigos de carácter se incrementan en 124 × 256, o lo que es lo mismo, en 31744. Veámoslo.

```
*BIGFONT 60,1,124,124

*0,4,greek
encima,debajo,modos,0

*31809,n,amy
... definición Alfa en mayúsculas, activada con "|A"

*31810,n,bmy
... definición Beta en mayúsculas, activada con "|B"

*31841,n,amn
... definición Alfa en minúsculas, activada con "|a"

*31842,n,bmn
... definición Beta en minúsculas, activada con "|b"

*31868,n,barrav
... definición de barra vertical, activada con "||"

...
```

CINCO.3.3.1. Utilizar estos tipos de letra grande

Si deseamos utilizar un tipo de letra grande para dibujar texto, debemos definir un estilo de texto y especificar el nombre de un archivo de tipo de letra que admita letra grande. Esto lo podemos comprobar cuando en el cuadro *Estilo de texto* esté disponible la *casilla Usar tipos de letra grandes*. Si así es, la activaremos y elegiremos el tipo de letra grande en la lista desplegable *Tipo de letra grande*:

Si se hace desde la línea de comandos, con las órdenes de **AutoCAD** expuestas anteriormente, debe indicarse el tipo de letra grande separado con una coma del tipo normal, así:

```
txt,greek
```

AutoCAD supone que el primer nombre corresponde al tipo de letra estándar y que el segundo es del tipo de letra grande. Si sólo se escribe un nombre, **AutoCAD** supone que se trata del tipo de letra estándar y suprime cualquier tipo de letra grande asociado.

Si utilizamos comas iniciales o finales al especificar los nombres de archivo de tipo de letra, podemos cambiar un tipo de letra y dejar los demás intactos, como se indica en la tabla siguiente.

Entrada	Resultado
<i>estándar, grande</i>	Se especifica el tipo de letra estándar y grande.
<i>estándar,</i>	Sólo se especifica el tipo de letra estándar (el grande permanece intacto).
<i>, grande</i>	Sólo se especifica el tipo de letra grande (el estándar permanece intacto).
<i>estándar</i>	Sólo se especifica el tipo de letra estándar (si es necesario se suprime el grande).
INTRO (<i>respuesta nula</i>)	Ningún cambio.

Cuando utiliza el comando **-ESTILO** (**-STYLE**), con su opción **?**, para presentar los estilos o revisar uno existente, **AutoCAD** muestra el archivo de tipo de letra estándar, una coma y el archivo de tipo de letra grande. Si el estilo sólo tiene un archivo de tipo de letra grande, aparece con una coma inicial, como en *,greek*.

AutoCAD primero busca los caracteres de una cadena de texto en el archivo de tipo de letra grande. Si no los localiza, los busca en el archivo de tipo de letra estándar.

CINCO.3.4. Soporte *PostScript*

Sabemos que **AutoCAD** contiene funciones de soporte *PostScript*. Estas funciones, además de muchas otras cosas, permite utilizar tipos de letra que utilicen la tecnología *PostScript*.

El archivo de soporte **ACAD.PSF**, en el directorio **\SUPPORT** de **AutoCAD**, es un fichero ASCII que contiene las asignaciones de archivos de forma a tipos de letra *PostScript* y las definiciones y procedimientos de codificación para dichos tipos de letra, entre otras. Los archivos de definiciones de tipos de letra *PostScript* tiene la extensión **.PFB** y hay que compilarlos con el mismo comando que los archivos de formas o tipos de letra estándar.

Por otro lado decir que el archivo de mapa de tipos de letra de **AutoCAD** (**FONTMAP.PS**) es un catálogo, o mapa, de todos los tipos de letra reconocidos por el intérprete *PostScript* de **AutoCAD** (**ACADPS.ARX**). Cuando se ejecuta **CARGAPS**, este archivo ASCII asigna los nombres de tipo de letra de idioma *PostScript* a los nombres de los archivos de definición de tipo de letra (**.PFB**) correspondientes. Los tipos de letra que han de cargarse automáticamente al especificarlos deben declararse en **FONTMAP.PS**.

Cuando se importa un archivo **.EPS** (*PostScript* Encapsulado) con **CARGAPS**, **ACADPS.ARX** usa **FONTMAP.PS** para localizar los archivos **.PFB** correspondientes a los tipos de letra indicados en el archivo **.EPS**. Si **ACADPS.ARX** encuentra un archivo **.PFB** en **FONTMAP.PS** pero no en el sistema, la aplicación muestra un mensaje de advertencia y utiliza un tipo de letra por defecto. Por lo que es posible declarar tipos de letra no instalados sin problema alguno.

El archivo **FONTMAP.PS** suministrado hace referencia a un juego de tipos de letra *Type 1* proporcionado con **AutoCAD**, así como a los tipos de letra distribuidos con *Adobe Type Manager for Windows*, *Adobe Plus Pack* y *Adobe Font Pack 1*. Si se han adquirido otros tipos de letra, se puede editar el archivo de mapa de tipos de letra y declararlos de la misma manera que los existentes.

Para obtener más información, véase el contenido del archivo FONTMAP.PS, incluidos los comentarios que explican la sustitución de tipos de letra.

CINCO.4. EJEMPLOS PRÁCTICOS DE FORMAS Y TIPOS DE LETRA

CINCO.4.1. Cuadrado con diagonales

```
*1,9,CUADIA2  
014,010,01C,5,018,012,6,016,0
```

CINCO.4.2. Subforma anterior y triángulos

```
*2,11,TRICUA  
5,7,1,3,2,012,01E,6,01E,012,0
```

CINCO.4.3. Número ocho simple

```
*00038,16,OCHO  
13,(0,5,127),(0,5,-127),(0,-5,-127),(0,-5,127),(0,0),0
```

CINCO.4.4. Letra G mayúscula románica

```
*00047,102,gmayus  
2,14,3,2,14,8,(-23,-42),14,4,2,14,5,8,(17,18),1,8,(1,-3),064,  
8,(-1,-3),026,8,(-3,1),028,8,(-3,-1),02A,02B,8,(-1,-3),05C,  
8,(1,-3),02D,02E,8,(3,-1),020,8,(3,1),022,2,8,(-7,18),1,029,02A,  
02B,8,(-1,-3),05C,8,(1,-3),02D,02E,02F,2,8,(7,8),1,08C,2,8,(1,8),  
1,08C,2,085,1,070,2,8,(2,-8),14,6,14,3,2,14,8,(23,-18),14,4,2,0
```

CINCO.4.5. Letra n minúscula gótica

```
*0006E,80,nminus  
2,14,8,(-9,-14),14,5,8,(2,12),1,010,01E,08C,029,010,02F,01E,2,8,  
(-2,13),1,01E,0AC,02F,2,8,(-5,11),1,022,02E,09C,02F,02A,2,0B4,1,  
8,(3,1),021,012,01E,02F,010,029,08C,01E,010,2,8,(-5,11),1,02F,  
0AC,01E,2,8,(-5,11),1,010,02F,09C,02E,022,2,8,(2,-2),14,6,  
14,8,(9,-9),0
```

CINCO.FIN. EJERCICIOS PROPUESTOS

- I. Crear una forma que represente un arco y una flecha.
- II. Crear una forma con el símbolo base para mecanizados.
- III. Créese una forma que represente un jalón con banderola, apta para situar estaciones topográficas.
- IV. Diseñar una forma que represente un helipuerto.
- V. Diseñar una forma que represente un vértice geodésico.
- VI. Crear una forma con el símbolo eléctrico para una resistencia.
- VII. Crear una letra f minúscula sencilla.

- VIII. Diseñese una letra Q mayúscula sencilla.
- IX. Créese una letra K mayúscula itálica.
- X. Crear un letra alfa griega minúscula.
- XI. Diseñar un archivo con múltiples formas y una fuente de tipo de letra nueva completa con todos los caracteres y signos necesarios para la acotación normalizada de un delineante mecánico. Inclúyanse en ambos archivos símbolos de tolerancia, símbolos de mecanizado, símbolos de acotación, flechas de cota y letras normalizadas.

EJERCICIOS RESUELTOS DEL MÓDULO CUATRO

EJERCICIO I

*TRAZOPUN, Trazos y dos puntos inclinadas
45, 0,0, 1, 1, 1,-.5,0,-.5,0,-.5

EJERCICIO II

*CRUZADAS, Líneas continuas que se cruzan
0, 0,0, 0, 1.414213562373
45, 0,0, 0, 1
135, 0,0, 0, 1

EJERCICIO III

*GRUP, Grupos de líneas continuas
90, 0,0, 0, 2
90, .5,0, 0, 2

EJERCICIO IV

*Celos, Cuadros de lado 1 y espacio 0.5
0,0,0,0,1.5,1,-.5
0,0,1,0,1.5,1,-.5
90,0,0,0,1.5,1,-.5
90,1,0,0,1.5,1,-.5

EJERCICIO V

*Ladrillos, Trama de ladrillos
0,0,0,0,.25
90,0,0,.25,.25,.25,-.25

EJERCICIO VI

*Estrellas, Estrellas de seis puntas
0, 0, 0, 0, 5.49926, 3.175, -3.175
60, 0, 0, 0, 5.49926, 3.175, -3.175
120, 1.5875, 2.74963, 0, 5.49926, 3.175, -3.175

EJERCICIO VII

(Ejercicio completo para resolver por técnicos y/o especialistas).

MÓDULO SEIS

Creación de archivos de ayuda

SEIS.1. INTRODUCCIÓN A LOS ARCHIVOS DE AYUDA

Imaginemos que, una vez aprendido todo lo necesario para crear una aplicación vertical para **AutoCAD**, con una base AutoLISP/DCL o VBA y unos archivos de definición de menús, de tipos de línea, letras, formas y demás que la acompañen, necesitamos informar al usuario de cuáles son los entresijos del conjunto que presentamos. Los tipos de comandos creados y su funcionamiento, el manejo de ciertos aspectos complejos etcétera, será necesario incluirlos en un archivo de ayuda para que el usuario de nuestro software sepa cómo manejarse con él.

Todos los programas, todas las aplicaciones, todo el software en general que se precie posee ficheros de ayuda que explican paso a paso unos, o más someramente otros, el funcionamiento del programa en sí. En este **MÓDULO SEIS** vamos a aprender a realizar archivos de ayuda para **AutoCAD** que acompañarán a nuestros diseños de aplicaciones o personalizaciones. Comentaremos algún que otro método, aunque a fondo únicamente veremos el propio de **AutoCAD** para creación de archivos de ayuda específicos del programa.

SEIS.2. LA AYUDA DE AutoCAD. FORMATO .AHP

Los archivos de ayuda propios de **AutoCAD** se escriben en archivos ASCII que deben tener la extensión .AHP. Toda la información contenida en un archivo .AHP ha de ser organizada por temas. Cada tema incluirá una serie de líneas de texto denominadas directrices que, a su vez, poseen un identificador de código y opcionalmente un título y/o una serie de palabras clave. A continuación se exponen todos los códigos posibles de forma conjunta y, después, iremos comentado cada uno mientras revisamos un ejemplo práctico. Veamos la tabla de códigos:

Código de directriz	Función	Aparece en...
\#	Identificador de tema	Se utiliza internamente (el usuario no puede verlo).
\\$	Título	Lista del historial y búsqueda de temas.
\K	Palabras clave	Lista de búsqueda.
\E	Final de archivo	Final de archivo de ayuda (el usuario no puede verlo).
\espacio_blanco	Comentario	Comentarios internos (se ignoran durante la visualización de la ayuda en pantalla).

Vamos a ver un ejemplo para esclarecer el tema. Imaginemos que queremos crear un archivo de ayuda para explicar una serie de comandos nuevos que hemos programado en AutoLISP. Los comandos en cuestión serán **HÉLICE**, **TUERCA** y **TORNILLO**, cada uno dibuja el objeto que describe su nombre, solicitando al usuario los datos necesarios.

Lo primero que haremos es crear un índice en el que se describirán todos los temas incluidos en la ayuda. Para ello, y en un editor ASCII escribiremos las siguientes líneas:
\#index

```
\$ÍNDICE DE CONTENIDOS
\Kíndice
1. Nuevo comando HÉLICE.\
2. Nuevo comando TUERCA.\
3. Nuevo comando TORNILLO.\
\E
```

Vamos a explicarlas ahora. Lo primero que nos encontramos es el identificador interno del tema. Tras el código \#, la palabra `index` (elegida por nosotros) identifica a lo que le sigue hasta encontrar otro identificador similar. Estos identificadores serán utilizados por **AutoCAD** para encontrar los diferentes temas, así como también los utilizaremos nosotros, como veremos más adelante. Son obligatorios antes de cada tema y han de ser nombres únicos — no se pueden repetir en un mismo archivo de ayuda, evidentemente— y sin espacios.

\\$ÍNDICE DE CONTENIDOS será la cadena que aparecerá a la hora de buscar un tema, como explicaremos a continuación. Parecido ocurre con la palabra clave `índice`, después del código \K.

A continuación escribimos el texto del tema en sí. Un carácter contrabarra (\) especifica la unión de líneas de distintos párrafos. Si al final de cada línea pulsamos simplemente `INTRO`, en un archivo `.AHP` esto se interpreta como cambio de párrafo, es decir, se deja una línea en blanco intermedia. Si no queremos que esto suceda habremos de introducir al final de la línea directriz dicho carácter de contrabarra.

El lector de la ayuda de **AutoCAD** está diseñado para trabajar con texto en formato DOS o UNIX EOL. Es decir, las líneas de texto del archivo fuente de ayuda deben concluir con una secuencia de retorno de carro. No se garantiza un correcto funcionamiento de archivos de ayuda escritos con otros protocolos EOL. El archivo fuente puede escribirse en ASCII de 7 u 8 bits, lo que significa que es fácilmente localizable. El lector de la ayuda de **AutoCAD** formatea el texto en el espacio disponible dentro del cuadro de lista de desplazamiento e inserta una línea en blanco entre párrafos. La ayuda de **AutoCAD** utiliza `INTRO` (secuencia `CRLF`) como separador de párrafos.

Por último introducimos el código \E de fin de archivo de ayuda. El archivo aún no está terminado, pero si queremos probarlo —que es lo que vamos a hacer— deberemos introducir este código para indicar el final de fichero y no producir un error.

NOTA: Nótese como los identificadores, los títulos y demás han de ir inmediatamente después de los códigos, sin espacios en blanco. Por otro lado, decir que se pueden introducir espacios interlineales para aclarar el archivo en su conjunto, aunque los espaciados reales entre párrafos vendrán definidos por la inclusión o no de los caracteres contrabarra explicados. Al final del archivo (tras \E) no es necesario introducir un `INTRO` para que funcione correctamente, como ocurría en otros archivos ASCII de definición de **AutoCAD**.

NOTA: Los códigos en archivos `.AHP` han de escribirse obligatoriamente en mayúsculas.

SEIS.2.1. Visualización del archivo en AutoCAD

Llegados a este punto conviene explicar la manera de cargar nuestro incipiente archivo de ayuda desde **AutoCAD**. Como hemos visto, estos archivos son simples ficheros de texto ASCII, sin embargo para poder ser utilizados habrán de ser manejados desde una aplicación AutoLISP (o ARX). Por ello, mencionaremos simplemente aquí la función AutoLISP que nos permite visualizar estos archivos de ayuda, dejando su estudio en profundidad para el **MÓDULO** de programación en AutoLISP.

La función en cuestión es **HELP**, y habrá de ser introducida en la línea de comandos de **AutoCAD** en el formato adecuado, es decir entre paréntesis. A esta función hay que proporcionarle como argumento el nombre del archivo **.AHP** de ayuda entre comillas (por ser una cadena de texto) y, opcionalmente y si fuera necesario, la ruta de acceso. Así, una vez guardado el archivo con la extensión **.AHP**, lo cargaremos introduciendo lo siguiente en línea de comandos (suponemos que se llama **PRUEBA.AHP**):

```
(HELP "prueba.ahp")
```

La extensión **.AHP** no es necesaria, aunque el autor de este curso siempre recomienda incluir todas la extensiones aún no siendo obligatorio, y todo ello por claridad.

Pero cuidado, ya que si tenemos que introducir una ruta de acceso o camino, la manera de hacerlo ha de ser en formato **AutoLISP**, esto es, escribiendo barras normales (/) en lugar de contrabarras (\) para separar directorios o carpetas. Por ejemplo:

```
(HELP "c:/autocad/ayuda/prueba.ahp")
```

También pueden ser utilizadas dos contrabarras seguidas (\\); la primera representa el carácter de introducción de códigos de control en **AutoLISP** y la segunda el propio carácter contrabarra en sí. Por ejemplo:

```
(HELP "c:\\autocad\\ayuda\\prueba.ahp")
```

De cualquiera de las formas sería válida la entrada.

NOTA: Como ya hemos de saber, para que un archivo no requiera ruta de acceso ha de encontrarse en uno de los directorios de soporte de **AutoCAD** definidos bajo **Herr>Preferencias...**

NOTA: Esta función de **AutoLISP** para llamar a archivos de ayuda puede ser incluida como macroinstrucción en opciones de menú o en botones de barras de herramientas, por ejemplo. De esta manera el usuario no habrá de escribir la línea explicada para recurrir a la ayuda de un programa nuevo. Como veremos más adelante, puede ser más conveniente en programas pequeños que manejen una interfaz de cuadro de diálogo en **DCL** o **VBA**, incluir la función de ayuda en un propio botón del cuadro, por ejemplo.

Tras cargar un archivo **.AHP** en **AutoCAD**, en el mismo directorio donde se encuentra dicho archivo se crea otro con el mismo nombre y de extensión **.HDX**. Este archivo es el denominado de índice de ayuda y es creado para gestionar con mayor rapidez la ayuda. Siempre se recomienda eliminar este archivo al introducir cambios en el **.AHP** para garantizar su actualización, sin embargo esto no es en absoluto necesario y puede llegar a resultar algo pesado. Aún así, al terminar por completo la edición de un fichero **.AHP** de ayuda de **AutoCAD**, se puede borrar y volver a cargar la ayuda para conseguir un **.HDX** final nuevo y actualizado.

Una vez realizado esto, y si no hay ningún problema, debería aparecer en pantalla la ventana de ayuda (*Ayuda de AutoCAD*) con nuestro índice creado. En la parte superior de esta ventana podemos distinguir cuatro botones principales. El botón **Índice** siempre nos llevará al primer tema definido en el archivo **.AHP** (bajo el primer identificador \#). Es por ello que suele ser práctica conveniente establecer un primer tema como índice de los demás.

El botón **Buscar**, por su lado, abre una segunda ventana en la que aparecen varios campos. La utilización de este cuadro es bien sencilla. Debemos escoger cualquier palabra del cuadro de lista superior y pulsar el botón **Mostrar temas**. Las palabras que aparecerán en este cuadro de lista serán las indicadas como palabras clave tras los códigos \k en el archivo

.AHP de definición. Una vez pulsado el botón aparecerán en el siguiente cuadro de lista (el inferior) todos los temas relacionados con la palabra clave en cuestión. Estos temas son los especificados tras los códigos \\$. Para visualizar un tema deberemos escogerlo de este segundo cuadro de lista y pulsar el botón *Ir a*. También podemos introducir una cadena en la casilla de edición superior.

Por lo que se ha dicho se comprende que es posible introducir la misma palabra clave en diferentes temas de archivo. Este es el típico funcionamiento de un archivo de ayuda Windows: tras introducir una palabra clave (\k) se nos muestran todos los temas (\\$) con los que está relacionada, es decir, donde está especificada.

El botón *Atrás* solamente estará disponible en el caso de que se haya visitado algún tema ya. Si así es, este botón nos llevará al tema anterior visualizado.

Y el botón *Historial* despliega otro cuadro de diálogo en el que se muestran todos los temas visitados y el orden en que fueron visionados. Así podemos elegir cualquiera de los vistos anteriormente y volver a él pulsado el botón *Ir a*.

En la parte central de la ventana de ayuda se muestra el texto correspondiente al tema en cuestión en el que nos hallamos. El cuadro general se cierra con el botón *Aceptar*.

NOTA: Cuando se estudie la función HELP de AutoLISP a fondo en el **MÓDULO** correspondiente, veremos otras características, como hacer que la ayuda arranque directamente en la ventana de búsqueda, por ejemplo.

SEIS.2.2. Introduciendo más temas

Por ahora sólo hemos escrito un índice en nuestro archivo de ayuda. Vamos ahora pues a introducir los diferentes temas que proporcionarán la ayuda real. El tema para el primer nuevo comando podría ser el siguiente:

```
\#helice
\$Nuevo comando HÉLICE
\Khélice;comandos nuevos;dibujo;paso;altura;precisión;vueltas;centro;radio
COMANDO HÉLICE\
-----
Este comando dibuja una curva helicoidal en 3D mediante una\
3DPOL. Para ello se sirve de un cuadro de diálogo que le\
solicitará los siguientes datos:
    * Radio inicial\
    * Radio final (si es diferente al inicial)\
    * Altura o paso\
    * Número de vueltas\
    * Precisión de cada vuelta en puntos
    * Centro de la hélice
Una vez introducidos los datos, pulse el botón "Aceptar"\
y la curva será dibujada.
Si se produce algún error en la introducción de datos,\
el programa presentará un mensaje de error en el propio\
cuadro de diálogo, y no se podrá continuar hasta que\
se subsane.
```

Aquí podemos apreciar la manera de separar entre sí las diferentes palabras clave, esto es con un carácter punto y coma (;). Las líneas se han sangrado con tabulaciones que, posteriormente serán de cuatro caracteres de longitud en la ventana de ayuda.

Si ahora nos fijamos en la ventana de búsqueda, podremos observar todas las nuevas palabras clave introducidas ordenadas por orden alfabético. Al elegir una de ellas aparecerá en la siguiente lista el tema asociado (al igual que en la lista del historial de temas visitados).

NOTA: El recurso de la contrabarra no se utiliza como se ha indicado aquí. Bajo el siguiente epígrafe de este **MÓDULO** se explicará su uso real.

Veamos la ayuda al siguiente comando nuevo creado:

```
\#tuerca
\$Nuevo comando TUERCA
\Ktuerca;comandos nuevos;dibujo;radio;centro
COMANDO TUERCA\
-----
Este comando dibuja una tuerca en planta.\
Para ello se sirve de un cuadro de diálogo que le\
solicitará los siguientes datos:
\ +++ Ahora se muestran los datos que se preguntarán +++
    * Radio interior\
    * Radio exterior\
    * Centro de la tuerca
Una vez introducidos los datos, pulse el botón "Aceptar"\
y la tuerca será dibujada.
\ +++ Ahora se explica el tratamiento de errores del cuadro +++
Si se produce algún error en la introducción de datos,\
el programa presentará un mensaje de error en el propio\
cuadro de diálogo, y no se podrá continuar hasta que\
se subsane.
```

Fijémonos ahora en la lista de búsqueda al pulsar Mostrar temas tras elegir, por ejemplo, la clave comando nuevo o la clave radio. Como ambas claves se encuentran definidas en ambos temas escritos hasta ahora, en la lista de estos aparecerán los dos. Tras elegir uno de ellos pulsaremos Ir a para visualizarlo.

Observando el código del archivo podemos ver también que se han introducido dos comentarios aclaratorios. Como hemos dicho estos comentarios hay que precederlos del carácter contrabarra y de un espacio blanco (\) como mínimo. Los comentarios en un archivo .AHP pueden colocarse en cualquier punto, a excepción de inmediatamente detrás de la directriz que contenga el código de claves \K, es decir, justo donde habría de comenzar el texto de la ayuda. Si así se hace, el comentario no será ignorado y aparecerá en pantalla.

Como último tema del archivo de ayuda de nuestro ejemplo, introduciremos el correspondiente al nuevo comando TORNILLO. Así, al final, el archivo .AHP completo sería el que se muestra a continuación:

```
\#index
\$ÍNDICE DE CONTENIDOS
\Kíndice
1. Nuevo comando HÉLICE.\
2. Nuevo comando TUERCA.\
3. Nuevo comando TORNILLO.\

\#helice
\$Nuevo comando HÉLICE
\Khélice;comandos nuevos;dibujo;paso;altura;precisión;vueltas;centro;radio
COMANDO HÉLICE\
-----
Este comando dibuja una curva helicoidal en 3D mediante una\
3DPOL. Para ello se sirve de un cuadro de diálogo que le\
```


solicitará los siguientes datos:

- * Radio inicial\
- * Radio final (si es diferente al inicial)\
- * Altura o paso\
- * Número de vueltas\
- * Precisión de cada vuelta en puntos
- * Centro de la hélice

Una vez introducidos los datos, pulse el botón "Aceptar"\
y la curva será dibujada.

Si se produce algún error en la introducción de datos,\
el programa presentará un mensaje de error en el propio\
cuadro de diálogo, y no se podrá continuar hasta que\
se subsane.

\#tuerca

\\$Nuevo comando TUERCA

\Ktuerca;comandos nuevos;dibujo;radio;centro

COMANDO TUERCA\

Este comando dibuja una tuerca en planta.\

Para ello se sirve de un cuadro de diálogo que le\
solicitará los siguientes datos:

\ +++ Ahora se muestran los datos que se preguntan +++

- * Radio interior\<
- * Radio exterior\<
- * Centro de la tuerca

Una vez introducidos los datos, pulse el botón "Aceptar"\
y la tuerca será dibujada.

\ +++ Ahora se explica el tratamiento de errores del cuadro +++

Si se produce algún error en la introducción de datos,\
el programa presentará un mensaje de error en el propio\
cuadro de diálogo, y no se podrá continuar hasta que\
se subsane.

\#tornillo

\\$Nuevo comando TORNILLO

\Ktornillo;comandos nuevos;dibujo;radio;centro;vástago;longitud

COMANDO TORNILLO\

Este comando dibuja un tornillo en alzado.\

Para ello se sirve de un cuadro de diálogo que le\
solicitará los siguientes datos:

- * Métrica\<
- * Longitud del vástago\<
- * Tipo

Una vez introducidos los datos, pulse el botón "Aceptar"\
y el tornillo será dibujado.

Si se produce algún error en la introducción de datos,\
el programa presentará un mensaje de error en el propio\
cuadro de diálogo, y no se podrá continuar hasta que\
se subsane.

\ +++ Final del archivo de definición +++

\E

NOTA: Como ya hemos comentado se podían haber introducido diferentes espaciados interlineales para dar una mayor claridad al archivo.

SEIS.2.3. Retornos suaves, tabulaciones y sangrías

El lector de ayuda de **AutoCAD** reconoce diferentes formatos especiales de escritura que luego interpreta para generar una presentación en pantalla. Dos de estos formatos ya los hemos visto: el retorno suave, que se sirve de la contrabarra (\) para evitar el interlineado blanco entre párrafos, y las tabulaciones, las cuales son convertidas a espaciados de cuatro caracteres en la ventana de ayuda de **AutoCAD**. El tercer formato dice relación a las sangrías de párrafo y puede ser asaz interesante para formatear nuestro texto.

Con respecto al retorno suave, comentar que, como ya hemos dicho, no se utiliza exactamente como se ha explicado anteriormente. La contrabarra realiza la unión entre dos párrafos diferentes. Si introducimos una contrabarra al final de cada línea estamos indicando que cambiamos de párrafo cada vez, y eso no es real.

La manera de escribir cada párrafo en un archivo .AHP es de forma continua, es decir, sin producir saltos de línea con retorno de carro hasta cambiar de párrafo. Comenzamos a escribir —por ejemplo en un editor como Microsoft Bloc de notas— y continuamos aunque se produzca un desplazamiento horizontal de la pantalla hacia la izquierda y ya no veamos el comienzo de la línea. En el momento en que queramos cambiar de párrafo podemos pulsar INTRO.

NOTA: Como sabemos, si en editores tipo Bloc de notas activamos la opción *Ajuste de línea* (en Bloc de notas se llama así, en otros puede que reciba otro nombre), la longitud de las líneas se adapta al tamaño de la ventana de texto, sea ésta cual sea. De este modo tenemos una total visualización del texto representado en varias líneas, aunque internamente sólo se encuentre en una (sin saltos de línea con retorno de carro).

Al escribir el texto así, posteriormente **AutoCAD** se encarga de darle el formato adecuado según las dimensiones de la ventana de ayuda. Aparentemente parece que el resultado es el mismo haciéndolo así o con caracteres contrabarra (\), sin embargo podremos comprobar que si queremos darle al texto algún formato especial como una sangría (que ahora veremos), sólo funcionará si lo hacemos correctamente, es decir, todo en una línea. Esto es evidente ya que si separamos con contrabarra se entiende que es otro párrafo.

Con respecto a las tabulaciones no haremos ningún comentario más que lo explicado, ya que es lo único relevante, es decir, que al introducir una tabulación en nuestro texto, ésta se interpretará y representará como un espaciado de cuatro caracteres.

Y por último, las sangrías de párrafo son interpretadas también de una manera especial por el lector de ayuda de **AutoCAD**. Para ajustar estos párrafos con sangría, el lector de la ayuda alinea el texto según la última tabulación. Este formato maneja las listas de puntos y las numeradas bastante bien, pero precisa de una técnica especial cuando la etiqueta inicial es más larga de lo habitual. En el siguiente ejemplo vemos cómo conseguir una sangría francesa de un párrafo (los símbolos → indican la situación de una tabulación del usuario; los símbolos ¶ indican la situación de un INTRO del usuario, donde no haya este símbolo no se produce salto de línea con retorno de carro):

```
→ 1. → Esto es un ejemplo de una sangría francesa. La manera de utilizar  
este método consiste en la introducción de una tabulación al principio de  
la línea, una etiqueta significativa y otra tabulación. A continuación se  
escribe el texto completo en una línea, es decir, sin saltos de carro.\¶  
→ 2. → Ahora podemos comenzar otra línea, y así sucesivamente...
```

El resultado de este código sería el siguiente:

```
1. Esto es un ejemplo de una sangría francesa. La manera de
```

- utilizar este método consiste en la introducción de una tabulación al principio de la línea, una etiqueta significativa y otra tabulación. A continuación se escribe el texto completo en una línea, es decir, sin saltos de carro.
2. Ahora podemos comenzar otra línea, y así sucesivamente...

Como sabemos, si no hubiéramos introducido la contrabarra al final del primer párrafo, ambos párrafos habrían estado separados por una línea en blanco.

Con las etiquetas de longitud de caracteres mayor de lo habitual la sangría no queda correcta si se utiliza este método. Es por ello que se recurre a un pequeño truco para amoldarlas a un formato agradable. Veamos un ejemplo (se utiliza la convención de símbolos mencionada anteriormente):

```
PUNTO PRIMERO → En este ejemplo se muestra la técnica precisa para\¶
→ → → realizar una sangría de etiquetas amplias. Esta técnica consiste
en escribir la etiqueta e introducir una tabulación. A continuación se
escribe la longitud aproximada de una línea y se realiza un cambio de
párrafo sin espaciado interlineal, esto es con el carácter contrabarra.
En la siguiente línea se introducen tres tabulaciones y se continúa
escribiendo todo en una sola línea (sin INTRO hasta el final del
párrafo).\¶
PUNTO SEGUNDO → Ahora podemos continuar con otro párrafo realizando\¶
→ → → la misma operación...
```

El resultado sería:

```
PUNTO PRIMERO En este ejemplo se muestra la técnica precisa para
realizar una sangría de etiquetas amplias. Esta técnica
consiste en escribir la etiqueta e introducir una
tabulación. A continuación se escribe la longitud aproximada
de una línea y se realiza un cambio de párrafo sin
espaciado interlineal, esto es con el carácter contrabarra.
En la siguiente línea se introducen tres tabulaciones y se
continúa escribiendo todo en una sola línea (sin INTRO hasta
el final del párrafo).\¶

PUNTO SEGUNDO Ahora podemos continuar con otro párrafo realizando
la misma operación...
```

Con todo lo explicado el principio del archivo de ayuda de nuestro ejemplo que hemos desarrollado anteriormente sería más lógico escribirlo de la siguiente forma (se utiliza la misma convención explicada para los símbolos):

```
\#index¶
\ $ÍNDICE DE CONTENIDOS¶
\ Kíndice¶
→ 1. → Nuevo comando HÉLICE.\¶
→ 2. → Nuevo comando TUERCA.\¶
→ 3. → Nuevo comando TORNILLO.\¶
¶
\#helice¶
\ $Nuevo comando HÉLICE¶
\ Khélice;comandos nuevos;dibujo;paso;altura;precisión;vueltas;centro;
radio¶
COMANDO HÉLICE\¶
-----¶
Este comando dibuja una curva helicoidal en 3D mediante una
```

```
3DPOL. Para ello se sirve de un cuadro de diálogo que le
solicitará los siguientes datos:¶
→ * Radio inicial\¶
→ * Radio final (si es diferente al inicial)\¶
→ * Altura o paso\¶
→ * Número de vueltas\¶
→ * Precisión de cada vuelta en puntos¶
→ * Centro de la hélice
Una vez introducidos los datos, pulse el botón "Aceptar"
y la curva será dibujada.¶
Si se produce algún error en la introducción de datos,
el programa presentará un mensaje de error en el propio
cuadro de diálogo, y no se podrá continuar hasta que
se subsane.¶
¶
\#tuerca¶
\$Nuevo comando TUERCA¶
\Ktuerca;comandos nuevos;dibujo;radio;centro¶
...
```

SEIS.2.4. Vínculos de hipertexto

Una característica muy interesante de los archivos .AHP de ayuda de **AutoCAD** es la posibilidad de incluir vínculos de hipertexto en los mismos. Estos vínculos se visualizarán encerrados entre pares de corchetes angulares (<< y >>) y al hacer doble clic sobre ellos mostrarán la pantalla de ayuda que tengan enlazada.

Existen dos tipos de vínculos de hipertexto para archivos .AHP: los denominados de salto de tema y los desplegables. La sintaxis para los primeros es:

```
<<Texto_del_hipervínculo>>Identificador_de_tema>
```

y la sintaxis para los desplegables es:

```
<<Texto_del_hipervínculo>>Identificador_de_tema]
```

La diferencia entre ambos es que los de salto de tema muestran la ayuda en la misma ventana del visor de ayuda de **AutoCAD**, mientras que los desplegables abren otra ventana para realizar la misma operación.

Así por ejemplo, nos habremos percatado al visualizar nuestro ejemplo de este **MÓDULO**, que resulta engorroso tener el índice delante y no poder acceder a los diferentes temas desde ahí mismo. Lo que hacíamos era dirigirnos a la ventana de búsqueda para visualizar los diferentes temas según palabras clave. De esta forma, si escribiéramos el índice así:

```
\#index
\$ÍNDICE DE CONTENIDOS
\Kíndice
<<1. Nuevo comando HÉLICE.>>helice>\
<<2. Nuevo comando TUERCA.>>tuerca>\
<<3. Nuevo comando TORNILLO.>>tornillo>\
```

en la pantalla se mostrarían las líneas encerradas entre los pares de corchetes angulares significando la presencia de un hiperenlace. Para acceder al tema enlazado según su identificador (la cadena tras \#) sólo hemos de hacer doble clic en la misma línea. Veremos

como aparece en la misma ventana de ayuda. Otra forma de acceder es haciendo un solo clic en la línea —veremos que se activa el texto debajo de la zona de texto de ayuda que dice *Escoger el tema:*— y pulsar el botón *Ir a* de la derecha inferior del cuadro (que hasta ahora no habíamos comentado).

Si, por el contrario, hubiéramos escrito el índice con enlaces desplegables así:

```
\#index
\ $ÍNDICE DE CONTENIDOS
\ Kíndice
<<1. Nuevo comando HÉLICE.>>helice]\
<<2. Nuevo comando TUERCA.>>tuerca]\
<<3. Nuevo comando TORNILLO.>>tornillo]\
```

los temas implícitos no aparecerían en la misma ventana de ayuda, sino en otra que se abre sobre la anterior llamada *Más ayuda sobre AutoCAD*. Para cerrar esta ventana habremos de pulsar el botón *Cerrar* inferior. De esta forma se cierra esta última ventana dejando al descubierto la anterior con el tema que tuviera en pantalla (en este caso el índice).

Como ya hemos podido dilucidar, los vínculos de salto de tema se utilizan para acceder a diferentes secciones desde un índice, por ejemplo, mientras que los vínculos desplegables se usan a la hora de mostrar aclaraciones o ayuda complementaria sobre un tema.

Si un hipervínculo o hiperenlace ocupa varias líneas, podemos hacer doble clic en cualquiera de las líneas (o un solo clic y pulsando el botón *Ir a*) para acceder a su tema enlazado. Por ejemplo:

```
<<Este vínculo al Tema tercero\
ocupa varias líneas en el archivo\
de definición de ayuda>>tema3>
```

Y si existen varios vínculos en una sola línea, habremos de elegir cuál queremos visitar rotando entre temas con el botón *Siguiente* de la parte inferior del cuadro de ayuda (tampoco comentado hasta ahora), y pulsando al final *Ir a*. Mientras pulsemos *Siguiente*, a la izquierda irán apareciendo los diferentes enlaces incluidos en la línea de definición (a la derecha de la etiqueta *Escoger tema:*); cuando hayamos elegido, como decimos, pulsaremos *Ir a*. Evidentemente, en estos casos un doble clic en una línea con varios enlaces implícitos no tendrá ningún efecto.

NOTA: No se pueden incluir vínculos en pantallas nacidas desplegables de otros vínculos, es decir, en una ventana *Más ayuda sobre AutoCAD* no se pueden escribir hiperenlaces; ojo, sí se puede, pero no funcionarán. Por lo tanto no interesa.

SEIS.2.5. Archivos de ayuda en directorios de sólo lectura

Como ya sabemos, por defecto **AutoCAD** crea el archivo de índice (.HDX) en el mismo directorio que el archivo de ayuda. No obstante, si se está utilizando un archivo de ayuda que se encuentra en un directorio de sólo lectura (como en una unidad de red restringida o en un CD-ROM), no se puede crear el archivo de índice. Para solucionar este problema habremos de crear un archivo de índice ficticio en otro directorio (con escritura permitida) del camino de soporte, así **AutoCAD** creará el archivo de índice real en ese otro directorio. Por ejemplo, si se introducen los siguientes comandos tras el *prompt* de MS-DOS, se crea un archivo de índice ficticio para el archivo MIAYUDA.AHP en el directorio \SUPPORT\ del programa:

```
echo dummy > \autoca~1\support\miayuda.hdx
```

A continuación se describe cómo **AutoCAD** ubica el archivo .HDX. Se puede determinar que **AutoCAD** sitúe dicho archivo en cualquier sitio de la ruta de soporte, colocando un .HDX ficticio en dicho directorio. Veamos la descripción:

Primero se busca el archivo de ayuda de **AutoCAD**. Si se puede escribir un archivo de índice en ese directorio se hace, de lo contrario, se continúa con lo siguiente. Se busca un archivo de índice con ese nombre en el camino de soporte. Si se encuentra un archivo de índice existente y se puede abrir para modificar se hace, de lo contrario, continúa buscando en el camino de soporte. Si no encuentra un archivo de índice que se pueda escribir, se continúa con lo siguiente. Se busca el archivo ACAD14.CFG en el camino de soporte. Si se encuentra y puede escribirse el archivo de índice en dicho directorio se hace, en caso contrario **AutoCAD** no podrá escribir un archivo de índice y no se mostrará el archivo de ayuda. Todas las veces que se intente cargar el archivo de ayuda se producirá el mismo proceso.

De todo ello se desprende que, al crear un archivo ficticio en un directorio de soporte, **AutoCAD** no tendrá problema alguno para crear en dicho directorio el archivo .HDX real.

NOTA: El mensaje Help index in: xxxxxx, siendo xxxxxx el camino donde se encuentra el archivo de índice, aparece en la línea de comandos para indicar dónde se ha escrito el archivo.

SEIS.3. FORMATO WINDOWS. ARCHIVOS .HLP

Los denominados archivos de ayuda *WinHelp* son los originales archivos .HLP de Microsoft Windows 95/98 y Windows NT. Estos archivos de ayuda podemos encontrarlos casi en el 100% de la aplicaciones que corren bajo entorno Windows y resultan muy cómodos de crear y visualizar.

NOTA: La norma sobre el formato de presentación de los textos de ayuda en Windows 98 ha variado con respecto a Windows 95, si embargo el fundamento sigue siendo el mismo.

Los archivos .HLP pueden contener texto, imágenes y objetos multimedia, por lo que la ayuda se transforma en un sistema eficaz de aprendizaje. Además, se puede potenciar su funcionalidad mediante macros *WinHelp* y funciones API, convirtiéndolos en una poderosa herramienta de documentación. Estos archivos suponen el formato ideal para documentación que no se actualiza con frecuencia.

NOTA: Las llamadas al API de Windows son llamadas a bibliotecas DLL. Cualquiera que entienda algo de programación visual para Windows sabrá a qué nos estamos refiriendo. Aunque tampoco nos importa demasiado.

Para crear archivos .HLP necesitamos una aplicación denominada Microsoft Help Workshop. Ésta es capaz de compilar archivos en formato RTF (texto enriquecido) para convertirlos en archivos *WinHelp* .HLP. Para conseguir esta aplicación nos podemos dirigir directamente a Microsoft; además también se proporciona en entornos de desarrollo como Microsoft Visual Basic 5.0.

NOTA: En esta sección no se pretende proporcionar un manual completo sobre el funcionamiento de Microsoft Help Workshop, sino sentar unas bases o directrices para la utilización de archivos de ayuda *WinHelp* propios con **AutoCAD**. La política que ha seguido el autor de este curso con respecto a la personalización y la programación de **AutoCAD**, consiste en explicar profundamente todo aquello que podemos tener a nuestro alcance, es decir, que se proporciona junto con **AutoCAD**. El resto —excepto honrosas excepciones—escapa a los objetivos del curso en sí y siempre será comentado, evidentemente está ahí, pero con menor alcance.

SEIS.3.1. Microsoft Help Workshop

Microsoft Help Workshop, como hemos dicho, compila archivos .HLP a partir de archivos de texto de formato enriquecido .RTF (con códigos específicos). Este formato es uno de los estándares de salida de Microsoft Word, por lo que esta aplicación se ha convertido en abanderado de desarrolladores de ayuda en formato *WinHelp*. No obstante es posible codificar estos archivos manualmente o con otras aplicaciones.

Justo antes de compilar un archivo de ayuda se crea un llamado archivo de proyecto .HPJ que contiene las definiciones de todos los archivos .RTF que se incluirán en el .HLP final, además de otros parámetros como ubicaciones de archivos gráficos, definiciones de macros, colores de la ventana de ayuda, etcétera.

Los archivos *WinHelp* .HLP, y aunque parezca lo contrario, no son ejecutables, es decir, no pueden ejecutarse por sí solos. Necesitan de una aplicación llamada Ayuda de Windows, cuyo ejecutable es WINHLP32.EXE, para poder ser corridos. Lo que ocurre es que podemos hacer doble clic en ellos porque su extensión (.HLP) está asociada con dicha aplicación. Esto ocurre con casi todos los archivos de las diferentes aplicaciones (y se puede personalizar en Windows), al igual que con los .DWG de **AutoCAD**.

Estos archivos .HLP utilizan el cuadro de diálogo *Temas de ayuda* de la aplicación Ayuda de Windows para mostrar en un principio las pestañas *Contenido*, *Índice* y *Buscar*. La pestaña *Contenido* proporciona una interfaz con los diferentes temas de ayuda contenidos en uno o varios archivos .HLP. Las entradas y presentación de la pestaña *Contenido* están definidas en un archivo .CNT de contenido. La potencia de personalización de estos archivos nos permitirá añadir nuevos temas a esta pestaña de contenidos, que no serán otra cosa que nuestros propios archivos .HLP.

SEIS.3.2. Añadiendo temas a la pestaña *Contenido*

Para añadir nuevos temas (archivos .HLP) a la pestaña *Contenido* del cuadro de ayuda propio de **AutoCAD**, lo que debemos hacer es crear un archivo de contenido llamado INCLUDE.CNT en el subdirectorio \HELP\ de **AutoCAD**. Este archivo contendrá únicamente secuencias :INCLUDE para llamar a otros archivos .CNT contenedores de los temas en cuestión; de la siguiente manera:

```
:INCLUDE nombre_archivo.CNT
```

Así, las siguientes líneas de un archivo INCLUDE.CNT incluirían los temas definidos en los diferentes archivos .CNT en la pestaña *Contenido* de la ayuda de **AutoCAD**:

```
:Include miayuda.cnt  
:Include help123.cnt  
:Include NCHélice.cnt  
:TuercaHelp.cnt
```

NOTA: Después de modificar el archivo INCLUDE.CNT o cualquier otro archivo .CNT al que se haga referencia, es necesario eliminar el archivo ACAD.GID para que los cambios surtan efecto. Más adelante se comentará la misión de cada tipo de archivo.

SEIS.3.3. Añadiendo temas a *Índice y Buscar*

Evidentemente también disponemos de la posibilidad de añadir nuestros archivos adicionales a las listas de búsqueda por palabras clave y de índice. Para ello utilizaremos la secuencia :INDEX en la forma expuesta en el siguiente ejemplo:

```
:Index Estándares=estándar.hlp
1 Estándares de oficina
2 Arquitectura
3 Vista general=arq_presen@estándar.hlp
3 Diseño=arq_diseño@estándar.hlp
3 Dibujo=arq_dibujo@estándar.hlp
3 Informes=arq_infor@estándar.hlp
2 Ingeniería
3 Vista general=ing_presen@estándar.hlp
3 Diseño=ing_diseño@estándar.hlp
3 Dibujo=ing_dibujo@estándar.hlp
3 Informes=ing_infor@estándar.hlp
2 Asesoría del personal
3 Vista general=pers_presen@estándar.hlp
```

Así se facilita el acceso al archivo ESTÁNDAR.HLP. No deberemos olvidar incluir una línea como la que sigue en el archivo INCLUDE.CNT dentro de la carpeta \HELP\ del directorio de **AutoCAD**:

```
:Include estándar.cnt
```

NOTA: Los números del ejemplo representan niveles de anidación: 1 es un tema de contiene otros tres temas —los tres números 2—, y los números 3 son los documentos de la ayuda.

Tampoco olvidaremos suprimir el archivo ACAD.GID.

A continuación se muestra una tabla con los tipos de archivos que utiliza el sistema de ayuda de Windows 95/98:

Tipo de archivo (extensión)	Descripción
.RFT	Archivo fuente para Microsoft Help Workshop.
.HPJ	Archivo de proyecto de Microsoft Help Workshop.
.HLP	Archivo objeto compilado de ayuda <i>WinHelp</i> .
.CNT	Archivo de contenido de Microsoft Help Workshop.
.GID	Archivo de configuración <i>WinHelp</i> . Archivo binario creado por el ejecutable WINHLP32.EXE. Contiene información sobre el archivo de ayuda que incluye vínculos de los archivos de contenido, nombres de todos los archivos de ayuda incluidos, palabras clave y ubicación de los archivos.
.FTS	Archivo de búsqueda de texto en la pestaña <i>Índice</i> . Archivo binario creado por el ejecutable WINHLP32.EXE la primera vez que ejecuta una búsqueda de texto en el índice.
.FTG	Archivo de grupos de búsqueda de texto. Archivo binario creado por el ejecutable WINHLP32.EXE.

Y a continuación se proporciona otra tabla que muestra los diferentes archivos de ayuda básicos que proporciona **AutoCAD** en su paquete de software:

Archivo	Contenido
ACAD.CNT	Contenido de temas de ayuda
ACAD.HLP	Referencia de comandos
ACAD_CG.HLP	Manual de personalización
ACAD_IG.HLP	Manual de instalación
ACAD_UG.HLP	Manual del usuario
ACAD_AG.HLP	Manual de <i>ActiveX Automation</i>
ACADAUTO.HLP	Referencia de <i>ActiveX Automation</i>
README.HLP	Documento <i>Readme</i>
DOCHECK.HLP	Publicaciones técnicas de AutoCAD

SEIS.3.4. Utilizar los archivos .HLP desde AutoCAD

Existen varias formas de llamar a un .HLP desde una macro de un botón, una opción de menú, un programa de AutoLISP o un programa VBA, entre otros. A lo largo de este curso se verán diferentes funciones y comandos que nos ayudarán a hacerlo. Por ahora digamos únicamente que con los archivos .HLP se puede utilizar la función HELP de AutoLISP al igual que con los .AHP.

También veremos que incluso podemos crear una orden externa de **AutoCAD** que ejecute la ayuda.

Por último, decir que en el CD-ROM de instalación de **AutoCAD**, en el directorio \BONUS\UTILS\ existe un pequeño programa que corre bajo MS-DOS llamado AHP2HLP.EXE. Este programa convierte archivos .AHP de ayuda de **AutoCAD** a archivos WinHelp .HLP. Evidentemente, el compilador HCW.EXE (*Help Compiler Workshop*) de Microsoft Help Workshop ha de estar presente, o en su defecto el HCP.EXE. Es muy fácil de utilizar.

SEIS.4. AYUDA EN FORMATO HTML

Este es el último tipo de formato de ayuda para **AutoCAD** que vamos a estudiar. Parece ser que últimamente se está produciendo un furor —no muy generalizado aún— por la presentación de la documentación en línea de las aplicaciones en formato HTML, esto es, en el mismo lenguaje que se definen los diferentes elementos de una página Web. Categóricamente HTML no puede ser considerado como un lenguaje de programación, ya que no posee las características propias de estos (posibilidad de definir y declarar variables y/o constantes, escritura de estructuras alternativas, de estructuras repetitivas, de estructuras condicionales y un largo etcétera). Sin embargo, el Lenguaje de Marcas de Hipertexto es una herramienta potente y versátil para la presentación de documentos en pantalla; con la posibilidad de introducción de textos e imágenes, así como de elementos multimedia.

La única desventaja —que no lo es— de HTML es la imperiosa necesidad de poseer instalado en nuestro equipo un navegador WWW o *browser* del tipo Microsoft Internet Explorer o Netscape Navigator (ambos gratuitos, por lo que por lo menos desembolso económico no nos supondrá).

Amén de todo esto, los archivos HTML son fácilmente actualizables sin necesidad de reinstalar o actualizar aspecto alguno en el sistema local. Además de la propia actualización *on-line* vía Internet.

SEIS.4.1. Añadiendo temas HTML a *Contenido*

Podemos añadir archivos HTML propios o vínculos a páginas Web de Internet a la pestaña *Contenido* del sistema de ayuda de **AutoCAD**, de forma similar a como lo hacíamos con los ficheros .HLP. Por ejemplo, para poder acceder con facilidad a la documentación HTML TEMA1.HTM, TEMA2.HTM y TEMA3.HTM, además de a varias páginas Web de la Red de Redes, crearemos el archivo MASAYUDA.CNT, por ejemplo, con el contenido siguiente:

```
1 Ayuda adicional
2 Nuestra información
3 Tema 1=!EF("tema1.htm"," ",4)
3 Tema 2=!EF("tema2.htm"," ",4)
3 Tema 3=!EF("tema3.htm"," ",4)
2 Sitios relacionados con la WWW
3 CodifInfo=!EF("http://www.CodifInfo.com"," ",4)
3 VigasRUs=!EF("http://www.VigasRUs.com"," ",4)
3 LosOtrosColaboradores=!EF("http://www.otros.es"," ",4)
```

A continuación añadiremos lo siguiente al archivo INCLUDE.CNT, en el directorio \HELP\ de **AutoCAD**:

```
:Include masayuda.cnt
```

Ahora, suprimiremos el archivo ACAD.GID. La próxima vez que se ejecute el archivo ACAD.HLP, seleccionando por ejemplo *Temas de Ayuda de AutoCAD* desde el menú 2, aparecerán las nuevas entradas a continuación de las existentes.

La forma de utilizar estos archivos es la misma que la anteriormente expuesta.

Y por último, recordar que en la sección ***HELPSTRINGS de los archivos de definición de menús se pueden introducir diferentes textos (pequeñas cadenas) de ayuda contextual. No es lo mismo que definir un archivo completo de ayuda en línea, pero sí habrá de servir como complemento.

SEIS.5. EJEMPLOS PRÁCTICOS DE ARCHIVOS DE AYUDA

NOTA: Se utilizan las mismas convenciones de sintaxis explicadas: → para las tabulaciones y ¶ para los INTRO; en el final de línea sin este último símbolo no se da un salto de línea con retorno de carro.

SEIS.5.1. Ayuda a nuevos comandos creados

```
\#indic¶
\$Índice¶
\KÍndice¶
<<Nuevo comando ESPIRAL.>>espiral>\¶
<<Nuevo comando CIRCCON.>>circon>\¶
<<Nuevo comando ESTRELLA.>>estrella>\¶
<<Nuevo comando HÉLICE3D.>>helic3d>\¶
¶
\#espiral¶
\$Nuevo comando ESPIRAL¶
\Kespiral;semicírculos;paso;radio;espiral, comandos nuevos:¶
```

```
COMANDO ESPIRAL\¶
-----¶
→ ESPIRAL dibuja una falsa espiral formada por semicírculos. Se genera
siempre en sentido horario, a partir de un centro, con un radio inicial y
un paso.¶
→ Abreviatura del comando: ESP\¶
→ Comando relacionado: <<HÉLICE3D>>helic3d>¶
¶
\#circon¶
\$Nuevo comando CIRCCON¶
\Kcircon;círculos;concéntricos;radio;circon, comandos nuevos:¶
COMANDO CIRCCON\¶
-----¶
→ CIRCCON dibuja círculos concéntricos respecto a un punto central.
El programa solicita el número de círculos concéntricos que serán
generados. Se señala luego el punto central y después los radios interior
y exterior. El programa distribuye los círculos concéntricos de forma
equidistante.¶
→ Abreviatura del comando: CC¶
¶
\#estrella¶
\$Nuevo comando ESTRELLA¶
\Kestrella;puntas;estrella, centro de;radio;estrella, comandos nuevos:¶
COMANDO ESTRELLA\¶
-----¶
→ ESTRELLA dibuja una estrella a base de líneas, indicando el centro, los
dos radios entre los que se distribuyen las puntas y el número de éstas.
Según ambos radios tengan el mismo signo o se indique uno de ellos
negativo, el aspecto de la estrella resultante cambia.¶
→ Abreviatura del comando: ET¶
¶
\#helic3d¶
\$Nuevo comando HÉLICE3D¶
\Khélice3d;cable;paso;sección;hélice3d, comandos nuevos:¶
COMANDO HÉLICE3D\¶
-----¶
→ HÉLICE3D dibuja una hélice o arrollamiento de cable en 3 dimensiones
mediante SUPREGLA. Presenta opciones para indicar un paso de hélice
constante, un paso proporcional al aumento o disminución de la sección, o
un apilamiento del cable. ¶
→ Abreviatura del comando: 3DH\¶
→ Comando relacionado: <<ESPIRAL>>espiral>¶
¶
\ *** /\ \ Final del fichero /\ \ ***¶
¶
\E
```

SEIS.5.2. Documentación sobre un comando nuevo

...

```
¶
\#cu¶
\$CU¶
\KCU; Conversión de unidades¶
→ (CU) → Esta rutina ofrece un menú de conversión de unidades,
ofreciendo las que se encuentran definidas en el archivo
<<ACAD.UNT>>unt]. Se puede utilizar en mitad de un comando cuando éste
```

solicita un valor numérico, llamando a (cu). La función (cu) no es un nuevo comando de AutoCAD sino una <<función de usuario>>funusr] que puede ser llamada desde mitad de un comando.¶

→ La función examina si existe el archivo <<ACAD.UNT>>unt] y en caso afirmativo llama a la función de búsqueda de grupos de unidades y, después solicita seleccionar la opción. Llama a continuación a la función de listado de unidades disponibles en el grupo seleccionado y por último a la función que solicita las unidades de origen y destino. La última expresión realiza la conversión mediante <<CVUNIT>>cvunit] y el valor obtenido es el devuelto por (cu).¶

```
¶  
\#unt¶  
\$ACAD.UNT¶  
\KACAD.UNT; Conversión de unidades¶
```

→ ACAD.UNT → Este archivo guarda las definiciones de conversión de unidades para la función CVUNIT de AutoLISP. El archivo es ASCII y, por lo tanto totalmente editable y personalizable. La manera de crear nuevos grupos de conversión de unidades se explica en el mismo archivo.¶

```
¶  
\#funusr¶  
\$Función de usuario¶  
\KFunción de usuario; Conversión de unidades¶
```

→ Función de usuario → Es una función con un determinado cometido, definida por el programador en una aplicación AutoLISP mediante la función inherente o "subr" DEFUN.¶

```
¶  
\#cvunit¶  
\$CVUNIT¶  
\KCVUNIT; Conversión de unidades¶
```

→ CVUNIT → Es una función inherente o subr de AutoLISP que convierte un valor de un tipo de unidades a otras. Su sintaxis es la siguiente:¶
(CVUNIT valor unidad_1 unidad_2)¶
siendo unidad_1 las unidades actuales de valor y unidad_2 las unidades a las cuales se desea convertir dicho valor.¶

...

\E

SEIS.FIN. EJERCICIOS PROPUESTOS

- I. Crear un archivo de ayuda de **AutoCAD** sencillo; cualquiera.
- II. Crear un archivo de ayuda con tabulaciones y sangrados.
- III. Diseñar un documento de ayuda con tabulaciones, sangrados y retornos suaves.
- IV. Diseñar un archivo .AHP con hipervínculos de las dos clases existentes.
- V. Diseñar una serie de documentos completos de ayuda para **AutoCAD** que expliquen el diseño de los patrones de sombreado creados para el mundo de la construcción en el último ejercicio propuesto del **MÓDULO CUATRO**.

EJERCICIOS RESUELTOS DEL MÓDULO CINCO

EJERCICIO I

*1,17,ARCFLE
12,(0,4,127),02C,05,02C,06,05,028,06,040,05,016,06,01A,0

EJERCICIO II

*2,6,MECA
05,015,06,013,020,0

EJERCICIO III

*3,4,JAL
034,01F,019,0

EJERCICIO IV

*4,20,HELI
054,010,02C,020,024,010,05C,018,024,028,02C,018,02,01A,01,074,060,07C,068,0

EJERCICIO V

*5,34,GEO
05,018,06,05,014,06,05,010,06,01C,02,01C,01,05,12,(0,4,127),06,05,12,
(0,4,-127),06,02,01C,01,038,064,060,06C,038,0

EJERCICIO VI

*6,15,RESIS
03,2,010,04,2,013,02D,023,02D,023,01D,03,2,010,0

EJERCICIO VII

*00066,22,fminus
2,14,8,(-2,-6),034,1,030,2,023,1,016,018,01A,05C,2,050,14,8,(-4,-3),0

EJERCICIO VIII

*00051,25,qmayus
2,14,8,(-2,-6),022,1,01E,01A,018,016,044,012,020,01E,03C,01A,01E,2,020,
14,8,(-4,-3),0

EJERCICIO IX

*0004B,83,kmayit
2,14,3,2,14,8,(-25,-42),14,4,2,8,(9,21),1,8,(-6,-21),2,8,(7,21),
1,8,(-6,-21),2,8,(19,21),1,8,(-17,-13),2,8,(7,4),1,8,(4,-12),2,
8,(-5,12),1,8,(4,-12),2,8,(-10,21),1,070,2,060,1,060,2,
8,(-25,-21),1,070,2,060,1,060,2,040,14,3,2,14,8,(-21,-21),14,4,2,
0

EJERCICIO X

*97,93,alfmin
2,14,3,2,14,8,-23,-28,14,4,2,14,5,8,10,14,1,8,-3,-1,42,43,8,-1,-3,60,45,
8,3,-1,32,33,50,8,2,3,67,8,1,3,2,152,1,41,42,43,8,-1,-3,60,45,47,2,8,3,
14,1,32,47,45,8,2,-8,45,30,2,229,1,30,45,8,2,-8,45,47,16,2,8,3,0,14,6,14,

3,2,14,8,23,-18,14,4,2,0

EJERCICIO XI

(Ejercicio completo para resolver por técnicos y/o especialistas).

MÓDULO SIETE

Creación de órdenes externas, redefinición y abreviaturas a comandos

SIETE.1. INTRODUCCIÓN

En este **MÓDULO** se van estudiar varias características de **AutoCAD** que nos permitirán acceder desde la línea de comandos a las llamadas órdenes externas, es decir, a la ejecución de comandos que nada tienen que ver con **AutoCAD**. Estos comandos pueden llamar a aplicaciones externas como un editor de texto o una base de datos de Windows, o incluso a comandos a nivel de sistema operativo MS-DOS, los cuales pueden resultarnos útiles en determinados momentos de la edición de un dibujo. Y todo ello desde el propio **AutoCAD**.

Asimismo se explicará la forma —simple sobremanera— de redefinir los comandos del programa, esto es, de asignar a las propias órdenes de **AutoCAD** funciones diferentes o ampliadas con respecto a las que ya poseen. De la misma manera veremos la posibilidad de seguir accediendo a la definición antigua de dichas órdenes, así como de anular la nueva definición creada.

Por último veremos la característica de creación de abreviaturas desde el teclado a los comandos más usados de **AutoCAD**. Podremos crear nuestros propios alias, modificar los existentes y/o añadir más definiciones a los actuales. Todo ello, acompañado de los habituales ejemplos explicativos, así como de los ejercicios de autoevaluación, nos ocupará la extensión de este **MÓDULO SIETE**.

SIETE.2. DEFINICIÓN DE COMANDOS EXTERNOS

Como ya se ha dicho, desde **AutoCAD** tenemos la posibilidad de realizar llamadas a comandos de MS-DOS y a utilidades y aplicaciones Windows, todo ello en tiempo de ejecución. Para que esas llamadas, escritas en la línea de comandos de **AutoCAD**, surtan el efecto deseado, deberemos definir un código que las realice efectivas. Las definiciones de llamadas a comandos externos se realizan en un archivo llamado `ACAD.PGP`, el cual se encuentra en el directorio `\SUPPORT\` del programa.

`ACAD.PGP` es un archivo ASCII, por lo que resulta fácilmente modificable y personalizable. Este archivo se busca en los directorios de soporte del programa y se carga (el primero que se encuentra si hay varios) cada vez que comenzamos un dibujo nuevo, abrimos uno existente o, evidentemente, arrancamos **AutoCAD**. En él se encuentran ya definidos los comandos externos que por defecto se suministran con **AutoCAD**, así como las abreviaturas a comandos que más adelante, y en este mismo **MÓDULO**, estudiaremos. Al abrir este archivo con cualquier editor ASCII, encontraremos primero la típica presentación de Autodesk, seguida primeramente de las definiciones de comandos externos. Se proporciona también una pequeña explicación para la creación de estos.

SIETE.2.1. Comandos externos a nivel MS-DOS

Una de las línea primeras de definición que podemos encontrar puede ser la siguiente:

```
CATALOGO, DIR /W, 0,Especificación de archivos: ,
```

Pero, ¿qué significa todo esto? ¿Cómo se definen estos comandos realmente? Vamos pues a explicar los diferentes parámetros de la definición.

Lo primero que tenemos que comentar es la posibilidad de introducir comentarios en este archivo, recurriendo, como en otros muchos casos de archivos de definición de **AutoCAD**, al carácter punto y coma (;) al principio de la línea. Al igual que en otros casos, con un solo punto y coma es suficiente. Las líneas que comiencen con ; serán ignoradas por **AutoCAD** a la hora de leer el archivo `ACAD.PGP`. Además, disponemos también de la ventaja de introducir líneas blancas como separadores interlineales para imprimir claridad a los textos, sin que esto afecte tampoco a su posterior proceso.

La sintaxis de definición de comandos externos MS-DOS en el archivo `ACAD.PGP` es la siguiente:

```
Nombre_comando, [solicitud_DOS], Indicador_bit, [*][Mensaje],
```

Pasaremos ahora a comentar cada uno de los parámetros incluidos.

NOTA: Las comas (,) son todas obligatorias.

- *Nombre_comando*. Es el nombre que le asignamos al comando que estamos creando. Este nombre será posteriormente reconocido por **AutoCAD** (como comando externo), por lo que no debe coincidir con el nombre de ningún comando inherente al programa.
- *Solicitud_DOS*. Se refiere al comando a nivel MS-DOS que se ejecutará el llamar desde **AutoCAD** a *Nombre_comando*. Es el nombre de una orden, proceso o programa con sus parámetros y modificadores si los necesitara. Es un parámetro opcional (ya veremos por qué).
- *Indicador_bit*. Es un bit con los siguientes significados base:
 - 0 (bit cero): inicia la aplicación y espera a que termine.
 - 1 (primer bit): no espera a que termine la aplicación.
 - 2 (segundo bit): ejecuta la aplicación minimizada.
 - 4 (tercer bit): ejecuta la aplicación en modo "oculto".

Algunos valores de bit pueden ser sumados, integrándose así unas acciones con otras. De esta forma, un valor de bit de 3 ejecutaría la aplicación minimizada y no esperaría a que terminara, y un valor de 5 la ejecutaría en modo oculto y no esperaría.

Todos los valores indicados (0, 1, 2 y 4) más las combinaciones expuestas (3 y 5) representan todos los valores posibles que podemos dar a *Indicador_bit*, ya que no se pueden realizar más combinados porque los bits 2 y 4 se excluyen mutuamente. Además, estos dos valores suelen evitarse, ya que hacen que **AutoCAD** no se encuentre disponible hasta que la aplicación haya finalizado.

- *Mensaje*. Permite definir un mensaje para la línea de comandos de **AutoCAD** que se mostrará al llamar a la orden externa. Lo que se escriba como respuesta a dicho mensaje será pasado como parámetro adicional al comando de DOS que se ejecutará.

Es por ello, que dicho mensaje ha de ser claro en su petición. Es opcional, aunque casi siempre necesario.

Si se prevé que en algún momento, los modificadores que introduzca el usuario en línea de comando como respuesta a dicho mensaje pueden incluir algún espacio en blanco entre ellos, habremos de preceder el parámetro *Mensaje* con un asterisco (*). Éste va marcado como opcional en la sintaxis porque únicamente se utiliza en estos casos.

NOTA: La última coma (,) tras *Mensaje*, al igual que las demás, es obligatoria. Esta coma se escribe al final por compatibilidad con antiguas versiones de **AutoCAD**. En dichas versiones se incluía un parámetro más que era un código de retorno interno.

Una vez visto esto, vamos a estudiar algunos ejemplos.

La siguiente línea en el archivo ACAD.PGP define un nuevo comando **FORMATO** que permite dar formato a una unidad de disco desde **AutoCAD**:

```
FORMATO,      FORMAT,      0,      Unidad a la que dar formato: ,
```

NOTA: Como vemos se pueden incluir espacios entre los parámetros para dar claridad a la línea. El espacio tras el mensaje lo que hace es dejar un espacio en la línea de comandos después del mensaje; así no se juntará éste con lo que escribamos después. Con los demás parámetros no se pueden incluir espacios blancos antes de la siguiente coma (sólo después).

El comando nuevo de **AutoCAD** será **FORMATO**; la llamada a MS-DOS es **FORMAT** (comando que da formato a discos); el indicador de bit es 0; el mensaje va al final seguido de la última coma.

El siguiente comando externo elimina el archivo indicado desde **AutoCAD**:

```
BORRA,      DEL,      4,      Archivo que se eliminará: ,
```

Se indica el indicador de bit 4 para que la eliminación del archivo se ejecute en un segundo plano no visible (no se muestra la ventana de MS-DOS).

NOTA: La utilización de mayúsculas o minúsculas en la sintaxis es indiferente.

Veamos un último ejemplo:

```
SHELL,      ,      1,      *Comando Sistema Operativo: ,
```

Este ejemplo de **SHELL** (y alguno más que se expondrá en los ejercicios propuestos) es el único que no ha de llevar un nombre de comando, porque se lo proporciona el usuario en línea de comandos tras el mensaje.

De esta forma sencilla podemos definir una serie de comandos externos DOS que nos interese particularmente para trabajar.

NOTA: Aunque no se incluya un parámetro opcional, hay que especificar la coma correspondiente.

NOTA: Como comentario de paso, decir que existe una aplicación ARX denominada **DOSLIB14.ARX** que se encuentra en el directorio `\BONUS\CADTOOLS\` en la carpeta donde tengamos instalado **AutoCAD**. Al cargarla proporciona una serie de funciones que representan todos los comandos de MS-DOS. Nada del otro mundo.

SIETE.2.2. Reiniciar el archivo ACAD.PGP

Cada vez que editemos y modifiquemos el archivo ACAD.PGP no es necesario volver a abrir un dibujo nuevo o existente, ni muchísimo menos cerrar y arrancar **AutoCAD** nuevamente para comprobar los cambios. Existe un comando de **AutoCAD** llamado REINICIA (REINIT en inglés) que permite realizar una reinicialización de tres parámetros: el puerto de Entrada/Salida de la tableta digitalizadora, su archivo y el archivo ACAD.PGP. Si no disponemos de digitalizador configurado, sus casillas estarán inhabilitadas.

Así pues, únicamente debemos escribir el comando, activar la casilla de verificación *Archivo PGP* en el cuadro de diálogo y pulsar el botón *Aceptar*. ACAD.PGP se carga de nuevo y ya podemos comprobar los cambios escribiendo algún nuevo comando definido.

SIETE.2.3. Comandos externos Windows

De parecida forma a los comandos MS-DOS, las aplicaciones Windows pueden ser ejecutadas desde **AutoCAD** mediante un comando externo.

La sintaxis de creación de estas definiciones en ACAD.PGP es:

```
Nombre_comando,START Solicitud_Windows,1,[*][Mensaje],
```

Como se puede observar los parámetros son parecidos a los ya explicados para comandos MS-DOS. La única diferencia es que el nombre de la aplicación Windows (*Solicitud_Windows*) ha de ir precedido del comando START de MS-DOS; y que el indicador de bit siempre ha de ser 1.

START, como decimos, se refiere al comando START de MS-DOS, no a una definición START como comando externo en ACAD.PGP. Por lo que si eliminamos esa definición (que viene por defecto con **AutoCAD**) siguen funcionando las llamadas a programas Windows. El indicador de bit 1 hace que cuando se salga del programa llamado, el control se devuelva a **AutoCAD**.

Cuando existe un mensaje casi siempre habremos de especificar el asterisco. Esto es debido a que hasta una simple ruta de acceso puede incluir espacios blancos en Windows.

NOTA: Como vemos esta definición es un caso particular de la anterior personalizada para aplicaciones Windows.

Veamos algunos ejemplos:

```
BLOC,      START NOTEPAD,    1,      *Archivo que editar: ,
CALC,      START CALC,       1,,
PAINT,     START PBRUSH      1,,
```

NOTA: Evidentemente, para escribir órdenes externas para **AutoCAD**, es necesario tener un mínimo conocimiento de cómo funcionan los comandos de MS-DOS, así como las llamadas a aplicaciones Windows en su caso.

SIETE.2.4. Los comandos de Windows START y CMD

Estos comandos de Windows son muy útiles a la hora de definir órdenes externas para **AutoCAD**.

El comando **START** inicia una ventana independiente y ejecuta un programa o comando especificado. Si se utiliza **START** sin parámetros, abre una nueva ventana de solicitud de comando. Este comando tiene varios parámetros de línea de comando que determinan la presentación de la nueva ventana. En Windows NT 3.51, el parámetro `/realtime` determina que la nueva ventana se visualice en la parte superior de la ventana gráfica de **AutoCAD** (en Windows 95/98 y NT 4.0 no es necesario). Esto es útil cuando deseamos indicar comandos en la solicitud de comandos de Windows.

Usaremos **START** sin parámetros para iniciar una aplicación Windows. **START** también es muy práctico para iniciar un documento asociado a una aplicación. Por ejemplo, se puede usar para abrir directamente un documento creado con un procesador de texto o un archivo HTML.

El comando **CMD** abre una ventana de solicitud de comando que actúa como un *shell* de **AutoCAD**. Esta ventana debe cerrarse antes de devolver el control a la solicitud de comando de **AutoCAD**.

Hay dos parámetros de línea de comando de gran utilidad con los comandos externos producidos con **CMD**: `/c` y `/k`. El parámetro `/c` ejecuta el comando especificado y después se detiene (se cierra la ventana). El parámetro `/k` ejecuta el comando especificado y continúa (la ventana permanece abierta). Al utilizar el parámetro `/k` deberemos cerrar pues la ventana de comandos.

Utilizaremos, en general, **START** para iniciar una nueva ventana o aplicación que constituirá un proceso distinto de **AutoCAD**. Usaremos **CMD** para ejecutar un archivo por lotes o guión de comando que no cree una ventana diferente o bien para crear una ventana que deba cerrarse para devolver el control a **AutoCAD**.

Veamos tres ejemplos explicativos de **CMD**:

```
RUN,          CMD /C,          0,      *Archivo por lotes que ejecutar: ,
LISTSET,      CMD /K SET,      0
DXB2BLK,      CMD /C DXBCOPY,  0,      Archivo .DXB: ,2
```

El nuevo comando **RUN** ejecuta un archivo por lotes o guión de comandos. El comando **CMD** seguido del parámetro `/c` abre una ventana de comandos, ejecuta el archivo por lotes y cierra la ventana.

El comando **LISTSET** muestra los parámetros de variable de entorno DOS actual. Dado que este ejemplo usa **CMD** con `/k` en lugar de **START**, debe cerrarse la ventana de comandos antes de volver a **AutoCAD**.

El comando **DXB2BLK** crea una definición de bloque a partir del archivo `.DXB` especificado. Un archivo `.DXB` convierte todos los objetos en líneas. Una ventaja del subproducto de este procedimiento es que facilita un método sencillo para descomponer objetos de texto en líneas, entre otras.

DXB2BLK pasa el nombre de archivo `.DXB` especificado al archivo por lotes **DXBCOPY**, que copia dicho nombre de archivo a `$CMD.DXB`. A continuación, **AutoCAD** crea un bloque a partir del archivo `.DXB` especificado. El nombre proporcionado en la solicitud se utiliza como el nuevo nombre de bloque. Para crear el archivo **DXBCOPY.CMD**, se debe teclear en la solicitud de MS-DOS:

```
echo copy %1.dxb $cmd.dxb > dxbcopy.cmd
```

Con esto se crea el archivo **DXBCOPY.CMD** en el directorio actual. Es necesario trasladar dicho archivo a un directorio que se encuentre en el camino (`PATH`) de MS-DOS, o especificar

explícitamente la ubicación en el archivo ACAD.PGP. Por ejemplo, si el archivo DXBCOPY.CMD está en D:\CAD, escribiremos lo siguiente en la sección de comandos externos del archivo ACAD.PGP.

```
DXB2BLK,      CMD /C D:\CAD\DXBCOPY, 0,      Archivo .DXB: ,2
```

Para crear un archivo .DXB, seleccionaremos *Archivo DXB de AutoCAD* como impresora actual e imprimiremos a dicho archivo.

SIETE.2.5. Visto lo visto nada funciona

Si hemos intentado ejecutar algunos de los comandos externos a nivel MS-DOS, nos habremos dado cuenta de que no funcionan como es debido, ya que en los que nos interesa que la ventana DOS permanezca en pantalla esto no ocurre.

El problema reside en que nadie se pone de acuerdo en el método real de definición de estas órdenes externas para el ya casi extinto MS-DOS. Pero existe un truco que comentamos ahora porque ya hemos visto las órdenes externas Windows. Dicho truco consiste en escribir **START** delante de la llamada al comando externo MS-DOS. Con los comandos internos de DOS no funcionará.

SIETE.3. ABREVIATURAS A COMANDOS

Además de poder definir órdenes externas, en el archivo ACAD.PGP se definen también las abreviaturas que utilizamos comúnmente en línea de comandos. Así, el poder escribir **L** en lugar de **LINEA**, **C** en vez de **CIRCULO** o **RR** en lugar de **RECORTA**, obedece a una línea de definición en el archivo ACAD.PGP.

La sintaxis para la creación de estas abreviaturas es:

```
Abreviatura,*Comando_equivalente
```

Así por ejemplo, la abreviatura de la orden **LINEA** viene definida así:

```
L,      *LINEA
```

Como se ve, se pueden incluir espaciados o tabulaciones tras la coma para mayor legibilidad. La coma (,) y el asterisco (*) son obligatorios.

Veamos otros ejemplos que se podrían incluir en el archivo:

```
TX,      *TEXTODIN  
VM,      *VMULT  
RT,      *REDIBT  
RN,      *REINICIA
```

Evidentemente, bajo aplicaciones de **AutoCAD** en inglés, el comando deberá ser el correspondiente anglosajón.

NOTA: Existe un truco para poder introducir comandos en castellano en una versión de **AutoCAD** en inglés. Precisamente consiste en definir, en el ACAD.PGP, los comandos en castellano como abreviaturas de los comandos en inglés, por ejemplo: **CIRCULO,*CIRCLE**; **LIMPIA,*PURGE**; **ACERCA,*ABOUT...** Esto hace que **AutoCAD** tenga que procesar un ACAD.PGP

muy grande, pero en un equipo medianamente rápido no se nota falta de velocidad alguna. Además, los comandos transparentes los podremos ejecutar en castellano sin más, colocando el apóstrofo de rigor delante de la equivalencia en castellano, por ejemplo 'ACERCA.

Unos consejos para crear abreviaturas son:

- No crear abreviaturas de más de tres caracteres; no merece la pena.
- Procúrese indicar la primera letra del comando, a poder ser, si no las dos primeras, las tres primeras, etcétera.
- También conviene indicar letras que recuerden al comando.
- Óbviese las letras DD en comandos que abren cuadros de diálogo y las poseen.
- Una abreviatura o alias deberá acortar el comando en al menos dos caracteres si dicho comando tiene 4 ó 5, y cuatro o cinco en comandos largos.
- Los comandos con tecla aceleradora asignada, tecla de función o acceso en la barra de estado no necesitan abreviaturas.
- Utilizar el guión (-) para diferenciar entre comandos del mismo nombre que ejecuten órdenes de línea de comandos y de letrero de diálogo.

NOTA: Se pueden indicar también en estas abreviaturas comandos definidos mediante AutoLISP o ADS, así como órdenes propias del gestor gráfico que se esté utilizando.

Existe una pequeña utilidad denominada *AutoCAD Alias Editor* que se proporciona con las rutinas de *Bonus*. Se puede ejecutar desde el menú desplegable (si está cargado) *Bonus>Tools>Command Alias Editor...*, y el ejecutable (ALIAS.EXE) se puede encontrar en el directorio \BONUS\CADTOOLS\ de la carpeta donde se haya instalado **AutoCAD**.

Esta utilidad propone una interfaz gráfica para la creación de alias o abreviaturas de comandos y de órdenes externas. Digna de tener en cuenta.

NOTA: En el archivo ACAD.PGP da lo mismo definir antes las abreviaturas que los comandos externos, que al revés.

SIETE.4. REDEFINICIÓN DE COMANDOS DE AutoCAD

NOTA: Esta última parte de este **MÓDULO** no tiene nada que ver con el archivo ACAD.PGP, pero por similitud temática con las definiciones que en él se encuentran (sobre creación de comandos y abreviaturas, como hemos visto) se ha incluido aquí.

Existe la posibilidad de redefinir cualquiera de la órdenes o comandos inherentes de **AutoCAD**, es decir, de proporcionar una definición totalmente diferente a la actual a un comando y hacer que al escribir su nombre realice la nueva función. Esto es lo que se conoce como redefinición de comandos.

De esta forma, podríamos "decirle" a **AutoCAD** que el comando LINEA ya no dibuje líneas como hacía, sino que haga lo que a nosotros nos interese. Al escribir la orden en la línea de comandos, o acceder a ella desde menús, macros o cualquier otro método válido, el efecto será el nuevo aplicado. De la misma manera, podremos eliminar la definición creada o, también, continuar accediendo a la definición base del comando aunque esté redefinido.

La definición nueva para un comando de **AutoCAD** ha de ser un programa AutoLISP o ADS. Es por ello que aquí explicaremos el método, pero no podremos darle una aplicación real hasta dominar, por ejemplo, la programación en AutoLISP, cosa que se aprenderá en su momento (**MÓDULO ONCE**).

El proceso de redefinición de un comando pasa por la anulación de la definición actual base. Para ello se utiliza el comando **ANULADEF** de **AutoCAD** (**UNDEFINE** para las versiones sajonas). Al teclear este comando se nos solicitará el nombre del comando de **AutoCAD** que queremos redefinir. Una vez introducido dicho nombre, el comando en cuestión estará inoperativo.

Antes o después de anular una definición deberemos cargar en memoria (mediante **APPLOAD**) el programa que sustituirá a la definición base. Este programa deberá contener una función de usuario con el mismo nombre que el comando que redefiniremos. Así por ejemplo, para anular la orden **LINEA** con una rutina **AutoLISP**, ésta deberá tener una función **C:LINEA**. De esta manera, al llamar a **LINEA** se ejecutará el programa implícito.

A pesar de todo ello, siempre es posible acceder a la antigua y habitual definición de un comando de **AutoCAD** redefinido. Para esto, únicamente deberemos escribir un punto (.) delante del comando en cuestión a la hora de llamarlo (desde línea de comandos, menús...), así por ejemplo:

.LINEA

NOTA: Esta forma de llamar a comandos, junto con el guión bajo que hace que se acepte en cualquier versión idiomática del programa, es la típica que ha de utilizarse en la creación de macros, programas **AutoLISP** y demás. De esta forma, nos aseguramos la compatibilidad en cualquier ordenador del mundo; sea cual fuere la versión idiomática de **AutoCAD** que se utilice y aunque los comandos en ella estén redefinidos. Por ejemplo: **_.LINEA**.

Para volver a la definición habitual de un comando redefinido, utilizaremos la orden **REDEFINE** (igual en inglés) de **AutoCAD**. A la pregunta del nombre de la orden, introduciremos el nombre de la anulada anteriormente mediante **ANULADEF**.

A partir de este momento, la orden funciona de forma habitual y el programa que sustituía su acción queda sin efecto.

NOTA: La manera de comportarse un programa **AutoLISP**, —a la hora de abrir un dibujo nuevo o de rearrancar **AutoCAD**— que redefine un comando es igual a la de las aplicaciones externas en general. Por ello, al cerrar **AutoCAD** la redefinición se perderá, así como puede perderse al abrir nuevos dibujos si un parámetro de configuración del programa está establecido con un determinado valor. Todo ello y las maneras de evitarlo se estudiará oportunamente más adelante.

SIETE.5. EJEMPLOS PRÁCTICOS DE COMANDOS EXTERNOS Y ABREVIATURAS

SIETE.5.1. Comandos externos MS-DOS

CREADIR,	MKDIR,	1,	*Ruta y nombre: ,
OCULTAR,	ATTRIB +H,	4,	Nombre del archivo: ,
DIRACAD,	DIR "C:\AutoCAD R14",	0,,	
SMART,	C:\DOS\STARTDRV,	4,	Unidad: ,
SETIME,	TIME,	0,	Hora: ,

SIETE.5.2. Comandos externos Windows

ESPACIO,	START CLEANMGR,	1,,	
PAINT,	START PBRUSH,	1,	*Archivo de mapa de bits: ,
IMAGEN,	START C:\ACD\ACDSEE32,	1,	*Directorio: ,
VB,	START C:\VB\VB5,	1,	*Archivo de proyecto: ,
C: ,	START C: ,	1,,	
ACAD,	START "C:\AutoCAD R14",	1,,	

SIETE.FIN. EJERCICIOS PROPUESTOS

- I. Crear un comando externo MS-DOS que permita comparar todos los archivos .LSP del directorio de trabajo actual con otro dado por el usuario (comando FC de MS-DOS).
- II. Crear un comando externo MS-DOS que elimine todos los archivos actuales en la cola de impresión (comando PRINT de MS-DOS con su modificador /T).
- III. Crear un comando externo MS-DOS que cambie el dispositivo terminal actual para trabajar desde un terminal remoto (comando CTTY de MS-DOS con su parámetro AUX).
- IV. Crear un comando externo MS-DOS que muestre la memoria libre del equipo (comando MEM de MS-DOS). Ofrecer la posibilidad de presentar la respuesta paginada (filtro |MORE de MS-DOS).
- V. Crear un comando externo MS-DOS que ejecute el archivo de proceso por lotes indicado por el usuario.
- VI. Crear un comando externo MS-DOS que permita abrir la unidad de disco especificada por el usuario.
- VII. Crear un comando externo Windows que ejecute Telnet con una conexión al sistema remoto indicado por el usuario (nombre de archivo TELNET.EXE).
- VIII. Crear un comando externo Windows que acceda a un sitio de la WWW indicado por el usuario mediante Microsoft Internet Explorer (nombre de archivo IEXPLORE.EXE).
- IX. Crear un comando externo Windows que ejecute el programa Información del sistema de Microsoft (nombre de archivo MSINFO32.EXE).
- X. Diseñese un completo grupo de órdenes externas para ejecutar desde **AutoCAD** todo el juego de comandos de MS-DOS. Asimismo, créese una batería de comandos externos, en la forma preferida por el creador, para ejecutar los programas más típicos para Windows 95/98. Hágase también un nuevo conjunto de abreviaturas o alias que abarque todo el grupo de comandos de **AutoCAD**.

EJERCICIOS RESUELTOS DEL MÓDULO SEIS

NOTA: Se utilizan las mismas convenciones de sintaxis explicadas en el **MÓDULO** anterior: → para las tabulaciones y ¶ para los INTRO; en el final de línea sin este último símbolo no se da un salto de línea con retorno de carro.

EJERCICIO I

```
\#rutinas
\ $Rutinas ExtraBonus
\KRutinas ExtraBonus 2000;ExtraBonus;2000;ORBITA;GIRATR;DESPLAZATR;ANULASOL;
  CAMBIASOL

Biblioteca de rutinas ExtraBonus 2000\
-----
Las nuevas rutinas ExtraBonus 2000 añadidas han sido diseñadas para realizarle
más fácil su trabajo diario en 3D con AutoCAD. En la biblioteca podemos
encontrar cinco comandos nuevos añadidos, todos ellos formados por otros tantos
programas en AutoLISP y Visual C++. El acceso a estos programas se realiza
mediante los nuevos comandos diseñados o mediante los menús nuevos
desplegables y/o barras de herramientas.

A continuación se presenta una lista con las diferentes nuevas rutinas,
indicando el nombre del nuevo comando asociado a la derecha:

RUTINA → → → COMANDO\
-----
ORBITA.ARX → ORBITA\
GIRA.ARX → → GIRATR\
DSPLZTR.ARX → DESPLAZATR\
ANUSOL.LSP → ANULASOL\
CHGSOL.LSP → CAMBIASOL\
\
\E
```

EJERCICIO II

```
\#purge
\ $Limpieza del dibujo actual desde la línea de comandos
\KLimpia;Purge;Limpieza;Máscara;Filtro;Comodín

Primer paso → Escriba la orden "LIMPIA" o "_PURGE" en la línea de comandos.\
→ → → Elija ahora lo que quiere limpiar: bloques, estilos de acotación,
tipos de línea, etcétera.

Segundo paso → Ahora debe elegir los nombres de elementos que desea limpiar\
→ → → Puede usar máscaras tipo comodines de MS-DOS, por ejemplo *, tr*,
*capa00*, capa?, ta??1*, etc...

Tercer paso → Elija si desea confirmar cada elemento que se eliminará\
→ → → Tras esto, sólo procede ir eliminando uno a uno o todos a la vez,
dependiendo de lo elegido.
\
\E
```

EJERCICIO III

```
\#bloques
\ $Diálogo de inserción de bloques
\KDiálogo de inserción de bloques;Inserción;Bloque

→ 1. → Teclee el comando INSERBLOQ para arrancar el cuadro de diálogo
"Inserción de bloques". También puede utilizar su correspondencia en el menú
desplegable "Bloques", en el menú de pantalla en la sección "Bloques" o en la
```


barra de herramientas "Bloques".\¶

→ 2. → Para elegir un directorio con bloques pulse el botón "Examinar..." Busque y escoja el directorio en el cuadro de diálogo correspondiente.\¶

→ 3. → Al volver al letrero elija el bloque que desea insertar de los iconos que verá en la zona derecha.\¶

→ 4. → Pulse el botón "Designar <" para designar un punto de inserción en pantalla.\¶

→ 5. → Al volver al cuadro, pulse "Aceptar" para insertar el bloque.\¶

¶

\E

EJERCICIO IV

\#ayuda¶

\\$Ayuda de AutoCAD¶

\KAyuda de AutoCAD;Ayuda;AHP;HLP;HTML¶

¶

→ <<1. Los archivos .AHP>>ahp>\¶

→ <<2. Los archivos .HLP>>hlp>\¶

→ <<3. Los archivos .HTML>>html>\¶

¶

\#ahp¶

\\$Archivos .AHP¶

\KAyuda de AutoCAD;Ayuda;AHP¶

¶

→ Los archivos .AHP son archivos de texto ASCII fácilmente definibles y que pueden proporcionar una muy valiosa ayuda. Estos archivos tienen la capacidad de admitir una serie de directrices de formato de texto, así como también es factible la inclusión de <<enlaces de hipertexto>>hiper] para que resulte sencilla la "navegación" por el documento completo de ayuda.¶

→ Véanse los formatos <<.HLP>>hlp> y <<.HTML>>html>.¶

¶

\#hlp¶

\\$Archivos .HLP¶

\KAyuda de AutoCAD;Ayuda;HLP¶

¶

→ Los archivos .HLP se producen mediante compilación de archivos de <<texto enriquecido>>rtf] .RTF. Para ello, es necesario un programa llamado "Microsoft Help Workshop", el cual realiza la compilación y nos ayuda a mejorar el documento, añadiendo imágenes, <<enlaces de hipertexto>>hiper] o índices de contenidos, por ejemplo.¶

→ Véanse los formatos <<.AHP>>ahp> y <<.HTML>>html>.¶

¶

\#html¶

\\$Archivos .HTML¶

\KAyuda de AutoCAD;Ayuda;HTML¶

¶

→ Los archivos .HTML son los indicados para la representación de páginas Web en la WWW. Son archivos fácilmente integrables en AutoCAD y que, además, permiten un amplio juego de posibilidades; como la inserción de imágenes o características multimedia.¶

→ Véanse los formatos <<.AHP>>ahp> y <<.HLP>>hlp>.¶

¶

\#hiper¶

\\$Enlaces de hipertexto¶

\KENlaces de hipertexto¶

¶

Los enlaces de hipertexto permiten saltar de un tema a otro dentro de un

archivo de ayuda. Así mismo, también pueden abrir una pantalla auxiliar para mostrar aclaraciones a términos o conceptos.¶

¶

\#rtf¶

\\$Archivos de texto enriquecido¶

\KArchivos de texto enriquecido¶

¶

Los archivos de texto enriquecido contienen códigos específicos que serán interpretados de determinada manera por Microsoft Help Workshop a la hora de compilarlos, formado así un fichero objeto final .HLP.¶

¶

\E

EJERCICIO V

(Ejercicio completo para resolver por técnicos y/o especialistas).

MÓDULO OCHO

Fotos, fototecas y archivos de guión

OCHO.1. LAS FOTOS DE AutoCAD

Una foto de **AutoCAD** no es sino un archivo con extensión `.SLD` que guarda una fotografía o instantánea de la visualización actual en pantalla gráfica. No puede considerarse como un archivo editable por el programa, ya que no se puede modificar ni imprimir, sino como una visualización destinada a la presentación de un proyecto o, inclusive, al intercambio con otras aplicaciones de autoedición o gráficos. Como decimos, **AutoCAD** sólo permite su visualización en pantalla.

La manera de obtener un archivo de foto pasa por la utilización del comando `SACAFOTO` (`MSLIDE` en versiones inglesas del programa). `SACAFOTO` obtiene pues una instantánea de la visualización actual en pantalla, que puede ser un detalle de un dibujo, el dibujo completo, una vista, etcétera, y lo guarda en un archivo con extensión `.SLD`, cuyo nombre y localización solicita al usuario mediante el letrero de diálogo correspondiente. `SACAFOTO` produce un redibujado de la pantalla antes de guardar el archivo en cuestión, por lo que únicamente los objetos visualizados en dicho momento formarán parte de la fotografía.

Dicha foto podrá ser utilizada posteriormente para mostrar en pantalla diversas visualizaciones de un proyecto, por ejemplo, sin necesidad de recurrir a la apertura de los correspondientes dibujos `.DWG` —que puede llevar largo tiempo— ni a la aplicación de puntos de vista a posteriori (o definidos anteriormente). El comando para visualizar fotos guardadas es `MIRAFOTO` (`VSLIDE` en inglés), el cual solicita al usuario el archivo y localización de la foto que desea mostrar (habrá de ser un archivo `.SLD`) mediante un cuadro de diálogo.

Al mostrar una foto mediante `MIRAFOTO`, susodicha aparecerá en pantalla ocultando el dibujo de la sesión actual. Esta foto es una especie de película que se interpone entre el dibujo y el usuario, es decir, el dibujo no desaparece, sino que podríamos decir que está “debajo”. Podremos dibujar sobre ella cualquier objeto, pero con un simple redibujado desaparecerá, manteniéndose los objetos de dibujo realizados sobre ella (si los hubiera) y apareciendo de nuevo el dibujo actual. La foto muestra exactamente la visualización, proporción y colocación de los objetos a los que se hizo `SACAFOTO` anteriormente.

Supongamos un ejemplo práctico. Tenemos varios dibujos de un proyecto completo que necesitamos presentar a un superior o en un concurso público de diseños. La práctica ilógica del neófito se correspondería con acarrear todos los archivos `.DWG` del proyecto (los cuales pueden ocupar varios megas de espacio en disco), llegar al ordenador de presentación, copiar o descomprimir todos los archivos del proyecto e ir abriéndolos uno a uno, mostrando lo que interese mostrar (con `ZOOM`, `ENCUADRE`, `PTOVISTA`, `DDVPOINT`...). Esto puede llevar mucho tiempo de presentación, sobre todo si los archivos son extensos y tardan en ser procesados por **AutoCAD**.

Lo lógico sería sacar fotos tranquilamente en nuestra casa o puesto de trabajo, eligiendo vistas, encuadres, acercamientos, y llevar únicamente los archivos `.SLD` para ir abriendo uno a uno de manera rápida e interactiva. Los archivos de foto ocupan muy poco y pueden ser procesados por **AutoCAD** en pocos segundos bajo plataformas lentas; décimas de segundo, o incluso menos, en ordenadores un poco rápidos.

NOTA: Como veremos más adelante, este proceso se puede automatizar mediante archivos de guión.

Pero las fotos de **AutoCAD** también pueden servirnos para otros propósitos importantes, ya que, como se comentó en su momento, los menús de imágenes poseen fotos en sus áreas de iconos (como el cuadro de elección de patrones de sombreado) y los cuadros de diálogo en DCL, como veremos, utilizan fotos para representar imágenes. La manera de utilizarlas en estos casos se ha estudiado o se estudiará en su momento.

OCHO.1.1. FOTOS DE MAYOR RENDIMIENTO

Aunque parezca sencilla la obtención de una foto, la verdad es que puede llegar a complicarse a la hora de introducirla en un menú de imagen o en un cuadro de diálogo en DCL.

Sacar la foto en sí no tiene ningún secreto, el problema se presenta cuando dicha foto no se visualiza como deseamos en un cuadro tan pequeño como puede ser el de un menú de imagen o, como decimos, el área de imagen de un cuadro DCL. Para ello, existe un truco poco divulgado que consiste en lo siguiente.

Una vez comprobado el espacio que disponemos para la inclusión de la foto, la obtención de ésta pasa por el encogimiento de la ventana de **AutoCAD** (la ventana general, normalmente). Esto se consigue haciendo clic en el botón *Restaurar* (el situado en el centro de los botones de control de la ventana cuando ésta está maximizada, en la barra de título) y adaptando la ventana (arrastrando por los bordes) hasta conseguir un área gráfica de dimensiones similares en proporción al cuadro de imagen en cuestión. Una vez conseguido esto, con la imagen bien centrada, procederemos a la obtención de la foto.

Si esto no se realiza así, puede que sobre un gran trozo de área gráfica alrededor del dibujo que se desea fotografiar. Al representar luego la foto en su cuadro de imagen, se adapta a él comprimiéndose uniformemente, por lo que el resultado final no será el esperado en absoluto.

Ahora, y antes de nada más, hemos de asimilar el concepto de fototeca.

OCHO.2. FOTOTECAS O BIBLIOTECAS DE FOTOS

Una fototeca o biblioteca de fotos es una agrupación o conjunto de fotos (.SLD) en un solo archivo de extensión .SLB (no confundir ambas extensiones). La generación de fototecas se realiza mediante un pequeño programa, que funciona bajo MS-DOS y que proporciona **AutoCAD** (versión 2.1), llamado SLIDELIB, cuyo ejecutable puede ser encontrado en el directorio \SUPPORT\ de **AutoCAD**; dicho ejecutable se llama SLIDELIB.EXE.

La manera de manejar SLIDELIB es bien sencilla. El programa no posee interfaz gráfica alguna y se maneja desde la línea de comandos de MS-DOS. Su sintaxis es la siguiente:

```
SLIDELIB fototeca [<archivo_fotos]
```

donde SLIDELIB es el nombre del ejecutable de la aplicación en cuestión, *fototeca* el nombre de la biblioteca de fotos que se pretende formar y *archivo_fotos* un parámetro opcional que dice relación a un archivo de texto ASCII que contendrá los nombres y, en su caso, las rutas de acceso o caminos de los archivos de fotos que se incluirán en la fototeca. Si especificamos este último parámetro es obligatorio incluir el carácter de redireccionamiento MS-DOS <, que permite que SLIDELIB lea el contenido del archivo directamente y cree la fototeca.

Por ejemplo, si utilizamos SLIDELIB así:

```
SLIDELIB MIFOT
```

el programa utilitario permitirá ir añadiendo el nombre de una foto en cada línea. Para finalizar habremos de pulsar CTRL+Z e INTRO un par de veces. Al final se creará una fototeca continente del grupo de fotos llamada MIFOT.SLB.

Cada nombre de foto que introduzcamos puede ir acompañado de su ruta de acceso si es necesario. Asimismo, podemos indicar un camino para la creación de la fototeca, por ejemplo:

```
SLIDELIB C:\ACAD\FOTOS\MIFOT
```

sin necesidad de incluir la extensión .SLB, como vemos. Tampoco es necesario incluir la extensión .SLD a la hora de introducir las fotos.

De la misma manera, y para evitar algún error al introducir los nombres de las fotos —ya que no se puede subsanar—, la práctica habitual aconseja utilizar el segundo método, es decir, la inclusión del parámetro *archivo_fotos* (con el carácter <). De esta forma, únicamente habremos de incluir en un archivo de texto —con cualquier extensión— el nombre y camino (si es necesario éste) de todas y cada una de las fotos (una foto en cada línea), para luego llamarlo desde la línea de comandos de la forma siguiente, por ejemplo:

```
SLIDELIB C:\ACAD\FOTOS\MIFOT.SLB<C:\ACAD\FOTOS\MIFOTTX.TXT
```

Como vemos, disponemos de la posibilidad de incluir ruta de acceso al archivo de texto también.

Lo que se suele hacer normalmente, para no liar mucho el proceso, es copiar a un directorio en el que trabajaremos todas las fotos, el archivo de texto con sus nombres y el ejecutable SLIDELIB.EXE. Así el contenido de un archivo de texto llamado FOTOS.FOT podría ser:

```
FOTO1.SLD  
FOTO2.SLD  
FOTO3.SLD  
FOTO4.SLD  
FOTO5.SLD  
FOTO6.SLD  
FOTO7.SLD  
FOTO8.SLD  
FOTO9.SLD
```

Y luego, en línea de comandos de MS-DOS escribiríamos simplemente:

```
SLIDELIB FTECA.SLB<FOTOS.FOT
```

NOTA: Como siempre resulta lógico escribir las extensiones para no confundirnos, aunque, como sabemos, no son obligatorias.

Un archivo de biblioteca de fotos ocupa aproximadamente el mismo espacio en disco que la suma de los espacios que ocupan las fotos, pero al compactarse la información en único archivo, puede ocurrir que en una fototeca con muchos archivos de foto el tamaño resultante sea considerablemente menor.

En cualquier caso, una vez creada la fototeca, se deben eliminar del disco los archivos de foto (quizá tras haber hecho copias de seguridad), si no la información estaría duplicada y el uso de la fototeca no tendría sentido alguno.

NOTA: Existe un pequeño truco para añadir rápidamente el conjunto de los nombres de las fotos al archivo de texto que luego procesará SLIDELIB. Este truco consiste en escribir en línea de comandos MS-DOS, y en el directorio donde están las fotos: `DIR/B>archivo.ext`, siendo `archivo.ext` el nombre del archivo ASCII de texto y su extensión. Con `DIR/B` invocamos al comando de MS-DOS `DIR` con su modificador `/B`, que hace que en la lista de archivos del directorio aparezcan únicamente los nombres y extensiones de los mismos (sin número de bytes, fecha de la última modificación, cabecera de volumen, etc.). El carácter MS-DOS de redireccionamiento `>` hace que la lista se escriba en un archivo de salida (en este caso), el cual será el que le indiquemos. Ingenioso.

OCHO.3. UTILIZACIÓN DE FOTOS Y FOTOTECAS

OCHO.3.1. En línea de comandos de AutoCAD

Como ya se ha comentado, la manera de visualizar una foto desde **AutoCAD** es invocando al comando `MIRAFOTO` (`VSLIDE`). Sin embargo, de esta manera no se pueden visualizar fotos que se encuentren incluidas en una biblioteca de fotos o fototeca.

OCHO.3.2. En macros

La manera de visualizar fotos desde una macro de un menú, de un botón de barra de herramientas o de un archivo de guión (que enseguida veremos) es similar a la utilizada desde la línea de órdenes, es decir, con la llamada a `MIRAFOTO`, pero en este caso es posible visualizar una foto que se encuentre incluida en una fototeca. La sintaxis para ello es la siguiente:

```
... MIRAFOTO fototeca(foto)...
```

Así pues, una macro de botón o menú que hiciera visualizar una foto de una fototeca podría ser:

```
^C^C_MIRAFOTO C:/MIFOT/FTECA(FOTO1)
```

NOTA: Recordar la manera de indicar caminos con la barra normal, no la inversa.

OCHO.3.3. En menús de imágenes

Cómo se utilizan fotos y fototecas en menús de imágenes ya se explicó en **el MÓDULO UNO** (véase), únicamente emplazamos en aquel momento al lector a este **MÓDULO** para aprender la creación de las mismas. La sintaxis nos recuerda a la de la utilización en macros, ya que la llamada a fototecas siempre se realiza de igual forma: con el nombre de la fototeca y el nombre de la foto seguido y entre paréntesis. Veamos un par de ejemplos recogidos en un submenú, uno con fototeca y otro sin ella:

```
***IMAGE
**Bloques
[Rocafot(foto-1,Lavabo)]^C^C_insert lavabo
[foto-23,Arbusto]^C^C_insert arbusto
...
```

NOTA: Las fotos y las fototecas a las que llaman los menús de imagen han de encontrarse obligatoriamente en el directorio \SUPPORT\ de **AutoCAD**.

OCHO.3.4. En patrones de sombreado. El programa SlideManager

Como se comentó en su momento (véase el **MÓDULO CUATRO**) la inclusión de nuestros propios patrones de sombreado en **AutoCAD** pasa por la necesidad de introducir la definición correspondiente en uno de los archivos de patrones de sombreado de **AutoCAD**, esto es, y según necesidades, en el `ACAD.PAT` o en el `ACADISO.PAT`. Es por ello, que las fotos que representan dichos patrones únicamente pueden ser integradas en la fototeca del programa: `ACAD.SLB`.

El problema reside en que las posibilidades de `SLIDELIB` son muy limitadas, ya que no es capaz de añadir o suprimir fotos a y de una biblioteca de fotos ya creada. Es por ello, que si deseamos añadir la foto al menú de imagen de patrones de sombreado de **AutoCAD** deberemos utilizar otro programa capaz de realizar dichas funciones. Existen varios, pero sin duda el más utilizado y difundido es el llamado `SlideManager` —cuyo archivo ejecutable se denomina `SLDMGR.EXE`—, el cual no se proporciona con **AutoCAD**, pero puede ser fácilmente conseguido en el CD-ROM de cualquier revista especializada o en Internet.

`SlideManager 5.15a` (la versión que utilizaremos para la explicación) es una pequeña aplicación basada en MS-DOS que posee una interfaz gráfica bastante intuitiva (esta versión está en inglés). Aquí no se explicará el programa al completo, sino únicamente sus funciones más importantes y características. El usuario que desee profundizar en él no tiene ningún problema en aprender más simplemente manejándolo, ya que es muy sencillo.

NOTA: Si al correr el programa da un error diciendo que no encuentra el archivo de ayuda, con un simple `INTRO` podremos entrar y el programa funcionará al 100%. Algunas distribuciones obvian este archivo.

El método de trabajo para añadir una foto a una fototeca existente es el siguiente (se trabaja con las teclas del cursor y/o con las teclas rápidas indicadas en los menús desplegables en otro color y con la tecla `INTRO`):

1. Desde el menú *File* se elige *Open Library* para abrir la fototeca deseada.
2. Se recorren los directorios hasta encontrar la biblioteca en cuestión, la cual se designará y aceptará con `INTRO`. En el cuadro *Current Settings* y tras la etiqueta *Selected Library* aparecerá el nombre y ruta de acceso de la fototeca.
3. Desde el menú *File* se escoge ahora *Open Slide*, para abrir la foto que queremos incluir en la fototeca ya seleccionada.
4. Se recorren los directorios hasta encontrar la foto en cuestión, la cual se designará y aceptará con `INTRO`. En el cuadro *Current Settings* y tras la etiqueta *Selected Slide* aparecerá el nombre y ruta de acceso de la foto.
5. Para añadir la foto a la fototeca, se elige *Add Slide* del menú *Library*. Si no hay ningún problema, la foto quedará añadida a la biblioteca de fotos.

El método para eliminar una foto de una biblioteca de fotos existente es:

1. Desde el menú *File* elegir *Open Library* para abrir la fototeca.
2. Desde el mismo menú *File* escoger *Select Library Entry*, para así seleccionar una de las entradas (fotos) de la fototeca.
3. Aparece un nuevo menú de persiana a la derecha representando todos y cada uno de los nombres de las fotos de la fototeca. Elegimos uno pulsando `INTRO`.

4. En el cuadro *Current Settings* habrá aparecido tanto la fototeca como la foto seleccionada.
5. Elegiremos ahora *Delete Entry* del menú *Library*; la foto quedará eliminada de la fototeca.

Como vemos, el programa utilitario es bastante sencillo de manejar, sólo hemos de desplazarnos con los cursores (o teclas rápidas) y seleccionar con **INTRO**.

Existen otras posibilidades de SlideManager; comentaremos las más importantes:

- *Change Directory* (menú *File*) cambia el directorio por defecto del programa. Se utiliza para situarnos en un directorio y no tener que andar buscando las fotos continuamente. En el cuadro *Current Settings*, tras la etiqueta *Default Dir* aparecerá el directorio seleccionado.
- *Close Slide* y *Close Library* (menú *File*) cierran la foto y la fototeca actual respectivamente.
- *Update Entry* (menú *Library*) actualiza una foto ya incluida en la fototeca existente.
- *Extract Entry* (menú *Library*) extrae una foto de la fototeca, pero no la borra como *Delete Entry*, sino que la convierte en un archivo unitario de foto **.SLD**.
- *Explode Library* (menú *Library*) descompone una fototeca, obteniendo los consiguientes archivos unitarios de foto **.SLD** de cada una de las fotos incluidas.
- *Merge Libraries* (menú *Library*) junta varias fototecas en una sola.
- *List Library* (menú *Library*) lista el contenido de una fototeca (el nombre de los archivos de foto incluidos), ya sea en pantalla o en impresora, según elijamos.
- *Browse Library* (menú *Library*) permite visualizar todas las fotos de una fototeca una a una, recorriéndolas con los cursores o las teclas **Re Pág** y **Av Pág**.
- *Show Slide* (menú *Display*) muestra la foto seleccionada.
- Desde el menú *Output* se pueden elegir distintas salidas para la foto: impresora, archivo **.DXF**, etcétera.
- Desde el menú *Settings* se configuran distintos parámetros del sistema para optimizar el trabajo con SlideManager: impresora, puerto en paralelo **LPT**, colores, visualización...
- La opción de menú *Quit* (no se despliega) sale de SlideManager.

El programa SlideManager también puede ser utilizado desde la línea de comandos de MS-DOS, añadiendo al nombre del ejecutable distintos parámetros (es más sencillo ejecutarlo con interfaz). A continuación se muestra una lista con las más importantes opciones:

Parámetro	Descripción
A	Añadir una foto a una fototeca
D	Eliminar una foto de una fototeca
E	Extraer una foto de una fototeca
L	Listar fotos de una fototeca
M	Juntar fototecas
U	Actualizar una foto de una fototeca
X	Descomponer una fototeca

OCHO.4. ARCHIVOS DE GUIÓN

Un archivo de guión de **AutoCAD**, también llamado *script*, es un archivo de texto ASCII que contiene una secuencia de comandos del programa. Al ser corrido, los comandos se van ejecutando uno por uno, al igual que lo haríamos en línea de comandos. Son pues archivos de automatización de procesos, con extensión **.SCR**.

Hemos incluido la explicación de estos *scripts* en este **MÓDULO** debido a su relación casi exclusiva con los archivos de foto y fototecas. Y es que los archivos de guión apenas se utilizan, excepto para presentaciones de fotos en pantalla.

Decíamos al comienzo de este **MÓDULO** que para realizar una presentación de un proyecto, lo lógico era obtener archivos de foto mediante SACA FOTO para luego visualizarlos uno por uno con MIRA FOTO. Aún así, decíamos, puede resultar un tanto engorroso ir abriendo una por una todas las fotos en **AutoCAD**. Para automatizar este proceso podemos utilizar un archivo de guión.

Imaginemos un ejemplo. Deseamos mostrar en una presentación una serie de fotos de **AutoCAD** que hemos obtenido de unas cuantas vistas de un dibujo. Para ello, y con las fotos sacadas, escribiremos un archivo de texto en cualquier editor ASCII (cuidando guardarlo como archivo de texto ASCII), así:

```
MIRAFOTO C:/FOTOS/FOTO1
MIRAFOTO C:/FOTOS/FOTO2
MIRAFOTO C:/FOTOS/FOTO3
MIRAFOTO C:/FOTOS/FOTO4-1
MIRAFOTO C:/FOTOS/FOTO4-2
MIRAFOTO C:/FOTOS/FOTO5
MIRAFOTO C:/FOTOS/FOTO6
```

Este archivo lo guardamos con el nombre PRESEN.SCR (siempre esta extensión).

Si las fotos estuvieran dentro de un fototeca, la manera de llamarlas sería la indicada para macros en la sección **OCHO.3.2.**. Así, por ejemplo:

```
MIRAFOTO C:/FOTOS/FOTECA(FOTO1)
MIRAFOTO C:/FOTOS/FOTECA(FOTO2)
MIRAFOTO C:/FOTOS/FOTECA(FOTO3)
MIRAFOTO C:/FOTOS/FOTECA(FOTO4-1)
MIRAFOTO C:/FOTOS/FOTECA(FOTO4-2)
MIRAFOTO C:/FOTOS/FOTECA(FOTO5)
MIRAFOTO C:/FOTOS/FOTECA(FOTO6)
```

NOTA: Si el programa **AutoCAD** donde se ejecutará el guión está en inglés, deberemos escribir los comandos en inglés (en este ejemplo VSLIDE). También podemos utilizar la notación internacional (,) y la notación por si los comandos estuvieran redefinidos (.).

Como vemos, esto nos recuerda mucho a las macros de botones o menús. La más importante diferencia estriba en que estos archivos son verdaderos *scripts* de **AutoCAD**, es decir, un conjunto de órdenes que se ejecutan de forma continuada (al igual que los archivos de procesamiento por lotes .BAT de MS-DOS o los *scripts* de UNIX), pero no son macros. Por ello, aquí no podemos incluir caracteres de punto y coma para simular un INTRO, por ejemplo; el INTRO se corresponde aquí solamente con el espacio blanco. Tampoco podemos introducir contrabarras para solicitar datos al usuario ni otras tantas ventajas de las macros.

En los archivos de guión todo ha de ir seguido como si de la línea de comandos de **AutoCAD** se tratara, o más o menos.

También nos percatamos de la utilización de la barra inclinada normal (/) en lugar de la contrabarra o barra inversa (\) para separar directorios. Así mismo, al igual que en todos los programas Windows, podemos introducir las rutas de acceso al estilo MS-DOS:

```
... C:/Archiv~1/Cabece~1.sld
```

o utilizando nombres largos de Windows, con comillas dobles:

```
... "C:/Archivos de programa/Cabecero del pórtico.sld"
```

Como podemos observar también, la inclusión de la extensión `.SLD` de archivo de foto no es necesaria (aunque sí conveniente).

Como último detalle inicial, decir que al final del archivo, tras la última letra de la última línea debe introducirse un retorno de carro o `INTRO`, para que todo funcione correctamente. Esto lo hemos visto en la mayoría de archivos ASCII para **AutoCAD** que influyen en aspectos de personalización.

OCHO.4.1. Ejecutando archivos *scripts*

La manera de ejecutar un archivo de guión es mediante el comando `SCRIPT` (igual en inglés) de **AutoCAD** (la abreviatura por defecto es `SR` en castellano y `SCR` en inglés). También se puede escoger la opción de menú *Herr.>Ejecutar guión...* En ambos casos se abre un cuadro de diálogo de gestión de archivos para buscar y seleccionar el archivo que deseamos ejecutar.

Si ejecutamos alguno de los ejemplos propuestos, veremos que las fotos se van mostrando una por una, pero cada una permanece muy poco tiempo en pantalla; apenas se puede seguir una visualización coherente. Para remediar esto se recurre al siguiente comando de **AutoCAD**.

OCHO.4.2. Retardos con `RETARDA`

El comando `RETARDA` (`DELAY` en inglés) de **AutoCAD** se utiliza única y exclusivamente para introducir una pausa o retardo en la ejecución de un archivo de guión, con una duración especificada.

Basta introducir un número tras `RETARDA`, especificando el retardo en milisegundos, para provocar una pausa antes de ejecutar el comando siguiente del guión. En nuestro primer ejemplo podríamos hacer así:

```
MIRAFOTO C:/FOTOS/FOTO1
RETARDA 2000
MIRAFOTO C:/FOTOS/FOTO2
RETARDA 2000
MIRAFOTO C:/FOTOS/FOTO3
RETARDA 2000
MIRAFOTO C:/FOTOS/FOTO4-1
RETARDA 2000
MIRAFOTO C:/FOTOS/FOTO4-2
RETARDA 2000
MIRAFOTO C:/FOTOS/FOTO5
RETARDA 2000
MIRAFOTO C:/FOTOS/FOTO6
```

Lo que provocaría una pausa de dos segundos (2000 milisegundos) tras mostrar cada foto y antes de mostrar la anterior. Esto hace que cada una de ellas permanezca en pantalla 2 segundos (excepto la última que se queda fija al final).

El número máximo de retardo que se puede incluir es 32767 (equivalente a casi 33 segundos).

OCHO.4.3. Repeticiones con RSCRIPT

El comando **RSCRIPT** se utiliza para volver a iniciar la ejecución del último archivo de guión ejecutado. **RSCRIPT** se puede utilizar desde la línea de comandos de **AutoCAD**, provocando un reinicio del último *script* ejecutado —sin necesidad de volver a buscarlo—. Sin embargo, lo más común es utilizarlo dentro del propio archivo de guión. De esta manera, se provoca que al llegar al final se produzca un bucle al inicio, repitiéndose indefinidamente el guión de forma automática. En estos casos, evidentemente **RSCRIPT** deberá colocarse al final (en la última línea) del archivo de guión. De cualquier otra forma, se produciría una vuelta al comienzo en un punto medio, evitando que se ejecuten los comandos que se encuentran por debajo.

En nuestro ejemplo anterior, podríamos hacer que las fotos se muestren continuamente y que, al llegar al final, se comience de nuevo por el principio. Así:

```
MIRAFOTO C:/FOTOS/FOTO1
RETARDA 2000
MIRAFOTO C:/FOTOS/FOTO2
RETARDA 2000
MIRAFOTO C:/FOTOS/FOTO3
RETARDA 2000
MIRAFOTO C:/FOTOS/FOTO4-1
RETARDA 2000
MIRAFOTO C:/FOTOS/FOTO4-2
RETARDA 2000
MIRAFOTO C:/FOTOS/FOTO5
RETARDA 2000
MIRAFOTO C:/FOTOS/FOTO6
RETARDA 2000
RSCRIPT
```

Nótese que se ha incluido un nuevo comando **RETARDA**, para evitar el paso repentino de la última foto a la primera (al igual que en los demás casos intermedios).

La manera de detener un guión en ejecución, sea por su condición de infinito (como acabamos de ver) sea por apetencia del usuario, es mediante la pulsación de la tecla **ESC** o de la tecla **RETROCESO**. Aún así, el mismo archivo de guión detenido puede ser reanudado. Para ello se utiliza el siguiente comando de **AutoCAD**.

OCHO.4.4. Reanudar con REANUDA

Este comando permite continuar con la ejecución de un archivo de guión después de haber sido interrumpido mediante la tecla **ESC** o la tecla **RETROCESO**. **REANUDA** se utiliza exclusivamente en línea de comandos.

También se puede producir una interrupción cuando existe algún error en la secuencia de comandos del archivo. En este caso, el guión se interrumpe involuntariamente en el punto del error. El usuario subsana dicho error introduciendo la opción o dato correcto para teclear después **REANUDA** y retomar la ejecución del *script* en el punto inmediatamente posterior a donde se produjo el error.

Este puede ser un procedimiento útil de depuración de archivos de guión, es decir, para localizar posibles errores y continuar la ejecución del guión, corrigiendo al final en el archivo todos los errores de una vez.

OCHO.4.5. Carga de fotos antes de su visualización

En estos casos especiales en los que los guiones muestran fotos en pantalla (que son prácticamente los únicos en los que se utilizan estos archivos), puede que se produzca un ligero retraso desde que una foto desaparece de la pantalla hasta que se carga y visualiza la siguiente. Para solucionar esto, **AutoCAD** ofrece la posibilidad de escribir un carácter asterisco (*) antes del nombre de la foto o de su ruta —o de la fototeca o de su ruta, en su caso—. De esta manera, la siguiente foto se carga mientras se visualiza la actual, produciéndose un ahorro de tiempo.

El *script* que estamos utilizando como ejemplo se escribiría así:

```
MIRAFOTO C:/FOTOS/FOTO1
MIRAFOTO *C:/FOTOS/FOTO2
RETARDA 2000
MIRAFOTO
MIRAFOTO *C:/FOTOS/FOTO3
RETARDA 2000
MIRAFOTO
MIRAFOTO *C:/FOTOS/FOTO4-1
RETARDA 2000
MIRAFOTO
MIRAFOTO *C:/FOTOS/FOTO4-2
RETARDA 2000
MIRAFOTO
MIRAFOTO *C:/FOTOS/FOTO5
RETARDA 2000
MIRAFOTO
MIRAFOTO *C:/FOTOS/FOTO6
RETARDA 2000
MIRAFOTO
RETARDA 2000
RSCRIPT
```

La explicación es la siguiente. Se visualiza FOTO1 y a la vez se carga en memoria FOTO2. Se produce el retardo de dos segundos y con el comando MIRAFOTO sin nombre de archivo se visualiza la última foto cargada, es decir FOTO2; pero a la vez, se carga en memoria FOTO3. Se produce el retardo y se visualiza dicha FOTO3; a la vez se carga FOTO4, y así sucesivamente. El único lapso de tiempo en blanco se puede notar únicamente entre la última y la primera de las fotos, evidentemente.

NOTA: Inexplicablemente los nombres largos tipo Windows no se pueden utilizar con el carácter asterisco en estos casos. Habrá que utilizar los truncados MS-DOS: los seis primeros caracteres, el carácter de tilde ~ (ALT+126) y un número de orden dependiendo de si el nombre truncado existe ya o no. Vaya usted a saber.

OCHO.4.6. Otros archivos de guión

Aunque, como ya se ha comentado, la aplicación principal de los archivos de guión se restringe a la visualización de fotos en cadena, estos archivos pueden contener cualquier comando de **AutoCAD**, formando una secuencia que ejecute diversas operaciones como, por

ejemplo, limpiar todos los bloques de un dibujo. Evidentemente, desde que existe la creación de macros en botones o menús, estos *scripts* son mucho menos utilizados.

De todos modos, sirviendo de ejemplo podemos ver el siguiente *script* que crea un efecto simpático en pantalla:

NOTA IMPORTANTE DE SINTAXIS: Es importantísimo colocar bien los espacios blancos que representan INTRO y procurar no confundirse, ya que aquí no se puede utilizar el punto y coma para realizar la misma acción. A continuación los espacios se representan por el símbolo ∪ para su fácil localización; se representa el retorno de carro con el símbolo ↵.

```
DESPLAZA∪∪∪∪3,0∪↵  
RSCRIPT↵
```

Antes de ejecutarlo dibujemos un objeto en pantalla (círculo, línea...). Veremos como se desplaza hacia la derecha simulando un movimiento continuo. Si dibujamos varios objetos, primero se “marchará” el último dibujado y, cuando desaparezca de pantalla, ocurrirá lo mismo con el anterior; y así sucesivamente. Este efecto es debido a que el filtro de designación ULTIMO sólo captura el último dibujado pero que se encuentre visible en pantalla. En el momento en que ya no está, y como el *script* se repite, se coge como último el anterior dibujado que esté visible.

Como podemos comprobar, cualquier comando de **AutoCAD** se puede incluir en un archivo de guión; o mejor digamos casi cualquiera, por lo que pueda pasar... Incluso el propio comando SCRIPT para ejecutar guiones puede ser incluido en un archivo de guión. De esta manera se puede llamar a un archivo *script* desde otro. Cuando esto ocurra, el lugar correcto para incluirlo será la última línea del archivo, ya que cuando el control de la secuencia de comandos pasa al segundo guión, ya no es posible volver atrás, es decir, al terminar la ejecución de este segundo guión no se devuelve el control al primero o padre (podríamos llamarlo así).

Como último apunte final, decir que todos los comandos vistos en esta sección (SCRIPT, RETARDA, RSCRIPT y REANUDA) son transparentes, por lo que con cualquier comando en ejecución, y mediante la antecesis del apóstrofo ('), se puede utilizar cualquiera de estas técnicas de creación de archivos de guión.

NOTA: La escritura de mayúsculas o minúsculas en archivos de guión es irrelevante.

NOTA: Una variable de sistema que puede venir bien a la hora de crear macros (menús, botones y/o guiones) es EXPERT. EXPERT controla la eliminación de determinadas preguntas para que no se visualicen durante la ejecución de un comando. En el **APÉNDICE B**, en la lista de variables de sistema, se pueden consultar todos los valores posibles de esta variable.

NOTA: Como veremos en su momento, en los archivos de guión también pueden ser incluidas expresiones DIESEL y/o AutoLISP.

OCHO.4.7. Ejecución de guiones en el arranque

Si se desea ejecutar automáticamente un archivo de guión en el momento de entrar en **AutoCAD**, se debe emplear el parámetro de arranque /b. Así, para ejecutar un *script* que se encuentre en un directorio de soporte y se llame INIC.SCR al iniciar el programa, en las propiedades del acceso directo a **AutoCAD** (botón derecho en el icono), en la pestaña *Acceso directo*, en la casilla Destino:, habría que escribir (se supone directorio de instalación del programa por defecto):

"C:\Archivos de programa\AutoCAD R14\acad.exe" /b Inic

NOTA: Los parámetros de arranque de **AutoCAD** se estudian todos ellos en el **APÉNDICE H** de este curso.

OCHO.5. EJEMPLOS PRÁCTICOS DE ARCHIVOS DE GUIÓN

NOTA: Se emplea la misma sintaxis que en el ejemplo propuesto anteriormente: espacios representados por el símbolo `␣` y localización de un `INTRO` con el símbolo `↵`.

OCHO.5.1. Ejemplo 1

```
color␣rojo↵
pol␣170,160␣170,280␣340,280␣340,160↵c↵
retarda␣2000↵
desplaza␣␣␣␣-45,-72↵
retarda␣2000↵
editpol␣␣␣␣3↵
```

OCHO.5.2. Ejemplo 2

```
_open␣_y␣c:/misdoc~1/autocad/proyec~1/pasador1.dwg↵
_audit␣_y↵
_purge␣_au*_␣_n↵
_z␣_e↵
```

NOTAS INTERESANTES:

1. Como se puede apreciar, disponemos aquí también de la posibilidad de crear guiones factibles de ser ejecutados en cualquier versión idiomática de **AutoCAD**.
2. Normalmente en macros —y también en programación— no es necesario incluir el guión que llama a la orden basada en línea de comandos (aunque para eso se inventaron y crearon), ya que al ser ejecutado exteriormente lo hace directamente sin cuadro de diálogo. En el caso de `OPEN` (`ABRE`) no existe el mismo comando con guión, sino que hay que establecer a 0 la variable de sistema `FILEDIA`.

OCHO.FIN. EJERCICIOS PROPUESTOS

- I. Ser capaz de visualizar las fotos en los menús de imagen de los ejercicios propuestos del **MÓDULO UNO** que así lo requieran.
- II. Ser capaz de introducir la fotos necesarias (y después visualizarlas) en el archivo correspondiente de **AutoCAD**, de los patrones de sombreado propuestos en el **MÓDULO CUATRO**.
- III. Crear un archivo de guión que dibuje dos objetos en pantalla y, a continuación, aplique las propiedades del último dibujado al primero.
- IV. Crear un archivo de guión que dibuje una línea en pantalla y alinee el SCP con respecto a ella.

- V. Hágase un archivo de guión que presente en pantalla un cubo de lado 50 en perspectiva isométrica.
- VI. Diseñese un conjunto de archivos de guión o *scripts* que rentabilicen el trabajo en un estudio de diseño de interiores.

EJERCICIOS RESUELTOS DEL MÓDULO SIETE

EJERCICIO I

COMPARA, START FC *.LSP, 1, Archivo que desea comparar: ,

EJERCICIO II

VACÍACOLA, START PRINT /T, 0,,

EJERCICIO III

TERMINAL, CTTY AUX, 0,,

EJERCICIO IV

MEMLIBRE, START MEM, 0,(Escribir |MORE para paginación o INTRO) ,

EJERCICIO V

BATCH, , 1, Archivo de proceso por lotes: ,

EJERCICIO VI

UNIDAD, START, 1, Unidad de disco: ,

EJERCICIO VII

TELNET, START Telnet, 1, Conexión: ,

EJERCICIO VIII

WWW, START IExplore, 1, http://, ,

EJERCICIO IX

INFO, START MSInfo32, 1,,

EJERCICIO X

(Ejercicio completo para resolver por técnicos y/o especialistas).

PARTE SEGUNDA

MÓDULO NUEVE

Lenguaje DIESEL y personalización de la línea de estado

NUEVE.1. INTRODUCCIÓN

En **AutoCAD**, además todo lo visto hasta ahora, es posible personalizar y modificar la línea de estado. Nos da la impresión, visto lo visto y en espera de lo que queda por ver, que **AutoCAD** no tiene casi ningún aspecto que no pueda ser personalizado; y efectivamente así es: los menús desplegables y contextuales, el menú de pantalla, las barras de herramientas, la acción de los botones del dispositivo señalador en cuestión, los tipos de línea, los patrones de sombreado, las formas y las fuentes, los archivos de ayuda, los comandos externos, las abreviaturas e incluso los propios comandos inherentes al programa pueden ser editados, modificados o personalizados sin mayor problema, como hemos visto. Pues ahora también la línea de estado. Y aún así, la verdadera potencia de personalización vendrá con la creación de letreros de diálogo en DCL y de programas utilitarios en AutoLISP o VBA.

La línea de estado, en los programas basados en Windows, es la que se encuentra ocupando la parte inferior de la pantalla. Además sólo suele ocupar una línea, de ahí su nombre. Esta área de pantalla proporciona información al usuario acerca de los parámetros y condiciones del programa.

En **AutoCAD**, la línea de estado muestra lo siguiente, de izquierda a derecha:

- Coordenadas del cursor, que además es botón de conmutación entre coordenadas rectangulares absolutas/polares relativas/Desactivado.
- Forzado de cursor, botón de Activado/Desactivado.
- Rejilla, botón de Activado/Desactivado.

- Modo Orto, botón de Activado/Desactivado.
- Modos de referencia, botón de Activado/Desactivado. Si no hay modos prefijados, el botón accede al comando `REFENT` (`OSNAP` en inglés) para fijar modos.
- Espacio Modelo Flotante/Papel, en entorno Espacio Papel.
- Variable `TILEMODE`, cambia entre Espacio Modelo Mosaico y Espacio Papel/Modelo Flotante.

Pues bien, esta es la línea de estado por defecto, pero nosotros podemos personalizarla añadiéndole elementos por la izquierda. Para ello utilizamos la variable de **AutoCAD** `MODEMACRO`.

NUEVE.2. LA VARIABLE MODEMACRO

La variable de sistema de **AutoCAD** `MODEMACRO` almacena una cadena de texto, la cual se refiere a la configuración de la línea de estado. Por defecto está vacía, lo cual indica que la línea de estado mantiene la configuración inicial básica (la anteriormente descrita).

Esta configuración básica no se puede modificar inicialmente, sino añadirle elementos por su izquierda. Para ello editaremos la variable `MODEMACRO` —como se sabrá, esto se puede hacer escribiendo directamente su nombre o con el comando `MODIVAR` de **AutoCAD** (`SETVAR` en inglés)— y le daremos un valor nuevo.

Una forma elemental de personalizar la línea de estado es añadiendo un texto fijo a la izquierda. Para ello, dicho texto lo habremos de introducir en la variable `MODEMACRO`. Por ejemplo, podemos editar la variable y escribir:

Personalización de la línea de estado

Ahora, a la izquierda de las coordenadas (en una casilla semejante) aparecerá el texto fijo Personalización de la línea de estado.

En principio la longitud de una cadena para `MODEMACRO` es de 255 caracteres, aunque mediante concatenación de cadenas en AutoLISP se pueden introducir más, como veremos en su momento. Sin embargo, y como estudiaremos enseguida, todo lo que se introduzca en esta variable no quiere decir que vaya a ser mostrado en la línea de comandos. Así que, el texto que aparecerá no tiene longitud de cadena fija, sino la impuesta quizá por el gestor gráfico configurado y las dimensiones del monitor, así como la configuración de fuentes y demás. De todas formas, si la cadena es muy larga, las casillas por defecto irán desapareciendo por la derecha a medida que crece nuestra propia configuración por la izquierda.

Para introducir una cadena nula habremos de escribir un punto (.) más un `INTRO` tras la solicitud de `MODEMACRO`.

La variable `MODEMACRO` no se guarda, y al comenzar un nuevo dibujo, abrir uno existente o arrancar **AutoCAD** se establece como nula. Por ello, las definiciones que se realicen en ella se perderán en dichos casos. Para que esto no ocurra habremos de recurrir a métodos del tipo de la función `S::STARTUP` o del archivo `ACAD.LSP`. Todo esto se explica claramente en la sección **ONCE.15.1.** del **MÓDULO ONCE** sobre programación en AutoLISP.

Como decimos, no todo lo que se introduce en `MODEMACRO` ha de ser texto fijo, sino que podemos construir cadenas que extraigan valores de variables y se actualicen en consecuencia, por ejemplo; u otras que según qué condiciones, muestren una cosa u otra. Ello se consigue con la definición de `MODEMACRO` mediante un lenguaje basado en cadenas de texto llamado DIESEL.

NUEVE.3. EL LENGUAJE DIESEL

El lenguaje DIESEL (*Direct Interpretively Evaluated String Expression Language*) es un lenguaje de cadenas de texto que se interpreta y evalúa por **AutoCAD** directamente. En este lenguaje todas las expresiones son cadenas de texto, y los resultados también lo son.

La sintaxis general de las expresiones DIESEL es la siguiente:

`$(función, argumento1, argumento2, ...)`

Como podemos apreciar es un lenguaje basado en signos de dólar \$, paréntesis () y comas ,. El carácter \$ especifica que se trata de una expresión DIESEL, es un identificador especial para ello. A continuación irá siempre un paréntesis de apertura y el nombre de la función DIESEL en cuestión. Tras una coma, toda la serie de argumentos necesitados por la función, separados todos ellos por comas también. Estos argumentos pueden ser valores, cadenas u otras expresiones DIESEL. Para finalizar, un paréntesis de cierre acaba la expresión.

NUEVE.3.1. Catálogo de funciones DIESEL

Veremos ahora el juego de funciones DIESEL de las que disponemos para construir expresiones. En ellas se indicará el nombre de la función en mayúsculas y los argumentos en cursiva y minúsculas. Los argumentos entre corchetes cursivos definen parámetros opcionales. Están ordenadas alfabéticamente.

NOTA: El número máximo de parámetros o argumentos admitidos en una función DIESEL es de diez, incluyendo el propio nombre de la función.

`$(+, valor1 [, valor2, ... valor9])`

Efectúa la suma de todos los valores indicados.

`$(-, valor1 [, valor2, ... valor9])`

Resta del primer valor indicado, todos los demás.

`$(*, valor1 [, valor2, ... valor9])`

Efectúa el producto de todos los valores indicados.

`$(/, valor1 [, valor2, ... valor9])`

Divide el primer valor indicado entre todos los demás.

`$(=, valor1, valor2)`

Compara ambos valores indicados. Si los valores son iguales el resultado es 1 (la condición es cierta); en caso contrario el resultado es 0 (la condición es falsa).

`$(<, valor1, valor2)`

Compara ambos valores indicados. Si el valor primero es menor que el segundo el resultado es 1 (la condición es cierta); en caso contrario el resultado es 0 (la condición es falsa).

`$(>, valor1, valor2)`

Compara ambos valores indicados. Si el valor primero es mayor que el segundo el resultado es 1 (la condición es cierta); en caso contrario el resultado es 0 (la condición es falsa).

`$(!=, valor1, valor2)`

Compara ambos valores indicados. Si los valores son diferentes el resultado es 1 (la condición es cierta); en caso contrario el resultado es 0 (la condición es falsa).

`$(<=, valor1, valor2)`

Compara ambos valores indicados. Si el valor primero es menor o igual que el segundo el resultado es 1 (la condición es cierta); en caso contrario el resultado es 0 (la condición es falsa).

`$(>=, valor1, valor2)`

Compara ambos valores indicados. Si el valor primero es mayor o igual que el segundo el resultado es 1 (la condición es cierta); en caso contrario el resultado es 0 (la condición es falsa).

`$(AND, valor1 [, valor2,... valor9])`

Operador lógico Y.

`$(ANGTOS, valor [, modo, precisión])`

El valor es editado como ángulo (lo convierte en) en el formato indicado en *modo* y con la precisión indicada en *precisión*. Los valores de *modo* posibles son:

Valor	Descripción
0	Grados sexagesimales
1	Grados/minutos/segundos
2	Grados centesimales
3	Radianes
4	Orientación geográfica (N, S, E u O)

`$(EDTIME, hora, formato)`

La hora o fecha indicada se edita con el formato indicado en *formato*. Los posibles valores para *formato* son (se proporciona un ejemplo de devolución de la función):

Valor	Tipo	Ejemplo de salida
D	día	8
DD	día	08
DDD	día	Jue
DDDD	día	Jueves
M	mes	7
MO	mes	07
MON	mes	Jul

Curso Práctico de Personalización y Programación bajo AutoCAD
Lenguaje DIESEL y personalización de la línea de estado

Valor	Tipo	Ejemplo de salida
MONTH	<i>mes</i>	Julio
YY	<i>año</i>	99
YYYY	<i>año</i>	1999
H	<i>horas</i>	4
HH	<i>horas</i>	04
MM	<i>minutos</i>	53
SS	<i>segundos</i>	17
MSEC	<i>milésimas de segundo</i>	506
AM/PM	<i>mañana/tarde</i>	AM
am/pm	<i>mañana/tarde</i>	am
A/P	<i>mañana/tarde</i>	A
a/p	<i>mañana/tarde</i>	a

La fecha y la hora actual se extraen de la variable `DATE` de **AutoCAD**. Esta variable, si intentamos mostrar su resultado directamente en la línea de estado —con la función `GETVAR` que veremos enseguida— (o guardarlo en una variable desde AutoLISP), devolverá un número real. Para convertirlo necesitamos utilizar la función `EDTIME`.

```
$(EQ, valor1, valor2)
```

Compara si ambas cadenas son idénticas. Si lo son devuelve 1 (condición verdadera), si no devuelve 0 (condición falsa).

```
$(EVAL, cadena)
```

La cadena indicada es evaluada por DIESEL, obteniendo el resultado correspondiente de esa evaluación.

```
$(FIX, valor)
```

Devuelve la parte entera del valor especificado.

```
$(GETENV, variable_entorno)
```

Extrae el valor de la variable de entorno indicada.

```
$(GETVAR, variable_sistema)
```

Extrae el valor de la variable de sistema de **AutoCAD** indicada.

```
$(IF, condición, acción_se_cumple [,acción_no_se_cumple])
```

Evalúa la expresión indicada en *condición* y, según sea cierta (1) o no (0), efectúa las acciones indicadas.

```
$(INDEX, índice, cadena)
```

Si *cadena* es una expresión DIESEL con elementos separados por comas, esta función extrae el elemento indicado según el orden en *índice*. El primer elemento tiene índice 0, el segundo 1, el tercero 2, y así sucesivamente.

Curso Práctico de Personalización y Programación bajo AutoCAD
Lenguaje DIESEL y personalización de la línea de estado

`$(LINELEN)`

Extrae el número máximo de caracteres que se pueden visualizar en la actual línea de estado de la pantalla gráfica. Se suele utilizar antes de modificar `MODEMACRO`, por ejemplo.

`$(NTH, índice, argumento1 [, argumento2,... argumento8])`

Evalúa el argumento cuyo índice se indica y obtiene su resultado. El primer argumento tiene índice 0, el segundo 1, el tercero 2, y así sucesivamente.

`$(OR, valor1 [, valor2,... valor9])`

Operador lógico *O*.

`$(RTOS, valor [, modo, precisión])`

El valor es editado como número real (lo convierte en) en el formato indicado en *modo* y con la precisión indicada en *precisión*. Los valores de *modo* posibles son:

Valor	Descripción
1	Científico
2	Decimal
3	Pies y pulgadas I (fracción decimal)
4	Pies y pulgadas II (fracción propia)
5	Fraccional

`$(STRLEN, cadena)`

Extrae la longitud en caracteres de la cadena indicada.

`$(SUBSTR, cadena, inicio [, longitud])`

Extrae una subcadena de la cadena indicada, comenzando por el número de carácter indicado en *inicio* y con la longitud en caracteres indicada en *longitud*. El primer carácter empieza a numerarse desde 1. Si se omite la longitud, se obtiene todo el resto de la cadena a partir del carácter especificado.

`$(UPPER, cadena)`

Convierte a mayúsculas la cadena indicada.

`$(XOR, valor1 [, valor2,... valor9])`

Operador lógico *O exclusivo*.

DIESEL, entre otras utilidades que también veremos, es eminentemente utilizado para pasar cadenas de personalización de línea de estado a la variable `MODEMACRO`. Si embargo, con DIESEL también podemos crear efectos muy interesantes en los menús desplegables o utilizarlo para ciertos menesteres en AutoLISP. Todo ello se estudiará en su preciso momento.

NUEVE.3.2. DIESEL para la línea de estado

Habiendo comprendido la funcionalidad de la variable `MODEMACRO` y las funciones `DIESEL` disponibles, podremos percatarnos de que podemos conjugar ambos conocimientos para personalizar la línea de estado de **AutoCAD** de una manera mucho más eficiente que con un solo texto fijo.

Por ejemplo, si queremos mostrar continuamente en la línea de comandos una serie de textos fijos, pero además el nombre del dibujo actual y el nombre de la capa actual, a la solicitud de `MODEMACRO` podríamos escribir:

```
DIBUJO ACTUAL  Nombre: $(getvar,dwgname) | Capa: $(getvar,clayer)
```

Los espacios que utilicemos fuera de las expresiones `DIESEL` serán interpretados y representados por **AutoCAD** como tales. Sin embargo, en las expresiones en sí se pueden dejar diversos espacios, pero sólo después de cada carácter de coma (,) como aclaración. Si se utilizan en otro sitio (tras el símbolo \$, después del paréntesis de abrir o antes del paréntesis de cerrar) `DIESEL` no reconoce la sintaxis y puede que represente símbolos extraños en la línea de estado, además de dos signos de interrogación final (??) como mensaje de error. Las escritura en mayúsculas o minúsculas es indiferente.

En el ejemplo propuesto de establecen varios textos como fijos, además de espaciados de separación. Por otro lado se introducen dos funciones `GETVAR` de `DIESEL` que tienen la capacidad de extraer el valor de las variables de sistema de **AutoCAD**. La primera variable extraída es `DWGNAME`, que guarda el nombre del dibujo actual, y la segunda es `CLAYER` que guarda el nombre de la capa actual. Todo ello, con los textos por delante y por detrás, podría quedar así —en determinado momento— en la línea de estado:

```
DIBUJO ACTUAL  Nombre: CABLE.DWG | Capa: SOMBREADO
```

Si durante la actual sesión de dibujo se cambia de capa, se abre un dibujo guardado o se guarda al actual con otro nombre, la variable `MODEMACRO` es reevaluada y la línea de estado se actualiza instantánea y automáticamente.

Sigamos con el mismo ejemplo. El problema es que, dependiendo de cómo hayamos guardado nuestro dibujo, puede ser que el nombre aparezca totalmente en minúsculas. Para evitar esto y asegurarnos que el resultado será el mostrado un poco más arriba, escribiríamos el siguiente valor más competente para `MODEMACRO`:

```
DIBUJO ACTUAL  Nombre: $(upper,$(getvar,dwgname)) | Capa: $(getvar,clayer)
```

De esta manera extraemos el valor de la variable `DWGNAME` con `GETVAR`, pero lo filtramos con `UPPER` para que lo convierta a mayúsculas. Con la capa no es necesario, ya que siempre se guarda en mayúsculas.

NOTA: Hemos de procurar no olvidarnos de ninguna coma, ya que si no `DIESEL` no reconoce la expresión y funciona como no es debido, además de mostrar sus caracteres ?? de error.

La función de `DIESEL` para la línea de estado casi se reduce exclusivamente a la captura de valores de variables de **AutoCAD** y proceso posterior. Veamos un ejemplo más:

```
Guardar cada $(getvar,savetime) minutos.
```

Este valor de MODEMACRO hará que aparezca en la línea de estado el mensaje del texto fijo con el valor de la variable SAVETIME, que guarda el valor de los minutos entre guardado y guardado automático.

Pero las expresiones DIESEL se pueden ir complicando un poco:

```
Ventana $(if,$(=,$(getvar,cvport),1),Papel.,Modelo $(getvar,cvport).)
```

En este caso se escribe el texto fijo Ventana, después se comprueba con la función condicional IF si la variable CVPORT es igual a 1. Si así fuera se escribe Papel., lo que quedaría Ventana Papel., indicándonos que estamos en Espacio Papel. Si CVPORT es distinto de 1, escribe Modelo y luego un número que es el de la propia ventana, lo que podría quedar Ventana Modelo 3.. Los puntos que se aprecian son de final de línea, dependiendo con qué termine: con Papel o con Modelo y número de ventana.

Al cambiar de ventana, sea Mosaico o Flotante, o entre Espacio Modelo y Papel, los valores se actualizarán.

Y las expresiones DIESEL se pueden complicar aún más:

NOTA IMPORTANTE DE SINTAXIS: Las expresiones que sean muy largas y no puedan ser introducidas correctamente en este formato (como la siguiente) se dividen en varias líneas, aunque el usuario debe saber que ha de formar una sola línea en la entrada de MODEMACRO.

```
DIESEL S.A. $(if,$(getvar,snapmode),Forzcursor:  
X=$(rtos,$(index,0,$(getvar,snapunit)),2,0)  
Y=$(rtos,$(index,1,$(getvar,snapunit)),2,0)  
Ángulo=$(angtos,$(getvar,snapang))°)
```

Esta expresión escribe un texto fijo de una empresa imaginaria llamada DIESEL S.A.. A continuación comprueba si la variable SNAPMODE está activada (igual a 1), es decir si está activado el Forzcursor. Esto se hace simplemente así: \$(if,\$(getvar..., esto es, no es necesario utilizar un operador de comparación, porque en realidad se determina si la variable existe, si es igual a 1. Si es 0, es como si no existiera y no se detecta.

Si el forzado de cursor no está activado no se hace nada más, simplemente se mantiene el texto inicial. En el momento en que se active esta característica, aparece un nuevo texto Forzcursor: X= que muestra el valor del intervalo en X, extraído como primer elemento (índice 0 en INDEX) de la variable SNAPUNIT, convertido a cadena (RTOS), ya que, como sabemos, a DIESEL sólo hay que darle cadenas, y estos valores son números reales, y pasado a decimal con precisión nula, sin decimales (modo 2 y precisión 0 de la función RTOS).

Posteriormente se escribe otro texto Y=, que muestra el intervalo en Y del Forzcursor y que está extraído de la misma manera que el anterior, pero ahora utilizando un índice 1 (segundo valor) para INDEX.

Por último, se escribe el último texto Ángulo= y se extrae el ángulo de forzado de la variable SNAPANG pasándolo a cadena con la función RTOS. Al final se coloca el símbolo de grados sexagesimales (°).

NOTA: Cuando hay que escribir estas funciones DIESEL tan extensas es fácil que nos equivoquemos y tengamos que repetir desde el principio. Existe un pequeño truco para evitar esta molestia, y es escribir la serie de funciones en un editor o procesador de texto, eso sí todo en una línea o, en su defecto, sin introducir caracteres INTRO. Al final, sólo habremos de copiar y luego pegar la línea en la línea de comandos de **AutoCAD**, tras la solicitud de

MODEMACRO. Si existe algún error, la editaremos tranquilamente y volveremos a repetir la operación.

NUEVE.3.3. Expresiones DIESEL en menús

Ya al hablar de la creación de menús para **AutoCAD**, en el **MÓDULO UNO**, se explicó por encima la posibilidad de incluir expresiones DIESEL en sus definiciones. Y es que el lenguaje DIESEL no sólo sirve para personalizar la línea de estado, sino que también puede ser incluido, en forma de funciones, en la definición de cualquier tipo de menú, como decimos.

Dentro de un archivo de menú pueden aparecer expresiones en DIESEL fundamentalmente en dos áreas o elementos: entre los corchetes de la cadena textual que será mostrada o fuera de ellos, esto último es, en la macro de ejecución de la opción de menú.

NUEVE.3.3.1. DIESEL entre corchetes

Esta parte fue la explicada someramente en el **MÓDULO** de creación de menús.

Como vimos allí, una serie de funciones DIESEL pueden ser indicadas, entre los corchetes que almacenan el texto que aparecerá en el menú en pantalla, casi exclusivamente para realizar que una opción esté disponible o no y/o para colocar un símbolo de activación delante de ella. Y decimos casi exclusivamente porque, aparte de este uso, no se le suele dar otro al lenguaje DIESEL en la definición de menús entre los corchetes.

Para no redundar en lo ya explicado, simplemente veremos el ejemplo que se exponía un poco ampliado:

```
[$(if,$(getvar,tilemode),~) Esp. Papel]espaciop  
[$(if,$(getvar,tilemode),~) Esp. Modelo]espaciom
```

Estas opciones dentro de un menú harán que los textos Esp. Papel y Esp. Modelo se encuentren no disponibles cuando o mientras TILEMODE tenga valor 1 (nos encontremos en el Espacio Modelo Mosaico). Recordemos que el carácter tilde (~), que sale pulsando la tecla ALT y, a la vez, 126 en el teclado numérico, hace que una opción se encuentre no disponible (gris o "apagada") dentro de un menú, con lo que no realizará acción alguna al ser pulsada.

En el momento en que TILEMODE valga 0 (pasemos a Espacio Papel/Modelo Flotante), las expresiones DIESEL se vuelven a evaluar y las opciones se encontrarán disponibles. Al estar ambas activas, si hacemos clic en cualquiera de ella se ejecutará el comando asociado, en este caso ESPACIOP para cambiar a Papel en la primera opción, y ESPACIOM para cambiar a Modelo Flotante en la segunda.

Sabemos también que, además de estar disponible o no una opción, tenemos la posibilidad de incluir una marca de verificación (✓) delante de la misma, dependiendo de si se encuentra activada o no. Para ello, hay que utilizar los caracteres de fin de exclamación y punto seguidos (!.). Veamos un ejemplo:

```
[$(if,$(getvar,gridmode),!.)Rejilla]^G  
[$(if,$(getvar,orthomode),!.)Orto]^O  
[$(if,$(getvar,snapmode),!.)Forzcursor]^B
```

Estas opciones de menú harán que delante de cada texto (Rejilla, Orto o Forzcursor) se coloque o no una marca de verificación, dependiendo si están activadas o no.

Para saber esto se recurre a la extracción de los valores correspondientes con `GETVAR` de las variables asociadas a cada característica.

Veamos un último ejemplo en el se utilizan los dos métodos. Un menú desplegable, por ejemplo, podría ser así:

```
***POP1
[&Guardado]
[->&Guardar cada]
  [$(if,$(=,$(getvar,savetime),20),!.)&20 min.]'savetime 20
  [$(if,$(=,$(getvar,savetime),30),!.)&30 min.]'savetime 30
  [$(if,$(=,$(getvar,savetime),45),!.)&45 min.]'savetime 45
  [<-$$(if,$(=,$(getvar,savetime),60),!.)&60 min.]'savetime 60
[--]
[$$(if,$(=,$(getvar,savetime),0),~)&No guardar]'savetime 0
```

Este ejemplo crea un menú de frecuencia de guardado automático, en el que se puede elegir el tiempo de esta característica de **AutoCAD**. Las cuatro primeras opciones establecen la variable `SAVETIME` a 20, 30, 45 ó 60 minutos. Además, comparando el valor que se acaba de introducir en la variable con él mismo, se activan o no las distintas marcas de verificación, que indican cuál de las opciones está activa. La última opción, establece `SAVETIME` a 0 minutos y se desactiva a sí misma. Esto se hace para que no esté disponible hasta que se elija otra vez una de las otras, evidentemente.

Por último, recordar que ambas estructuras de opciones de menú, ya sea la de estar disponible o no o la de la marca de verificación, se pueden combinar dentro de una misma opción concatenando los signos (`~!.).` También decir que en los dispositivos gráficos que no admiten la señal `✓`, se puede utilizar cualquier otro carácter, como la letra `c`, por ejemplo (`!c`).

NUEVE.3.3.2. DIESEL en la macro

Como decimos, una expresión DIESEL puede aparecer también en una macro de menú (o incluso de un botón de barra de herramientas).

La manera de realizar estas llamadas es similar a la que utilizábamos para los submenús, pero empleando un nombre de sección especial `M`, e incluyendo a continuación la expresión DIESEL. La sintaxis pues sería:

`$M=expresión_DIESEL`

Veamos un ejemplo. La siguiente expresión define una opción de menú que es un conmutador entre Espacio Modelo Flotante y Espacio Papel:

```
[EM/EP]^C^C^P$M=$(if,$(and,$(=,$(getvar,tilemode),0),
$(=,$(getvar,cvport),1)),Espaciom,Tilemode 0 Espaciop)
```

Primeramente se efectúan dos cancelaciones y se desactiva el eco de menú (`^C^C^P`). A continuación viene la llamada a la expresión DIESEL (`$M=`). Dicha expresión incluye una función condicional `IF` que examina los valores de `TILEMODE` y `CVPORT`. Si es 0 `TILEMODE`, y al mismo tiempo `CVPORT` vale 1, esto significa que nos encontramos en el entorno Espacio Papel, por lo que se conmuta a Espacio Modelo Flotante con el comando `ESPACIOM`. En caso contrario, se desactiva `TILEMODE` (por si se encontrara con valor 1) y se cambia a Espacio Papel con el comando `ESPACIOP` de **AutoCAD**.

NUEVE.3.4. Expresiones DIESEL en botones

La manera de utilización de expresiones en DIESEL en la macro de un botón de una barra de herramientas es exactamente la misma que la vista para la macro de una opción de menú. Por ejemplo, el último ejemplo explicado (conmutador EM/EP), podría haber sido incluido con la misma sintaxis (excepto, evidentemente, la opción de menú en sí) en una macro dentro de la definición de un botón.

NUEVE.3.5. Expresiones DIESEL en archivos de guión

Decir que también es posible escribir funciones DIESEL dentro de un archivo de guión o *script*. Estos archivos como sabemos, ejecutan por lotes las líneas incluidas en ellos como si de su escritura en la línea de comando se tratara, por lo que habrá que invocar a la variable MODEMACRO anteriormente. Por ejemplo:

```
MODEMACRO  
Capa actual: $(getvar,clayer)
```

NUEVE.3.6. Variables USER_n1 a USER_n5 y MACROTRACE

Como hemos podido comprobar, el lenguaje DIESEL es muy limitado con respecto a otros entornos de desarrollo mucho más avanzados. Y es que en un principio, para **AutoCAD**, se incluyó como apoyo a los demás aspectos personalizables. Por no disponer, no disponemos en DIESEL ni de la ventaja de declarar variables de usuario y asignarles valores posteriormente.

Se incluyen en **AutoCAD** quince variables independientes de usuario que son: de USER1 a USER5, de USERR1 a USERR5 y de USERS1 a USERS5. Las variables USER_i pueden almacenar valores enteros, USERR valores reales y USERS cadenas de texto. Estas últimas son las que más nos interesan, ya que DIESEL es un lenguaje que trabaja con cadenas de texto, como ya sabemos.

Dichas variables podemos utilizarlas en cualquier momento, ya sea para almacenar un dato y después recuperarlo, ya sea para cualquier otro menester; están a disposición del usuario. Así mismo pueden ser utilizadas a la hora de programar en DIESEL, ya que a ellas se puede acceder como a cualquier otra variable de sistema en **AutoCAD**. Todas ellas se vacían igual que MODEMACRO, esto es, con un punto (e INTRO).

En DIESEL no existe ninguna función que permita editar o modificar los valores de las variables de sistema, por lo que habremos de recurrir al comando MODIVAR (SETVAR en el programa en inglés) de **AutoCAD**. Aún así, todo este proceso resulta muy engorroso y asaz limitado en el fondo, ya que sólo puede utilizarse en macros y combinando la rutina DIESEL con las llamadas a MODIVAR.

Simplemente que quede la idea de estas quince variables que podemos utilizar como memoria temporal.

NOTA: Las variables USERS1 a USERS5 se utilizan también para pasar valores de AutoLISP a expresiones DIESEL.

Por otro lado, decir que la variable MACROTRACE es una herramienta de depuración para expresiones DIESEL. Al activarse (valor 1), todas las expresiones, incluidas las utilizadas en

los menús (y botones) y en la línea de estado, son evaluadas. De esta manera se comprueba si se han escrito correctamente. No obstante, conviene evitar la evaluación excesiva.

NUEVE.4. EJEMPLOS PRÁCTICOS EN DIESEL

NUEVE.4.1. Línea de estado 1

Usuario: \$(getvar,loginname) | Código de idioma ISO: \$(getvar,locale)

NUEVE.4.2. Línea de estado 2

Último punto: \$(getvar,lastpoint) | Último ángulo: \$(if,\$(<=,\$(getvar,lastangle),0),negativo o cero,\$(getvar,lastangle))

NUEVE.4.3. Línea de estado 3

Versión de AutoCAD: \$(substr,\$(getvar,acadver),1,2) | Directorio raíz de instalación: \$(substr,\$(getvar,acadprefix),1,3)

NUEVE.4.4. Visibilidad de objetos *Proxy*

```
***POP1
[&Objetos Proxy]
[$(if,$(=,$(getvar,proxyshow),1),!.)&Todo]'proxyshow 1
[$(if,$(=,$(getvar,proxyshow),2),!.)&Borde]'proxyshow 2
[--]
[$(if,$(=,$(getvar,proxyshow),0),!.~)&Nada]'proxyshow 0
```

NUEVE.4.5. Orden de objetos

```
***POP15
[&Ordenación de objetos]
[$(if,$(=,$(getvar,sortents),1),!.,$(if,$(=,$(getvar,sortents),0),~))
Ordena para la &designación]'sortents 1
[$(if,$(=,$(getvar,sortents),2),!.,$(if,$(=,$(getvar,sortents),0),~))
Ordena para la &referencia]'sortents 2
[$(if,$(=,$(getvar,sortents),4),!.,$(if,$(=,$(getvar,sortents),0),~))
Ordena para el re&dibujado]'sortents 4
[$(if,$(=,$(getvar,sortents),8),!.,$(if,$(=,$(getvar,sortents),0),~))
Ordena para las &fotos]'sortents 8
[$(if,$(=,$(getvar,sortents),16),!.,$(if,$(=,$(getvar,sortents),0),~))
Ordena para el re&generado]'sortents 16
[$(if,$(=,$(getvar,sortents),32),!.,$(if,$(=,$(getvar,sortents),0),~))
Ordena para la &trazado]'sortents 32
[$(if,$(=,$(getvar,sortents),64),!.,$(if,$(=,$(getvar,sortents),0),~))
Ordena para la &PostScript]'sortents 64
[--]
[$(if,$(=,$(getvar,sortents),0),!.)De&sactiado]'sortents 0
[$(if,$(!=,$(getvar,sortents),0),!.)Acti&vado]'sortents 1
```

NOTAS INTERESANTES:

1. Este ejemplo no contempla todos los posibles valores de la variable SORTENTS, ya que se admite la suma de bits para la compaginación.

NUEVE.4.6. Ventanas en mosaico y flotantes

```
***POP1  
[&Ventanas]  
[Mosaic&o/Flotantes]^C^C$M=$(if,$(=,$(getvar,tilemode),0),_mview,_vports)
```

NUEVE.FIN. EJERCICIOS PROPUESTOS

- I. Añadir a la línea de estado un texto que especifique si el modo de forzado del cursor (Forzcursor) está o no activado, así como si está establecido en isométrico o no.
- II. Añadir a la línea de estado de **AutoCAD** un texto que indique la coordenada X y la coordenada Y del centro de la vista de la ventana gráfica actual. Así mismo, y por delante, se indicará que número de ventana es.
- III. Añadir a la línea de estado la fecha y hora actuales en el formato deseado por el creador.
- IV. Crear un menú desplegable que permita seleccionar las distintas modalidades de sombra. Al elegir cada una de ellas se activará por delante una marca de verificación. Además, habrá una opción que permita realizar el SOMBRA con la opción elegida.
- V. Diseñese un menú desplegable que permita ejecutar los comandos de configuración de ventanas en mosaico y múltiples o flotantes. Según nos encontremos en Espacio Modelo Mosaico o Espacio Papel/Modelo Flotante, la opción correspondiente estará activa y la otra no.
- VI. Crear un conmutador en un menú para alternar entre sombreado de doble o simple rayado para patrones de usuario.
- VII. Crear un completo conjunto de expresiones DIESEL para la línea de estado que permita informar en diversos momentos de distintas situaciones del programa. La visualización de unas u otras se controlará desde un menú cualquiera. Deberá estar personalizado para un técnico en diseño de jardines.

EJERCICIOS RESUELTOS DEL MÓDULO OCHO

NOTA: Se emplea la misma sintaxis que en los ejemplos del **MÓDULO** anterior: espacios representados por el símbolo ∪ y localización de un INTRO con el símbolo ↵.

EJERCICIO I

(...)

EJERCICIO II

(...)

EJERCICIO III

```
circle0100,100010↵  
select010↵  
line010,1250250,100↵  
matchprop0100↵
```

EJERCICIO IV

```
line0140,1720260,570↵  
ucs0001↵
```

EJERCICIO V

```
box010,10,100050↵  
vpoint0non0*1,-1,1↵
```

EJERCICIO VI

(Ejercicio completo para resolver por técnicos y/o especialistas).

MÓDULO DIEZ

Lenguaje DCL; personalización y creación de cuadros de diálogo

DIEZ.1. LENGUAJE DCL

A partir de la versión 12 de **AutoCAD** existe un lenguaje de programación de letreros de diálogo denominado DCL, el cual nos permitirá crear los cuadros necesarios para que nuestras aplicaciones AutoLISP sean más vistosas y coherentes. El lenguaje AutoLISP será estudiado en el siguiente **MÓDULO** de este curso.

Con DCL (*Dialog Control Language*, lenguaje de control de letreros de diálogo) se crean archivos de texto ASCII con extensión `.DCL` que contienen el diseño geométrico aparente completo de un cuadro de diálogo que queremos crear. No obstante, los elementos definidos en estos archivos han de controlarse desde rutinas en AutoLISP o C. Por esto, en este **MÓDULO** vamos a estudiar en profundidad cómo crear un cuadro de diálogo, e incluso lo cargaremos e inicializaremos para verlo en **AutoCAD**, pero hacerlo funcionar, no se explicará hasta el **MÓDULO ONCE**, en un apartado dedicado a tal efecto.

El aspecto de los cuadros definidos en cuanto a la forma de presentar las casillas, los recuadros, la flecha del cursor, etc., depende del dispositivo gráfico conectado. Lo que el usuario diseña mediante DCL es la distribución de los diferentes tipos de elementos o componentes dentro del cuadro.

Para simplificar el número de parámetros que definen la forma y distribución de los elementos dentro del cuadro, existen posiciones predefinidas en filas y columnas (no es necesario indicar coordenadas X e Y). Además, existe una herramienta de PDB (*Programmable Dialogue Box*) con elementos (*tiles*) predefinidos, que se pueden utilizar en la creación de cuadros personalizados.

DIEZ.2. ESTRUCTURA JERARQUIZADA DE DISEÑO

La estructura de un fichero DCL es una férrea disposición jerárquica de elementos o *tiles* que, desde un comienzo común, se van subdividiendo en otros elementos hasta llegar a la parte mínima de representación. Por ejemplo, observemos el cuadro de la página siguiente.

La manera de estructurar este cuadro sería la siguiente. Esencialmente está dividido en cuatro filas: la primera se corresponde con la de las áreas *Mallas* y *Sólidos* (de izquierda a derecha), la segunda corresponde a la casilla de verificación *Diálogo al imprimir*, la tercera a la casilla *Gestión de archivos* y la cuarta, y última, a la de los botones *Aceptar* y *Cancelar*.

A su vez, la primera fila se subdivide en dos columnas, el área *Mallas* y el área *Sólidos*. Y a su vez, cada área se vuelve a subdividir en otros elementos: el área *Mallas* en dos casillas de edición y un botón (3 elementos), y el área *Sólidos* en dos casillas de edición, una casilla de verificación y un botón (4 elementos). Las demás filas ya no se dividen más.

Pues bien, la forma de diseñar un fichero `.DCL` se basa precisamente en esa división estructurada, evidentemente recurriendo a la sintaxis propia del lenguaje. Esta sintaxis recuerda un tanto a la de programación en C o C++.



DCL se basa en la escritura de lo que se denominan *tiles*, componentes o elementos. Estos componentes —que iremos viendo poco a poco— son nombres mnemotécnicos precedidos de un carácter de dos puntos (:). Tras el *tile* se escribe una llave de apertura ({) que englobará a todos los argumentos y *tiles* incluidos que contenga el anterior, acabando con una llave de cierre (}). Dichos elementos incluidos, o los propios argumentos del elemento externo, se separan con caracteres punto y coma (;). Los comentarios se pueden indicar con una doble barra (//) al principio de la línea o en la forma */*comentario*/* para incluirlos en la mitad de una línea. Además, se suelen sangrar la líneas para mostrar claramente el carácter de anidación de elementos. Están admitidos espacios y tabuladores como separadores por claridad.

DIEZ.3. TÉCNICA DE DISEÑO

La organización básica de los diversos componentes de un cuadro es en filas y columnas. Estas se definen con los elementos `:row` y `:column`. La utilidad de programación distribuye los componentes dentro de ellas, de manera que tiendan a ocupar todo el espacio de manera simétrica y ordenada. Es por ello que deberemos distribuir la información del cuadro para que quede simétrica. Así por ejemplo, en el cuadro mostrado al principio de esta página, las dos áreas de la primera fila no contienen el mismo número de elementos, por eso no queda muy vistoso. Determinados atributos como `fixed_width` o `fixed_height` permiten restringir el espacio ocupado por los componentes.

Existen algunas reglas que debemos (podemos) seguir para el correcto diseño estético y funcional de los cuadros, que se resumen en los siguientes puntos:

- Distribuir los elementos en una disposición que facilite su empleo, de forma que resulte cómodo con el cursor desplazarse de uno a otro. Por ejemplo, para una serie de botones excluyentes es preferible disponerlos en una columna que en una fila. El cursor recorre más espacio entre botón y botón en el último caso. Respetar el orden natural de introducción de datos, como por ejemplo X, Y y Z para las coordenadas.

- Repartir los elementos de manera homogénea por la superficie del cuadro, sin apilamientos. Respetar la coherencia con los cuadros existentes en **AutoCAD**. Así, las casillas que llaman a un subcuadro tienen un título que termina en puntos suspensivos como *Mostrar...* y las que ocultan el cuadro para señalar algo en pantalla tienen un carácter < como en *Designar* <. El título de casillas de edición termina en dos puntos. También conviene que la

primera letra de los títulos se ponga en mayúsculas. No poner un punto al final de los títulos de casillas.

— Asegurar que los cuadros son reversibles, es decir, que el usuario pueda hacer pruebas sin miedo a cometer actos irreparables. Informarle en todo momento de las consecuencias de su utilización, en forma de avisos o errores en la línea inferior del cuadro (elemento `errtile`) o cuadros de advertencia en los casos graves mediante la función de AutoLISP `ALERT` (se estudiará en el siguiente **MÓDULO**). Examinar todos los datos introducidos por el usuario para detectar valores fuera de rango. Desactivar componentes cuando proceda, para que el usuario vea que no puede utilizarlos.

— Disponer de teclas rápidas para la mayoría de elementos del cuadro. Estas se indican mediante el carácter `&` (como en los menús) dentro del texto del atributo `label`. Si hay que incluir abreviaturas en los textos, que sean fácilmente comprensibles. Incluir en las casillas valores por defecto razonables.

— Tener en cuenta que el tamaño de los cuadros se mide en número de caracteres. Influye decisivamente la fuente de letra y la resolución especificada en el gestor gráfico. Una mayor resolución hace que se puedan mostrar cuadros más grandes. Pero una fuente de letra grande, hace que los cuadros ocupen más superficie en pantalla. Así, un mismo cuadro podría no caber en determinadas configuraciones gráficas. Procurar si es posible que el cuadro quepa en la resolución básica de VGA o SVGA (640 × 800 puntos).

Todo esto puede sonarnos raro o no entender algo de momento, pero poco a poco iremos repasándolo e incidiendo en ello mientras creemos nuestros propios cuadros de diálogo.

DIEZ.4. LAS HERRAMIENTAS

A partir de aquí vamos a ir explicando las características y componentes propios del lenguaje DCL, es decir la manera de llevar a la práctica todo lo expuesto hasta ahora.

La forma elegida para explicar este lenguaje de programación es, al entender del autor de este curso, la más coherente. En un principio vamos a explicar todos los *tiles* que existen para programar en DCL, después se detallarán los argumentos que podemos utilizar con dichos *tiles* y, por último, se establecerá una correspondencia entre los *tiles* y los argumentos, es decir, con cuáles de ellos podemos usar qué argumentos.

Tras todo esto, podemos empezar a volcar lo aprendido en la creación de cuadros de diálogo. Lo haremos con múltiples ejemplos. Este **MÓDULO** no está previsto para un aprendizaje de principio a fin, sino más bien para empezar estudiando los ejemplos y, mientras se estudian y comprenden las explicaciones de ellos, ir continuamente atrás a conocer la sintaxis de cada elemento y de cada argumento. Al final el aprendizaje será óptimo.

Las convenciones utilizadas para las sintaxis en este **MÓDULO** van a ser las siguientes:

- Sintaxis recuadradas para su fácil búsqueda y detección.
- Nombres de *tiles* y argumentos predefinidos en minúsculas normales.
- Texto en *itálica minúscula* indica un nombre representativo.
- Puntos suspensivos en *itálica* indican la posibilidad de indicar más argumentos.
- Una barra vertical indica doble posibilidad.

Comencemos pues.

DIEZ.4.1. Los *tiles* o elementos

Existe distintos tipos de *tiles* que vamos a ir viendo a lo largo de este **MÓDULO DIEZ**. Todos ellos se pueden clasificar en cuatro grandes grupos o categorías:

1. Grupos de componentes (*Tile Clusters*): No se puede seleccionar el grupo sino sólo los componentes individuales dentro del grupo. No pueden tener acciones asignadas, salvo los grupos de botones excluyentes. Son los siguientes:

Cuadro de diálogo en su conjunto, :dialog
Columna, :column
Columna encuadrada o enmarcada, :boxed_column
Fila, :row
Fila encuadrada o enmarcada, :boxed_row
Columna de botones de selección excluyentes, :radio_column
Columna de botones excluyentes enmarcada, :boxed_radio_column
Fila de botones de selección excluyentes, :radio_row
Fila de botones excluyentes enmarcada, :boxed_radio_row

2. Componentes individuales de acción (*Tiles*): al ser seleccionados, se ejecuta la acción asignada. Esta acción se asigna a través de la clave (atributo *key*) que identifica cada componente. Son los siguientes:

Casilla de activación o conmutador, :toggle
Botón de acción, :button
Botón excluyente, :radio_button
Casilla de edición, :edit_box
Botón de imagen o icono, :image_button
Casilla o recuadro de lista, :list_box
Lista desplegable, :popup_list
Barra de exploración o deslizador, :slider

3. Componentes decorativos e informativos: no realizan acciones ni pueden designarse. Muestran información en forma de texto o imágenes. Son:

Croquis o Imagen, :image
Texto, :text
Componente en blanco, spacer
Componente sin tamaño, spacer_0
Componente de tamaño 1, spacer_1
Concatenación de textos, :concatenation
Partes de una concatenación, :text_part
Párrafo de textos, :paragraph

4. Botones de salida y componentes de error: realizan las acciones de validar o anular el cuadro y el tratamiento de errores. Son:

Mensaje de error, errtile
Casilla de *Aceptar*, ok_only
Casillas *Aceptar* y *Cancelar*, ok_cancel
Casillas *Aceptar*, *Cancelar* y *Ayuda...*, ok_cancel_help
Las tres casillas, más mensaje de error, ok_cancel_help_errtile
Las tres casillas, más casilla información, ok_cancel_help_info

Pasamos ahora a ver los atributos y, luego, se explicará cada uno de los *tiles* y se indicará qué atributos admite.

DIEZ.4.2. Los atributos predefinidos

Los atributos de un componente de cuadro definen su aspecto y funcionalidad. Su nombre está normalizado en inglés y cada uno acepta un valor que puede ser uno de los siguientes:

- Valor entero: generalmente representa un tamaño en número de caracteres. Por ejemplo `width=18`.

- Valor real: en estos casos hay que indicar el dígito correspondiente a la unidad (no vale `.7` sino que debe ser `0.7`). Por ejemplo `aspect_ratio=0.85`.

- Cadena de texto entrecomillada: se distinguen las mayúsculas y minúsculas. No es lo mismo `key="defecto"` que `key="Defecto"`. Si la cadena contiene comillas en su interior, hay que indicarla mediante el código de control `\`. Además, se admite `\\` para indicar el carácter contrabarra, `\n` para indicar un cambio de línea y `\t` para indicar una tabulación.

- Palabra reservada: son términos normalizados en inglés, que se indican sin comillas. También se distinguen mayúsculas y minúsculas. No es lo mismo `alignment=centered` que `alignment=Centered`. Normalmente todas se indican en minúsculas, la válida es la primera.

Además de los atributos predefinidos, el usuario puede crear sus propios atributos. Todos los valores asignados a elementos del cuadro, y los obtenidos desde un cuadro tras la actuación del usuario, son siempre cadenas de texto. Por este motivo, en los programas de AutoLISP que hacen funcionar el cuadro, se recurre a las funciones de conversión desde y a cadenas de texto, como `ATOI`, `ATOF`, `ITOA`, `RTOS`, etcétera, que ya estudiaremos.

Cada tipo de elemento admite varios atributos, pero no todos. Existen atributos específicos, como por ejemplo `edit_limit` que sólo se utiliza en las casillas de edición (elemento `:edit_box`).

A continuación se verán todos los atributos disponibles con sus correspondientes explicaciones.

DIEZ.4.2.1. Atributos de título, clave y valor inicial

`label="cadena" ;`

Puede ser el título del cuadro, una fila o columna enmarcada, una casilla, una imagen, etc. El valor es un texto entre comillas. La tecla aceleradora se indica mediante `&` (el atributo `mnemonic` también sirve para especificar una tecla aceleradora). El título puede ser un espacio en blanco o una cadena nula.

`key="cadena" ;`

Permite definir la acción que se efectuará sobre o desde el elemento en el programa AutoLISP que hace funcionar el cuadro. Es un texto entre comillas. Debe especificarse en todos aquellos componentes que actúan. Debido a que distingue mayúsculas, conviene indicarlo siempre en minúsculas para que no haya problemas, aunque puede hacerse de otro modo si luego no nos equivocamos en el programa AutoLISP.

```
value="cadena";
```

Es el valor presentado al inicializar el cuadro. Se indica como un texto entre comillas. En general, se modificará durante la utilización del cuadro por parte del usuario.

```
list="cadena";
```

Líneas de texto que situar inicialmente en las listas desplegables o casillas de lista (componentes :popup_list y :list_box). Se presentan al inicializar el cuadro, aunque pueden ser modificados durante la utilización del mismo. Para separar las diferentes líneas de texto, se emplea el código de cambio de línea \n. Dentro de las líneas se puede incluir el carácter tabulador \t.

```
alignment=palabra_reservada;
```

Controla la justificación de un componente dentro de su grupo. Así, una casilla puede estar centrada en el espacio disponible de una fila, o alineada a la derecha por ejemplo. Los valores posibles son left, right o centered para la justificación horizontal y top, bottom o centered para la justificación vertical. Si no se indica, los valores por defecto son left y centered respectivamente.

```
children_alignment=palabra_reservada;
```

Controla la alineación de todos los componentes de un grupo, por ejemplo dentro de una fila o columna enmarcada, o una fila o columna de botones excluyentes. Los valores posibles son los mismos que para alignment.

```
color=valor_entero|palabra_reservada;
```

Sólo se utiliza con elementos :image e :image_button. Puede ser un valor entero (número de color de **AutoCAD**) o una de las palabras reservadas que se indican en la siguiente tabla:

Palabra reservada	Color
dialog_line	Color de línea actual de los cuadros de diálogo.
dialog_foreground	Color de primer plano del cuadro de diálogo actual (para texto).
dialog_background	Color de fondo del cuadro de diálogo actual.
graphics_background	Fondo actual de la pantalla de gráficos de AutoCAD (normalmente equivale a 0).
graphics_foreground	Primer plano actual de la pantalla de gráficos de AutoCAD (normalmente equivale a 7).
black	Color 0 de AutoCAD (negro).
red	Color 1 de AutoCAD (rojo).
yellow	Color 2 de AutoCAD (amarillo).
green	Color 3 de AutoCAD (verde).
cyan	Color 4 de AutoCAD (ciano).
blue	Color 5 de AutoCAD (azul).
magenta	Color 6 de AutoCAD (magenta).
white	Color 7 de AutoCAD (blanco).

```
is_bold=palabra_reservada;
```

Los valores posibles son `true` (texto en negrita) o `false` (texto normal, valor por defecto).

```
password_char="carácter";
```

Se utiliza para evitar que otros usuarios vean el valor introducido en la casilla (por ejemplo, en una casilla de solicitud de contraseña). En ésta aparecerá el carácter especificado en `password_char`, en lugar de los caracteres introducidos por el usuario durante la utilización del cuadro.

```
layout=palabra_reservada;
```

Los valores posibles son `horizontal` (por defecto) y `vertical`. Para `:slider`.

DIEZ.4.2.2. Atributos de tamaño

```
width=valor;
```

Es un número (puede ser real, aunque generalmente se usa entero) que especifica la anchura mínima en número de caracteres de texto. Esta anchura se ampliará automáticamente si el componente lo necesita, a no ser que se indique un atributo del tipo `fixed_...`. Si se omite, el componente utiliza la anchura mínima para que quepa.

```
height=valor;
```

Tiene el mismo significado y formato que `width`. En las casillas y botones de imagen debe especificarse.

```
fixed_width=palabra_reservada;
```

Especifica si la anchura de un componente puede rellenar el espacio disponible. Los valores son `true` y `false` (por defecto). Si se indica `true`, el componente mantiene su tamaño sin ampliarse para ocupar el espacio disponible.

```
fixed_height=palabra_reservada;
```

Especifica si la altura de un componente puede rellenar el espacio disponible. Tiene el mismo significado y opciones que `fixed_width`.

```
fixed_width_font=palabra_reservada;
```

Especifica si un cuadro de lista o lista desplegable mostrará texto con un tipo de letra de caudal fijo, es decir, la separación entre caracteres es siempre la misma (por ejemplo la "i" y la "m" ocupan el mismo espacio). Se utiliza para facilitar la alineación de las columnas con tabulaciones. Los valores son `true` y `false` (por defecto).

```
edit_width=valor;
```

Es la anchura en caracteres de la parte visible de edición o introducción de datos para las casillas de edición. Controla los caracteres que se muestran en la ventana de la casilla, aunque el usuario podría introducir más. Si no se indica, la casilla ocupa todo el espacio. En

caso contrario, la casilla se alinea por la derecha. Puede ser un número real, aunque generalmente sea un entero.

```
children_fixed_width=palabra_reservada;
```

Especifica si todos los componentes de un grupo pueden ocupar toda la anchura disponible. Si algún componente tiene atribuida una anchura específica mediante `width` ésta se respeta. Los valores posibles son `true` y `false` (por defecto). En principio, conviene no utilizar este atributo.

```
children_fixed_height=palabra_reservada;
```

Especifica si todos los componentes de un grupo pueden ocupar toda la altura disponible. Tiene el mismo significado y opciones que `children_fixed_width`.

```
aspect_ratio=valor_real;
```

Es el cociente entre la anchura y la altura de la imagen en una casilla o botón de imagen. Si se indica 0, el componente toma el tamaño de la imagen.

DIEZ.4.2.3. Atributos de limitaciones de uso

```
edit_limit=valor_entero;
```

Es el número máximo de caracteres que el usuario puede introducir en una casilla de edición. Se indica un número entero entre 1 y 256. Por defecto es un valor 132. El número de caracteres visibles en la casilla (controlado por `edit_width`) será generalmente menor. Si el usuario introduce más caracteres, éstos se irán desplazando en la ventana visible de la casilla, aceptándose todos hasta llegar al máximo establecido por `edit_limit`.

```
min_value=valor_entero;
```

Es el valor mínimo de un deslizador en su extremo o tope inicial. Se indica un número entero con signo. Valor por defecto = 0. El mínimo posible es -32768.

```
max_value=valor_entero;
```

Es el valor máximo de un deslizador en su extremo o tope final. Se indica un número entero con signo. El valor por defecto es 10000. El máximo posible es 32767. El formato es el mismo que `min_value`.

```
small_increment=valor_entero;
```

Es el valor incremental mínimo para el desplazamiento del cursor deslizando. Se obtiene pulsando las flechas en los extremos del deslizador. Por defecto es una centésima del rango total. Así, un deslizador cuyos valores permitidos son entre 0 y 100, ofrecerá un incremento mínimo de 1 en un principio.

```
big_increment=valor_entero;
```

Es el valor incremental mayor para el desplazamiento del cursor deslizando. Se obtiene pulsando a un lado y otro del cursor deslizando, sobre la barra del deslizador. El valor por

defecto es un décimo del rango total. Un deslizador cuyos valores son entre 0 y 100, ofrecerá un incremento de 10 por defecto. El formato es el mismo que `small_increment`.

DIEZ.4.2.4. Atributos de funcionalidad

```
action="(expresión_AutoLISP)";
```

Expresión de AutoLISP que ejecuta la acción que se efectuará cuando se pulsa el componente. También se denomina "retorno de llamada". Debe ir entre comillas. Si desde el programa en AutoLISP se utiliza `ACTION_TILE` (se verá en el próximo **MÓDULO**) ésta tiene preferencia sobre la acción especificada en `action`.

```
mnemonic="carácter";
```

Carácter de tecla aceleradora para el componente. Debe ser una cadena de un sólo carácter entre comillas. El carácter debe ser uno de los que forman el título (atributo `label`). Si el usuario pulsa la tecla con ese carácter (da igual mayúsculas o minúsculas), se activa el componente pero no se selecciona. Resulta más sencillo indicar las teclas aceleradoras mediante `&` (al igual que en los menús hacíamos) en el título especificado en `label`.

```
initial_focus="cadena";
```

Identificación clave o *key* (argumento `key`) del componente del cuadro que se presenta iluminado al entrar en el cuadro y que por lo tanto recibe la pulsación inicial del teclado.

```
allow_accept=palabra_reservada;
```

El componente se activa al pulsar la tecla de aceptación (normalmente `INTRO`). Los valores posibles son `true` y `false` (por defecto). Al aceptar un cuadro, se considera pulsado también el botón por defecto (aquel que tiene definido `is_default` como `true`).

```
is_default=palabra_reservada;
```

El componente se selecciona al pulsar `INTRO`. Los valores posibles son `true` y `false` (por defecto). Sólo un componente del cuadro puede tener este atributo como `true` (por defecto, es la casilla de validación *Aceptar*).

```
is_cancel=palabra_reservada;
```

El componente se selecciona al cancelar con `ESC`. Los valores posibles son `true` y `false` (por defecto). Sólo un componente del cuadro puede tener este atributo como `true` (por defecto es la casilla de cancelación *Cancelar*). El formato es el mismo que el de `is_default`.

```
is_enabled=palabra_reservada;
```

El componente aparece habilitado o no inicialmente. Los valores posibles son `true` (por defecto) y `false`. Si se define como `false`, el componente se muestra atenuado en gris, indicando que se encuentra inhabilitado.

```
is_tab_stop=palabra_reservada;
```

El componente se activa al desplazarse el usuario por el cuadro mediante el tabulador. Los valores posibles son `true` (por defecto) y `false`. Si se define como `false`, el componente no se activa con la tecla de tabulado y ésta lo pasa por alto.

```
tabs ="cadena";
```

Posiciones de tabulación en número de caracteres. Se utiliza en las casillas de lista y listas desplegables para alinear verticalmente columnas de texto (por ejemplo en el cuadro de control de capas, la casilla de lista con los nombres, estados, colores y tipos de línea). Se indica una cadena de texto con las posiciones de tabulación.

```
tab_truncate=palabra_reservada;
```

Los valores posibles son `true` y `false` (por defecto). Si se define como `true`, el texto de un cuadro de lista o lista desplegable se truncará al rebasar la posición de una tabulación existente.

```
multiple_select=palabra_reservada;
```

Establece la posibilidad de seleccionar más de un elemento en una lista de casilla o lista desplegable. Los valores posibles son `true` y `false` (por defecto). Si se indica `true` se permite iluminar o seleccionar varios elementos. En caso contrario sólo uno.

DIEZ.4.3. Los *tiles* y sus atributos

DIEZ.4.3.1. Grupos de componentes

```
:dialog {atributos...}
```

Es el componente que engloba a todos los demás y define el cuadro de diálogo. Los atributos permitidos son:

<code>label</code>	Título en el borde superior del letrero.
<code>value</code>	Título opcional; invalida al anterior.
<code>initial_focus</code>	Clave (atributo <code>key</code>) del elemento que se activa inicialmente al desplegarse el cuadro en pantalla.

```
:column {atributos...}
```

Agrupar en una columna a todos los componentes incluidos en su definición. Los atributos permitidos son:

<code>label</code>	Título en el borde superior de la columna.
<code>width</code>	Anchura mínima.
<code>height</code>	Altura mínima.
<code>fixed_width</code>	Anchura fija.
<code>fixed_height</code>	Altura fija.
<code>children_fixed_width</code>	Anchura fija para todos los componentes.
<code>children_fixed_height</code>	Altura fija para todos los componentes.
<code>alignment</code>	Alineación si está incluida en otro elemento.
<code>children_alignment</code>	Alineación para todos los componentes.

```
:boxed_column {atributos...}
```

Agrupar a todos los componentes incluidos en su definición en una columna enmarcada o encuadrada mediante un rectángulo. Los atributos permitidos son los mismos que los de :column. label indica el título en el borde superior del marco. Si se indica como título un espacio en blanco, label = " ", se deja un espacio encima de la columna. Si se indica cadena nula, label = "", no existe ese espacio.

```
:row {atributos...}
```

Agrupar en una fila a todos los componentes incluidos en su definición. Los atributos posibles y su funcionalidad son los mismos que para :column.

```
:boxed_row {atributos...}
```

Agrupar a todos los componentes incluidos en su definición en una fila enmarcada o encuadrada mediante un rectángulo. Los atributos y la consideración para label, igual que en :boxed_column.

```
:radio_column {atributos...}
```

Agrupar en una columna a una serie de botones de acción excluyentes. Cada botón será un componente :radio_button. Sólo uno de los botones puede encontrarse activado cada vez.

key	Clave de acción para la columna. Se puede indicar una clave, al contrario que en :column o :boxed_column.
value	Valor de clave (atributo key) del botón inicialmente activado.

```
:boxed_radio_column {atributos...}
```

Agrupar a una serie de botones de acción excluyentes en una columna enmarcada dentro de un rectángulo. Su funcionamiento es similar al de :radio_column.

label	Título en el borde superior del marco. Las mismas posibilidades que en :boxed_column.
key	Clave de acción para la columna. Se puede indicar una clave, al contrario que en :column o :boxed_column.
value	Valor de clave (atributo key) del botón inicialmente activado.

```
:radio_row {atributos...}
```

Agrupar en una fila a una serie de botones de acción excluyentes. Cada botón será un elemento :radio_button. Su funcionamiento y atributos son los de :radio_column.

```
:boxed_radio_row {atributos...}
```

Agrupar a una serie de botones de acción excluyentes, en una fila enmarcada dentro de un rectángulo. Cada botón será un elemento :radio_button. Su funcionamiento y atributos son los de :boxed_radio_column.

DIEZ.4.3.2. Componentes individuales de acción

:toggle {atributos...}

Casilla de activación o conmutador que puede valer 0 ó 1. Si se encuentra activada, se muestra con una señal en su interior en forma de x o ✓.

label	Título a la derecha de la casilla.
value	Estado inicial de la casilla. Por defecto es 0, desactivada. Si es 1 la casilla está activada.
key	Clave de acción asignada a la casilla.
action	Acción asignada por defecto a la casilla.
width	Anchura mínima.
height	Altura mínima.
fixed_width	Anchura fija.
fixed_height	Altura fija.
alignment	Alineación de la casilla, respecto al elemento en el que está incluida.
is_enabled	Estado de habilitación.
is_tab_stop	Tabulador tiene o no en cuenta esta casilla.

:button {atributos...}

Botón con un texto en su interior que efectúa una acción al ser pulsado. Los atributos posibles y su funcionalidad son los mismos que para :toggle añadiéndose los siguientes y con su característica propia en label:

mnemonic	Carácter de tecla mnemotécnica para seleccionar.
is_default	Botón seleccionado o no al pulsar INTRO.
is_cancel	Botón seleccionado o no al pulsar ESC.
label	Texto que aparece dentro del botón.

:radio_button {atributos...}

Botón componente dentro de una fila o columna de botones excluyentes. Su valor puede ser 0 ó 1. Como sólo uno de ellos puede estar activado en cada grupo del tipo :...radio... se activará el último al que se haya asignado valor 1. Atributos los de :toggle, con las siguientes puntualizaciones y excepciones:

label	Título a la derecha del botón.
value	Estado inicial del botón. Por defecto es 0, desactivado. Si es 1 el botón está activado.
mnemonic	Carácter de tecla mnemotécnica.

:edit_box {atributos...}

Casilla para que el usuario introduzca datos, con un título a su izquierda, y una ventana donde el usuario escribe valores o modifica los existentes. El título se justifica por la derecha. Atributos los de :toggle más las siguientes puntualizaciones y añadidos:

label	Título a la izquierda de la casilla.
value	Contenido inicial de la casilla. Aparece justificado a la izquierda.

mnemonic	Carácter mnemotécnico.
allow_accept	La casilla se activa al pulsar INTRO.
edit_width	Anchura en caracteres de la ventana de edición.
edit_limit	Número máximo de caracteres que el usuario puede introducir.

:image_button {atributos...}

Presenta una imagen o icono en lugar de texto. Al ser designado, se devuelven las coordenadas del punto de designación. Esto resulta útil para asignar diferentes acciones a zonas de la imagen. Por ejemplo, este mecanismo se utiliza en el cuadro del comando de **AutoCAD** DDVPOINT. El tamaño se especifica mediante una anchura y altura, o mediante uno de ambos atributos y `aspect_ratio`. Atributos los de `:toggle` más los siguientes:

mnemonic	Carácter mnemotécnico.
allow_accept	Activado o no a pulsar INTRO.
aspect_ratio	Relación entre altura y anchura de la imagen.
color	Color de fondo de la imagen.

:list_box {atributos...}

Casilla que contiene una lista de términos organizados en filas. Si hay más filas de las que caben en el tamaño de la casilla, se despliega un cursor deslizante para moverse por la lista. El atributo `value` contiene los números de orden de todos los términos inicialmente iluminados o seleccionados en la lista. Sus atributos son los de `:toggle`, teniendo en cuenta las siguientes consideraciones:

label	Título encima de la casilla de lista.
value	Cadena entrecomillada que contiene cero ("") o más números enteros separados por espacios. Cada número es un índice (se empieza a contar desde 0) que designa el elemento de la lista que aparece seleccionado inicialmente. Si <code>multiple_select</code> es false, sólo un elemento puede estar seleccionado.
mnemonic	Carácter de tecla mnemotécnica.
allow_accept	Activada o no al pulsar INTRO.
list	Lista de términos inicialmente incluidos en la casilla.
tabs	Posiciones de tabulación para presentar los términos en columnas.

:popup_list {atributos...}

Casilla con una flecha a su derecha que, al señalarse despliega una lista. Sólo uno de los términos de la lista puede estar designado. Sus atributos y funcionalidad son los mismos que para `:list_box`, exceptuando `allow_accept` y `multiple_select`, y añadiendo `edit_width`:

label	Título a la izquierda de la casilla.
value	Cadena entrecomillada que contiene un número entero (por defecto 0) con el índice del elemento de la lista que aparece seleccionado inicialmente en la ventana de la casilla, cuando no está desplegada la lista.
edit_width	Anchura en caracteres de la ventana de lista.

```
:slider {atributos...}
```

Barra deslizante, con un cursor que se desplaza y flechas en sus extremos. Los atributos son los mismos que para `:toggle`, con las siguientes consideraciones y añadidos:

<code>value</code>	Cadena entrecomillada que contiene un número entero con el valor inicial del cursor deslizante. Por defecto es el establecido en <code>min_value</code> .
<code>layout</code>	Orientación horizontal o vertical de la barra.
<code>min_value</code>	Valor mínimo en el extremo inicial.
<code>max_value</code>	Valor máximo en el extremo final.
<code>small_increment</code>	Incremento mínimo al pulsar las flechas de los extremos.
<code>big_increment</code>	Incremento mayor al señalar la barra a ambos lados del cursor.

DIEZ.4.3.3. Componentes decorativos e informativos

```
:image {atributos...}
```

Rectángulo en el que se presenta una imagen vectorial (debe ser un archivo `.SLD` de *foto* de **AutoCAD**). Se debe especificar una anchura y altura, o bien uno de ambos atributos y `aspect_ratio`. Sus atributos son los mismos que para `:image_button` excepto `allow_accept`.

```
:text {atributos...}
```

Componente que muestra una cadena de texto. Para los textos fijos, se especifica un atributo `label` con su contenido. Para los textos variables, se especifica una clave de acción mediante el atributo `key`. Los atributos posibles son los siguientes:

<code>label</code>	Texto fijo.
<code>key</code>	Clave para texto variable.
<code>value</code>	Estado inicial.
<code>height</code>	Altura mínima.
<code>width</code>	Anchura mínima.
<code>fixed_height</code>	Altura fija.
<code>fixed_width</code>	Anchura fija.
<code>alignment</code>	Alineación respecto a donde está incluido.
<code>is_bold</code>	Si es <code>true</code> muestra el texto en negrita.

```
:concatenation { :text_part... }
```

Línea de texto formada por varios componentes `:text_part` sucesivos. Se utiliza para presentar un mensaje que consta de varias partes que pueden variar independientemente entre sí en tiempo real. No tiene atributos.

```
:text_part {atributos...}
```

Componente de texto que forma parte de `:concatenation`. Los atributos que puede incluir son `label`, con el contenido del texto, y `key`.

```
:paragraph { :text_part... | :concatenation... }
```

Conjunto de componentes `:text_part` o `:concatenation` dispuestos en vertical. Se utiliza para construir párrafos. No tiene atributos.

```
:spacer { atributos... }
```

Componente en blanco que se utiliza para mejorar el aspecto cuando la distribución homogénea y simétrica de los elementos de un cuadro no ofrece una estética adecuada. El tamaño se especifica mediante los atributos `height`, `width`, `fixed_height`, `fixed_width` y `alignment`.

```
spacer_0;
```

Se trata de un componente predefinido (por eso se indica como los atributos, sin llaves, sin `:` y con `;` al final) sin tamaño concreto, que se inserta para situar un espacio en una zona determinada y ensanchar el resto de componentes.

```
spacer_1;
```

Componente predefinido de tamaño unitario, es decir, el mínimo visible para el usuario. Se utiliza por los mismos motivos que `spacer_0`.

DIEZ.4.3.4. Botones de salida y componentes de error

```
errtile;
```

Componente predefinido (se indica como los atributos, sin llaves, sin `:` y con `;` al final) de error que aparece como una línea de texto en la parte inferior del cuadro. Tiene asignada la clave (atributo `key`) `error`.

```
ok_only;
```

Componente predefinido que contiene una `:row` con el botón de validación *Aceptar*. Tiene asignada la clave `accept`.

```
ok_cancel;
```

Componente predefinido que contiene una `:row` con los botones *Aceptar* y *Cancelar*. Tienen asignadas las claves `accept` y `cancel`.

```
ok_cancel_help;
```

Componente predefinido que contiene una `:row` con los botones *Aceptar*, *Cancelar* y *Ayuda*.... Tienen asignadas las claves `accept`, `cancel` y `help`.

```
ok_cancel_help_errtile;
```

Componente predefinido que contiene dos :row, una con los botones *Aceptar*, *Cancelar* y *Ayuda...* y otra con el componente de error. Tienen asignadas las claves *accept*, *cancel*, *help* y *error*. Es la combinación más habitual en la mayoría de cuadros de diálogo.

```
ok_cancel_help_info;
```

Componente predefinido que contiene una :row con los botones *Aceptar*, *Cancelar*, *Ayuda...* e *Info....* Tienen asignadas las claves *accept*, *cancel*, *help* e *info*.

NOTA: Uno de estos elementos *ok_...* habrá de existir siempre en un cuadro de diálogo para que éste funcione correctamente. La no presencia de uno de ellos, producirá un error al cargarse el letrero.

DIEZ.4.4. Elementos predefinidos

Para mayor facilidad existen, como acabamos de ver, unos aspectos y tamaños predefinidos de cuadros, que **AutoCAD** asume por defecto, y que están almacenados en dos archivos: *BASE.DCL* y *ACAD.DCL* (en el directorio *\SUPPORT*).

Los elementos o componentes predefinidos se indican como si fueran un atributo más. Por ejemplo, el elemento predefinido para incluir la casilla de validación *Aceptar*. Este elemento se llama *ok_button* y en la versión española de **AutoCAD** se ha cambiado el texto que visualiza por *Aceptar*, en lugar de *OK*. Su definición en el archivo *BASE.DCL* llama a otro elemento predefinido en el mismo archivo, como se muestra a continuación:

```
retirement_button : button {  
    fixed_width = true;  
    width = 8;  
    alignment = centered;  
}  
ok_button : retirement_button {  
    label = "Aceptar";  
    key = "accept";  
    is_default = true;  
}
```

En primer lugar, el elemento *:retirement_button* define una casilla (instrucción DCL *:button*), con el atributo *fixed_width = true* para que la casilla no se extienda todo el espacio disponible en el cuadro sino sólo el tamaño del texto *Aceptar*, con el atributo *width = 8* que es la longitud de la casilla, y con el atributo *alignment = centered* para que la casilla ocupe el centro de la fila del cuadro en que se utiliza. El atributo *key* asocia una clave de acción al elemento. Desde el programa en AutoLISP que controla el cuadro, el valor *accept* permitirá especificar una acción para efectuar al salir del cuadro. Ese valor predefinido del atributo *key* hace referencia siempre a las casillas de validación de todos los cuadros.

A continuación, el elemento *ok_button* llama a *:retirement_button* con sus atributos y añade tres nuevos atributos: *label = "Aceptar"* que es el texto en el interior de la casilla, *key = "accept"* que es el texto asociado a la aplicación, e *is_default = true* para que acepte la casilla al terminar el cuadro con *INTRO*.

De forma similar, el elemento *ok_cancel* se definiría:

```
ok_cancel:retirement_button {  
  label = "Cancelar";  
  key    = "cancel";  
  is_cancel = true;  
}
```

La casilla `ok_button` existe en un componente predefinido llamado `ok_only`. Ambas casillas juntas, existen en un grupo predefinido en fila, con el nombre `ok_cancel`. Añadiendo una tercera casilla `Ayuda...`, existe también como grupo predefinido con el nombre `ok_cancel_help`. De esta forma, a base de módulos incluidos unos en otros, se pueden utilizar en los cuadros elementos comunes predefinidos.

DIEZ.5. PROGRAMANDO CUADROS DCL

Una vez llegado a este punto, y tras ver todos los *tiles* y argumentos que podemos utilizar, vamos a llevar todo a la práctica comenzando a programar nuestros propios letreros de diálogo. Como ya hemos comentado, estos letrero por ahora no funcionarán, porque hay que controlarlos desde AutoLISP, tema que se explicará en el **MÓDULO ONCE**. Pero podremos ver como quedan en pantalla y, una vez dominado el proceso de diseño, podremos subir un escalón más y hacer que funcionen, como decimos, desde un programa en AutoLISP.

El primer ejemplo que vamos a estudiar es el más sencillo que podemos encontrar. Se trata de un cuadro de diálogo que simplemente muestre un texto dentro de él y un botón *Aceptar* para cerrarlo.

DIEZ.5.1. Ejemplo sencillo: letrero informativo

Lo que queremos conseguir es el cuadro de diálogo de la figura siguiente:



Lo primero que vamos a ver es un esquema jerárquico de la división del cuadro en elementos. Este esquema conviene hacerlo siempre antes de diseñar un cuadro, sobre todo cuando es complejo, para ir guiándonos a la hora de escribir el código en DCL. A continuación examinaremos dicho código perteneciente a este cuadro, acompañado de una explicación que se proporciona.

Análisis jerárquico

```
:dialog  
|__ :row  
|   |__ :column  
|   |   |__ :text  
|   |   |__ ok_only
```

Código

```
prueba:dialog {label="Prueba en DCL";  
  :row {
```

```
:column {  
  :text {label="Esto es una prueba";alignment=centered;}  
  ok_only;  
}  
}
```

Explicación

Lo primero que se hace es comenzar el cuadro con el *tile* :dialog, el cual ha de ir siempre precedido del nombre identificador que le queremos dar al cuadro. Este nombre será el que luego se maneje desde AutoLISP. Un carácter { abre dicho *tile* para introducir todo el resto del programa. :dialog siempre habrá de estar al principio de un programa en DCL y lo contendrá por completo. El atributo label para :dialog da un nombre al letreto de diálogo en su parte superior (barra de título); es la zona de color generalmente azul (navy). Al final, siempre un carácter ; que separa argumentos.

A continuación, y sin indicar otros atributos para :dialog porque no nos interesa, definimos una fila con :row. Dentro de ella se encuentra una columna definida con :column. Dentro de esta columna, un texto centrado en ella (alignment=centered) que *dice Esto es una prueba* (label). Entre argumentos seguimos separando con ;, incluso al final, ya que lo que venga después, en el fondo, son argumentos del *tile* que engloba a todo, de :dialog.

Dentro de la misma columna insertamos también un botón *Aceptar*, el cual, al ser un elemento predefinido no comienza con :, y se centra automáticamente. Al final cerramos todas la llaves abiertas, la de :column, la de :row y la de :dialog.

¿Por qué hemos definido los elementos dentro de una columna, que se encuentra a su vez dentro de una fila, y no dentro de dos filas separadas? La razón es simple. Si el código de este programa hubiera sido así:

```
prueba:dialog {label="Prueba en DCL";  
  :row {  
    :text {label="Esto es una prueba";alignment=centered;}  
  }  
  :row {  
    ok_only;  
  }  
}
```

tanto el texto como el botón *Aceptar* se situarían donde se encuentran en el ejemplo anterior, es decir, uno debajo del otro, pero alineados ambos a la izquierda. ¿Por qué? Sencillamente porque se encuentran situados en dos filas y, al indicar que queden centrados (alignment para el texto y por defecto en ok_only), se centrarán con respecto a la fila, es decir con una justificación vertical centrada.

Si se centra un elemento con respecto a una columna, se centrará dejando espacio a su izquierda y a su derecha (se coloca en el centro de la columna). Por el contrario, si se centra un elemento con respecto a una fila, se centrará dejando espacio arriba y abajo (se coloca en el centro de la fila). El resultado del programa anterior es el siguiente:



Por otro lado, los elementos `ok_...` se centran también si no se encuentran dentro de ninguna fila ni columna. Así, el ejemplo válido (no este anterior) también podía haberse conseguido así:

```
prueba:dialog {label="Prueba en DCL";
  :row {
    :column {
      :text {label="Esto es una prueba";alignment=centered;}
    }
  }
  ok_only;
}
```

NOTA: Para que un cuadro funcione, y como hemos dicho ya anteriormente, siempre debe existir alguno de los elementos `ok_...`.

Hablando ahora de la disposición del código en el archivo ASCII, decir que lo más lógico es indicarlo como se muestra en estos ejemplos. Los *tiles* que se encuentren dentro de otros suelen separarse de los anteriores por un `INTRO`, y cuando un *tile* supone el fin de una ramificación del árbol jerárquico, se coloca entero en una línea (con sus atributos y llave de fin). Además, se pueden indicar tabuladores y espaciados para sangrar líneas (como se ve) y proporcionar claridad a la jerarquía. También puede haber espacio de claridad en cualquier otro punto: después y antes de los caracteres `=`, entre argumentos y etcétera.

Por último, decir que el `;` tras `ok_only` es necesario, aunque después no aparezcan más argumentos.

DIEZ.5.1.1. Cómo cargar y visualizar el cuadro

En el **MÓDULO** siguiente trataremos todo lo referente a la interacción de AutoLISP con los cuadros de diálogo en DCL. Sin embargo, y para que podamos ver nuestro nuevo cuadro en pantalla, vamos a explicar muy por encima los pasos que debemos seguir para cargar un cuadro en **AutoCAD**.

1º. Guardar nuestro archivo de definición con extensión `.DCL`, por ejemplo, `PRUEBA.DCL`, y en un directorio de un disco, por ejemplo en `C:\DCL\` del disco duro.

2º. En la línea de comandos de **AutoCAD** escribir la siguiente línea de AutoLISP y pulsar `INTRO` (hay que escribirla como se indica: con los paréntesis, espacios, comillas, barras inclinadas normales y demás):

```
(load_dialog "c:/dcl/prueba.dcl")
```

Evidentemente, la ruta de acceso y el archivo corresponderán a nuestro nombre y ubicación propios.

Esto devolverá un índice de carga (un número en la línea de comandos). Por ejemplo 64. La primera vez que se cargue un cuadro será 1, luego 2, 3, 4...

3º. Escribir esta otra línea ahora:

```
(new_dialog "prueba" 64)(start_dialog)
```

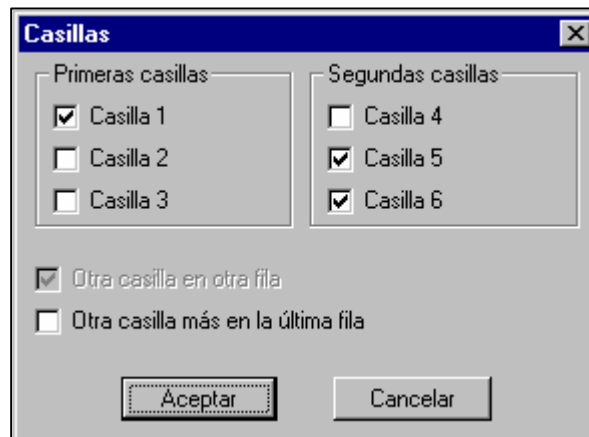
Lo que ahora aquí es `"prueba"` se corresponde con el nombre indicado inmediatamente antes del `tile :dialog`, no con el nombre de archivo. De todas formas, se recomienda hacer

coincidir estos dos nombres para evitar confusiones. Lo que aquí es 64 es el índice devuelto en la operación anterior realizada. Debemos indicar el que nos haya devuelto AutoLISP a nosotros. Entre el primer paréntesis de cierre y el segundo de apertura no hay espacio.

Tras esto el cuadro se muestra en pantalla y responde.

DIEZ.5.2. Ejemplo con casillas de verificación

En el siguiente ejemplo vamos a definir un cuadro de diálogo como el siguiente:



Análisis jerárquico

```
:dialog
|__ :row
|  |__ :boxed_column
|  |  |__ :toggle
|  |  |__ :toggle
|  |  |__ :toggle
|  |__ :boxed_column
|  |  |__ :toggle
|  |  |__ :toggle
|  |  |__ :toggle
|__ spacer_1
|__ :row
|  |__ :toggle
|__ :row
|  |__ :toggle
|__ spacer_1
|__ ok_cancel
```

Código

```
// Ejemplo con casillas de verificación

verif:dialog {label="Casillas";
  :row {
    :boxed_column {label="Primeras casillas";
```

```
:toggle {label="Casilla 1";value=1;}
:toggle {label="Casilla 2";}
:toggle {label="Casilla 3";}
}
:boxed_column {label="Segundas casillas";
:toggle {label="Casilla 4";}
:toggle {label="Casilla 5";value=1;is_tab_stop=false;}
:toggle {label="Casilla 6";value=1;is_tab_stop=false;}
}
}
spacer_1;
:row {:toggle {label="Otra casilla en otra fila";value=1;is_enabled=false;}}
:row {:toggle {label="Otra casilla más en la última fila";}}
spacer_1;
ok_cancel;
}

// Fin del ejemplo
```

Explicación

Como podemos ver este ejemplo ya está un poco más elaborado. Tras el comienzo de rigor, podemos ver en la estructura jerárquica proporcionada que este letrero de diálogo se divide principalmente en tres filas. La primera fila está a su vez dividida en dos columnas, que en este caso son enmarcadas (:boxed_column), cada una con su título superior y con tres casillas de verificación dentro.

Las otras dos filas no se subdividen en columnas y únicamente tienen un elemento :toggle cada una. Al final existe un *tile* predefinido ok_cancel, que muestra los botones de *Aceptar* y *Cancelar*.

Apreciamos algo nuevo en este cuadro con respecto al anterior; es la introducción de dos elementos predefinidos spacer_1, los cuales introducen una línea (altura de 1 carácter) para separar las dos casilla inferiores tanto de las :boxed_column (por arriba) como de los botones de ok_cancel (por abajo). Como ya hemos comentado anteriormente, los cuadros de diálogo se “autocomponen” en el espacio que ocupan, es decir, el hecho de que indiquemos dos líneas en blanco con spacer_1 no quiere decir que el cuadro vaya a resultar más grande, sino que los elementos incluidos en él se apelotonarán entre sí para dejar dicho espacio libre.

La diferencia que existe entre spacer_1 y spacer_0, es que éste último no tiene un tamaño concreto —como spacer_1 de tamaño un carácter— sino que se inserta para situar un espacio en una zona determinada y ensanchar el resto de los componentes.

NOTA: Pruébese a definir el cuadro sin los *tiles* spacer_1.

Vemos también en el ejemplo, que algunas casillas aparecen por defecto señaladas. Esto se consigue haciendo el atributo value igual a 1. value hace que al iniciar el cuadro los elementos tengan el valor de la cadena indicada por defecto. Un elemento :toggle no contiene cadenas, ya que es una especie de conmutador, y todos estos conmutadores tienen dos estados: señalado o no señalado, valor 1 y valor 0 respectivamente.

De todas formas, en ejemplos como el que veremos a continuación, el cual contiene conmutadores para variables de **AutoCAD**, lo más lógico sería que el cuadro se inicializara mostrando por defecto los valores actuales en el sistema. Esto sólo se consigue desde AutoLISP y será tratado en el próximo **MÓDULO**.

Lo siguiente que podemos apreciar a simple vista es la no disponibilidad de la penúltima de las casillas, la que dice *Otra casilla en otra fila*. Esto es debido al atributo `is_enabled=false`. Este atributo permite habilitar o inhabilitar al inicio del cuadro los diversos componentes que lo admiten. Nótese que, aunque esté inhabilitado, la casilla está activada (`value=1`), esto es perfectamente factible.

En la práctica, lo más lógico será inhabilitar o habilitar los elementos según qué condiciones del sistema desde AutoLISP.

Por último existe otra característica en la que quizá no hemos reparado al observar el cuadro pero que está ahí. Sólo debemos pulsar repetidamente la tecla `TAB` para recorrer cada uno de los elementos del letrero (como es típico en cuadros tipo Windows) para darnos cuenta de que tanto *Casilla 5* como *Casilla 6* las pasa por alto. El atributo que define esta característica es `is_tab_stop` cuando es igual a `false`. Por defecto es `true` en todos los *tiles*; solamente es `false` en los que se indica expresamente —como aquí— o en los que están inhabilitados con `is_enabled=false`, que acabamos de comentar.

DIEZ.5.3. Letrero de control de variables de AutoCAD

Veamos ahora un pequeño ejemplo de cuadro en el que se combinan varios elementos. Se corresponde con el primer ejemplo que aparece en este **MÓDULO** (figura primera), el de control de variables de **AutoCAD** denominado *Variables* y que se encuentra en la página segunda de este **MÓDULO DIEZ**.

Existen numerosas variables de sistema de **AutoCAD** que, o bien no se encuentra definido ningún acceso directo a ellas en ningún cuadro de diálogo, o bien nos los encontramos dichos accesos desperdigados por entre los menús o botones de las barras de herramientas del programa. El cuadro que estudiaremos a continuación recoge una serie de estas variables, proporcionando un buen y rápido acceso a las mismas.

Análisis jerárquico

```
:dialog
|__ :row
|  |__ :boxed_column
|  |  |__ :edit_box
|  |  |__ :edit_box
|  |  |__ spacer_1
|  |  |__ :button
|  |__ :boxed_column
|  |  |__ :edit_box
|  |  |__ :edit_box
|  |  |__ spacer_1
|  |  |__ :toggle
|  |  |__ :button
|__ :row
|  |__ :toggle
|__ :row
|  |__ :toggle
|__ :row
|  |__ ok_cancel
```

Código

//Archivo .DCL de control de variables de AutoCAD.

```
variables:dialog {label="Variables";
:row {
:boxed_column {label="Mallas";
:edit_box {label="SURFTAB&1";edit_width=3;edit_limit=3;key="Surf1";}
:edit_box {label="SURFTAB&2";edit_width=3;edit_limit=3;key="Surf2";}
spacer_1;
:button {label="De&fecto";fixed_width=true;alignment=centered;key="Def1";}
}
:boxed_column {label="Sólidos";
:edit_box {label="Isolíneas";edit_width=2;edit_limit=2;key="Iso";
mnemonic="s";}
:edit_box {label="Suavizado";edit_width=4;edit_limit=8;key="Suav";
mnemonic="v";}
spacer_1;
:toggle {label="Si&lueta";key="Sil";}
:button {label="Defe&cto";fixed_width=true;alignment=centered;key="Def2";}
}
}
spacer_1;
:row {:toggle {label="&Diálogo al imprimir";key="Dia";}}
:row {:toggle {label="&Gestión de archivos";key="Ges";}}
spacer_1;
:row {ok_cancel_help;}
}
```

Explicación

Descubrimos aquí otro nuevo *tile*, :edit_box. :edit_box define una casilla de edición en la que el usuario introducirá datos, en este caso valores a las variables en cuestión. A estas casillas, entre otros, casi siempre les acompañan dos argumentos muy típicos para ellas, los cuales son edit_with y edit_limit. edit_with controla el tamaño físico de la casilla, o sea, el tamaño en caracteres que nosotros veremos en el letrero de diálogo. Pero, como sabemos, a veces el tamaño físico aparente poco tiene que ver con el número de caracteres que podemos introducir en una casilla.

Muchas veces, como decimos, una casilla tiene una longitud aparentemente pequeña y, sin embargo, debemos introducir en ella una cadena de caracteres extensa. Lo que ocurre es que, al llegar escribiendo al final aparente de la casilla, se produce un *scroll* o desplazamiento hacia la izquierda y, mientras se van escondiendo los primeros caracteres introducidos, podemos seguir escribiendo por la derecha hasta un tope. Este tope real de escritura es el que se controla mediante edit_limit, argumento también indicado en las casillas editables del ejemplo.

Un nuevo argumento que podemos observar es key. key define una palabra clave para cada elemento del cuadro que luego será utilizada desde AutoLISP para controlar dicho elemento. Todos los componentes que interactúan en un letrero de diálogo deben poseer un argumento key. Así, y después en AutoLISP, podremos indicar que "tal elemento con determinada key realice tal acción" al determinarse unas características concretas en la sesión actual de dibujo.

NOTA: Todo esto se estudiará en el **MÓDULO** siguiente, por ahora que quede afianzada la idea.

Con respecto a la clave introducida en el atributo key, decir que se realiza distinción entre mayúsculas y minúsculas. Por ejemplo, si le damos una clave "Suav" a un *tile* y,

después, desde AutoLISP indicamos "suav", el elemento no será reconocido y no funcionará correctamente.

Examinemos ahora los elementos `:button` introducidos. Estos elemento definen botones que, al ser pulsados realizarán una determinada acción controlada desde AutoLISP. El argumento `label` indica el nombre que ocupará el espacio del botón. El argumento `fixed_width`, por su lado, indica si un elemento tiende o no a ocupar el espacio disponible. En este caso, si se establece como `true`, o no se establece (`true` es la opción predeterminada), el botón ocupará todo el espacio del que dispone de lado a lado de `:boxed_column`. El efecto no es muy atractivo —pruébese a quitarle dicho atributo o a declararlo como `true`— por lo que se suele establecer como `false` `fixed_width` en los botones de acción, lo que hace que el tamaño físico de los mismos se trunque con respecto al tamaño de texto que lleva dentro.

En el código que define este cuadro de diálogo podemos apreciar los caracteres `&` introducidos en las cadenas de etiquetas `label`. Estos caracteres funcionan de la misma manera que lo hacían en la definición de menús, vista en el **MÓDULO UNO** de este curso. Es decir, el carácter que sigue a `&` será una definición de tecla rápida o mnemotécnica (que aparecerá subrayada) la cual, con solo ser pulsada (o con `ALT` dependiendo del momento), hará acceder directamente al elemento que la lleve en su definición. Pero ojo, es una acceso rápido no una tecla de acción. Por ejemplo, los botones definidos que establecerían una configuración por defecto de las variables del cuadro, al pulsar una de las teclas mnemotécnicas que llevan implícitos serán designados (se dibuja una pequeña línea de puntos alrededor del texto) pero no pulsados. Para pulsarlos habría que presionar `INTRO`.

Las consideraciones que debemos observar a la hora de definir estas teclas son las mismas que en los menús. Por ejemplo, no podemos definir dos iguales en un mismo letrero (si existen dos iguales sólo funcionará la primera definida en el archivo `.DCL` o irán rotando), ni letras acentuadas, etcétera.

Otra forma de definir estas teclas mnemotécnicas es con el atributo `mnemonic`. Podemos ver un par de ejemplos en el código anterior (en las `:edit_box` *Isolineas* y *Suavizado*). `mnemonic` se hace igual a un carácter (si se indica más de uno únicamente se toma el primero) que ha de corresponder con uno existente en el atributo `label`. Si no existe en la cadena de `label` dicho carácter, la definición será ignorada. Si existen dos caracteres iguales en `label`, se toma como válido el primero.

NOTA: Cuidado con los botones *Ayuda* e *Info...* que vienen con los caracteres "u" e "l" predefinidos como teclas mnemotécnicas.

Por último decir que el `tile` predefinido `ok_button_help`, que muestra los tres botones correspondientes, está introducido dentro de la definición de una `:row`. Como sabemos esto no es necesario, pero se suele establecer así por mayor estructuración y claridad.

NOTA: En este cuadro de este último ejemplo existen un par de aspectos que demuestran lo que nunca se debe hacer a la hora de diseñar un letrero de diálogo, y si queremos que resulte vistoso. No es que quede mal del todo, pero apréciase la falta de gusto al incluir más elementos en la `:boxed_column` de la derecha que en la de la izquierda. También apréciase lo desigual de las dos `:edit_box` de la `:boxed_column` derecha. Podían haberse hecho iguales limitando después su tamaño real mediante `edit_limit`.

DIEZ.5.4. Parámetros de control de una curva

Vamos a crear ahora un cuadro que controla los parámetros de creación de una curva helicoidal en 3D mediante una polilínea. El programa AutoLISP que controla por completo este

cuadro se explicará, como ejemplo también en el **MÓDULO** siguiente. El letrero es el siguiente:



Análisis jerárquico

```
:dialog
|__ :row
|  |__ :image
|  |__ :boxed_column
|    |__ :radio_row
|    |  |__ :radio_button
|    |  |__ :radio_button
|    |__ :edit_box
|    |__ :edit_box
|__ :row
|  |__ :boxed_column
|  |  |__ :edit_box
|  |  |__ :popup_list
|  |__ :boxed_column
|  |  |__ :radio_row
|  |  |  |__ :radio_button
|  |  |  |__ :radio_button
|  |  |__ :edit_box
|  |  |__ :edit_box
|__ :row
|  |__ ok_cancel
|__ :row
|  |__ errtile
```

Código

```
helice:dialog {label="Hélice con Polilínea 3D";
:row {
:  :image {width=20;aspect_ratio=0.8;color=0;fixed_height=true;key=img;}
:  :boxed_column {label="Radios";
:    :radio_row {
```

```
:radio_button {label="&Iguales";value="1";key=igu;}
:radio_button {label="&Diferentes";key=dif;}
}
:edit_box {label="Radio i&nicial:";edit_width=6;fixed_width=true;
           key=radin;}
:edit_box {label="Radio &final:    ";edit_width=6;fixed_width=true;
           is_enabled=false;key=radif;}
spacer_1;
}
}
:row {
  :boxed_column {label="Vueltas";fixed_width=true;
    :edit_box {label="Nº &vueltas:";edit_width=2;edit_limit=2;key=nv;}
    :popup_list {label="&Precisión:";edit_width=8;list="8 ptos.\n16 ptos.\n24
               ptos.\n32 ptos.";key=pre;}
    spacer_1;
  }
  :boxed_column {label="Paso/Altura";
    :radio_row {
      :radio_button {label="P&aso";value="1";key=bpas;}
      :radio_button {label="Altu&ra";key=balt;}
    }
    :edit_box {label="Pas&o:";edit_width=8;key=pas;}
    :edit_box {label="Al&tura:";edit_width=8;is_enabled=false;key=alt;}
  }
}
:row {ok_cancel;}
:row {errtile;}
}
```

Explicación

El cuadro es muy parecido a los anteriores, únicamente se ha pretendido introducir elementos que no se habían visto aún. Como hemos podido venir comprobando, la estructura de un cuadro de diálogo es siempre la misma. Como los argumentos son comunes a todos los *tiles*, simplemente hemos de adaptar su significado a cada uno de ellos.

En este último ejemplo que vamos a comentar se han introducido, como decimos, elementos nuevos. El primero con el que nos tropezamos es `:image`. Este *tile* define un hueco para insertar una imagen que será una foto .SLD de **AutoCAD**. Los argumentos mínimos para definir un hueco de imagen son los que se corresponden a la altura y a la anchura. Esto podemos definirlo con `width` y `height`, o con cualquiera de los dos y `aspect_ratio`. La diferencia consiste en que, con los dos argumentos `width` y `height` damos a la imagen una altura y una anchura numérica como tal, y con cualquiera de estos y `aspect_ratio`, le proporcionamos una altura o una anchura y un factor de proporción de aspecto que hará que el cuadro —y por ende la imagen— sea más o menos cuadrado o más o menos rectangular. Un `aspect_ratio` de 1 define un cuadrado.

Lo normal es introducir también un atributo `color` con `:image` para darle un color de fondo.

El atributo `fixed_height=true` ajusta el cuadro un poco. Y es que, como sabemos, la técnica de construcción de un letrero de diálogo mediante DCL no es una ciencia exacta. Al final, el cuadro se reajusta y establece un poco a sí mismo dependiendo de los valores mayores de una columna y/o una fila, del número de elementos, etcétera.

Otro nuevo *tile* es `:radio_row`. `:radio_row` define una fila de botones excluyentes. Éstos son esas pequeñas casillas circulares que sólo puede estar una activada en cada grupo.

Cada elemento de una `:radio_row` ha de ser un `:radio_button`. Estos últimos se definen con su `label` correspondiente y, en este caso, con un `value=1` en uno de los botones para que aparezca señalado al abrir el cuadro. Al otro no hace falta ponerle `value=0`, pues al ser botones excluyentes aparecerá sin señalar por defecto. Existe también una `:radio_column` con el mismo significado que `:radio_row` pero en formato de columna.

También hemos introducido el elemento `:popup_list`, que define una lista desplegable. Aparte de otros argumentos ya conocidos, `:popup_list` introduce el argumento `list`, que declara todos los elementos que contiene la lista desplegable. Estos elementos han de ir seguidos y separados por el código de control `\n` que define un salto de línea.

Por último tenemos un elemento predefinido muy utilizado que es `errtile`. `errtile` define al final de cuadro de diálogo (normalmente) una línea de errores. Simplemente deja un hueco en el que irán apareciendo los errores pertinentes según cometa algún fallo el usuario. Todo esto, evidentemente, se controla desde AutoLISP.

Así como el botón de *Aceptar* tiene definida la clave (atributo `key`) `accept`, el botón *Cancelar* la clave `cancel`, el botón *Ayuda...* la clave `help` y el botón *Info...* la clave `info`, la línea de error `errtile` tiene definida la clave `error`. Todo esto lo utilizaremos para interactuar desde AutoLISP.

NOTA: Los tres espacios blancos en *Radio final:* hacen que las casillas `:edit_box` queden alineadas.

NOTA: Al cargar un archivo DCL y cuando se produce un error, a veces se crea un archivo denominado `ACAD.DCE` que puede ayudarnos a depurar errores de sintaxis.

Pues bien, hasta aquí este **MÓDULO** de programación en DCL. Ciertamente es que no se han mostrado ejemplos de todos los elementos, pero quedan todos explicados. Es trabajo del lector aplicar lo aprendido a diversos ejemplos propios de letreros de diálogo. La mecánica es en todos la misma y quien hace uno no tiene ningún problema en hacer más. En el **MÓDULO ONCE**, sobre programación en AutoLISP, se estudiará la manera de hacer que estos cuadros que hemos diseñado funcionen en condiciones. Allí se verá algún ejemplo más de algún letrero de diálogo.

DIEZ.FIN. EJERCICIOS PROPUESTOS

- I. Escribese el código DCL necesario para mostrar los siguientes letreros de diálogo expuestos:

a)

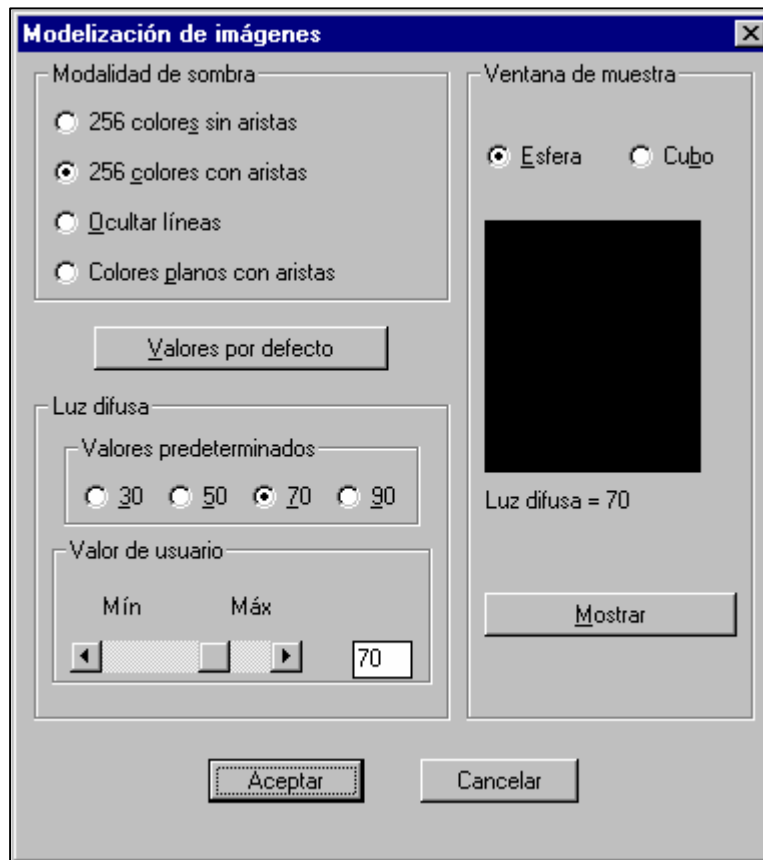
b-1)



b-2) (Este cuadro saldrá al pinchar el botón Mostrar... del anterior.)



c)



EJERCICIOS RESUELTOS DEL MÓDULO NUEVE

NOTA IMPORTANTE DE SINTAXIS: Las expresiones que sean muy largas y no puedan ser introducidas correctamente en este formato se dividen en varias líneas, aunque el usuario debe saber que ha de formar una sola línea en la entrada de MODEMACRO.

EJERCICIO I

Forzado: \$(if,\$(=,\$(getvar,snapmode),0),DESACTIVADO,ACTIVADO) | Estilo:
\$(if,\$(=,\$(getvar,snapstyl),0),ESTÁNDAR,ISOMÉTRICO)

EJERCICIO II

ID de Ventana: \$(getvar,cvport) | Centro de Ventana, X:
\$(index,0,\$(getvar,viewctr)) Y: \$(index,1,\$(getvar,viewctr))

EJERCICIO III

\$(edtime,\$(getvar,date),DDD", " DD MON YYYY - H:MMam/pm)

EJERCICIO IV

```
***POP1
[Som&bra]
[$(if,$(=,$(getvar,shadedge),0),!.)&256 colores]'shadedge 0
[$(if,$(=,$(getvar,shadedge),1),!.)&256 &resaltar]'shadedge 1
[$(if,$(=,$(getvar,shadedge),2),!.)&16 ocultar]'shadedge 2
[$(if,$(=,$(getvar,shadedge),3),!.)&16 re&llenar]'shadedge 3
[--]
[&Sombra]^C^C_.shade
```

EJERCICIO V

```
***POP7
[&Ventanas]
[$(if,$(=,$(getvar,tilemode),0),~)En &mosaico]^C^C_.vports
[$(if,$(=,$(getvar,tilemode),1),~)&Flotantes]^C^C_.mview
```

EJERCICIO VI

```
***POP3
[Som&breado]
[Doble/Simple rayado]^C^C$M=$(if,$(=,$(getvar,hpdouble),0),'hpdouble
1','hpdouble 0)
```

EJERCICIO VII

(Ejercicio completo para resolver por técnicos y/o especialistas).

PARTE TERCERA

MÓDULO ONCE

Programación en AutoLISP

ONCE.1. INTRODUCCIÓN

A parte de todo lo visto en cuestión de personalización, **AutoCAD** ofrece al usuario la posibilidad de crear programas y aplicaciones verticales totalmente funcionales. Estos programas podrán ser distribuidos por el creador, eso sí, siempre correrán bajo **AutoCAD**.

La capacidad para hacer un programa nos lleva mucho más allá de la simple personalización de menús o patrones de sombreado, nos lleva a un mundo totalmente integrado en **AutoCAD** desde donde podremos diseñar nuestros propios comandos, manipular dibujos o incluso acceder a la Base de Datos interna del programa.

AutoCAD proporciona diversas interfaces de programación de aplicaciones que vamos a comentar ahora de forma somera.

ONCE.1.1. AutoLISP, ADS, ARX, VBA y Visual Lisp

ONCE.1.1.1. Entorno AutoLISP

Dentro de lo que es la programación existen diversas interfaces para crear programas para **AutoCAD**. El más antiguo y, quizá el más utilizado hasta ahora, es AutoLISP. AutoLISP es una adaptación del lenguaje de programación LISP que forma parte íntima e integrada con **AutoCAD**, esto es, **AutoCAD** posee internamente un propio intérprete de LISP.

El lenguaje LISP está basado en lo que se denominan listas de símbolos. Es un lenguaje de alto nivel (como BASIC) y que, en principio, fue diseñado para la investigación en el campo

de la inteligencia artificial. AutoLISP es un lenguaje evaluado, es decir, está a un paso entre los lenguajes interpretados, por un lado, y los lenguajes compilados, por otro.

Como hemos dicho, **AutoCAD** provee al usuario de un propio intérprete de AutoLISP interno. Este intérprete, al que habría que llamarle *evaluador*, se encarga —precisamente— de evaluar las expresiones incluidas en el código fuente de un programa. Estos programas pueden introducirse directamente desde la línea de comandos de **AutoCAD**, o bien cargarse en memoria a partir de un programa completo escrito es un archivo de texto ASCII. Dichos archivos tendrán habitualmente la extensión `.LSP`.

De la programación en AutoLISP hablaremos largo y tendido en este **MÓDULO**.

ONCE.1.1.2. Entorno ADS

ADS (*AutoCAD Development System*) es un entorno de programación el lenguaje C para **AutoCAD**. Las aplicaciones programadas de esta manera han de ser llamadas desde un propio programa AutoLISP, ya que **AutoCAD** las considera como funciones externas.

El código fuente de estas aplicaciones ha de escribirse y compilarse en un compilador externo de C, es decir, **AutoCAD** no provee de un entorno de programación para estas aplicaciones, como ocurre con AutoLISP o VBA. El programa objeto ejecutable `.EXE` (normalmente), producto de la compilación del fuente, podrá ser cargado desde **AutoCAD**.

NOTA: Este entorno está ya obsoleto en la versión 14 y los programas desarrollados en él deberán adaptarse a ARX mediante el entorno ADSRX.

ONCE.1.1.3. Entorno ARX

ARX (*AutoCAD Runtime Extension*) es desde la versión 13 de **AutoCAD** otro entorno de programación C que posibilita, al contrario que ADS, una comunicación directa de las aplicaciones con **AutoCAD**. De esta forma, no es necesaria la intervención de AutoLISP como intermediario. La aplicación Render, en `RENDER.ARX`, es un ejemplo de programa en este formato.

La no integración de un entorno de programación en C dentro del software de **AutoCAD** ha hecho que su utilización para desarrollar aplicaciones para el programa no esté muy extendida. Es necesario, pues, adquirir un compilador de C complementario para poder desarrollar en este lenguaje aplicaciones ADS o ARX, lo que no ocurre con AutoLISP o VBA. Por este motivo, tampoco se tratará en este curso.

NOTA: En la versión 14 de **AutoCAD** existe el comando `ARX` para cargar, descargar y obtener información sobre aplicaciones ARX sin necesidad de utilizar AutoLISP.

ONCE.1.1.4. Entorno VBA

VBA (*Visual BASIC for Applications*) es un novedoso entorno, incluido en la versión 14, que parece hacer furor entre los programadores últimamente. La programación en Visual BASIC para Windows viene siendo, desde hace unos años, una de las herramientas más versátiles y, a la vez, más sencillas y utilizadas en el mundo de la programación informática. El usuario puede desarrollar sus programas en un compilador Visual BASIC externo o utilizar el propio módulo VBA que incluye **AutoCAD**. Este módulo contiene la sintaxis del lenguaje, un depurador y un entorno de desarrollo. Así, el programador, puede programar rutinas VBA e ir probándolas en una sesión de **AutoCAD** mientras se van depurando.

VBA será ampliamente tratado en el **MÓDULO DOCE**.

ONCE.1.1.5. Entorno Visual Lisp

A partir de la Versión 14 existe un nuevo entorno de desarrollo denominado Visual Lisp que permite realizar aplicaciones en AutoLISP de una manera más rápida y efectiva. Este entorno proporciona herramientas para desarrollar y depurar las rutinas y compilarlas como aplicaciones ARX. También dispone de su propio evaluador, que emula al de AutoLISP, además de un completo control de codificación y seguridad de las rutinas creadas.

El entorno de Visual Lisp es un módulo que se carga bajo demanda. No está incluido en el propio núcleo de **AutoCAD**, como ocurre con el evaluador de AutoLISP. El nuevo conjunto de funciones incorporadas en Visual Lisp permite trabajar en diferentes áreas y niveles que incluyen funciones añadidas de AutoLISP, funciones de acceso al sistema operativo y E/S de archivos, funciones de carga y vinculación de objetos y bases de datos, almacenamiento directo de listas en un archivo de dibujo, acceso al conjunto de objetos ActiveX de **AutoCAD** y tecnología basada en ObjectARX que no necesita la presencia de **AutoCAD** para su ejecución.

De esta manera, el entorno de Visual Lisp permite a los desarrolladores la programación en ARX y ActiveX.

ONCE.2. CARACTERÍSTICAS DE AutoLISP

Como ya hemos dicho, LISP (*LISt Processing*) es un lenguaje de programación que se remonta a los años cincuenta y que fue desarrollado para la investigación de inteligencia artificial. La base de su funcionamiento es el manejo de listas, en lugar de datos numéricos como otros lenguajes. AutoLISP es una implantación LISP en **AutoCAD**.

Una lista es un conjunto de símbolos. El símbolo es la unidad mínima básica de una lista, y puede ser una variable, una función inherente a AutoLISP, una función de usuario, un dato constante... Las listas elaboradas mediante símbolos son evaluadas y procesadas para obtener un resultado.

Para programar en **AutoCAD**, este lenguaje proporciona una serie de posibilidades como la facilidad para manejar objetos heterogéneos (números, caracteres, funciones, entidades u objetos de dibujo, etcétera), la facilidad para la interacción en un proceso de dibujo, la sencillez del lenguaje y su sintaxis, y otras que hacen de él una herramienta muy útil y sencilla de manejar y aprender.

Como también hemos dicho ya, el lenguaje AutoLISP (y LISP) es un lenguaje evaluado, y no interpretado o compilado. Los lenguajes interpretados son leídos palabra a palabra por el ordenador, al ser introducidas, y cada una de ellas convertida a lenguaje máquina. Esto hace que sea sencilla su edición y detección de errores de sintaxis u otros; por el contrario, hace que sean muy lentos (ejemplo: Microsoft QBASIC). Los códigos de los lenguajes compilados son escritos por completo y, antes de su ejecución final, es necesario compilarlos, convirtiéndolos así en código fuente ejecutable y comprensible por la máquina. Este tipo de lenguajes hace que su ejecución sea más rápida y pura pero, en contra, resulta más difícil su depuración (ejemplos: Microsoft QuickBASIC o Borland C++).

Los lenguajes evaluados —AutoLISP— están a caballo entre unos y otros. No son tan rápidos como los compilados pero son más flexibles e interactivos que estos. Es posible, por ejemplo, construir un programa con AutoLISP que sea capaz de modificarse a sí mismo bajo determinadas circunstancias; ésta es la base de los llamados *Sistema Expertos*.

El mecanismo evaluador de AutoLISP es la propia lista: conjunto de símbolos separados entre sí por, al menos, un espacio blanco y encerrados entre paréntesis. Esto es, desde el momento que existe una expresión encerrada entre paréntesis, AutoLISP la evalúa e intenta ofrecer un resultado.

AutoLISP es un subconjunto del lenguaje *Common LISP*. Como ha sido diseñado para trabajar desde **AutoCAD**, se han seleccionado las características de LISP más adecuadas para este fin y, además, se han añadido otras nuevas, sobre todo en la manipulación de objetos de dibujo, acceso a la Base de Datos de **AutoCAD** e interacción gráfica general.

Los programas en AutoLISP son simples archivos de texto ASCII, con la extensión habitual **.LSP**. Una vez hecho el programa, habremos de cargarlo desde el propio editor de dibujo de **AutoCAD**. También es posible escribir líneas de código AutoLISP desde la línea de comandos del programa, como veremos en breve.

Es posible la creación de órdenes nuevas que llamen a programas en AutoLISP, así como la redefinición de comandos propios de **AutoCAD**, como por ejemplo **LINEA** o **DESPLAZA**. Pero una de las más importantes potencialidades de AutoLISP es el acceso directo a la Base de Datos interna de **AutoCAD**. Toda la información de un dibujo, como deberíamos saber, no se guarda como objetos de dibujo en sí, o sea, cuando salvamos un **.DWG**, en disco no se guardan los círculos, líneas, etcétera, sino una relación o base de datos donde se dice dónde aparece un círculo o una línea, con qué coordenadas de origen y final, con qué radio o diámetro, tipo de línea, color... Podremos pues desde AutoLISP acceder a dicha base de datos para modificarla, editarla o para exportar datos, por ejemplo, a una base de datos externa.

ONCE.2.1. Tipos de objetos y datos en AutoLISP

Atendiendo a sus características podemos definir varios tipos de objetos y datos en AutoLISP, que son los de la tabla siguiente:

Objeto o dato	Descripción
<i>Lista</i>	Objeto compuesto de un paréntesis de apertura, uno o más elementos separados por, al menos, un espacio en blanco y un paréntesis de cierre. Los elementos de una lista pueden ser símbolos, valores concretos (constantes numéricas o cadenas) o listas incluidas. Por ejemplo: (DEFUN seno (x) (SETQ xr (* PI (/x 180.0))) (SETQ s (SIN xr))), se trata de una lista que contiene cinco elementos: una función inherente de AutoLISP DEFUN, un nombre de función de usuario seno, una lista con un único elemento (x) y dos listas SETQ con tres elementos cada una.
<i>Elemento</i> <i>Símbolo</i>	Cualquiera de los componentes de una lista. Cualquier elemento de una lista que no sea una constante. Puede ser el nombre de una variable, un nombre de función definida por el usuario o un nombre de función de AutoLISP.
<i>Enteros</i>	Valores numéricos enteros, sin punto decimal. Internamente son números de 32 bits con signo, sin embargo, en la transferencia de datos con AutoCAD son valores de 16 bits

Objeto o dato	Descripción
<i>Reales</i>	con signo, comprendidos por lo tanto entre -32768 y 32767. Valores numéricos con coma flotante de doble precisión. Esto representa un mínimo de 14 dígitos representativos. Se pueden expresar en notación científica exponencial mediante <i>e</i> o <i>E</i> .
<i>Cadenas</i>	Valores textuales que contienen cadenas de caracteres en código ASCII. Deben ir entre comillas y se permite una longitud máxima de 132 caracteres. Se pueden construir cadenas mayores como veremos.
<i>Descriptores de archivo</i>	Valores que representan un archivo abierto para lectura o escritura.
<i>Nombres de objetos de dibujo</i>	Valores que representan el "nombre" hexadecimal de un objeto de la Base de Datos.
<i>Conjuntos designados de AutoCAD</i>	Valores que representan un conjunto de selección de objetos de dibujo.
<i>Funciones de usuario</i>	Símbolo que representa el nombre de una función definida por el usuario.
<i>Función inherente o subrutina</i>	Símbolo con el nombre de una función predefinida de AutoLISP. Se suelen denominar con la abreviatura inglesa <i>subrs</i> . Pueden ser internas o externas.

ONCE.2.2. Procedimientos de evaluación

La base de todo intérprete de LISP es su algoritmo evaluador. Éste analiza cada línea de programa y devuelve un valor como resultado. La evaluación sólo se realizará cuando se haya escrito una lista completa y ésta podrá ser cargada desde un archivo de texto o tecleada directamente en la línea de comandos de **AutoCAD**.

El primer elemento de la lista es comparado con todos los nombres de funciones inherentes base o *subrs* internas de AutoLISP, con todos los nombres de *subrs* externas cargadas desde aplicaciones ADS o ARX y en su caso con todos los nombres de funciones de usuario cargadas en memoria. Si el nombre es reconocido, se realiza la evaluación de la expresión de AutoLISP. Esta puede consistir en asignar un valor a una variable, cargar en memoria una función de usuario, escribir un resultado en pantalla o en un archivo, dibujar un objeto gráfico, etc.

El primer elemento de la lista debe ser por tanto un nombre de función. El resto de elementos se consideran argumentos de dicha función. La evaluación en AutoLISP se realiza de acuerdo a las siguientes reglas.

Primera:

Las listas se evalúan quedando determinadas por el primer elemento. Si éste es un nombre de función inherente o subrutina, los elementos restantes de la lista son considerados como los argumentos de esa función. En caso contrario se considera como un nombre de función definida por el usuario, también con el resto de elementos como argumentos. Cuando los elementos de una lista son a su vez otras listas, éstas se van evaluando desde el nivel de anidación inferior (las listas más "interiores"). El valor resultante en cada evaluación de las listas "interiores", es utilizado por las listas "exteriores". Por ejemplo:


```
(SETQ sx (SIN (* PI (/ x 180.0))))
```

La lista más “interior” contiene como primer elemento el nombre de la función de AutoLISP / que realiza el cociente o división del siguiente elemento entre los restantes. La evaluación de dicha lista devuelve el resultado del cociente, en este caso el valor contenido en la variable *x* dividido entre el número real 180.0. Dicho valor es utilizado como elemento en la lista circundante que empieza por la función de AutoLISP * que realiza una multiplicación o producto. Esta lista se evalúa ofreciendo como resultado el producto de *PI* (3,14159) por el valor anterior. A su vez el resultado interviene como argumento en la lista que empieza por la función de AutoLISP SIN, que es evaluada obteniéndose el seno. Este interviene por último como argumento de la lista más exterior que empieza por la función de AutoLISP SETQ , cuyo resultado es asignar o almacenar en la variable *sx* el valor del seno. Este valor es devuelto por la lista de SETQ como resultado final.

Segunda:

Puesto que cada lista contiene un paréntesis de apertura y otro de cierre, cuando existen listas incluidas en otras, todos los paréntesis que se vayan abriendo deben tener sus correspondientes de cierre. Si se introduce desde el teclado una expresión en AutoLISP a la que falta por cerrar algún paréntesis, se visualiza un mensaje del tipo *n>* donde *n* es un número que indica cuántos paréntesis faltan por cerrar. Se pueden introducir por teclado esos paréntesis y subsanar el error. Por ejemplo:

```
Comando: (SETQ sx (SIN (* PI (/ x 180.0)))
2> ))
0.523599
```

Tercera:

También es posible evaluar directamente un símbolo (extraer por ejemplo el valor actual contenido en una variable), introduciendo por teclado el signo de cerrar admiración seguido del nombre del símbolo. Esta evaluación se puede producir incluso en mitad de un comando. Por ejemplo, para suministrar como ángulo para un arco el valor contenido en la variable *x*, se responde a la solicitud de **AutoCAD** con *!x*. Por ejemplo:

```
Comando: !sx
0.523599
```

Cuarta:

Los valores enteros, reales, cadenas de texto y descriptores de archivos, devuelven su propio valor como resultado. Los nombres de funciones inherentes o subrutinas de AutoLISP devuelven un número interno hexadecimal (suele cambiar de una sesión de dibujo a otra). Por ejemplo:

!72.5	devuelve	72.5
!"Inicio"	devuelve	"Inicio"
!SETQ	devuelve	<Subr: #1a93e24>

Quinta:

Los símbolos de variables participan con el valor que contienen (que les está asociado) en el momento de la evaluación. Por ejemplo:

!x	devuelve	30
!sx	devuelve	0.523599

Sexta:

Determinadas funciones de AutoLISP pueden devolver un valor nulo, que se representa mediante la expresión `nil`. Por ejemplo:

```
Comando: (PROMPT "Bienvenido a AutoLISP\n")
Bienvenido a AutoLISP
nil
```

La función `PROMPT` escribe en la línea de comando el mensaje especificado y devuelve `nil`. El código `\n` equivale a `INTRO`.

ONCE.2.3. Archivos fuente de programas

Las expresiones contenidas en un programa de AutoLISP pueden introducirse directamente desde el teclado durante la edición de un dibujo de **AutoCAD**, escribirlas en un fichero de texto ASCII o ser suministradas por una variable del tipo cadena, como ya se ha dicho varias veces.

Para una correcta evaluación, las expresiones deben cumplir unos requisitos de sintaxis, que se pueden resumir en los siguientes puntos:

- Una expresión puede ser tan larga como se quiera. Puede ocupar varias líneas del archivo de texto.

- Los nombres de símbolos pueden utilizar todos los caracteres imprimibles (letras, números, signos de puntuación, etc.) salvo los prohibidos que son: () . ' " ;

- Los nombres de símbolos no son sensibles a las mayúsculas. Así, `seno` y `SENO` representan el mismo nombre. Los nombres pueden contener números, pero no estar formados exclusivamente por números. Así, `1pt`, `pt-1`, `p12` son válidos como nombres de variables, pero no `21`, que será interpretado como un valor numérico constante.

- Los caracteres que terminan un nombre de símbolo o un valor explícito (una constante numérica o de texto) son: paréntesis de apertura y cierre, apóstrofo, comillas, punto y coma, espacio en blanco o final de línea en el archivo. Estos caracteres sirven de separación entre elementos de una lista.

- Los espacios en blanco de separación entre símbolos son interpretados como un solo espacio entre cada par de símbolos. Se recuerda que es necesario un espacio en blanco para separar un símbolo del siguiente, siempre que no haya paréntesis, apóstrofo, comillas o punto y coma. Debido a la longitud de las expresiones de AutoLISP y a la profusión de paréntesis que dificultan su lectura, suele ser norma habitual realizar sangrados en las líneas del archivo de texto, para resaltar los paréntesis interiores de los exteriores. Todos los espacios añadidos son interpretados como uno solo.

- Los valores explícitos (constantes) de números pueden empezar con el carácter `+` o `-` que es interpretado como el signo del número.

- Los valores de constantes de números reales deben empezar con una cifra significativa. El carácter punto (.) se interpreta como el punto decimal. También se admite `+` o `-` para el signo y `e` o `E` para notación exponencial o científica. No es válida la coma decimal, ni tampoco abreviar como en `.6` (hay que escribir `0.6`).

— Los valores de constantes con cadenas de texto son caracteres que empiezan y terminan por comillas. Dentro de las cadenas se pueden incluir caracteres de control mediante la contrabarra (\). Los códigos permitidos son:

\\	Carácter contrabarra (\).
\"	Carácter comillas (").
\e	Carácter de escape.
\n	Nueva línea o retorno de carro.
\r	INTRO
\t	Carácter de tabulador TAB
\nnn	Carácter cuyo código octal (no ASCII, que es decimal) es <i>nnn</i> .
\U+xxxx	Secuencia de caracteres de código <i>Unicode</i> .
\M+nxxxx	Secuencia de caracteres multibyte.

Los códigos deben ir en minúsculas. Para incluir en una cadena un código ASCII hay que calcular su valor octal. Por ejemplo, el carácter dólar \$ es ASCII 36; su valor octal será 44 y en la cadena habrá que indicar el código de control \44.

— El apóstrofo (') se puede utilizar como abreviatura del comando QUOTE. El comando QUOTE devuelve el literal del símbolo. Es decir, cuando en una expresión un símbolo aparece precedido por apóstrofo o se le aplica la función de AutoLISP QUOTE, no se evalúa con el valor que contiene en ese momento sino que devuelve el propio nombre literal del símbolo.

— Se pueden incluir comentarios en un archivo de texto con programas y expresiones en AutoLISP, comenzando la línea del archivo con un punto y coma (;). A partir de donde se encuentre un punto y coma hasta el final de la línea, AutoLISP considera que son comentarios y no los tiene en cuenta. También se pueden incluir comentarios en mitad de una línea u ocupando varias de ellas, si se sitúan entre los caracteres ;| y |;. Por ejemplo:

```
;Función de estado actual del dibujo.
(DEFUN funcion_1 (x / pt1 pt2)
  (SETQ refnt0 ;| modos de referencia actualmente
    activados |; (GETVAR "osmode"))
...
)
```

Los comentarios son útiles tanto para el autor del programa como para futuros usuarios que accedan al archivo con el fin de modificarlo. Suele ser habitual situar al principio del archivo el título, autor y fecha de creación. Posteriormente, una explicación general del programa, explicaciones particulares de cada función intermedia, usos de variables, etc. Como muchas de las configuraciones de pantalla de texto disponen de un ancho de 80 columnas, conviene que las líneas del archivo de texto no sobrepasen los 80 caracteres.

ONCE.2.4. Variables predefinidas

Existen unos valores de símbolos de AutoLISP predefinidos. Son los siguientes:

- **PI**. Es el valor del número real PI, es decir: 3,141592653589793.
- **PAUSE**. Es una cadena de texto que consta de un único carácter contrabarra. Se utiliza para interrumpir un comando de **AutoCAD** después de haberlo llamado mediante la función de AutoLISP COMMAND. Esto permite al usuario introducir algún dato.
- **T**. Es el símbolo de *True*, es decir, cierto o verdadero (valor 1 lógico). Se utiliza para establecer que determinadas condiciones se cumplen.

- Por último el valor de nada, vacío o falso (0 lógico) se representa en AutoLISP por `nil`. Este valor aparece siempre en minúsculas y no es propiamente un símbolo, ya que no está permitido acceder a él.

ONCE.3. PROGRAMANDO EN AutoLISP

A partir de ahora vamos a comenzar a ver poco a poco la manera de ir haciendo nuestros programas en AutoLISP. Vamos a seguir un orden lógico de menor a mayor dificultad, por lo que la estructura puede llegar a parecer un poco caótica para alguien que conozca el lenguaje. Tampoco es objetivo de este curso profundizar en un método complejo de programación, sino proponer unas bases para comenzar a programar que, con imaginación y horas de trabajo, podrá convertirnos en programadores expertos de AutoLISP.

Todo lo visto hasta ahora resulta la parte árida de la programación; parece que no sirven de mucho esos conocimientos teóricos mientras no pasemos a la práctica. De aquí en adelante podremos ir entendiendo las 2 primeras secciones de este **MÓDULO** y, si algo se quedó en el tintero o algo hay que repetir de todo lo expuesto hasta aquí, se completará y se explicará o repetirá.

Comencemos, pues, con la programación en AutoLISP para **AutoCAD**.

ONCE.3.1. Convenciones de sintaxis

Las convenciones utilizadas para las sintaxis en este **MÓDULO** van a ser las siguientes:

- Sintaxis recuadrada para su fácil búsqueda y detección.
- Nombre del comando o función AutoLISP en mayúsculas.
- Argumentos en minúscula itálica, representados por un nombre mnemotécnico.
- Argumentos opcionales encerrados entre corchetes itálicos (que no han de escribirse).
- Puntos suspensivos en itálica indican la posibilidad de indicar más argumentos.

ONCE.4. OPERACIONES NUMÉRICAS Y LÓGICAS

Explicaremos aquí la manera en que se realizan en AutoLISP las operaciones matemáticas, de comparación y lógicas. El buen aprendizaje de estas técnicas nos será tremendamente útil a la hora de lanzarnos a la programación pura.

ONCE.4.1. Aritmética básica

Para realizar las cuatro operaciones aritméticas básicas existen cuatro funciones AutoLISP que son `+`, `-`, `*` y `/`, estas se corresponden con la suma, resta, multiplicación y división.

La función de suma tiene la siguiente sintaxis:

`(+ [valor1 valor2 valor3...])`

Esto es, primero se indica el nombre de la función, como siempre en AutoLISP, que en este caso es `+` y luego los argumentos de la misma, es decir, aquí los valores de los distintos sumandos.

Esta función devuelve el resultado aditivo de todos los valores numéricos especificados como argumentos de la función. Por ejemplo:

```
(+ 14 10 20)
```

devolvería el valor 44. Para hacer la prueba únicamente debemos escribir dicho renglón en la línea de comandos de **AutoCAD**, pulsar `INTRO` y comprobar el resultado.

NOTA: Al introducir el primer carácter de apertura de paréntesis, **AutoCAD** reconoce que se está escribiendo una expresión en AutoLISP, por lo que nos permitirá utilizar los espacios necesarios de la sintaxis sin que se produzca un `INTRO` cada vez, como es habitual. Recordemos que todos los elementos de una lista de AutoLISP han de ir separados por lo menos con un espacio blanco. Probemos diferentes sintaxis utilizando más espacios, o tabuladores, y comprobemos que el resultado es el mismo; se interpretan los espacios o tabuladores como un único carácter de espacio en blanco.

NOTA: Hagamos la prueba de no introducir el paréntesis final de la lista indicada. Comprobaremos lo explicado en la segunda regla de la sección **ONCE.2.2**.

Con la función `+` podemos indicar valores enteros o reales. Si todos los valores son enteros el resultado será entero, pero si uno o varios de ellos son reales —o todos ellos—, el resultado será real. Esto significa que únicamente es necesario introducir un valor real para recibir una respuesta real. Por ejemplo, si introducimos la siguiente línea en la línea de comandos en **AutoCAD**:

```
(+ 14 10 20.0)
```

el resultado será:

```
44.0
```

o sea, un número real.

Esto aquí parece irrelevante, pero comprenderemos su utilidad al hablar, por ejemplo, de la división.

Si indicamos un solo sumando con esta función, el resultado es el valor del propio sumando. Por ejemplo:

```
(+ 23)
```

devuelve:

```
23
```

Y si se escribe la función sin argumentos, el resultado es 0 (función sin argumentos: `(+)`).

Los valores indicados en la función de suma pueden ser directamente valores numéricos o nombres de variables numéricas declaradas anteriormente, por ejemplo:

```
(+ 10.0 x total)
```

En esta función, 10.0 es un valor constante real y `x` y `total` son dos nombres de variables que han debido ser anteriormente declaradas; ya aprenderemos a declarar variables. Si la variable no existiera se produciría un error *bad argument type* de AutoLISP.

Otros ejemplos con números negativos:

(+ 10 -23)	devuelve -13
(+ -10 -10)	devuelve -20

NOTA: Si se produce algún error de sintaxis u otro, podemos acudir al final de este **MÓDULO** para ver una relación de los mensajes de error de AutoLISP.

La función de resta, por su lado, tiene la siguiente sintaxis:

`(- [valor1 valor2 valor3...])`

Esta función devuelve la diferencia del primer valor con todos los demás indicados. Así por ejemplo:

`(- 10 5)`

da como resultado 5 y la siguiente expresión:

`(- 10 5 2)`

da como resultado 3. Esto es producto de restar $10 - 5 = 5$ y, luego, $5 - 2 = 3$; o lo que es lo mismo $10 - (5 + 2) = 3$.

Al igual que en la suma, si se indican valores enteros el resultado será entero, si se indica uno real (con uno es suficiente) el resultado es real, si se indica un solo valor se devuelve el mismo valor y si se escribe la función sin argumentos se devuelve 0. Así pues, si queremos un resultado real efectuado con números enteros para posteriores operaciones, deberemos indicar uno de los valores entero; de la siguiente manera, por ejemplo:

`(- 10 5.0 2)`

o cualquier otra combinación posible de uno o más números enteros.

Como se ha explicado para la suma, los valores de los argumentos para la resta pueden ser constantes, eso sí, siempre numéricas, o variables:

`(- tot num1 num2)`

Llegados a este punto, podemos suponer ya las diferentes combinaciones que podremos realizar con las distintas funciones aritméticas. Por ejemplo, es factible la evaluación de la siguiente expresión:

`(+ 12 (- 2 -3))`

cuyo resultado es 11. O sea, y como hemos explicado, se realizarán las operaciones de dentro a fuera. En este ejemplo, se suma la cantidad de 12 a la diferencia $2 - 3$, esto es, $12 + (2 - 3) = 11$. Como vemos, existen dos listas, una interior anidada a la otra que es, a la vez, argumento de la lista exterior. Ocurre lo mismo con nombres de variables:

`(- fer1 (+ -sum1 sum2) 23.44)`

Con respecto al producto su sintaxis es la siguiente:

`(* [valor1 valor2 valor3...])`

Se evalúa el producto de todos los valores numéricos indicados como argumentos. Como anteriormente, si un valor es real el resultado es real. Un solo valor como argumento devuelve el mismo valor. Ningún valor devuelve 0. Veamos un ejemplo:

```
( * 12 3 4 -1 )
```

El resultado es -144. Veamos otros ejemplos:

```
( * 2 3 )  
( * val ( - vax vad ) )  
( - ( * 12 2 ) 24 )  
( + ( - -10 -5 ) ( * 3 total 23 ) )
```

NOTA: Si escribimos más paréntesis de los necesarios por la derecha se nos mostrará un mensaje de error. Recordar que si no los escribimos nos da la opción de escribirlos después, así como el número de ellos que faltan. De todas formas, consúltense el final del **MÓDULO** para la explicación de los mensajes de error.

La sintaxis de la división es la que sigue:

```
( / [valor1 valor2 valor3...] )
```

La función / realiza el cociente del primer valor numérico por todos los demás, es decir, divide el primer número por el producto de los demás. Por ejemplo:

```
( / 10 2 )
```

da como resultado 5. Y el ejemplo siguiente:

```
( / 100 5 5 )
```

da como resultado 4, es decir, $100 / 5 = 20$ y, luego, $20 / 5 = 4$; o lo que es lo mismo, $100 / (5 * 5) = 4$.

Otros dos ejemplos:

```
( / 24 ( * ( + 10.0 2 ) 12 ) )  
( / 12 2 1 )
```

Con respecto al cociente debemos realizar las mismas observaciones anteriores, esto es, si se indica un solo valor se devuelve el mismo valor, si se indica la función sin argumentos se devuelve 0 y si se indican valores enteros sólo se devuelven valores enteros. Esto último cobra especial sentido en el caso de las divisiones, ya que el cociente entre dos números enteros puede ser un número real. Veamos el siguiente ejemplo:

```
( / 15 7 )
```

Si introducimos esta línea el resultado será 2. El motivo es que, como hemos especificado valores enteros, el resultado se muestra en forma de número entero, con la parte decimal o mantisa truncada. Para asegurarnos de recibir una respuesta correcta (con decimales significativos), deberemos introducir uno de los valores —o todos ellos, pero con uno es suficiente— como valor real, de la siguiente forma:

```
( / 15 7.0 )
```

Ahora el resultado será 2.14286. El número entero podría haber sido el otro:

```
( / 15.0 7 )
```

Esto debemos tenerlo muy en cuenta a la hora de realizar operaciones cuyo resultado vaya a ser parte integrante de otra operación —o no— que puede devolver decimales.

Vemos otros ejemplos de divisiones:

```
(/ -12.0 7.8 210)
(/ (+ (- 23.3 32) 12.03) (/ (* (+ 1.01 2.01) 100)))
(+ datos (/ grupo (* 100 2)))
```

NOTA: Evidentemente, la división por 0 produce un error de AutoLISP: *divide by zero*.

1ª fase intermedia de ejercicios

• Realizar mediante AutoLISP las siguientes operaciones aritméticas (están expresadas en notación informática sencilla de una línea):

```
— (50 + 5) / 2
— (200 * 5 - 3) / (4 / 2)
— (10.23 - (12.03 / 3)) * (12 + (2 * -2) - ((12.5 / 2) * 2.65))
— (19 + 23) / (10 + (23 / (23 / 19)))
— ((-20 / 5) - 1) / (15.5 * ((15.5 - 1) / 12))
```

ONCE.4.2. Matemática avanzada

Fuera aparte de las funciones aritméticas de nivel básico, programando en AutoLISP podemos realizar operaciones matemáticas complejas como raíces cuadradas o senos. Vamos a explicar aquí las funciones que controlan estas operaciones.

Las dos primeras que veremos son sencillas de entender y utilizar. Ambas se refieren al incremento, positivo o negativo, de una unidad a un valor numérico.

`(1+ valor)`

Esta sintaxis corresponde a la función de incremento positivo de una unidad al valor indicado. Así, si queremos incrementar en 1 el valor 576, haremos:

```
(1+ 576)
```

Esto equivale a `(+ 576 1)` pero es de una forma más cómoda; el resultado es 577.

NOTA: La función se denomina 1+ en sí, por lo que no existe espacio entre 1 y +.

Podemos realizar estos incrementos con nombres de variable:

```
(1+ n)
```

lo que incrementará en una unidad el valor de *n*. Esto puede ser especialmente necesario para controlar los llamados *contadores-suma* en programación, del tipo:

```
(SETQ sum (1+ sum))
```

Esto ya lo veremos a la hora de declarar variables.

La siguiente función resta (incremento negativo) una unidad al valor numérico indicado. Equivale a `(- valor 1)`, pero de una forma más cómoda.

`(1- valor)`

Por ejemplo:

`(1- 32)`

su resultado es 31.

Otros ejemplos de estas dos funciones:

```
(1- n)
(1- 67.90)
(1- -23)
(1+ -34.0)
(1+ (+ tuttoto 1))
(1- (* 2 2))
(1- (* (/ 32 2) (+ 10 1.0)))
```

`(ABS valor)`

Esta función `ABS` devuelve el valor absoluto del número indicado o expresión indicada. De esta forma, la siguiente expresión:

`(ABS -23)`

devuelve 23.

Las siguientes expresiones tienen el siguiente efecto indicado:

<code>(ABS -25.78)</code>	devuelve 25.78
<code>(ABS 45)</code>	devuelve 45
<code>(ABS 0)</code>	devuelve 0
<code>(ABS -13)</code>	devuelve 13
<code>(ABS (/ 2 3.0))</code>	devuelve 0.666667
<code>(ABS (/ 2 -3.0))</code>	devuelve 0.666667

`(FIX valor)`

`FIX` trunca un valor a su parte entera (positiva o negativa), es decir, de un número real con decimales devuelve únicamente su parte entera. Pero, cuidado, no se produce redondeo, sólo un truncamiento.

Ejemplos:

<code>(FIX 32.79)</code>	devuelve 32
<code>(FIX -12.45)</code>	devuelve -12
<code>(FIX (/ 10 3.0))</code>	devuelve 3
<code>(FIX (/ 10 -3.0))</code>	devuelve -3

`(REM valor1 valor2 [valor3...])`

Esta función `AutoLISP` devuelve el resto del cociente (módulo) de los dos valores introducidos en principio. Por ejemplo, la siguiente expresión devuelve 6 como resultado:

`(REM 20 7)`

Dicho 6 es el resto que resulta de dividir $20 / 7$. Si aplicamos la regla de la división (dividendo es igual a divisor por cociente más resto): $20 = 7 * 2 + 6$, vemos que se cumple correctamente.

Si se especifican más de dos valores, el resto anterior es dividido entre el actual, devolviendo el nuevo resto de la nueva división. Por ejemplo:

```
(REM 20 7 4)
```

da como resultado 2. El primer resto 6 se calcula de la forma explicada en el ejemplo anterior y, el resultado final 2, se produce al dividir dicho primer resto entre el tercer valor 4. Al dividir $6 / 4$, nos da un resultado (que es igual a 1) y un resto 2 (valor final obtenido). Y así sucesivamente.

Otros ejemplos:

```
(REM -1 2)
(REM 0 23)
(REM (* 23 2) (- (+ 1 1) 45.5))
(REM 54 (* 3 -4))
```

Pasemos ahora a ver las funciones trigonométricas, esto es, cómo calcularlas mediante AutoLISP. La primera sintaxis se refiere al seno de un ángulo y es la siguiente:

```
(SIN ángulo)
```

La función SIN devuelve el seno de un ángulo expresado en radianes. Ejemplos:

```
(SIN 1)           devuelve 0.841471
(SIN (/ PI 2))    devuelve 1.0
```

NOTA: Como sabemos PI es un constante de AutoLISP, por lo que no hace falta declararla como variable; ya tiene valor propio y es 3.14159. Aún así, se puede calcular su valor exacto mediante la expresión: $PI = 4 * \arctan 1$.

```
(COS ángulo)
```

COS devuelve el coseno de un ángulo expresado en radianes. Ejemplos:

```
(COS PI)           devuelve -1.0
(COS (* 3 4))      devuelve 0.843854
```

NOTA: Nótese que PI es un valor real, por lo que el resultado será real.

```
(ATAN valor1 [valor2])
```

Esta función ATAN devuelve el arco cuya tangente es valor1 expresada en radianes, es decir, realiza el arco-tangente de dicho valor. Por ejemplo:

```
(ATAN 1.5)         devuelve 0.98
```

Si se indica un segundo valor (valor2), ATAN devuelve el arco-tangente de valor1 dividido por valor2. Esto permite indicar la razón entre los lados de un triángulo recto, es decir, escribir la tangente directamente como cociente del seno entre el coseno. Si valor2 es 0, el valor devuelto será igual a $PI / 2$ o a $-PI / 2$ radianes, dependiendo del signo de valor1.

Ejemplos:

```
(ATAN 1 1)
(ATAN 1 (* 2 -4.5))
```

Estas son las tres funciones trigonométricas de AutoLISP. En este punto se nos plantean un par de problemas: ¿cómo calculo las restantes funciones trigonométricas? y ¿cómo convierto grados sexagesimales en radianes y viceversa?

La segunda cuestión es sencilla, ya que basta aplicar al fórmula $rad = grados * PI / 180$ para convertir grados en radianes. La operación inversa es fácilmente deducible.

La primera pregunta tiene una respuesta no menos sencilla, y es que en la mayoría — por no decir todos— de los lenguajes de programación únicamente nos proporcionan estas funciones trigonométricas básicas y, a partir de ellas, podemos calcular las funciones trigonométricas derivadas inherentes. La manera se explica a continuación mediante notación sencilla de una línea:

Función derivada	Notación
Secante (sec x)	1 / cos (x)
Cosecante (cosec x)	1 / sen (x)
Arco-seno (arcsen x)	arctag (x / $\sqrt{1 - x^2}$)
Arco-coseno (arccos x)	1.5707633 – arctag (x / $\sqrt{1 - x^2}$)
Arco-secante (arcsec x)	arctag ($\sqrt{x^2 - 1}$) + signo (x) – 1) * 1.5707633
Arco-cosecante (arccsc x)	arctag (1/ $\sqrt{x^2 - 1}$) + signo (x) – 1) * 1.5707633
Arco-cotang. (arccotag x)	1.5707633 – arctag (x)

NOTA: El símbolo ^ significa exponenciación. $\sqrt{}$ es raíz cuadrada. *signo* (x) se refiere al signo del valor; si éste es positivo *signo* (x) valdrá 1, si es negativo valdrá –1 y si es cero valdrá 0. No debemos preocuparnos ahora por esto, ya que aprenderemos en breve o más adelante —con mayor soltura— a realizar exponenciaciones, raíces cuadradas y operaciones con signos.

Sigamos, pues, ahora con otras diferentes funciones que nos ofrece AutoLISP a la hora de realizar operaciones matemáticas. La siguiente dice referencia a las raíces cuadradas; su sintaxis es:

```
(SQRT valor)
```

Esta función devuelve el resultado de la raíz cuadrada del valor indicado, ya sea un guarismo simple o una expresión matemática, como siempre. Así por ejemplo, veamos unas expresiones con sus correspondientes evaluaciones:

```
(SQRT 4)           devuelve 2.00
(SQRT 2)           devuelve 1.4142
(SQRT (* 2 6))     devuelve 3.4641
```

La intención de extraer una raíz cuadrada de un número negativo produce el error *function undefined for argument* de AutoLISP.

Por otro lado, la sintaxis para la función exponencial es la siguiente:

```
(EXPT base exponente)
```

EXPT devuelve el valor de *base* elevado a *exponente*. De esta forma, para elevar 5 al cubo (igual a 125), por ejemplo, escribiremos:

(EXPT 5 3)

Otro ejemplo:

(EXPT 2.3 7.23)

De esta forma, como sabemos, podemos resolver el resto de raíces (cúbicas, cuartas, quintas...) existentes. Ya que raíz cúbica de 32 es lo mismo que 32 elevado a $1/3$, podemos escribir la siguiente expresión:

(EXPT 32 (/ 1 3))

Así también:

(EXPT 20 (/ 1 5))

(EXPT 10 (/ (+ 2 4) (- v23 rt sw2)))

(EXPT 3 (/ 1 2))

NOTA: El intento de extraer raíces negativas de cualquier índice producirá el mismo error explicado en `SQRT`.

`(EXP exponente)`

Esta función devuelve la constante (número) e elevada al exponente indicado. Se corresponde con el antilogaritmo natural. Por ejemplo:

(EXP 1) devuelve 2.71828

`(LOG valor)`

`LOG` devuelve el logaritmo neperiano o natural (en base e) del valor indicado. Por ejemplo:

(LOG 4.5) devuelve 125.0000

`(GCD valor_entero1 valor_entero2)`

Esta sintaxis se corresponde con la función de AutoLISP `GCD`, que devuelve el máximo común denominador de los dos valores indicados. Estos valores han de ser obligatoriamente enteros, de no ser así, AutoLISP devuelve *bad argument type* como mensaje de error. Veamos unos ejemplos:

(GCD 45 80)	devuelve 5
(GCD 80 70)	devuelve 10
(GCD (* 10 10) (/ 70 2))	devuelve 5

Si se indica un entero negativo el mensaje de error de AutoLISP es *improper argument*.

Las dos últimas funciones matemáticas que veremos pueden sernos de gran ayuda a la hora de programar. Una de ellas (`MAX`) devuelve el mayor de todos los números indicados en la lista. Su sintaxis es:

`(MAX valor1 valor2...)`

Los valores pueden ser números enteros o reales, y también expresiones matemático-aritméticas. Así por ejemplo:

```
(MAX 78.34 -12 789 7)
```

devolverá 789.0, ya que es el número mayor. Lo devuelve como real por la aparición de decimales en el elemento 78.34. Como sabemos, con la sola aparición de un valor real en una lista, el resultado es real.

Si el elemento mayor de la lista es una expresión matemática, se devolverá su resultado, no la expresión en sí, por ejemplo:

```
(MAX (* 10 10) 5)
```

devolverá 100 como resultado (10 * 10).

Otro ejemplo:

```
(MAX -5 -7 -9)
```

devolverá -5.

```
(MIN valor1 valor2...)
```

La función MIN, por su lado, devuelve el menor de todos los valores indicados en lista. Las demás consideraciones son análogas a la función anterior. Ejemplos:

(MIN 1 2 3 4 7)	devuelve 1
(MIN 23.3 7 0)	devuelve 0.0
(MIN (/ 7 3) 0.56)	devuelve 0.56

Ejemplos de MAX y MIN con variables:

```
(MIN x y z)  
(MIN (+ x1 x2) (+ y1 y2) (+ w1 w2) (+ z1 z2))
```

Y hasta aquí todas las funciones que tienen que ver con operaciones matemáticas. Pasaremos, tras unos ejercicios propuestos, a ver las operaciones de comparación, muy interesantes y sencillas de comprender.

2ª fase intermedia de ejercicios

- Realizar mediante AutoLISP las siguientes operaciones matemáticas (están expresadas en notación sencilla de una línea):

- $\sqrt{(20 - 3) * (8 - 2)}$
- $1 + 78.8 + ((78.8 ^ 2) / 2) + ((78.8 ^ 3) / 3)$
- $(\text{sen}(\sqrt{(80 * 28.002)} - \cos(\text{PI} / 2))) / (\text{PI} - (1 / 2))$
- $\arccos(100 / 2)$
- $(124.6589 * (e ^ 2.3)) / (7 * \sqrt{2})$
- $\ln(45 * (7 / 2))$
- $(23.009 / 78.743) ^ (56.00123 - 1)$

- Realícense ejercicios de cálculo de valores mayores y menores de listas, así como de máximos comunes denominadores.

- Realizar un par de ejercicios de incremento y decremento de una unidad a valores.

NOTA: Las operaciones en general siguen en AutoLISP la jerarquía de las operaciones matemáticas: paréntesis internos, paréntesis externos, operadores unitarios (signos), potenciación, multiplicación y división, suma y resta, operadores relacionales (mayor que, menor que...) y operadores lógicos (álgebra de Boole). Y cuando existen varios operadores en el mismo nivel, se ejecutan de izquierda a derecha. Ahora mismo veremos operadores relacionales o de comparación y, luego, el álgebra de Boole en AutoLISP.

ONCE.4.3. Operaciones relacionales

Las funciones que veremos a continuación se denominan relacionales o de comparación, y es que comparan valores, ya sean numéricos o textuales (cadenas) emitiendo un resultado verdadero o falso, según la comparación. Estas funciones son conocidas por todos (igual, mayor que, menor o igual que...), sólo queda determinar cómo se utilizan y cuál es su sintaxis en AutoLISP.

Como hemos dicho el resultado de la evaluación solo puede ser uno de dos: T (True) que representa el verdadero o cierto, o nil que representa el falso o nulo.

NOTA: Con la devolución nil por parte de AutoLISP nos empezamos a familiarizar ahora y la veremos muchas veces.

Comencemos por el igual o igual que, cuya sintaxis es la siguiente:

`(= valor1 [valor2...])`

La función = compara todos los valores especificados —uno como mínimo—, devolviendo T si son todos iguales o nil si encuentra alguno diferente. Los valores pueden ser números, cadenas o variables (numéricas o alfanuméricas). Así por ejemplo:

<code>(= 5 5)</code>	devuelve T
<code>(= 65 65.0)</code>	devuelve T
<code>(= 7 54)</code>	devuelve nil
<code>(= 87.6 87.6 87.6)</code>	devuelve T
<code>(= 34 34 -34 34)</code>	devuelve nil

Veamos ahora algún ejemplo con cadenas:

<code>(= "hola" "hola")</code>	devuelve T
<code>(= "casa" "cAsa")</code>	devuelve nil
<code>(= "H" "H" "H" "H")</code>	devuelve T
<code>(= "hola ahora" "hola ahora")</code>	devuelve nil

NOTA: Nótese, como adelanto, que las cadenas literales han de ir encerradas entre comillas, como en casi todos los lenguajes de programación.

Con variables declaradas, que ya veremos, sería de la misma forma. Si sólo se indica un valor en la lista, AutoLISP devuelve T.

NOTA: Hay que tener en cuenta que esta función sólo compara valores y no listas o expresiones. Si, por ejemplo, se tienen dos variables pt1 y pt2 con dos puntos que son listas de tres elementos (una coordenada X, una coordenada Y y una coordenada Z), para comparar

la igualdad de ambos habría que recurrir a una función lógica como `EQUAL`, que veremos un poco más adelante.

`(/= valor1 [valor2...])`

Esta función `/=` (distinto o desigual que) devuelve `T` si alguno o algunos de los valores comparados de la lista son diferentes o distintos de los demás, por ejemplo en los siguientes casos:

```
(/= 2 3)
(/= "texto" "textos")
(/= (* 2 2) (* 2 4) (* 2 3))
```

Devuelve `nil` si todos los valores son iguales, por ejemplo:

```
(/= "casa" "casa" "casa")
(/= "1 2 3" "1 2 3" "1 2 3" "1 2 3" "1 2 3")
(/= 32 32 32 32)
(/= (* 10 10) (* 25 4))
```

Si únicamente se indica un valor, AutoLISP devuelve `T`.

`(< valor1 [valor2...])`

Esta sintaxis se corresponde con la comparación menor que. Es una función AutoLISP que devuelve `T` si efectivamente el primer valor comparado es menor que el segundo. Si existen diversos valores, cada uno ha de ser menor que el siguiente para que AutoLISP devuelva `T`. Si no se devuelve `nil`. Veamos algunos ejemplos:

<code>(< 2 3)</code>	devuelve <code>T</code>
<code>(< 3 4 5 89 100)</code>	devuelve <code>T</code>
<code>(< 3 -4 5 6)</code>	devuelve <code>nil</code>
<code>(< (* 2 2) (/ 5 3))</code>	devuelve <code>nil</code>

En el caso de cadenas o variables alfanuméricas (las que contienen cadenas), la comparación se efectúa según el valor de los códigos ASCII. Por lo tanto, será el orden alfabético ascendente (de la A a la Z) la manera de considerar de menor a mayor los caracteres, teniendo en cuenta que el espacio blanco es el carácter de menor valor y que las letras mayúsculas son de menor valor que las minúsculas. Ejemplos:

<code>(< "a" "b")</code>	devuelve <code>T</code>
<code>(< "z" "h")</code>	devuelve <code>nil</code>
<code>(< "A" "a" "b")</code>	devuelve <code>T</code>
<code>(< "f" "S")</code>	devuelve <code>nil</code>

Si las cadenas tienen más caracteres se comparan de la misma forma:

<code>(< "abc" "abd")</code>	devuelve <code>T</code>
<code>(< "abc" "ab")</code>	devuelve <code>nil</code>

No es posible comparar cadenas literales con números; AutoLISP devuelve un mensaje de error que dice `bad argument type`. Con variables que contienen valores numéricos o literales se realizaría de la misma manera:

```
(< valor1 valor2 total)
(< -12 -7 km hrs)
```

```
(< autor1 autor2 autor3 auto4 autor5)
```

```
(<= valor1 [valor2...])
```

Esta es la función menor o igual que. Funciona de la misma forma que la anterior pero teniendo en cuenta que devolverá T si cada valor es menor o igual que el anterior. Si no devolverá nil. He aquí unos ejemplos:

```
(<= 10 30 30 40 50 50) devuelve T
(<= 12.23 12.23 14)     devuelve T
(<= 56 57 57 55)        devuelve nil
```

Las demás consideraciones son idénticas a las de la función precedente.

```
(> valor1 [valor2...])
```

Al igual que en la comparación de menor que, pero de manera inversa, esta función devuelve T si cada valor especificado, sea numérico sea cadena, es mayor que el siguiente, esto es, si se encuentran ordenados de mayor a menor. Si no devuelve nil. Por ejemplo:

```
(> 10 5 4.5 -2) devuelve T
(> "z" "gh" "ab") devuelve T
(> 23 45)        devuelve nil
```

Otros ejemplos:

```
(> saldo divid)
(> pplanta ppiso pcubierta)
```

```
(>= valor1 [valor2...])
```

Similar a los anteriores, establece la comparación mayor o igual que. Se devolverá T si y sólo si cada valor es mayor o igual que el que le sucede, si no, nil. Las demás consideraciones son idénticas a las otras funciones similares explicadas. Ejemplos:

```
(>= 33 23 23 12 12 54) devuelve nil
(>= 24 24 24 23 23 0.01 -3) devuelve T
```

3ª fase intermedia de ejercicios

- Indicar el resultado de AutoLISP (T o nil) ante las siguientes proposiciones:

```
— (= 23 23.0)
— (= 48.0 (* 6 8))
— (= "AutoLISP" "autolisp" "aUtOlIsP")
— (/= (/ 7 2) (/ 2 7))
— (/= "libro" "libro ")
— (< 3 5 6 (+ 5 -67))
— (<= "A" "A" "bc" "zk" "zk")
— (> "coche" "mesa")
— (>= "coche" "cohecito")
— (>= "cochu" "coche" "coche" "cocha")
— (>= "á" "á" "a")
```


ONCE.4.4. Operaciones lógicas

Además de lo estudiado hasta ahora, existen cuatro operaciones lógicas referidas al álgebra de Boole. Estas operaciones son el Y lógico, el O lógico, la identidad y el NO lógico. Además, existe una quinta función que veremos al final denominada de identidad de expresiones y que es un poco especial.

Las cuatro funciones que vamos a ver actúan como operadores lógicos y devuelven, al igual que las anteriores, únicamente los resultados T (cierto) o nil (falso).

`(AND expresión1 [expresión2...])`

Esta función realiza el Y lógico de una serie de expresiones indicadas que representan otras tantas condiciones. Esto significa que evalúa todas las expresiones y devuelve T si ninguna de ellas es nil. En el momento en que alguna es nil, abandona la evaluación de las demás y devuelve nil. Es decir, se deben cumplir todas y cada una de las condiciones. Veamos un ejemplo:

```
(AND (<= 10 10) (>= 10 10))    devuelve T
```

Esto significa que, si se cumple la condición de la primera lista (<= 10 10) y, además, se cumple la de la segunda lista (>= 10 10) devolverá T. Como esto es así, devuelve T.

De otra forma, si una de las condiciones no se cumple, devuelve nil, por ejemplo en el siguiente caso:

```
(AND (= 10 10) (> 10 10))
```

La primera condición es verdadera (10 es igual a 10), pero la segunda es falsa (10 no es mayor que 10). Como una ya no se cumple se devuelve nil. Han de cumplirse todas las condiciones para que sea el resultado verdadero. Veamos otros dos ejemplos:

```
(AND (= 10 10) (> 23 22.9) (/= "camión" "camioneta"))    devuelve T
(AND (<= "A" "a") (= 5 7))                                devuelve nil
```

No tiene mucho sentido indicar una sola expresión con esta función. Las dos siguientes son idénticas y producen el mismo resultado:

```
(AND (= 20 -20))
(= 20 -20)
```

Ambas devuelven nil.

`(OR expresión1 [expresión2...])`

Realiza un O lógico de una serie de expresiones que representan otras tantas condiciones. Evalúa las expresiones y devuelve nil si todas ellas son nil. En el momento en que encuentre una respuesta distinta de nil, abandona la evaluación y devuelve T. Ésta es precisamente la mecánica del O lógico, es decir, basta que se cumpla una de las condiciones para que la respuesta sea verdadera o cierta.

El siguiente ejemplo compara números y devuelve nil:

```
(OR (< 20 2) (> 20 2))
```

O sea, si es menor 20 que 2 —que no lo es— o si es mayor 20 que dos —que sí lo es—, devuelve T. El cumplirse una de las dos condiciones es condición suficiente para que devuelva T. Veamos otro ejemplo:

```
(OR (= 20 2) (> 2 20))           devuelve nil
```

En este caso ninguna de las dos condiciones se cumplen (ambas son nil), así que el resultado final será nil.

Como en el caso de la función AND, no tiene sentido utilizar una sola expresión, ya que el resultado sería el mismo que al escribirla sola. Veamos otros ejemplos:

```
(OR (>= 30 30 20 -5) (<= -5 -5 -4 0))           devuelve T
(OR (< (* 2 8) (* 2 3)) (= (/ 8 2) (* 4 1)))      devuelve T
(OR (= "carro" "carreta") (= "casa" "caseta") (= 2 3)) devuelve nil
```

Recapitulando, y para afianzar estos dos últimos conocimientos, decir que AND obliga a que se cumplan todas las condiciones para devolver T. Sin embargo, a OR le basta con que una de ellas se cumpla para devolver T. Digamos, en lenguaje coloquial, que AND es “si se cumple esto, y esto, y esto, y... es válido”, y OR es “si se cumple esto, o esto, o esto, o... es válido”.

Veamos ahora otra función lógica para comparar expresiones. Se llama EQUAL y su sintaxis es la siguiente:

```
(EQUAL expresión1 expresión2 [aproximación])
```

Esta función compara las dos expresiones indicadas, si son idénticas devuelve T, si difieren en algo devuelve nil.

A primera vista puede parecer igual a la función = (igual que) estudiada, sin embargo, ésta únicamente comparaba valores; EQUAL tiene la capacidad de poder comparar cualquier expresión o lista de expresiones. De esta forma, podemos utilizar EQUAL de la misma forma que =, así:

```
(EQUAL 2 2)           devuelve T
(EQUAL -3 5)          devuelve nil
```

Pero no tiene mucho sentido, ya que tenemos la función =. Reservaremos EQUAL para lo expuesto, es decir, para la comparación de listas de expresiones.

Así pues, y adelantándonos a algo que veremos un poco más tarde, diremos que la expresión de las coordenadas de un punto 3D se escribiría de la siguiente forma:

```
'(20 20 10)
```

El apóstrofo es la abreviatura de la función QUOTE de AutoLISP, que toma como literales, y sin evaluar, las expresiones que le siguen. De esta forma, para comparar la identidad de dos puntos haríamos, por ejemplo:

```
(EQUAL '(20 20 10) '(20 20 10))           devuelve T
(EQUAL '(20 -5 10) '(20 20 10))          devuelve nil
```

NOTA: La función QUOTE se ve ampliada en la sección **ONCE.5.1**.

El argumento optativo *aproximación* se utiliza cuando se comparan expresiones cuyos resultados son números reales y puede haber una pequeña diferencia decimal que no queramos considerar desigual. Con este argumento suministramos a la función un valor de aproximación decimal respecto al cual se creerán iguales los resultados. Por ejemplo:

```
(EQUAL 23.5147 23.5148)           devuelve nil
(EQUAL 23.5147 23.5148 0.0001)    devuelve T
```

`(NOT expresión)`

La función NOT devuelve el *NO* lógico, es decir, si algo es verdadero devuelve falso y viceversa. Así, cuando el resultado sea distinto de nil (T), devolverá nil; cuando el resultado sea nil, devolverá T. Por ejemplo:

```
(NOT (= 2 2))      devuelve nil
(NOT (/= 2 2))     devuelve T
```

`(EQ expresión1 expresión2)`

Esta función no es propiamente lógica, sino que se denomina de identidad de expresiones. Aún así, la introducimos en este apartado por su similitud con las anteriores.

EQ compara las dos expresiones (sólo dos y ambas obligatorias) indicadas y devuelve T si ambas son idénticas o nil en caso contrario. Se utiliza sobre todo para comparar listas y ver si hay igualdad estructural.

La diferencia de EQ con EQUAL es que ésta última compara los resultados de evaluar las expresiones, mientras que EQ compara la identidad estructural de las expresiones sin evaluar. Por ejemplo, y adelantando la función SETQ que enseguida veremos, podemos hacer lo siguiente:

```
(SETQ list1 '(x y z))
(SETQ list2 '(x y z))
(SETQ list3 list2)
(EQ list1 list2)      devuelve T
(EQ list2 list3)      devuelve nil
```

Se observa que list1 y list2 son exactamente la misma lista por definición, están declaradas con SETQ y por separado, siendo sus elementos iguales. Pero list3 es, por definición, igual a list2 y no a list3, aunque sus elementos sean iguales. Es por ello que, en la segunda evaluación, EQ devuelve nil.

NOTA: Comprenderemos enseguida el funcionamiento y base de la función SETQ, no hay preocuparse.

4ª fase intermedia de ejercicios

- Indicar el resultado de AutoLISP (T o nil) ante las siguientes proposiciones:

```
— (AND (= (* 20 20) (/ 800 2)) (> 300 200 500))
— (AND (>= "a" "a") (>="z" "a") (>= " " " ") (>= " " " "))
— (AND (OR (= 2 2) (> 3 6)) (OR (= 7 5) (= 0 0)))
— (EQUAL (AND (= 1 10) (= 1 1)) (OR (>= 3 2 1 0) (<= 0 -2)))
— (OR (AND (= 1 1) (= 2.0 2)) (OR (NOT (= 1 1)) (= 2.0 2)))
```

NOTA: Apréciese la capacidad de poder anidar y combinar expresiones.

Y hasta aquí llega esta parte de funciones matemáticas, lógicas y de comparación. Probablemente el lector estará pensando que de poco sirve lo expuesto hasta ahora: qué más dará que una expresión matemática me dé un resultado si luego no puedo operar con él; que importará que una proposición lógica me devuelva `T` o `nil` si no me sirve para otra cosa.

Paciencia... En el mundo de la programación hay que empezar desde abajo y, aseguramos que un buen dominio abstracto de lo visto hasta ahora proporcionará un gran nivel de soltura a la hora de programar de verdad. Seguramente, además, aquella persona que sepa programar en algún lenguaje existente, habrá comprendido algo más —ya que todos son muy parecidos—. El neófito comenzará a ver las cosas claras inmediatamente.

A partir de la siguiente sección comenzaremos a ver para qué sirve todo esto y cómo utilizarlo prácticamente en programas propios.

ONCE.5. CREAR Y DECLARAR VARIABLES

Una vez visto lo visto, vamos a ver como podemos introducir valores en variables para no perderlos. A esto se le llama declarar variables.

Una variable es un espacio en memoria donde se guardará, con un nombre que indiquemos, un valor concreto, una cadena de texto, un resultado de una expresión, etcétera. El comando para declarar variables en AutoLISP es `SETQ` y su sintaxis es la que sigue:

```
(SETQ nombre_variable1 expresión1 [nombre_variable2 expresión2...])
```

De esta manera introducimos valores en nombres de variables, por ejemplo:

```
(SETQ x 12.33)
```

Esta proposición almacena un valor real de 12,33 unidades en una variable con nombre `x`.

Al escribir una función `SETQ` atribuyendo a una variable un valor, AutoLISP devuelve dicho valor al hacer `INTRO`. AutoLISP siempre tiene que devolver algo al ejecutar una función.

Como indica la sintaxis, podemos dar más de un valor a más de un nombre de variable a la vez en una función `SETQ`, por ejemplo:

```
(SETQ x 54 y 12 z 23)
```

En este caso, AutoLISP devuelve el valor de la última variable declarada. Esto no es muy recomendable si las expresiones o elementos de la lista son muy complicados, ya que puede dar lugar a errores. A veces, aunque no siempre, es preferible utilizar tantas `SETQ` como variables haya que declarar que hacerlo todo en una sola línea.

Si declaramos una variable que no existía, se crea y se guarda en memoria con su valor; si la variable ya existía cambiará su valor por el nuevo.

NOTA: Al comenzar un dibujo nuevo, abrir uno existente o salir de **AutoCAD**, el valor de las variables se pierde de la memoria.

Podemos, como hemos dicho, atribuir valores de cadena a variables de la siguiente forma:

```
(SETQ ciudad "Bilbao")
```

y combinar cadenas con valores numéricos y/o expresiones:

```
(SETQ ciudad "Bilbao" x (+ 23 45 23) v1 77.65)
```

De esta forma, se guardará cada contenido en su sitio. Las variables que contienen cadenas textuales han de ir entre comillas dobles (" "). A estas variables se las conoce en el mundo de la informática como variables alfanuméricas o cadenas, y pueden contener cualquier carácter ASCII. Las otras variables son numéricas, y únicamente contendrán datos numéricos.

NOTA: De forma diferente a otros lenguajes de programación, en AutoLISP no hay que diferenciar de ninguna manera los nombres de variables numéricas de los de variables alfanuméricas o cadenas.

Para comprobar nosotros mismos el valor de una variable declarada, y como se expuso al principio de este **MÓDULO ONCE**, podemos evaluarla directamente introduciendo su nombre, en la línea de comandos, precedido del carácter de cierre de exclamación (!). Por ejemplo, declaradas las variables anteriores (ciudad, x y v1), podemos examinar su valor de la siguiente manera:

```
!ciudad          devuelve "Bilbao"
!x               devuelve 91
!v1              devuelve 77.65
```

Así pues, imaginemos que queremos escribir unas pequeñas líneas de código que calculen el área y el perímetro de un círculo, según unos datos fijos proporcionados. Podríamos escribir la siguiente secuencia en orden, acabando cada línea con INTRO:

```
(SETQ Radio 50)
(SETQ Area (* PI Radio Radio))
(SETQ Perim (* 2 PI Radio))
```

De esta forma, si ahora tecleamos lo siguiente se producen las evaluaciones indicadas:

```
!area            devuelve 7853.98
!perim           devuelve 314.159
```

NOTA: Como sabemos es indiferente el uso de mayúsculas y minúsculas. Además, decir que podemos (lo podríamos haber hecho con la variable `Area`) introducir tildes y/o eñes en nombres de variable pero, por compatibilidad, es lógico y mucho mejor no hacerlo.

NOTA: Es posible declarar variables con nombres de funciones inherentes de AutoLISP, pero cuidado, si hacemos estos perderemos la definición propia de la misma y ya no funcionará, a no ser que cambiemos de sesión de dibujo. Así mismo, tampoco debemos reasignar valores diferentes a constantes (que en realidad son variables, porque podemos cambiarlas) propias de AutoLISP como `PI`. La siguiente función que veremos nos ayudará a evitar esto.

NOTA: Si queremos ver el valor de una variable no declarada, AutoLISP devuelve `nil`.

Al estar los valores guardados en variables, podemos utilizarlos para otras operaciones sin necesidad de volver a calcularlos. Teniendo en cuenta el último ejemplo, podríamos hacer:

```
(+ area perim)
```

para que devuelva el resultado de la adición de las dos variables. O incluso, podemos guardar dicho resultado en otra variable, para no perderlo, así por ejemplo:

```
(SETQ total (+ area perim))
```

Después podremos comprobar su valor escribiendo `!total`.

Lo que no podemos es realizar, por ejemplo, operaciones matemáticas con variables alfanuméricas entre sí, o con numéricas y alfanuméricas mezcladas (aunque las cadenas contengan números no dejan de ser cadenas textuales). Veamos la siguiente secuencia y sus resultados:

(SETQ x 34)	devuelve 34
(SETQ y "ami")	devuelve "ami"
(SETQ z "guitos")	devuelve "guitos"
(SETQ w "12")	devuelve "12"
(SETQ p 10)	devuelve 10
(+ x p)	devuelve 44
(+ p y)	devuelve error: bad argument type
(+ x w)	devuelve error: bad argument type
(+ y z)	devuelve error: bad argument type

En otros lenguajes de programación podemos concatenar cadenas de texto con el símbolo de suma `+`, en AutoLISP no. AutoLISP ya posee sus propios mecanismos —que ya estudiaremos— para realizar esta función. Tampoco podemos, como vemos, operar con cadenas y valores numéricos, sean como sean y contuvieren lo que contuvieren.

Veamos algunos ejemplos más de `SETQ`:

```
(SETQ ancho (* l k) largo (+ x1 x2) alto (* ancho 2))
```

NOTA: Como vemos, podemos definir una variable con una expresión que incluya el nombre de otra definida anteriormente, aunque sea en la misma línea.

```
(SETQ x (= 20 20))
```

Esta variable `x` guardaría el valor verdadero (`T`).

```
(SETQ zon (* (/ 3 2) 24 (EXPT 10 4)))  
(SETQ f (1+ f))
```

Este último ejemplo es lo que se denomina, en el mundo de la programación informática, un contador-suma. Guarda el valor de `f` más una unidad en la propia variable `f` (se autosuma 1).

Cambiando a otra cosa, vamos a comentar la posibilidad de perder la definición de una función AutoLISP por declarar una variable con su nombre. Existe una función que muestra todos los símbolos actuales definidos. Esta función es:

```
(ATOMS-FAMILY formato [lista_símbolos])
```

`ATOMS-FAMILY`, como decimos, muestra una lista con todos los símbolos definidos actualmente. En esta lista entran tanto las *subrs* (funciones inherentes) de AutoLISP como las funciones y variables definidas y declaradas por el usuario cargadas en la actual sesión de dibujo. De esta forma podemos consultar dicha lista para ver si tenemos la posibilidad de dar ese nombre de variable que estamos pensando. Ahí tendremos todas las funciones propias e inherentes, además de las variables ya creadas.

Como podemos observar en la sintaxis, esta función necesita un parámetro o argumento obligatorio llamado *formato*. *formato* sólo puede tomar dos valores: 0 ó 1. Si es 0, los símbolos se devuelven en una lista, separando cada nombre de otro por un espacio en blanco. Si es 1, los símbolos se devuelven entre comillas (separados también por espacios blancos) para su mejor comparación y examen. Cuestión de gustos; la verdad es que no se encuentra un símbolo tan fácilmente entre la marabunta de términos.

Pero con el argumento optativo podemos depurar o filtrar al máximo la búsqueda; de esta manera es de la que más se utiliza. Con *lista_símbolos* hacemos que se examinen solamente los nombres que incluyamos en la lista. Estos símbolos habrán de ir encerrados entre comillas y ser precedidos del apóstrofo (') por ser una lista literal. El resultado es otra lista en la que, los símbolos ya existentes aparecen en su sitio, en el mismo lugar de orden donde se escribieron y, en el lugar de los no existentes aparece *nil*.

Si por ejemplo queremos saber si el nombre de variable *total* existe ya como símbolo, sea función inherente, propia o variable ya declarada, y deseamos el resultado como simple lista escribiremos:

```
(ATOMS-FAMILY 0 '("total"))
```

y AutoLISP, si no existe el símbolo, devolverá:

```
(nil)
```

Si aún no hemos declarado ninguna variable y escribimos:

```
(ATOMS-FAMILY 0 '("tot" "setq" "w" ">=" "sqrt" "suma"))
```

AutoLISP devolverá:

```
(nil SETQ nil >= SQRT nil)
```

Y si lo escribimos así (con 1 para *formato*):

```
(ATOMS-FAMILY 1 '("tot" "setq" "w" ">=" "sqrt" "suma"))
```

AutoLISP devolverá:

```
(nil "SETQ" nil ">=" "SQRT" nil)
```

ONCE.5.1. A vueltas con el apóstrofo (')

Ya hemos utilizado un par de veces este símbolo y, también, hemos explicado por encima su función. Vamos ahora a ampliar esa información.

El símbolo de apóstrofo (') no es otra cosa, como ya se comentó, que una abreviatura de la función *QUOTE* de AutoLISP. Dicha función tiene la siguiente sintaxis de programación:

<code>(QUOTE expresión)</code>

o también:

<code>('expresión)</code>

NOTA: Nótese que tras `QUOTE` hay un espacio pero, si se utiliza el apóstrofo no hay que introducirlo.

Esta función se puede utilizar con cualquier expresión de AutoLISP. Lo que hace es evitar que se evalúen los símbolos y los toma como literales. Devuelve siempre el literal de la expresión indicada, sin evaluar. Por ejemplo:

<code>(QUOTE (SETQ x 22.5))</code>	devuelve <code>(SETQ x 22.5)</code>
<code>(QUOTE hola)</code>	devuelve <code>HOLA</code>
<code>(QUOTE (+ 3 3 3))</code>	devuelve <code>(+ 3 3 3)</code>

Hay que tener cuidado al utilizar el apóstrofo de abreviatura de `QUOTE`, ya que desde la línea de comandos no lo vamos a poder utilizar. Recordemos que **AutoCAD** sólo reconoce que estamos escribiendo algo en AutoLISP en la línea de comandos cuando comenzamos por el paréntesis de apertura `(`, o a lo sumo por la exclamación final `!`, para evaluar variables directamente. Expresiones como las siguientes:

```
'(DEFUN diblin () "Nada")
'a
'var12
```

sólo podremos introducirlas desde un archivo ASCII (como veremos en seguida).

Pues este comando es muy utilizado a la hora de introducir directamente, por ejemplo, las coordenadas de un punto, ya que estas coordenadas son en el fondo una lista y que no ha de ser evaluada. Por ejemplo `'(50 50)`.

Lo mismo nos ha ocurrido con la lista de `ATOMS-FAMILY`. Ésta no ha de evaluarse (no tiene otras funciones añadidas, es simplemente un grupo de cadenas), por lo que ha de introducirse como literal.

Una lista que no tiene función añadida, por ejemplo `(50 50 -23)` produce un error de `bad function` en AutoLISP, a no ser que se introduzca como literal:

<code>(QUOTE (50 50 -23))</code>	devuelve <code>(50 50 -23)</code>
----------------------------------	-----------------------------------

NOTA: En la mayoría de las funciones de AutoLISP, al introducir un literal de expresión la haremos con el apóstrofo directamente, ya que con `QUOTE` no funcionará. `QUOTE` sólo tendrá validez cuando se utilice solo, sin más funciones.

ONCE.6. PROGRAMANDO EN UN ARCHIVO ASCII

Hasta ahora hemos visto muchos ejemplos de funciones en AutoLISP, pero todos ellos los hemos tecleado desde la línea de comandos de **AutoCAD**. Esto resulta un poco engorroso, ya que si quisiéramos volver a teclearlos tendríamos que escribirlos de nuevo. Sabemos que existe la posibilidad de copiar y pegar en línea de comandos, aún así es pesado tener que volver a copiar y pegar cada una de las líneas introducidas.

Existe la posibilidad de crear archivos ASCII con una serie de funciones AutoLISP (programa) que se vayan ejecutando una detrás de otra al ser cargado, el programa, en **AutoCAD**. Ésta es la verdadera forma de trabajar con AutoLISP. La escritura en línea de comandos está relegada a pruebas de funcionamiento de funciones.

Con este método, no sólo tenemos la posibilidad de editar una línea y correrlas (ejecutarlas) bajo **AutoCAD**, sino que además podremos elaborar programas extensos que

tendremos la posibilidad de cargar desde disco en cualquier sesión de dibujo, en cualquier momento.

Incluso, como veremos, es factible la creación de órdenes o comandos para **AutoCAD** que, siendo no otra cosa que programas en AutoLISP, podremos ejecutar con sólo teclear su nombre. Estos programas manejarán la Base de Datos de **AutoCAD**, operarán con objetos de dibujo, utilizarán cuadros de diálogo o no como interfaz, y un larguísimo etcétera. La programación en AutoLISP, unida a estructuras de menús, tipos de línea, patrones de sombreado y demás estudiado en este curso, nos permitirá llegar a crear verdaderas aplicaciones verticales para **AutoCAD**.

Pero para desarrollar un programa en un archivo ASCII y luego poder cargarlo en **AutoCAD**, no debemos simplemente escribir las expresiones que ya hemos aprendido y punto. Hay que seguir una lógica y hay que indicarle a **AutoCAD**, al principio del programa, que estamos escribiendo un programa en AutoLISP, precisamente.

Un archivo ASCII puede contener varios programas o funciones de usuario en AutoLISP. Se suelen escribir procurando no sobrepasar los 80 caracteres por línea para su edición más cómoda y, además, se suelen sangrar en mayor o menor medida las entradas de algunas líneas, dependiendo de la función —ya nos iremos familiarizando con esto— para dar claridad al programa.

Un programa de AutoLISP se compone de una serie de funciones AutoLISP que se ejecutan una detrás de la otra produciendo diferentes resultados. El caso sería el mismo que ir introduciendo renglón a renglón en la línea de comandos. Pero en un archivo ASCII hay que introducir todas las funciones dentro de la lista de argumentos de otra que las engloba. Esta función es DEFUN y su sintaxis es:

`(DEFUN nombre_función lista_argumentos expresión1 [expresión2...])`

DEFUN define una función de usuario. Su paréntesis de apertura es lo primero que debe aparecer en un programa AutoLISP y su paréntesis de cierre lo último tras todas las funciones intermedias (después puede haber otros DEFUN).

nombre_función es el nombre que le vamos a dar a nuestra función y *lista_argumentos* es una lista de argumentos globales o locales para la función. Los argumentos o variables globales son aquellos que se almacenan en memoria y permanecen en ella; son todas las variables que hemos definiendo hasta ahora. Estas variables pueden ser utilizadas por otros programas AutoLISP o ser evaluadas directamente en línea de comandos mediante el carácter !.

Los símbolos locales son variables temporales. Estas se almacenan en memoria sólo de manera temporal, hasta que se termina la función en curso. Una vez ocurrido esto desaparecen y no pueden ser utilizados por otros programas ni evaluados en línea de comandos. Estos símbolos locales han de estar indicados en la lista después de una barra (/). Esta barra tiene que estar separada del primer símbolo local por un espacio en blanco y del último símbolo global —si lo hubiera— por un espacio blanco también. Veamos unos ejemplos:

<code>(DEFUN func (x)...</code>	variable global: x
<code>(DEFUN func (x y)...</code>	variables globales: x y
<code>(DEFUN func (x / u z)...</code>	variable global: x variables locales: u z
<code>(DEFUN func (/ x s)...</code>	variables locales: x s

Si el símbolo local se encontrara ya creado antes de ser utilizado en la función definida, recupera el valor que tenía al principio una vez terminada la función. Si no se especifican

como locales al definir una función, todos los símbolos declarados con `SETQ` dentro de ella son globales.

NOTA: De momento vamos a olvidarnos de variables globales y locales, ya que todas las funciones que definamos por ahora tendrán una lista de argumentos vacía. Más adelante se profundizará en este tema.

Después de esto, aparecerán todas las expresiones del programa, o sea, las funciones de AutoLISP o de usuario ya definidas que formen el conjunto del programa. Al final, deberá cerrarse el paréntesis de `DEFUN`.

Así pues, ya podemos crear nuestro primer programa en AutoLISP. Este programa calculará la raíz cuadrada de un número, definidos anteriormente en una variables. Veamos cómo es el pequeño programa:

```
(DEFUN () Raiz
  (SETQ X 25)
  (SQRT X)
)
```

Vamos a comentarlo un poco. Definimos, lo primero, la función llamada `Raiz` con una lista de argumento vacía. A continuación, asignamos con `SETQ` el valor 25 a la variable `x` y calculamos su raíz cuadrada. Al final, cerramos el paréntesis de `DEFUN`. Simple.

NOTA: La razón para sangrar las líneas se debe a la comodidad de ver qué paréntesis cierran a qué otros. De un golpe de vista se aprecia perfectamente.

NOTA: Es irrelevante la utilización de mayúsculas o minúsculas en la programación en AutoLISP (excepto en cadenas literales, lógicamente).

Podíamos haber hecho el programa sin variable, simplemente poniendo el valor tras la función de la raíz cuadrada, pero es otro modo de recordar y practicar. Escribámoslo y guardémoslo con extensión `.LSP`. Como nombre es recomendable darle el mismo que a la función, es decir, que el nombre del archivo quedaría así: `RAIZ.LSP`. Esto no tiene por qué sentar cátedra.

Vamos ahora a cargar nuestra nueva función en **AutoCAD**. El procedimiento es sencillo y siempre el mismo. Desde *Herr.>Cargar aplicación...* accedemos al *cuadro Cargar archivos AutoLISP, ADS y ARX*. En este cuadro, pinchando en *Archivo...* se nos abre un nuevo cuadro para buscar y seleccionar el archivo. Tras seleccionarlo (y pulsar *Abrir*) volveremos al cuadro anterior donde pulsaremos el botón *Cargar*. De esta forma cargamos el archivo para poder ser utilizado.

NOTA: Si en este cuadro comentado activamos la casilla *Guardar lista*, tendremos accesibles en la lista *Archivos a cargar* todos los archivos cargados desde la activación de la casilla. De esta forma podremos modificar un archivo `.LSP` y, rápidamente, volver a cargarlo escogiéndolo de esta lista y pulsando *Cargar*. Realmente la lista se guarda en un archivo llamado `APpload.DFS` y que estará guardado en el directorio al que haga referencia el acceso directo que arranca **AutoCAD** en su casilla *Iniciar en:*.

El botón *Descargar* descarga de memoria la aplicación designada y, el botón *Suprimir*, elimina una entrada de la lista.

NOTA: Este cuadro de diálogo aparece también con el comando `APpload` de **AutoCAD**.

Una vez cargada la función sólo queda ejecutarla. Para ello deberemos indicarla entre paréntesis, esto es (en la línea de comandos):

(RAIZ)

y **AutoCAD** devuelve:

2.23607

La razón de que haya que ejecutarlas entre paréntesis es porque es una función AutoLISP; es una función definida por el usuario, pero no deja de ser AutoLISP. Pero existe una forma de no tener que escribir los paréntesis para ejecutar una nueva orden de usuario. Esta forma consiste en colocar justo delante del nombre de la nueva función los caracteres C: (una "c" y dos puntos). De la siguiente manera quedaría con el ejemplo anterior:

```
(DEFUN ( ) C:Raiz
  (SETQ X 25)
  (SQRT X)
)
```

Así, únicamente habríamos de escribir en la línea de comandos:

RAIZ

para que devuelva el mismo resultado. De esta forma, RAIZ es un nuevo comando totalmente integrado en **AutoCAD**, el cual podríamos ejecutar desde la línea de comandos o hacer una llamada a él desde un botón de una barra de herramientas, o desde una opción de menú, etcétera.

NOTA: Las funciones definidas mediante este método no admiten variables globales, sólo locales.

NOTA: Las mayúsculas o minúsculas son también irrelevantes a la hora de llamar a un función de usuario, al igual que ocurre con los comandos de **AutoCAD**.

5ª fase intermedia de ejercicios

- Realizar un programa AutoLISP que calcule la suma de los diez primeros números.
- Realizar un programa que compare valores mayores.
- Realizar un programa que asigne valores a 3 variables y luego las multiplique todas entre sí.

ONCE.7. CAPTURA Y MANEJO BÁSICO DE DATOS

ONCE.7.1. Aceptación de puntos

Tras lo estudiado parece ser que vamos entrando poco a poco y de lleno en el mundo de la programación en AutoLISP. Sin embargo, aún puede parecernos algo ilógico el poder realizar un programa que calcule una serie operaciones con cantidades fijas, sin poder variar de números cada vez que se ejecute el programa, por ejemplo. En esta sección **ONCE.7.** vamos a aprender la forma que tenemos de pedirle datos al usuario para luego operar con ellos. Comenzaremos por los puntos.

Todo lo que se refiere a captura de datos, tiene en AutoLISP un nombre propio que es *GET*. . . . Si nos damos cuenta, se ha indicado con punto suspensivos porque "*GET*" como tal no

existe como función, sino una serie de ellas que comienzan con esas letras. Pues bien, todas estas funciones del tipo *GET*... nos proporcionarán la posibilidad de preguntar al usuario por un texto, por el valor de una distancia, por la situación de un punto, etc. para luego operar a nuestro antojo con dichos valores.

La primera función de este tipo que vamos a estudiar tiene la sintaxis:

```
(GETPOINT [punto_base] [mensaje])
```

GETPOINT solicita un punto al usuario. Esta función aguarda a que se introduzca un punto, bien sea por teclado o señalando en pantalla como habitualmente lo hacemos con **AutoCAD**, y devuelve las coordenadas de dicho punto en forma de lista de tres valores reales (X, Y y Z). Para probarla podemos escribir en la línea de comandos:

```
(GETPOINT)
```

A continuación, señalamos un punto (o lo digitamos) y AutoLISP devuelve las coordenadas de dicho punto. Estas coordenadas, como hemos dicho, están en forma de lista, es decir, entre paréntesis y separadas entre sí por espacios en blanco (es una típica lista de AutoLISP como hemos visto alguna ya).

La potencia de esta función se desarrolla al guardar las coordenadas indicadas en una variable, para que no se pierdan. En el momento en que capturamos los datos y los almacenamos en una variable ya podemos utilizarlos posteriormente. Para almacenar los datos utilizaremos la función SETQ estudiada, de la siguiente manera por ejemplo:

```
(DEFUN C:CapturaPunto ()  
  (SETQ Punto (GETPOINT))  
)
```

Como sabemos, para ejecutar esta nueva orden habrá que escribir en la línea de comandos de **AutoCAD**:

```
CAPTURAPUNTO
```

Con el argumento opcional *mensaje* de GETPOINT tendremos la posibilidad de incluir un mensaje en la línea de comandos a la hora de solicitar un punto. Así, podríamos variar un poco el programa anterior de la siguiente manera:

```
(DEFUN C:CapturaPunto ()  
  (SETQ Punto (GETPOINT "Introducir un punto: "))  
)
```

De esta forma se visualizará el mensaje indicado (siempre entre comillas) a la hora de solicitar el punto.

El argumento *punto_base* permite introducir un punto base de coordenadas (2D ó 3D), a partir del cual se visualizará una línea elástica hasta que indiquemos un punto. Viene a ser algo así como la manera de dibujar líneas en **AutoCAD**: se indica un punto y la línea se "engancha" a él hasta señalar el segundo. De todas formas no tiene nada que ver. Para indicar este punto de base lo podemos hacer mediante una variable que contenga un punto o directamente con una lista sin evaluar (con apóstrofo), como vimos:

```
(GETPOINT '(50 50) "Introducir un punto: ")
```

NOTA: Apréciase el espacio tras ...punto: . Es puramente decorativo. Produciría mal efecto al aparecer en pantalla el mensaje si no estuviera este espacio. Pruébese.

Pero, ¿qué hacemos ahora con este punto? Hemos comenzado a ver la manera de obtener datos del usuario, pero poco podremos hacer si no somos capaces de procesarlos después, al margen de las típicas —que no inútiles— operaciones matemáticas y de comparación. Para avanzar un poco más, vamos a hacer un inciso en la manera de capturar datos y vamos a ver la función `COMMAND` de AutoLISP.

La función `COMMAND` permite llamar a comandos de **AutoCAD** desde AutoLISP. Sus argumentos son las propias órdenes de **AutoCAD** y sus opciones correspondientes. La manera de indicarle estas órdenes y opciones del programa a la función `COMMAND` es entre comillas dobles (" "), aunque también podremos indicar puntos en forma de lista (o no), valores en formato de expresión matemática y otros. La sintaxis de `COMMAND` es la siguiente:

```
(COMMAND [comando] [opciones...])
```

Así por ejemplo, podemos ejecutar la siguiente función desde la línea de comandos:

```
(COMMAND "linea" '(50 50) '(100 100) "")
```

Esto es, ejecutar el comando `LINEA`, darle 50,50 como primer punto y 100,100 como segundo punto. Al final, un `INTRO` (" ") para acabar la orden. La base es exactamente la misma que cuando escribíamos la macro de un botón: hay que ir escribiendo comandos y opciones como si fuera directamente en línea de comandos. La diferencia es que no hay que introducir ningún carácter para indicar un `INTRO`, simplemente al escribir "`LINEA`" se ejecuta el comando, o al escribir `'(50 50)` se introduce el punto. Es por ello que, al final haya que escribir un par de comillas dobles (sin espacio intermedio) para acabar la orden `LINEA`, y es que estas comillas indican un `INTRO`.

Como vemos, la manera de escribir las coordenadas de un punto es mediante un lista sin evaluar (con apóstrofo). Pero es perfectamente lícito (sólo con la función `COMMAND`) introducirlas como algo que se escribiría por teclado, es decir, de la siguiente forma:

```
(COMMAND "linea" "50,50" "100,100" "")
```

como ocurre con el comando `LINEA`. Esto no lo podremos hacer con el resto de funciones.

NOTA: Al igual que en las macros y en los menús, sería más recomendable, por aquello del soporte idiomático del programa en AutoLISP, escribir funciones como la anterior de esta otra forma: `(COMMAND "_line" '(50 50) '(100 100) "")`.

Así pues, podríamos reciclar nuestro ejemplo de `GETPOINT` de la siguiente forma:

```
(DEFUN C:DibCirc ()  
  (SETQ Centro (GETPOINT "Introducir un punto: "))  
  (COMMAND "_circle" Centro "10")  
)
```

Este programa pedirá un punto al usuario y dibujará un círculo de radio 10 con centro en dicho punto. Sencillo.

NOTA: Si damos un nombre de un comando de **AutoCAD** a una función definida por nosotros, recordar lo explicado en el **MÓDULO NUEVE** sobre redefinición de órdenes. Y si queremos realizar un programa que sea totalmente compatible con todas las versiones idiomáticas de **AutoCAD** y, además, evitar la posibilidad de que en una máquina haya órdenes predefinidas, utilizaremos los comandos con el guión de subrayado y el punto juntos, de la forma: `_.line`.

NOTA: Las órdenes de **AutoCAD** que leen directamente información del teclado, como TEXTODIN (DTEXT) o BOCETO (SKETCH), no funcionan correctamente con la función COMMAND, por lo que no se pueden utilizar. Si se utiliza una llamada a la orden SCRIPT mediante COMMAND deberá ser la última llamada.

Al principio de este **MÓDULO** vimos que existían tres variables o símbolos predefinidos de AutoLISP. Entre ellas estaba PAUSE, y dijimos que se utilizaba con la función COMMAND. La forma de hacerlo es introducir este símbolo predefinido como argumento de COMMAND, esto hará que el comando en curso, al que haya llamado la función, se interrumpa para que el usuario introduzca algún dato. La mecánica es la misma que se utilizaba al escribir un carácter de contrabarra en las macros de los menús o los botones de barras de herramientas. Por ejemplo:

```
(COMMAND "_circle" '(50 50) pause)
```

Este ejemplo situará el centro de un círculo en el punto de coordenadas 50,50 y esperará a que el usuario introduzca el radio (o diámetro), sea por teclado o indicando en pantalla. Podemos hacer zooms, encuadres y demás (siempre transparentes) hasta introducir lo solicitado, momento en el cual se devolverá el control a la función COMMAND y terminará el comando.

NOTA: De hecho, el símbolo PAUSE contiene el valor predefinido de contrabarra. Únicamente deberemos evaluarlo en línea de comandos (!pause) para comprobarlo. El resultado de esta evaluación será "\\ ", ya que \\, como está indicado en el punto 8. de la sección **ONCE.2.3.** (en este **MÓDULO**), es el código para el carácter contrabarra. Por compatibilidad, podemos introducir la cadena "\\ " en lugar de PAUSE con funciones COMMAND.

Terminado el inciso de la función COMMAND, vamos a seguir explicando otra función similar a GETPOINT. Esta nueva se llama GETCORNER y su sintaxis es la siguiente:

```
(GETCORNER punto_base [mensaje])
```

La misión de GETCORNER es exactamente la misma que la de GETPOINT (solicitar y aceptar un punto), la única diferencia es la forma de visualizar dinámicamente el arrastre. Con GETCORNER, en lugar de ser una línea elástica (como ocurría con GETPOINT con punto base), es un rectángulo elástico. Esto nos lleva a deducir que esta función necesita obligatoriamente que se indique un punto de base para el rectángulo (vemos en la sintaxis que es argumento obligatorio). Así:

```
(GETCORNER '(50 50))
```

situará la esquina primera del rectángulo elástico en coordenadas 50,50 y esperará que se señale, o se indique por teclado, el punto opuesto por la diagonal. Devolverá el punto señalado por el usuario en forma de lista.

El punto base se expresa respecto al SCP actual. Si se indica un punto de base 3D no se tiene en cuenta su coordenada Z, evidentemente: siempre se toma como tal el valor actual de la elevación.

El argumento *mensaje* funciona de la misma forma que con GETPOINT, es decir, escribe el texto en línea de comandos al solicitar el punto. Veamos un pequeño ejemplo con esta función:

```
(DEFUN C:Caja ()  
  (SETQ Esq1 '(100 100))
```

```
(SETQ Esq2 (GETCORNER Esq1 "Indique 2º punto de la diagonal del  
rectángulo: "))  
(COMMAND "rectang" Esq1 Esq2)  
)
```

Este ejemplo dibuja un rectángulo cuya diagonal se sitúa entre el punto 100,100 y el designado por el usuario. Al final, AutoLISP devuelve `nil`. Esto no significa que haya habido algún fallo, sino que, como dijimos, AutoLISP siempre ha de devolver algo, cuando no hay nada que devolver, el resultado será `nil`.

La separación en dos de la tercera línea es únicamente problema de espacio en estas páginas. Al escribirlo en un archivo ASCII deberemos hacerlo todo seguido, en este caso. En otros casos, si el mensaje que presentaremos en pantalla excede el número de caracteres que caben en la línea de comandos, podemos recurrir al código `\n`, expuesto al principio de este **MÓDULO** con el resto de los códigos admitidos. `\n` representa un salto de línea con retorno de carro, pero no un `INTRO`. De esta forma, el programa anterior mostraría la siguiente línea en pantalla:

```
Indique 2º punto de la diagonal del rectángulo:
```

Pero si lo escribimos de la siguiente forma, por ejemplo:

```
(DEFUN C:Caja ()  
  (SETQ Esq1 '(100 100))  
  (SETQ Esq2 (GETCORNER Esq1 "Indique 2º punto\nde la diagonal\ndel  
rectángulo: "))  
  (COMMAND "rectang" Esq1 Esq2)  
)
```

mostrará:

```
Indique 2º punto  
de la diagonal  
del rectángulo:
```

NOTA IMPORTANTE DE SINTAXIS: Mientras no se indique lo contrario, si se separan las líneas en la escritura de los programas de estas páginas, es exclusivamente por falta de espacio. En la práctica, al escribir un programa en un editor ASCII, cada vez que damos un `INTRO` para saltar a la línea siguiente, para el intérprete de AutoLISP es un espacio en blanco. Por eso si escribimos lo siguiente:

```
(DEFUN C:MiProg  
  (SETQ X 5)  
  (COMM  
  AND "linea" X '(10 10) "")  
)
```

el resultado de la tercera línea, que podemos ver en el historial de la línea de comandos pulsando `F2` para conmutar a pantalla de texto, será el siguiente:

```
(COMM AND "linea" X '(10 10) "")
```

lo que producirá un error `null function` de AutoLISP. Sin embargo, si el programa fuera:

```
(DEFUN C:MiProg  
  (SETQ X 5)  
  (COMMAND  
  "linea" X '(10 10) ""))
```

)

y siempre que tras COMMAND no exista ningún espacio, el resultado sería:

```
(COMMAND "línea" X '(10 10) "")
```

que es perfectamente correcto. Si lo que queremos es separar en líneas textos literales que aparecerán por pantalla (por que no caben en una sola línea), utilizaremos el código \n explicado. Por lo general, escribiremos todas las líneas seguidas en el archivo de texto, a no ser que nos resulte incómoda su extremada longitud para la edición.

ONCE.7.2. Captura de datos numéricos

Siguiendo con las funciones de solicitud de datos, vamos a pasar ahora a explicar cómo preguntar por datos numéricos al usuario. Para este tipo de misión disponemos en AutoLISP de dos funciones, GETINT y GETREAL.

```
(GETINT [mensaje])
```

La función GETINT —cuya sintaxis se indica— solicita y acepta un número entero introducido por el usuario. El valor de dicho número ha de estar comprendido entre -32768 y 32767. Si se introduce un valor real o un dato no numérico, AutoLISP devuelve un mensaje de error indicando que ha de ser un número entero y solicita un nuevo número. El mensaje de error proporcionado es similar (aunque no igual) al que produce el comando MATRIZ (ARRAY en inglés) de **AutoCAD** al introducir un número con decimales (real) cuando pregunta por número de filas o de columnas.

mensaje proporciona la posibilidad de escribir un mensaje a la hora de solicitar el valor; es opcional. Como todos los textos literales y cadenas, el mensaje indicado irá encerrado entre comillas dobles. Un ejemplo:

```
(GETINT "Introduzca el número de vueltas de la rosca: ")
```

```
(GETREAL [mensaje])
```

GETREAL es totalmente similar a la función anterior, salvo que acepta números reales. Estos números pueden tener todos los decimales que se quiera introducir, separado de la parte entera por el punto decimal (.). Si se introduce un número entero se toma como real, es decir, con un decimal igual a 0 (28 = 28.0) y, si se introduce un carácter no numérico se produce un error de AutoLISP, proporcionando la opción de repetir la entrada. El argumento mensaje funciona igual que con GETINT.

Veamos un ejemplo de un pequeño programa con GETINT y GETREAL:

```
;Programa que realiza el producto
;entre un número entero y un número real.
(DEFUN C:Producto (); Comienzo de la función de usuario.
  (SETQ Ent (GETINT "Introduzca un número entero: ")); Número entero.
  (SETQ Real (GETREAL "Introduzca un número real: ")); Número real.
  (* Ent Real); Producto.
); Fin de función de usuario.
;Fin del programa
```

Como vemos, los comentarios (precedidos del carácter ;) se pueden incluir en cualquier parte del programa. Como se explicó en el punto 10. de la sección **ONCE.2.3.**,

también podemos incluir comentarios en medio de las líneas utilizando los caracteres `;` para la apertura y `;` para el cierre (son los caracteres de punto y coma y barra vertical). De la siguiente forma:

```
(SETQ X ;| se guarda en x |; 5 ;|el valor 5|;)
```

O incluso en varias líneas:

```
(SETQ X ;| se guarda  
en x |; 5 ;|el valor 5|;)
```

NOTA: Al contrario de cómo ocurría en los archivos ASCII de personalización, en un archivo de código AutoLISP no se hace necesario un `INTRO` al final de la última línea para que funcione el programa. Aunque no viene mal introducirlo por comodidad y para no perder la costumbre.

ONCE.7.3. Distancias y ángulos

Las tres funciones siguientes nos permitirán solicitar distancias y ángulos al usuario. La función `GETDIST` acepta el valor de una distancia introducida y su sintaxis es la siguiente:

```
(GETDIST [punto_base] [mensaje])
```

El valor de la distancia puede ser introducida por teclado o directamente indicando dos puntos en pantalla, como muchas distancias en **AutoCAD**. Si se introduce por teclado el formato ha de ser el establecido por el comando `UNIDADES (UNITS)`. Pero independientemente de este formato, `GETDIST` devuelve siempre un número real.

mensaje funciona como en todas las funciones explicadas. Y *punto_base* permite incluir un punto de base a partir del cual se visualizará una línea elástica hasta introducir un segundo punto para la distancia.

Veamos un ejemplo con `GETDIST`:

```
(DEFUN C:Circulo2 ()  
  (SETQ Centro (GETPOINT "Introduzca el centro del círculo: "))  
  (SETQ Radio (GETDIST Centro "Introduzca el radio del círculo: "))  
  (COMMAND "_circle" Centro Radio)  
)
```

Este ejemplo pide el centro de un futuro círculo y, al pedir el radio ya está “enganchado” a dicho centro; se introduce el segundo punto del radio y el círculo se dibuja. Al final AutoLISP devuelve `nil`.

NOTA: Pruébese que podemos utilizar los modos de referencia a objetos (Punto Final, Punto Medio, Centro...), los filtros (`.XY`, `.YZ`...) y demás con todos los pequeños programas que estamos aprendiendo a hacer.

```
(GETANGLE [punto_base] [mensaje])
```

`GETANGLE` espera a que el usuario introduzca un ángulo y devuelve su valor. Dicho ángulo puede ser introducido por teclado —según formato actual de `UNIDADES (UNITS)`— o mediante dos puntos en pantalla con el cursor. El valor devuelto siempre será un número real en radianes. Hay que tener en cuenta que los ángulos se devuelven considerando como origen el indicado en la variable de **AutoCAD** `ANGBASE`, pero medidos en el sentido antihorario

(independientemente de lo que especifique la variable `ANGDIR`). Se utiliza esta función sobre todo para medir ángulos relativos.

NOTA: El orden de introducción de los puntos (si se hace con el dispositivo señalador) influye en el ángulo medido. Por ejemplo, si desde un punto A a otro B se miden 30 grados, desde el punto B al A se medirán 210 grados.

Si se indica un punto base se muestra la típica línea elástica. Si se escribe un punto de base 3D, el ángulo se mide sobre el plano XY actual únicamente. Si no se indica punto de base se solicitan los dos puntos y se calcula el ángulo de la línea que une ambos en radianes.

`mensaje` funciona como en las funciones anteriores. Veamos un pequeño ejemplo:

```
(DEFUN C:GiraSCP ()
  (SETQ AngRad (GETANGLE "Introduzca un ángulo: "))
  (SETQ AngGrad (/ (* AngRad 180) PI))
  (COMMAND "_ucs" "_x" AngGrad)
)
```

El programa solicita el ángulo para imprimir un giro al SCP con respecto al eje X y lo guarda en `AngRad` (como sabemos el resultado de `GETANGLE` es en radianes). Después guarda en `AngGrad` la conversión del ángulo pedido a grados sexagesimales. Por último, gira el SCP el ángulo en cuestión alrededor del eje X.

`(GETORIENT [punto_base] [mensaje])`

La función inherente a AutoLISP `GETORIENT` funciona de forma parecida a la anterior. La diferencia con `GETANGLE` estriba en que, `GETORIENT` devuelve los ángulos con el origen 0 grados siempre en la posición positiva del eje X del SCP actual y el sentido positivo antihorario, independientemente de los valores de las variables `ANGBASE` y `ANGDIR` de **AutoCAD**. Se utiliza esta función sobre todo para medir ángulos absolutos.

Al igual que con `GETANGLE`, el valor devuelto es siempre en radianes y, si el punto de base es 3D, el ángulo se mide sobre el plano XY actual.

Para comprender bien la diferencia entre ambas funciones de captura de ángulos vamos a ver un ejemplo simple. Si tuviéramos el origen de ángulos definido en el eje Y negativo y el sentido positivo como horario, lo que entendemos por un ángulo de 45 grados (con respecto a la horizontal), produciría un valor de 45 grados con la función `GETORIENT` y un valor de 135 grados con la función `GETANGLE` (ambos en radianes).

Si indicamos dos puntos en pantalla que unidos describan una línea a 45 grados (con respecto a la horizontal), el ángulo se mide desde el origen indicado en `UNIDADES (UNITS)` con `GETANGLE` y desde el lado positivo del eje X con `GETORIENT` (las 3 de la esfera de un reloj) hasta dicha línea y siempre en sentido antihorario (con ambas funciones). De ahí los dos tipos de resultado.

Evidentemente, si indicamos un ángulo por teclado el resultado siempre será el mismo.

El ejemplo de la función anterior puede aplicarse a ésta. Habremos de tener mucho cuidado a la hora de entrar los ángulos señalando puntos, debido a las características de ambas funciones, ya que pueden generar resultados erróneos de giro del SCP.

ONCE.7.4. Solicitud de cadenas de texto

Con AutoLISP también tenemos la posibilidad de solicitar, y posteriormente procesar, cadenas de texto. La función para realizar esto es `GETSTRING`. Podemos ver su sintaxis a continuación:

`(GETSTRING [T] [mensaje])`

`GETSTRING` acepta una cadena de caracteres introducida por teclado y devuelve dicha cadena, precisamente en forma de cadena (entre comillas). Ejemplo:

`(GETSTRING)`

Si introducimos las siguientes cadenas devuelve lo que se indica:

AutoCAD	devuelve "AutoCAD"
123456	devuelve "123456"
INTRO	devuelve ""

El argumento opcional `T` (o equivalente) de la función especifica la posibilidad de introducir espacios blancos en la cadena. `T` es el símbolo predefinido del que hemos hablado más de una vez; es el carácter de cierto o verdadero. Si no se incluye, o se incluye otro u otros cualesquiera, `GETSTRING` no aceptará espacios blancos y, en momento en que se introduzca uno se tomará como un `INTRO` y se acabará la función. Si se incluye este argumento, `GETSTRING` aceptará espacios blancos y sólo será posible terminar con `INTRO`.

mensaje actúa como siempre. Veamos unos ejemplos:

```
(GETSTRING "Introduce un texto sin espacios: ")
(GETSTRING T "Introduce cualquier texto: ")
(GETSTRING (= 3 3) "Introduce cualquier texto: ")
(GETSTRING (/= 3 3) "Introduce un texto sin espacios: ")
```

Si se introduce una contrabarra (`\`) en cualquier posición, en dicha posición se devuelven dos contrabarras (`\\`) que, como sabemos, es el código para el carácter contrabarra. Esto será útil a la hora del manejo de archivos, que ya estudiaremos.

NOTA: Como se ha visto en el tercero de los primeros ejemplos de esta función, si se introduce un `INTRO` (o un espacio también si no se admiten), AutoLISP devuelve una cadena vacía (`""`). Si se admiten espacios y sólo se teclean espacios, se devuelven dichos espacios como cadena.

NOTA: Si se introducen más de 132 caracteres, AutoLISP sólo devuelve los 132 primeros, desechando los restantes.

ONCE.7.5. Establecer modos para funciones *GET...*

Antes de ver la última función de este tipo, y para comprender su funcionamiento, hemos de introducir una nueva función AutoLISP muy utilizada y versátil. Esta función es `INITGET` y su sintaxis es:

`(INITGET [modo] [palabras_clave])`

La función `INITGET` especifica el modo en que va a operar la siguiente función del tipo *GET...*, esto es, la primera que se encuentre tras ella. Este modo se indica con el argumento

modo, y es un número entero cuyo valor especifica un bit de control que determina los valores no permitidos para la siguiente función *GET*. . . . Los valores son los que siguen:

Valor de bit	Modo
1	No admite valores nulos, es decir, <i>INTRO</i> como respuesta.
2	No admite el valor cero (0).
4	No admite valores negativos.
8	No verifica límites, aunque estén activados.
16	(No se utiliza).
32	Dibuja la línea o el rectángulo elásticos con línea de trazos en lugar de continua.
64	Hace que la función <i>GETDIST</i> devuelva distancias 2D.
128	Permite introducir datos arbitrarios por teclado. Tiene prioridad sobre el valor 1.

Para ver la manera de utilizar esto pongamos un ejemplo. Imaginemos que queremos solicitar un número al usuario para realizar una cierta copia de objetos. Dicho número, evidentemente, habrá de ser entero (utilizaremos *GETINT*), pero además no puede ser negativo (no podemos copiar un objeto -24 veces). Pues para controlar dicho filtro, escribiremos lo siguiente:

```
(INITGET 4); Establece que el siguiente GETINT no admita valores negativos
(GETINT "Introduzca el número de copias: "); Solicita el número de copias
```

De esta forma, si el usuario introduce un número negativo, AutoLISP devuelve un mensaje de error diciendo que el número ha de ser positivo, y vuelve a solicitarlo.

Pero siguiendo con nuestro ejemplo, nos percatamos de que tampoco se podría introducir un valor de cero, porque no se puede copiar un objeto 0 veces. Habremos de indicarle también a *GETINT* que tampoco admita el cero como respuesta. Para especificar varios valores a la vez debemos sumarlos. Así pues, como el modo de no admitir negativos es el 4 y el de no admitir el cero es el 2, el valor final del bit sería un 6 ($4 + 2 = 6$). De esta forma haríamos:

```
(INITGET 6)
(GETINT "Introduzca número de copias: ")
```

Y para "redondear" el ejemplo, que menos que evitar que introduzca un *INTRO* a la pregunta, es decir, obligar al usuario a introducir un valor:

```
(INITGET 7)
(GETINT "Introduzca número de copias: ")
```

Esto es resultado de sumar los tres valores de bits correspondientes ($1 + 2 + 4 = 7$).

El modo en *INITGET* también puede indicarse como suma de valores de bits. Por ejemplo, el último caso podría haberse escrito así:

```
(INITGET (+ 1 2 4))
(GETINT "Introduzca número de copias: ")
```

Si se vuelve a utilizar ahora otra función del tipo *GET*. . . habría que especificar otro *INITGET* si fuera necesario, ya que cada uno sólo afecta a la función *GET*. . . que le sigue y solamente a esa.

NOTA: INITGET siempre devuelve nil.

Los modos establecidos con INITGET sólo se tienen en cuenta para las funciones GET... con las cuales tienen sentido. Así, no tiene sentido establecer un modo que no admita valores negativos con una función GETPOINT, puesto que ésta devuelve un punto como una lista de tres elementos y las listas no pueden ser negativas por definición. En la siguiente tabla se muestran los modos que tienen sentido con las diferentes funciones tipo GET...

Función	Valores de bits de modo con sentido para la función					
GETINT	1	2	4			128
GETREAL	1	2	4			128
GETDIST	1	2	4	32	64	128
GETANGLE	1	2		32		128
GETORIENT	1	2		32		128
GETPOINT	1			8	32	128
GETCORNER	1			8	32	128
GETSTRING						
GETKEYWORD	1					128

El valor 128 es el más específico. Se utiliza para tener la posibilidad de responder a una solicitud de función GET... por ejemplo con una expresión AutoLISP. Ésta se aceptaría como una cadena de texto y se podría evaluar posteriormente mediante la función EVAL (tal como veremos). Como el valor 128 tiene prioridad sobre el valor de bit 1, se aceptaría también un INTRO en la forma de una cadena de texto vacía "".

Veamos un pequeño programa de ejemplo de todo esto e intentemos comprenderlo. El listado es el que sigue:

```
; Nuevo comando CircEjes de AutoCAD

(DEFUN CircEjes (/ Centro Radio)
  (INITGET 1)
  (SETQ Centro (GETPOINT "Centro del círculo: "))
  (INITGET (+ 1 2 4))
  (SETQ Radio (GETDIST Centro "Radio del círculo: "))
  (COMMAND "_circle" Centro Radio)
  (INITGET 1)
  (COMMAND "_line" Centro "_qua" "\\ " "")
  (COMMAND "_line" Centro "_qua" "\\ " "")
  (COMMAND "_line" Centro "_qua" "\\ " "")
  (COMMAND "_line" Centro "_qua" "\\ " ")
)

(DEFUN C:CircEjes ()
  (CircEjes)
)

; Fin de CircEjes
```

En este ejemplo se observan algunas prácticas habituales a la hora de desarrollar programas en AutoLISP. La primera dice relación a la estructuración de programas. Como se ve, primero se define una función de usuario CircEjes y luego el propio comando de **AutoCAD** (C:CircEjes) con el mismo nombre. Esto se suele realizar así, primero por estructurar los programas: la definición de la orden de usuario (C:CircEjes) no contiene la secuencia de rutinas del programa en sí, sino una llamada a las funciones de usuario (en este caso sólo una, CircEjes) necesarias para ejecutar la orden. Y, segundo, por claridad estructural a la hora de observar el listado de un programa: podemos acceder directamente al

comando de usuario (`C:CircEjes`) para ver como va llamando sucesivamente a diferentes funciones e ir comprobando cada una de ellas. Por lo tanto, bajo la definición del comando en sí, únicamente aparecerán llamadas a funciones (entre paréntesis porque son funciones de AutoLISP sin los caracteres `C:`) y algunas otras funciones que ya veremos; por ejemplo para establecer valores de variables antes y después de las llamadas, etcétera.

Por otro lado también podemos apreciar la declaración de las dos variables que utilizará el programa como locales en los argumentos de `DEFUN`. Como se explicó, la manera de declarar variables locales es con una barra `y`, después de un espacio, sus nombres separados también por espacios. Y la diferencia que había con las globales, es que las locales desaparecen de memoria en cuanto se acaba de ejecutar el programa, o sea, no ocuparán memoria inútilmente. Si al acabar el programa intentamos evaluar alguna en la línea de comandos mediante el carácter `!`, el resultado será `nil`.

Otra de las características importantes de definir, por un lado la función de usuario y por el otro el comando de **AutoCAD**, es la posibilidad de introducir variables globales en los argumentos de las funciones creadas con `DEFUN`. Recordemos que si definimos con `DEFUN` una función del tipo `C:` (comando de **AutoCAD**) no se pueden introducir variables globales en sus argumentos. Recordemos también que si no se introduce argumento alguno, todas las variables declaradas con `SETQ` serán globales (seguirán en memoria al terminar el programa). Esto puede ser útil cuando otro programa AutoLISP necesita de esas variables para funcionar.

Por último, hemos de recordar también que si introducimos variables globales y locales juntas como argumentos de `DEFUN`, entre la última global y la barra, y entre la barra y la primera local habrá de haber un espacio blanco. Además siempre estarán en el orden global-local.

El programa del ejemplo en sí es un poco rudimentario, pero con los conocimientos que poseemos hasta ahora no podemos hacer más. Simplemente dibuja (de un modo un poco chapucero y manual) un círculo con sus cuatro ejes —sólo hasta el círculo, sin sobresalir—. Pide el centro y el radio, dibuja el círculo y, luego, va dibujando líneas (cuatro) desde el centro hasta un punto que ha de indicar el usuario. El punto que debe indicarse ha de ser un cuadrante (aunque no es obligatorio, pero si no se indica el programa no funcionará bien). El modo de referencia Cuadrante se activa automáticamente tras “engancharse” la línea al centro del círculo.

ONCE.7.5.1. Palabras clave

Hemos dejado un poco colgada la explicación de `INITGET`, a falta de explicar el segundo de sus argumentos `palabras_clave`. Vamos a ver para que sirve exactamente.

El argumento `palabras_clave` es una cadena de texto que define una serie de respuestas alternativas para las funciones del tipo `GET....`. Vamos a ver un ejemplo para entender esto. Imaginemos que queremos solicitar un punto para el final de una línea, lo haremos con `GETPOINT`. Sabemos que dicho punto lo podemos introducir directamente en pantalla o por teclado. Pero, en este caso, también nos interesa que se acepten otros caracteres, por ejemplo una “H” para deshacer el tramo último y una “C” para cerrar. Como bien sabemos, si introducimos un carácter no numérico con `GETPOINT` AutoLISP nos dará un mensaje de error. Pero si definimos qué caracteres se pueden aceptar (los expuestos) `GETPOINT` los capturará sin ningún problema. Esto lo realizamos desde `INITGET`. Vamos a ver el ejemplo:

```
(INITGET 1 "H C")
(GETPOINT "Introducir nuevo punto: ")
```

En este caso GETPOINT no acepta INTRO como respuesta (valor de bit 1), pero si aceptará un carácter H o un carácter C (da lo mismo mayúsculas que minúsculas). Los caracteres han de indicarse entre comillas —es una cadena— y separados entre sí por un espacio blanco.

Si las respuestas posibles tienen una o varias letras como abreviatura mínima necesaria para identificar dicha respuesta, se indican esas letras en mayúsculas en INITGET y el resto en minúsculas. Por ejemplo:

```
(INITGET "desHacer Cerrar")
```

Este ejemplo aceptará como respuesta válida H (o h), C (o c) y las palabras completas DESHACER y CERRAR (tanto mayúsculas como minúsculas). Es exactamente lo mismo que podemos apreciar en el comando MATRIZ (ARRAY en inglés) de **AutoCAD**, por ejemplo. Tras teclearlo y designar los objetos correspondientes, el comando expone:

```
Matriz Rectangular o Polar (<R>/P):
```

Podemos escribir R, P, Rectangular o Polar, cualquiera de las opciones. Si ésta fuera una orden de usuario, la función INITGET podría haber sido:

```
(INITGET "Rectangular Polar")
```

Hemos de tener cuidado con esto de las abreviaturas porque a veces, incluso con los propios mensajes que emite **AutoCAD**, parece que hay mucha gente que tiene problemas. En el caso siguiente:

```
(INITGET "DESactivar")
```

no se admite ni D ni DE, sino DES (como abreviatura válida) y la palabra completa DESACTIVAR; tanto mayúsculas como minúsculas todas las opciones.

La abreviatura es pues el mínimo número de caracteres en que debe coincidir la respuesta del usuario con la alternativa indicada en INITGET. A partir de ahí, se admiten más caracteres por parte del usuario hasta la longitud de la respuesta alternativa completa. Por ejemplo:

```
(INITGET 1 "Cont")
```

admitiría C, CO, CON o CONT (mayúsculas o minúsculas), pero no CONTINUA o CONTINUAR.

NOTA: Es norma lógica indicar, como mínimo, el bit 1 en estos tipos de INITGET con palabras clave para que no se admita el INTRO como respuesta, ya que es necesaria una de las opciones.

También el posible indicar la abreviatura junto a la respuesta completa en mayúsculas y separada por una coma (aunque recomendamos el método anterior). Por ejemplo:

```
(INITGET 1 "DESACTIVAR,DES")
```

equivale a:

```
(INITGET 1 "DESactivar")
```

NOTA: Todas las funciones GET... admiten palabras clave.

Un ejemplo sencillo aunque no funcional:

```
(DEFUN C:Prueba ()  
  (INITGET 1 "desHacer Cerrar")  
  (GETPOINT "desHacer/Cerrar/<Primer punto>: ")  
)
```

Este ejemplo no realiza nada, y es que aún no hemos aprendido a procesar estos datos de palabras clave. Pero es una buena muestra de lo que sería un mensaje típico de **AutoCAD** en la línea de comandos. El programa muestra:

```
desHacer/Cerrar/<Primer punto>:
```

La opción por defecto (entre corchete angulares) es señalar un punto en pantalla (o por teclado), aunque podemos acceder a otras dos opciones alternativas mediante los caracteres **H** y **C** (o sus palabras completas) respectivamente. Al acceder a estas opciones el programa no hace nada y es que, como decimos, hace falta procesar esta entrada de usuario (ya se verá).

Al indicar cualquiera de las opciones anteriores, AutoLISP devuelve la palabra clave, entre comillas, correspondiente indicada en `INITGET`.

Pues vista esta característica de las palabras clave, ya podemos estudiar la última de las funciones `GET...`, la cual emplazamos para después de `INITGET` y es `GETKEYWORD`. La sintaxis de `GETKEYWORD` es:

```
(GETKEYWORD [mensaje])
```

Esta función solicita únicamente una de una serie de palabras clave indicadas en `INITGET` de la forma explicada. Sólo sirve para palabras clave, y nada más. Solicitará dichas palabras y, si no se introduce alguna de ellas, da un mensaje de error y vuela a indicar la solicitud. Por ejemplo:

```
(INITGET 1 "Sí No")  
(GETKEYWORD "Cerrar el muro (Sí/No): ")
```

Tras esta pregunta podríamos teclear, como sabemos, `O S O N O SÍ O NO` (incluso el "sí" sin tilde y tanto mayúsculas como minúsculas). Pero sólo podríamos realizar dichas entradas, ninguna más.

La función `GETKEYWORD` devuelve, como cadena de texto, la opción especificada tal y como se indicó en `INITGET`. Si no se especificó ninguna devuelve `nil`.

6ª fase intermedia de ejercicios

- Realizar un programa que dibuje aros (arandelas sin relleno) solicitando el centro, el diámetro interior y el diámetro exterior.
- Realizar el mismo ejercicio anterior pero solicitando el centro, el radio intermedio del aro (mitad entre interior y exterior) y el grosor del mismo.
- Practicar la característica de palabras clave con algún ejercicio inventado, aunque no se procesen las entradas de estas palabras.

ONCE.8. ACCESO A VARIABLES DE AutoCAD

Vamos a explicar ahora, en esta sección, el control que podemos tener desde AutoLISP con respecto a las variables de sistema de **AutoCAD**.

Para lo que se refiere a este control tenemos a nuestra disposición dos funciones muy importantes y utilizadas en la programación en AutoLISP. Estas funciones son `GETVAR` y `SETVAR`. Si revisamos el **MÓDULO NUEVE** de este curso, acerca de la programación en lenguaje DIESEL, recordaremos la misión que realizaba la función `GETVAR` de este lenguaje. Pues exactamente la misma (y con el mismo nombre) realiza bajo AutoLISP. Lo único que varía es la sintaxis de la función, debido a las exigencias propias de AutoLISP, pero tampoco demasiado, es la siguiente:

```
(GETVAR nombre_variable)
```

Con `GETVAR` extraemos o capturamos el valor actual de la variable de sistema o acotación de **AutoCAD** indicada en *nombre_variable*, o sea, de cualquier variable del programa.

Como sabemos, **AutoCAD** funciona internamente con multitud de variables (**APÉNDICE B**) que controlan prácticamente todos los aspectos del programa. El que posea el conocimiento y habilidad de manejo de las variables de **AutoCAD**, se puede decir que posee el control casi al 100% sobre él. Pues desde AutoLISP accederemos al contenido de dichas variables para luego procesarlo, o simplemente como información.

El nombre de la variable habrá de ir entre comillas, por ser cadena. Vemos un ejemplo:

```
(GETVAR "pickfirst")
```

Esta expresión devolverá el valor de la variable de sistema de **AutoCAD** `PICKFIRST`, que controla la llamada designación *Nombre-Verbo*.

Otros ejemplos:

```
(GETVAR "blipmode")  
(GETVAR "aperture")  
(GETVAR "blipmode")  
(GETVAR "dimtad")  
(GETVAR "modemacro")
```

NOTA: Si la variable indicada no existe, AutoLISP devuelve `nil`.

Por su lado, `SETVAR` realiza la acción contraria, es decir, introduce o asigna un valor a una variable de **AutoCAD**. Su sintaxis es:

```
(SETVAR nombre_variable valor)
```

`SETVAR` asignará *valor* a *nombre_variable*, según esta sintaxis, y devolverá *valor* como respuesta. El nombre de la variable en cuestión deberá ir entre comillas, al igual que con `GETVAR`, y el valor que se le asigne deberá ser coherente con la información que puede guardar la variable. Si no es así, AutoLISP devuelve el error `AutoCAD rejected function`.

NOTA: En el **APÉNDICE B**, además de la lista de las variables de sistema y acotación de **AutoCAD**, se muestra también el significado de cada una de ellas y el tipo de valor que pueden guardar, así como el rango de éste o las opciones disponibles.

Veamos algún ejemplo:

```
(SETVAR "filletrad" 2)
```

```
(SETVAR "proxygraphics" 0)
(SETVAR "attdia" 1)
```

Si no existe la variable se devuelve el mismo error que si se le introduce un valor erróneo.

El funcionamiento de SETVAR cuando un comando se encuentra en curso es completamente transparente, es decir, sería como utilizar el comando MODIVAR (SETVAR en inglés, igual que la función) de **AutoCAD** de manera transparente, con el apóstrofo delante. En estos casos puede suceder que la modificación de la variable sólo surta efecto en la siguiente orden o en la siguiente regeneración.

Un ejemplo de total transparencia podría ser:

```
(COMMAND "_erase") (SETVAR "pickbox" 2)
```

COMMAND llama al comando BORRA (ERASE) de **AutoCAD**, el cual se queda esperando en Designar objetos. Después SETVAR cambia el valor de la mira de designación a un valor de 2. Este cambio se efectúa de manera transparente, y la orden BORRA sigue pidiendo designar objetos, pero ahora visualiza la mirilla con el nuevo tamaño de mira de designación.

Evidentemente no se puede cambiar el valor de una variable que sea de sólo lectura. Si se intenta, se producirá el mismo error antes comentado en dos ocasiones.

NOTA: Para algunas variables como ANGBASE y SNAPANG, el valor de las mismas se interpreta en radianes al acceder mediante AutoLISP, mientras que si se accede con MODIVAR, desde la línea de comandos (o tecleando el nombre de la variable), su valor se considera en grados. Cuidado con esto. La misma consideración para GETVAR.

Un ejemplo práctico y muy usado es la posibilidad de desactivar el eco de la línea de comandos en la ejecución de programas AutoLISP. Este eco (variable CMDECHO) evitará que las funciones de AutoLISP vayan devolviendo números, cadenas y demás a lo largo de la ejecución. Y antaño, cuando las marcas auxiliares (variable BLIPMODE) venían activadas por defecto en **AutoCAD**, se utilizaba mucho la posibilidad de desactivarlas para producir unas rutinas "limpias". Veamos en uno de los ejemplos vistos hace poco:

```
(DEFUN CircEjes (/ Centro Radio)
  (INITGET 1)
  (SETQ Centro (GETPOINT "Centro del círculo: "))
  (INITGET (+ 1 2 4))
  (SETQ Radio (GETDIST Centro "Radio del círculo: "))
  (COMMAND "_circle" Centro Radio)
  (INITGET 1)
  (COMMAND "_line" Centro "_qua" "\\\" \"")
  (COMMAND "_line" Centro "_qua" "\\\" \"")
  (COMMAND "_line" Centro "_qua" "\\\" \"")
  (COMMAND "_line" Centro "_qua" "\\\" \"")
)

(DEFUN C:CircEjes ()
  (SETVAR "cmdecho" 0)
  (SETVAR "blipmode" 0)
  (CircEjes)
  (SETVAR "cmdecho" 1)
  (SETVAR "blipmode" 1)
)
```

Podemos observar otra aplicación a la hora de estructurar la programación. El comando de **AutoCAD** (`C:Circle`) sólo contiene la llamada a la función que realiza toda la tarea y las definiciones de los valores de las variables pertinentes antes de la propia llamada; restaurando sus valores al final del programa (tras la ejecución de la función).

ONCE.9. ESTRUCTURAS BÁSICAS DE PROGRAMACIÓN

En el mundo de los lenguajes de programación existen un par de estructuras que, con todas sus variantes, son consideradas las estructuras básicas o elementales a la hora de programar. Estas estructuras son las condicionales (o alternativas) y las repetitivas. Dentro de cada una de ellas pueden existir variantes, como decimos, que realicen el trabajo de distinta forma. Por ejemplo, y si sabemos programar algo en BASIC (QuickBASIC) pensaremos en *IF... THEN... ELSE*, *WHILE... WEND* o *SELECT CASE* como estructuras alternativas o condicionales y en *FOR... NEXT* o *GOSUB... RETURN* como estructuras repetitivas. Hay más, en este y en todos los lenguajes, cada una operando a su manera, pero todas dentro del mismo grupo.

Pues en AutoLISP también disponemos de una serie de funciones que nos van a permitir jugar con la posibilidad de ejecutar determinados tramos de nuestro programa si se da una condición, o repetir una serie de funciones un determinado número de veces, etcétera.

Vamos a empezar pues con la primera.

```
(IF condición acción_se_cumple [acción_no_se_cumple])
```

La función *IF* establece una condición en forma de expresión evaluada. Si dicha condición se cumple, es decir si el resultado es distinto de *nil*, entonces pasa a evaluar la expresión contenida en *acción_se_cumple*. En este caso devuelve el resultado de esta expresión.

Si la condición no se cumple, es *nil*, entonces pasa a evaluar el contenido de la expresión en *acción_no_se_cumple*, si es que existe (es opcional). El contenido en este caso de la acción si es que se cumple sería obviado, al igual que el contenido de la acción si no se cumple cuando se cumple.

Si no se indica *acción_no_se_cumple* y la condición no se cumple (no evalúa *acción_se_cumple*), AutoLISP devuelve *nil*.

Veamos un ejemplo para aclararnos un poco:

```
(DEFUN C:Personal ()
  (SETQ Nombre (GETSTRING T "Introduce tu nombre: "))
  (IF (= Nombre "Jonathan")
    (SETVAR "blipmode" 0)
    (SETVAR "blipmode" 1)
  )
)
```

Este pequeño programa ha podido ser diseñado para que pregunte por un nombre, que guardará en la variable (global) *Nombre*. Después se pregunta: si *Nombre* es igual a *Jonathan*, entonces se establece la variable *BLIPMODE* a 0, si no, se establece *BLIPMODE* a 1. Dependiendo del nombre que tecleemos se realizará una acción u otra.

Otro ejemplo:

```
(DEFUN C:Compara ()
```

```
(SETQ Punto1 (GETPOINT "Primer punto: "))
(SETQ Punto2 (GETPOINT "Segundo punto: "))
(IF (EQUAL Punto1 Punto2)
  (PROMPT "Son iguales.")
  (PROMPT "No son iguales."))
)
```

Este ejemplo acepta dos puntos introducidos por el usuario. Si dichos punto son iguales (comparados con `EQUAL`) el resultado de la comparación es cierto (`T`) por lo que se escribe el mensaje *Son iguales.* (*acción_se_cumple*); si no lo son, el resultado es `nil` y pasa directamente a escribir *No son iguales.* (*acción_no_se_cumple*).

NOTA: Hemos conjeturado el funcionamiento de `PROMPT`. Aún así, lo veremos inmediatamente.

Como ya se ha dicho, la acción que se realiza si no se cumple la condición no es obligatorio ponerla. Así, podemos realizar un pequeño ejercicio en el que no haga nada ni no se cumple la condición:

```
(DEFUN C:Prueba ()
  (SETQ X (GETDIST "Distancia primera: "))
  (SETQ Y (GETDIST "Distancia segunda: "))
  (IF (>= X Y)
    (SETQ X (1+ X))
  )
)
```

Este ejemplo pregunta por dos distancias, si la primera es mayor o igual que la segunda, incrementa en una unidad esa distancia primera, si no, no se realiza absolutamente nada.

La función `IF` debe llevar dos argumentos como mínimo, la condición o comparación y la acción si dicha condición se cumple. La acción si no se cumple es opcional, como sabemos. Por ello, si lo que queremos es indicar una opción si no se cumple y evitar que realice algo si se cumple, habremos de indicar una lista vacía en este primero argumento:

```
(IF (EQUAL Pto1 Pto2) () (PROMPT "No son iguales."))
```

Si no se hace esto, tomaría la segunda acción como primera y no produciría el resultado esperado.

Existe una pequeña restricción en torno a la función `IF`, y es que únicamente permite un elemento o expresión en cada uno de sus argumentos. Por ejemplo, si hubiéramos querido indicar en el ejemplo `C:Prueba` un incremento de uno para `X` y, además un incremento de 7.5 para `Y`, todo ello si la condición se cumple, no habríamos podido hacerlo todo seguido. Para subsanar este pequeño inconveniente existe una función que enseguida veremos.

Antes vamos a explicar esa función `PROMPT` que hemos dejado un poco en el aire.

`(PROMPT cadena)`

`PROMPT` escribe la cadena de texto especificada en la línea de comandos de **AutoCAD** y devuelve `nil`. Ejemplos:

<code>(PROMPT "Hola")</code>	devuelve <code>Holanil</code>
<code>(PROMPT "Hola, soy yo")</code>	devuelve <code>Hola, soy yonil</code>
<code>(PROMPT "1 + 2")</code>	devuelve <code>1 + 2nil</code>

```
(PROMPT " ")           devuelve nil
(PROMPT "              ") devuelve          nil
```

Se observa que el mensaje se devuelve sin comillas.

NOTA: En configuraciones de dos pantallas, PROMPT visualiza el mensaje en ambas. Es por ello preferible a otras funciones de escritura que ya veremos más adelante.

Volvamos ahora sobre el siguiente ejemplo, ya expuesto anteriormente:

```
(DEFUN C:Compara ()
  (SETQ Punto1 (GETPOINT "Primer punto: "))
  (SETQ Punto2 (GETPOINT "Segundo punto: "))
  (IF (EQUAL Punto1 Punto2)
    (PROMPT "Son iguales.")
    (PROMPT "No son iguales."))
  )
)
```

Podemos apreciar, al correr este programa, un par de cosas. La primera es que no existe salto de línea en ningún momento de la ejecución. Una salida final de este ejercicio podría aparecer así (tras indicar los dos puntos en pantalla):

```
Primer punto: Segundo punto: No son iguales.nil
```

Esto hace realmente poco vistoso el desarrollo de una aplicación.

El segundo problema es la devolución de nil al final de una función PROMPT. Al igual que en el caso anterior, desmejora la vistosidad del programa. Para solucionar estos dos problemas vamos a exponer dos funciones, TERPRI y PRIN1. La primera (TERPRI) la explicamos a continuación y, la segunda (PRIN1), indicamos donde escribirla y no vamos a decir nada más de ella, porque la comentaremos a fondo cuando estudiemos las operaciones con archivos, que es para lo realmente sirve.

```
(TERPRI)
```

Como apreciamos, TERPRI es una función sin argumentos. La misión que tiene es la de mover el cursor al comienzo de una nueva línea. Se utiliza para saltar de línea cada vez que se escribe algún mensaje en el área de comandos de **AutoCAD**, a no ser que la función que escriba el mensaje salte de línea por sí sola, que las hay, ya veremos.

Así por ejemplo, podemos variar el ejemplo anterior así:

```
(DEFUN C:Compara ()
  (SETQ Punto1 (GETPOINT "Primer punto: ")) (TERPRI)
  (SETQ Punto2 (GETPOINT "Segundo punto: ")) (TERPRI)
  (IF (EQUAL Punto1 Punto2)
    (PROMPT "Son iguales.")
    (PROMPT "No son iguales."))
  )
)
```

El resultado será bastante más claro, al saltar a la línea siguiente después de cada petición.

NOTA: Podríamos haber escrito cada función TERPRI en un renglón aparte del programa, pero se suelen indicar así por estructuración: para especificar después de qué mensaje salta a una nueva línea.

Existe otro método, como deberíamos saber ya (ver el principio de este **MÓDULO**) para saltar de línea. Es la inclusión de los caracteres `\n`. Pero esto se utiliza para separar cadenas en diferentes líneas. Así, el ejemplo que venimos proponiendo podemos escribirlo:

```
(DEFUN C:Compara ()
  (SETQ Punto1 (GETPOINT "Primer punto: \n"))
  (SETQ Punto2 (GETPOINT "Segundo punto: \n"))
  (IF (EQUAL Punto1 Punto2)
    (PROMPT "Son iguales.")
    (PROMPT "No son iguales."))
  )
)
```

Pero el resultado es distinto: hace la petición del punto y salta a una nueva línea antes de que lo introduzcamos.

Por otra parte, la función `PRIN1` la escribiremos como norma general al final de cada programa para producir un final "limpio" del mismo:

```
(DEFUN C:Compara ()
  (SETQ Punto1 (GETPOINT "Primer punto: ")) (TERPRI)
  (SETQ Punto2 (GETPOINT "Segundo punto: ")) (TERPRI)
  (IF (EQUAL Punto1 Punto2)
    (PROMPT "Son iguales.")
    (PROMPT "No son iguales."))
  )
  (PRIN1)
)
```

De esta forma evitamos el mensaje `nil` al final de la ejecución.

NOTA: Como ya hemos comentado, hablaremos profundamente de `PRIN1` cuando llegue el momento, ya que tiene diversas funciones y ésta es una característica especial derivada de ellas. Por ahora, tomemos como norma lo dicho y creámonoslo sin más.

Siguiendo ahora con las estructuras alternativas que habíamos apartado un poco para ver estas funciones de escritura y salto de línea, pasemos al estudio de `PROGN`.

```
(PROGN expresión1 [expresión2...])
```

Esta función admite como argumentos todas las expresiones indicadas y las evalúa secuencialmente, devolviendo el valor de la última evaluada.

La siguiente expresión:

```
(PROGN (+ 2 3) (- 1 2) (/= 23 23) (SETQ s 5.5))
```

equivale a indicar todas las expresiones que incluye en sus argumentos de forma separada y continuada dentro de un programa o en la línea de comandos. Es decir, los siguientes dos ejemplos son idénticos, en cuanto a resultado:

```
(DEFUN C:Ejem1 ()
  (SETQ X 5 Y 23.3)
  (+ X Y)
  (- X Y)
  (/ X Y)
  (* X Y)
)
```

y

```
(DEFUN C:Ejem2 ()
  (PROGN
    (SETQ X 5 Y 23.3)
    (+ X Y)
    (- X Y)
    (/ X Y)
    (* X Y)
  )
)
```

Entonces, ¿para qué puede servir PROGN? PROGN se utiliza en funciones cuyo formato sólo admite una expresión en determinados argumentos y nosotros deseamos indicar más. Un ejemplo muy claro es el de la función IF. Como hemos explicado, existe esa pequeña restricción de IF que únicamente permite especificar una expresión en cada uno de sus argumentos. Con PROGN tendremos la posibilidad de especificar más de una acción, tanto si se cumple la condición como si no. Veamos un pequeño ejemplo primero y después otro más elaborado que servirá de pequeño repaso de muchos aspectos vistos hasta ahora.

```
(DEFUN C:Condic ()
  (SETQ Valor (GETREAL "Introduce un valor: "))
  (IF (> Valor 100)
    (PROGN
      (PROMPT "Es un valor mayor de 100.") (TERPRI)
    )
    (PROGN
      (PROMPT "Es un valor menor de 100,") (TERPRI)
      (PROMPT "¿qué te parece?")
    )
  )
  (PRIN1)
)
```

De esta manera, cada argumento de la función IF ejecuta no sólo una expresión, sino varias. En realidad únicamente ejecuta una, PROGN, que es lo que admite IF, pero ella es una que permite evaluar más una dentro de sí misma.

Veamos ahora el ejemplo siguiente. Tiene relación con un ejercicio propuesto anterior, pero con mucho más jugo.

```
(DEFUN Aro (/ Centro Radio Grosor Rint Rext Dint Dext Op)
  (SETQ Centro (GETPOINT "Centro del aro: ")) (TERPRI)
  (SETQ Radio (GETDIST "Radio intermedio: ")) (TERPRI)
  (SETQ Grosor (GETDIST "Grosor del aro: ")) (TERPRI)
  (INITGET "Hueco Relleno")
  (SETQ Op (GETKEYWORD "Aro Hueco o Relleno (<H>/R): ")) (TERPRI)
  (IF (OR (= Op "Hueco") (= Op \n))
    (PROGN
      (SETQ Rint (- Radio (/ Grosor 2)))
      (SETQ Rext (+ Radio (/ Grosor 2)))
      (COMMAND "_circle" Centro Rext)
      (COMMAND "_circle" Centro Rint)
    )
    (PROGN
      (SETQ Dint (* (- Radio (/ Grosor 2))2))
      (SETQ Dext (* (+ Radio (/ Grosor 2))2))
      (COMMAND "_donut" Dint Dext Centro "")
    )
  )
)
```

```
)  
)  
)  
  
(DEFUN C:Aro (  
  (SETVAR "cmdecho" 0)  
  (Aro)  
  (SETVAR "cmdecho" 1)  
  (PRIN1)  
)  
  
(PROMPT "Nuevo comando Aro definido.") (PRIN1)
```

Explicuemos el ejemplo. El programa dibuja aros, huecos o rellenos, solicitando el centro del mismo, su radio intermedio y su grosor.

Se crea una nueva función de usuario a la que se atribuyen una serie de variables locales —las que luego serán utilizadas—. Se pregunta por los tres datos determinantes para el dibujo de aro (centro, radio intermedio y grosor), los cuales se guardan en tres variables (Centro, Radio y Grosor). A continuación se inicializa (INITGET) el siguiente GETKEYWORD para que admita dos palabras claves (Hueco y Relleno) con sus respectivas abreviaturas. Nótese que no se indica ningún código para que no admita un INTRO por respuesta, ya que luego nos será útil.

Pregunta el programa si el aro que va a dibujar será hueco o relleno. Por defecto se nos ofrece la opción correspondiente a hueco (entre corchetes angulares <> para indicarlo como los comandos típicos de **AutoCAD**). Aquí para tomar la opción por defecto podremos pulsar directamente INTRO (lo normal en **AutoCAD**), por ello nos interesaba antes poder aceptar un INTRO. Además podremos elegir teclear la opción segunda o la primera.

Seguidamente hemos de controlar la entrada del usuario que se ha guardado en la variable Op. Para ello utilizamos una función IF que nos dice que, si Op es igual a Hueco (o a h, hu, hue, huec, tanto mayúsculas como minúsculas; recordemos que la salida de GETKEYWORD es la indicada completa en el INITGET) o (OR) igual a un INTRO (\n, opción por defecto), se realizará todo lo contenido en el primer PROG. Si no, se pasará a evaluar lo contenido en el segundo PROG (argumento *acción_no_se_cumple* de IF). De esta forma el usuario sólo tiene dos alternativas, aro hueco o aro relleno. Si escribe otra cosa no será aceptada por GETKEYWORD. Así, al indicar luego en el IF que si la opción no es la de aro hueco pase por alto el primer argumento, sabremos de buena tinta que lo que no es Hueco ha de ser forzosamente Relleno.

En la secuencia de funciones para un aro hueco, se calculan el radio interior y exterior del mismo y se dibujan dos círculos concéntricos que representan el aro. Por su lado, en la secuencia para un aro relleno, se calculan los diámetros interior y exterior y se dibuja una arandela. La razón para calcular diámetros aquí es que el comando ARANDELA (DONUT en inglés) de **AutoCAD** solicita diámetros y no radios.

Tras cerrar todos los paréntesis necesarios —el del último PROG, el del IF y el de DEFUN— se pasa a crear el comando propio para **AutoCAD** (C:Aro). De desactiva el eco de mensajes en la línea de comandos, se llama a la función (Aro), se vuelve a activar el eco y se introduce una expresión PRIN1 para un final "limpio" del programa (sin nil ni ningún otro eco o devolución de AutoLISP).

Por último, y fuera de cualquier DEFUN, se introduce una función PROMPT que escribe un mensaje en la línea de comandos. Todas las funciones de AutoLISP que no estén contenidas dentro de los DEFUN en un programa se ejecutan nada más cargar éste. Por ello, al cargar este programa aparecerá únicamente el mensaje Nuevo comando Aro definido. Y al ejecutar el

comando, escribiendo `Aro` en línea de comandos, este `PROMPT` no se evaluará al no estar dentro de ningún `DEFUN`.

El `PRIN1` detrás de este último `PROMPT` hace que no devuelva `nil`. Tampoco se ejecutará al correr el programa, ya que está fuera de los `DEFUN`, sino sólo al cargarlo. Es por ello, que para el programa en sí se utilice otro `PRIN1`, el expuesto antes e incluido en el segundo `DEFUN`.

`(COND (condición1 resultado1) [(condición2 resultado2)...])`

La función `COND` de AutoLISP que vamos a ver ahora establece varias condiciones consecutivas asignando diferentes resultados a cada una de ellas. Es decir, es una generalización de la función `IF` que, sin embargo, resulta más cómoda a la hora de establecer diversas comparaciones. Veamos un ejemplo sencillo:

```
(DEFUN Compara ()
  (SETQ X (GETREAL "Introduce el valor de X entre 1 y 2: "))
  (COND ((= X 1) (PROMPT "Es un 1.") (TERPRI))
        ((= X 2) (PROMPT "Es un 2.") (TERPRI))
        ((< X 1) (PROMPT "Es menor que 1, no vale.") (TERPRI))
        ((> X 2) (PROMPT "Es mayor que 2, no vale.") (TERPRI))
        (T (PROMPT "Es decimal entre 1 y 2.") (TERPRI)))
)
```

Se establece una serie de comparaciones que equivaldría a una batería de funciones `IF` seguidas. En la última condición no es una lista, sino el valor de cierto `T`. Esto garantiza que, si no se han evaluado las expresiones anteriores se evalúen las de esta última lista. Y es que `COND` no evalúa todas las condiciones, sino que va inspeccionándolas hasta que encuentra una que sea diferente de `nil`. En ese momento, evalúa las expresiones correspondientes a esa condición y sale del `COND`, sin evaluar las siguientes condiciones aunque sean `T`.

Si se cumple una condición y no existe un resultado (no está especificado), `COND` devuelve el valor de esa condición.

Una aplicación muy típica de `COND` es el proceso de las entradas por parte del usuario en un `GETKEYWORD`. Por ejemplo:

```
(DEFUN Proceso ()
  (INITGET 1 "Constante Gradual Proporcional Ninguno")
  (SETQ Op (GETKEYWORD "Constante/Gradual/Proporcional/Ninguno: "))
  (COND ((= Op "Constante") (Constante))
        ((= Op "Gradual") (Gradual))
        ((= Op "Proporcional") (Proporcional))
        ((= Op "Ninguno") (Ninguno)))
)
...
```

En este ejemplo se toman como condiciones las comparaciones de una respuesta de usuario frente a `GETKEYWORD`, haciendo llamadas a funciones diferentes dentro del mismo programa según el resultado.

NOTA: Como podemos observar, los paréntesis indicados en la sintaxis tras `COND` son obligatorios (luego cerrarlos antes de la segunda condición). Estas listas engloban cada condición y resultado por separado.

NOTA: Como observamos en el primer ejemplo, con `COND` podemos especificar más de una expresión para el resultado de una comparación, y sin necesidad de `PROGN`. La primera lista se toma como condición y todas las demás, hasta que se cierre el paréntesis que engloba a una condición con sus respectivos resultados, se toman como resultados propios de dicha condición.

Y continuando con las estructuras básicas de la programación, vamos a ver ahora una muy recurrida y usada; se trata de `REPEAT`. `REPEAT` representa la estructura repetitiva en AutoLISP y sus sintaxis es la siguiente:

`(REPEAT veces expresión1 [expresión2...])`

Esta función repite un determinado número de veces (especificado en *veces*) la expresión o expresiones que se encuentren a continuación, hasta el paréntesis de cierre de `REPEAT`. El número de repeticiones ha de ser positivo y entero. `REPEAT` evaluará dicho número de veces las expresiones contenidas y devolverá el resultado de la última evaluación. Veamos un ejemplo:

```
(DEFUN Poligonal ()
  (SETQ Vert (GETINT "Número de vértices de la poligonal: "))
  (SETQ Lin (- Vert 1))
  (SETQ Pto1 (GETPOINT "Punto primero: "))
  (REPEAT Lin
    (SETQ Pto2 (GETPOINT "Siguiente punto: "))
    (COMMAND "_line" Pto1 Pto2 "")
    (SETQ Pto1 Pto2)
  )
)
```

El ejemplo pide el número de vértices de una poligonal que se dibujará con líneas. Evidentemente el número de líneas que se dibujarán será el número de vértices menos uno, por lo que se establece en la variable `Lin` dicho valor. Tras pedir el primer punto se comienza a dibujar las líneas en la estructura repetitiva (tantas veces como líneas hay). Lo que hace la línea `(SETQ Pto1 Pto2)` es actualizar la variable `Pto1` con el valor de `Pto2` cada vez que se dibuja una línea. De esta forma se consigue tomar como punto de la primera línea el punto final de la anterior.

`(WHILE condición expresión1 [expresión2...])`

La función `WHILE` establece estructuras repetitivas al igual que `REPEAT`. La diferencia estriba en que `WHILE` proporciona un control sobre la repetición, ya que la serie de expresiones (o única expresión como mínimo) se repetirá mientras se cumpla una determinada condición especificada en *condición*.

Mientras el resultado de la condición sea diferente de `nil` (o sea `T`), `WHILE` evaluará las expresiones indicadas. En el momento en que la condición sea igual a `nil`, `WHILE` terminará, dejando de repetirse el ciclo. Veamos el anterior ejemplo de `REPEAT` un poco más depurado con `WHILE`:

```
(DEFUN Poligonal ()
  (SETQ Vert (GETINT "Número de vértices de la poligonal: "))
  (SETQ Lin (- Vert 1))
  (SETQ Pto1 (GETPOINT "Punto primero: "))
  (WHILE (> Lin 0)
    (SETQ Pto2 (GETPOINT "Siguiente punto: "))
    (COMMAND "_line" Pto1 Pto2 "")
    (SETQ Pto1 Pto2)
  )
)
```

```
(SETQ Lin (1- Lin))
)
)
```

De esta forma se establece una estructura repetitiva controlada por el número de líneas, el cual va decrementándose en -1: (SETQ Lin (1- Lin)) cada vez que se repite el proceso. Mientras Lin sea mayor de 0 se dibujarán líneas, en el momento en que no sea así se terminará el proceso.

WHILE se utiliza mucho para controlar entradas de usuario y procesar errores, por ejemplo:

```
...
(SETQ DiaCj (GETREAL "Diámetro de la cajera: "))
(SETQ Dia (GETREAL "Diámetro del agujero: "))
(WHILE (> Dia DiaCj)
  (PROMPT "El diámetro del agujero debe ser menor que el de la cajera.\n")
  (SETQ Dia (GETREAL "Diámetro del agujero: "))
)
...
```

Existe una forma muy particular de usar funciones como WHILE o IF. Vemos el ejemplo siguiente:

```
(DEFUN Haz (/ ptb pt)
  (INITGET 1)
  (SETQ ptb (GETPOINT "Punto de base: ")) (TERPRI)
  (WHILE (SETQ pt (GETPOINT ptb "Punto final (INTRO para terminar): ")) (TERPRI)
    (COMMAND "_line" ptb pt "")
  )
)
```

El ejemplo dibuja segmentos rectos en forma de haz de rectas desde un punto de base a diversos puntos que es usuario introduce. Examinemos cómo se realiza la comparación en el WHILE. De suyo la comparación no existe como tal, pero sabemos que WHILE continúa mientras no obtenga nil. Ahí está el truco. En el momento en el pulsemos INTRO, pt guardará nil, por lo que WHILE no continuará. Si introducimos puntos, WHILE no encuentra nil por lo que realiza el bucle.

A continuación vamos a ver tres funciones que no se refieren a repetición de expresiones en sí, sino a repeticiones de proceso con elementos de listas. Estas tres funciones son FOREACH, APPLY y MAPCAR.

```
(FOREACH variable lista expresión)
```

Esta función procesa cada elemento de una lista (*lista*) aplicándole una expresión (*expresión*) indicada. Para ello se utiliza un símbolo (*variable*) que debe aparecer en dicha expresión. El funcionamiento es el siguiente: se toma cada elemento de la lista y se hace intervenir en la expresión en los lugares donde aparece el símbolo. Después se evalúa cada una de las expresiones resultantes para cada elemento de la lista. Vamos a estudiar un ejemplo:

```
(FOREACH Var '(10 20 30) (* 2 Var))
```

Lo que se pretende aquí es multiplicar cada uno de los elementos de la lista por 2. De esta forma, y como hemos explicado, en principio se define una variable (*Var*). Esta variable será sustituida por cada uno de los elementos de la lista que sigue en la expresión del final.

Así, Var es sustituida por 10, por 20 y por 30 respectivamente en la expresión del producto que se indica en último lugar.

Al final, FOREACH devuelve el resultado de la última expresión evaluada.

Veamos otro ejemplo que dibuja líneas desde el punto 0,0 hasta cuatro puntos 2D indicados en una lista:

```
(FOREACH Pto '((10 10) (20 20) (25 40) (100 170)) (COMMAND "_line" "0,0"
Pto ""))
```

```
(APPLY función lista)
```

APPLY aplica la función indicada a todos los elementos de una lista también indicada. Ejemplo:

```
(APPLY '* '(2 3 4))
```

Este ejemplo aplica la función * inherente a AutoLISP a la lista especificada. El resultado habría sido el mismo que si hubiéramos escrito:

```
(* 2 3 4)
```

aunque en determinadas situaciones puede ser interesante su uso.

Se aprecia que tanto la lista (como ya sabíamos) como la función indicada han de llevar un apóstrofo delante al ser literales. La función puede ser una *subr* de AutoLISP o una función definida previamente por el usuario.

```
(MAPCAR función lista1... listan)
```

Por su lado, MAPCAR aplica la función indicada a elementos sucesivos de listas. Por ejemplo, supongamos n listas cada una con un número m de elementos. MAPCAR aplicará la función especificada al primer elemento (1-1, 2-1,... n -1) de cada lista (*lista1*, *lista2*,... *listan*) y el resultado será guardado como primer elemento de la lista de resultado. Después realiza lo mismo con los m elementos de las n listas. El resultado final será una lista cúmulo de los resultados parciales. Veamos un ejemplo sencillo:

```
(MAPCAR '+ '(8 2 3) '(2 1 1) '(0 0 0))
```

El resultado será:

```
(10 3 4)
```

Las mismas consideraciones en cuanto a literales que para APPLY.

A continuación vamos a estudiar aquí una función que no es que tenga que ver con estas últimas, pero se suele utilizar con ellas, sobre todo con APPLY y MAPCAR. Esta función es:

```
(LAMBDA argumentos expresión1 [expresión2...])
```

LAMBDA define una función de usuario sin nombre. Su formato y funcionamiento es el mismo que DEFUN, pero al no tener nombre sólo puede utilizarse en el momento de definirla y no puede ser llamada posteriormente. Se utiliza cuando se necesita definir una función sólo momentáneamente y no se desea ocupar espacio en memoria de manera innecesaria.

LAMBDA devuelve el valor de la última expresión evaluada, lo mismo que DEFUN. Se puede usar en combinación con APPLY y MAPCAR —como decíamos— para aplicar una función temporal a los elementos de una o varias listas:

```
(APPLY '(LAMBDA (x y z) (/ (- x y) z))
      '(25 5 2))
```

En el ejemplo se define una función temporal con tres variables. Su cometido es restarle *y* a *x* y dividir el resultado entre *z*. Se aplica esa función con APPLY a la lista que suministra los tres argumentos requeridos. El resultado será $(25 - 5) / 2$, es decir 10.

De manera similar se utiliza con MAPCAR, cuando se quiere obtener una lista de resultados. Por ejemplo una función para dibujar líneas entre una serie de puntos iniciales y una serie de puntos finales podría ser:

```
(MAPCAR '(LAMBDA (pin pf) (COMMAND "linea" pin pf ""))
      (LIST pin1 pin2 pin3)
      (LIST pf1 pf2 pf3))
```

7ª fase intermedia de ejercicios

- Realizar un programa que dibuje círculos concéntricos. La aplicación solicitará el centro de la serie de círculos, al número de círculos y el radio interior y exterior del conjunto. Los círculos se dispondrán de manera equidistante.
- Realizar un programa que dibuje círculos concéntricos a partir de un círculo base. Los radios de los demás círculos se irán introduciendo a medida que se dibujan (por el usuario).

ONCE.10. MANEJO DE LISTAS

En esta sección, y avanzando un poco más en este curso, vamos a ver una serie de funciones de AutoLISP muy sencillas que se utilizan para el manejo de listas. Ya hemos visto en varios ejemplos tipos de listas, como las de las coordenadas de un punto, por ejemplo. Aprenderemos ahora a acceder o capturar todo o parte del contenido de una lista, así como a formar listas con diversos elementos independientes. El tema es corto y fácilmente asimilable, pero no por ello menos importante, ya que esta característica se utiliza mucho en la programación de rutinas AutoLISP, sobre todo a la hora de acceder a la Base de Datos interna de **AutoCAD**.

Lo primero que vamos a ver es cómo acceder a elementos de una lista. Para ello disponemos de una serie de funciones que iremos estudiando desde ahora.

```
(CAR lista)
```

La función CAR de AutoLISP devuelve el primer elemento de una lista. Si se indica una lista vacía `()` se devuelve `nil`, si no se devuelve al valor del elemento. Veamos un ejemplo. Si queremos capturar la coordenada X, para su posterior proceso, de un punto introducido por el usuario, podríamos introducir las líneas siguientes en nuestro programas:

```
(SETQ Coord (GETPOINT "Introduce un punto: "))
(SETQ X (CAR Coord))
```

De esta manera, guardamos en la variable `x` el primer elemento de la lista guardada en `Coord`, es decir la coordenada X del punto introducido por el usuario.

Recordemos que si se emplean listas directamente, éstas han de ir indicadas como literales (precedidas del apóstrofo):

```
(CAR '(5 20 30))
```

Si la lista sólo tiene un elemento se devuelve dicho elemento. Vemos unos ejemplos:

<code>(CAR '(/ 1 2.2) -80.2 -23.002 (* 2 3.3))</code>	devuelve <code>(/ 1 2.2)</code>
<code>(CAR '(34.45 décimo -12))</code>	devuelve 34.45
<code>(CAR '(x y z))</code>	devuelve <code>x</code>
<code>(CAR '(3))</code>	devuelve 3

`(CDR lista)`

Esta función devuelve una lista con los elementos segundo y siguientes de la lista especificada. Esto es, captura todos los elementos de una lista excepto el primero (desde el segundo, inclusive, hasta el final) y los devuelve en forma de lista. Si se especifica una lista vacía, `CDR` devuelve `nil`. Ejemplos:

<code>(CDR '(8 80.01 -23.4 23 34.67 12))</code>	devuelve <code>(80.01 -23.4 23 34.67 12)</code>
<code>(CDR '(x y z))</code>	devuelve <code>(y z)</code>
<code>(CDR (CAR '((1 2 4) (3 5 7) (8 1 2))))</code>	devuelve <code>(2 4)</code>

Si se indica una lista con dos elementos, `CDR` devuelve el segundo de ellos pero, como sabemos, en forma de lista. Para capturar una segunda coordenada Y de un punto 2D por ejemplo, habríamos de recurrir a la función `CAR` —vista antes— para obtener dicho punto. Véanse estos dos ejemplos:

<code>(CDR '(30 20))</code>	devuelve <code>(20)</code>
<code>(CAR (CDR '(30 20)))</code>	devuelve 20

De esta manera, es decir, con la mezcla de estas dos funciones se puede obtener la coordenada Y de cualquier punto, o el segundo elemento de cualquier lista, que es lo mismo:

<code>(CAR (CDR '(20 12.4 -3)))</code>	devuelve 12.4
<code>(CAR (CDR '(34 -23.012 12.33)))</code>	devuelve -23.012
<code>(CAR (CDR '(23 12)))</code>	devuelve 12
<code>(CAR (CDR '(10 20 30 40 50 60)))</code>	devuelve 20

Si se especifica una lista con sólo un elemento, al igual que con listas vacías se devuelve `nil`.

NOTA: Si la lista es un tipo especial de lista denominado *par punteado* con sólo dos elementos (se estudiará más adelante), `CDR` devuelve el segundo elemento sin incluirlo en lista alguna. Este tipo de listas es fundamental en la Base de Datos de **AutoCAD**, como se verá en su momento, y de ahí la importancia de estas funciones para acceder a objetos de dibujo y modificarlos.

Las funciones siguientes son combinaciones permitidas de las dos anteriores.

`(CADR lista)`

Esta función devuelve directamente el segundo elemento de una lista. Equivale por completo a `(CAR (CDR lista))`. De esta forma resulta mucho más cómoda para capturar segundos elementos, como por ejemplo coordenadas Y. Ejemplos:

<code>(CADR '(10 20 34))</code>	devuelve 20
<code>(CADR '(23 -2 1 34 56.0 (+ 2 2)))</code>	devuelve -2
<code>(CADR '(19 21))</code>	devuelve 21
<code>(CADR '(21))</code>	devuelve nil
<code>(CADR '())</code>	devuelve nil

El resto de las funciones más importante se explicarán con un solo ejemplo, el siguiente:

```
(SETQ ListaElem '((a b) (x y)))
```

`(CAAR lista)`

`(CAAR ListaElem)` devuelve A

Equivale a `(CAR (CAR ListaElem))`.

`(CDAR lista)`

`(CDAR ListaElem)` devuelve (B)

Equivale a `(CDR (CAR ListaElem))`.

`(CADDR lista)`

`(CADDR ListaElem)` devuelve nil

Equivale a `(CAR (CDR (CDR ListaElem)))`.

`(CADAR lista)`

`(CADAR ListaElem)` devuelve B

Equivale a `(CAR (CDR (CAR ListaElem)))`.

`(CADDAR lista)`

`(CADDAR ListaElem)` devuelve A

Equivale a `(CAR (CDR (CDR (CAR ListaElem))))`.

Y así todas las combinaciones posibles que podamos realizar. Como se ha visto, para obtener el tercer elemento (coordenada Z por ejemplo) de una lista utilizaremos `CADDR`:

`(CADDR '(30 50 75))` devuelve 75

En el ejemplo anterior, esta función habíamos visto que devolvía `nil`. Esto es porque era una lista de dos elementos, y si el elemento buscado no existe se devuelve, `nil`. Por ejemplo:

`(CDDDR '(30 60 90))` devuelve nil

La manera de construir funciones derivadas es bien sencilla. Todas comienzan con C y terminan con R. En medio llevan la otra letra, ya sea la A de CAR o la D de CDR, tantas veces como se repita la función y en el mismo orden. Veamos unos ejemplos:

```
CAR-CAR-CAR = CAAAR, p.e. (CAR (CAR (CAR ListaElem)))
CDR-CDR-CDR-CAR = CDDAR, p.e. (CDR (CDR (CDR (CAR ListaElem))))
CAR-CDR-CAR-CAR-CDR-CDR = CADAADDR, p.e. (CAR (CDR (CAR (CAR (CDR (CDR
                                         ListaElem))))))
```

Y así sucesivamente. Todas estas combinaciones son extremadamente útiles, tanto para manejar listas en general como para gestionar directamente la Base de Datos de **AutoCAD**.

Veamos ahora otra función muy útil y versátil.

`(LIST expresión1 [expresión2...])`

La función LIST reúne todas las expresiones indicadas y forma una lista con ellas, la cual devuelve como resultado. Se debe indicar al menos una expresión.

Imaginemos que queremos formar una lista de las tres coordenadas de un punto obtenidas por separado y guardadas en tres variables llamadas X, Y y Z. X vale 10, Y vale 20 y Z vale 30. Si hacemos:

```
(SETQ Punto (X Y Z))
```

AutoLISP devuelve error: bad function. AutoLISP intenta evaluar el paréntesis porque es una lista. Al comenzar comprueba que X no es ninguna función y da el mensaje de error.

Si hacemos:

```
(SETQ Punto '(X Y Z))
```

La lista con las tres coordenadas se guarda en Punto, pero ojo, como un literal. Si introducimos ahora lo siguiente el resultado será el indicado:

```
!Punto           devuelve (X Y Z)
```

Para ello tenemos la función LIST por ejemplo. Hagamos ahora lo siguiente:

```
(SETQ Punto (LIST X Y Z))
```

Y ahora, al hacer lo que sigue se devuelve lo siguiente:

```
!Punto           devuelve (10 20 30)
```

Hemos conseguido introducir valores independientes en una lista asignada a una variable.

Vamos a ver un ejemplo de un programa que utiliza estas funciones. El listado del código es el siguiente:

```
(DEFUN Bornes (/ pti dia ptf ptm)
  (INITGET 1)
  (SETQ pti (GETPOINT "Punto inicial de conexión: "))(TERPRI)
  (INITGET 5)
  (SETQ dia (GETREAL "Diámetro de bornes: "))(TERPRI)
  (WHILE (SETQ ptf (GETPOINT "Punto de borne (INTRO para terminar): "))
```



```
(TERPRI)
(SETQ ptm (LIST (CAR pti) (CADR ptf)))
(COMMAND "_line" pti "_non" ptm "_non" ptf "")
(COMMAND "_donut" "0" (+ dia 0.0000001) "_non" ptf "")
)
)

(DEFUN c:bornes ()
  (SETVAR "cmdecho" 0)
  (Bornes)
  (SETVAR "cmdecho" 1)(PRIN1)
)

(PROMPT "Nuevo comando BORNES definido")(PRIN1)
```

NOTA: En programas que definan más de una función (este no es el caso), sin contar la que empieza con `c:`, deberemos de poner cuidado a la hora definir variables locales. Si lo hacemos por ejemplo en un `DEFUN` y luego otro necesita de esas variables, el segundo no funcionará. Las variables locales únicamente funcionan para su función, es decir para su `DEFUN`. La forma de conseguir que fueran variables locales compartidas —sólo dentro del propio programa— sería declarándolas en el `DEFUN` que sea comando de **AutoCAD** (`c:`).

Este último ejemplo solicita los datos necesarios y comienza el bucle de `WHILE`. La condición es un tanto extraña pero fácil de comprender. Sabemos que `WHILE` acepta una condición como válida si no devuelve `nil`, por lo tanto la condición es el propio valor de la variable `ptf`. Al darle un valor mediante `GETPOINT`, `WHILE` continuará. En el momento en que pulsemos `INTRO` para terminar el programa, `ptf` no tendrá valor, será `nil`, por lo que `WHILE` no prosigue y acaba.

El bucle lo que realiza es guardar en la variable `ptm` el valor de una lista, formada mediante la función `LIST`, y que guarda el primer elemento de la lista guardada en `pti` (punto inicial de conexión), es decir la coordenada X, y el segundo elemento de la lista guardada en `ptf` (punto de situación del borne), la coordenada Y. Después se dibujan la línea vertical y horizontal de conexión y el borne en el extremo (mediante `ARANDELA`).

8ª fase intermedia de ejercicios

- Realizar un programa que dibuje rectángulos con grosor y con esquinas redondeadas. Se solicitará al usuario el grosor del rectángulo, el radio de redondeo de las esquinas y la primera y segunda esquina del rectángulo en sí.
- Realícese un programa que dibuje ventanas con celosía en cruz. Al usuario se le solicitará el grosor de rectángulo exterior y la anchura de los marcos. Así también, evidentemente, la posición de dos vértices opuestos por una de las diagonales del rectángulo.

ONCE.11. FUNCIONES DE CONVERSIÓN DE DATOS

De lo que hablaremos en esta sección es de la posibilidad que tenemos mediante AutoLISP de conversión de los tipos de datos disponibles para utilizar, esto es, valores enteros, valores reales, ángulos, distancias y cadenas de texto alfanumérico. Además, y en último término, se explicará una función que es capaz de convertir cualquier valor de un tipo de unidades a otro.

Con lo que comenzaremos será con una función capaz de convertir cualquier valor (entro o real) en un valor real. Esta función es la siguiente:

(FLOAT valor)

valor determina el número que queremos convertir. Si es real lo deja como está, si el entero lo convierte en real. Veamos unos ejemplos:

(FLOAT 5)	devuelve 5.0
(FLOAT 5.25)	devuelve 5.25
(FLOAT -3)	devuelve -3.0
(FLOAT 0)	devuelve 0

(ITOA valor_entero)

Esta otra función convierte un valor entero, y sólo entero, en una cadena de texto que contiene a dicho valor. Por ejemplo:

(ITOA 5)	devuelve "5"
(ITOA -33)	devuelve "-33"

ITOA reconoce el signo negativo si existe y lo convierte en un guión.

Esta función resultará especialmente útil cuando se explique en este mismo **MÓDULO** la interacción con letreros de diálogo en DCL. Además, puede servir para introducir valores de variables en una concatenación de cadenas, por ejemplo, que próximamente veremos.

NOTA: Si se especifica un número real o una cadena como argumento de ITOA se produce un error de AutoLISP.

(RTOS valor_real [modo [precisión]])

RTOS convierte valores reales en cadenas de texto. Al contrario que ITOA, RTOS admite números enteros. Veamos algún ejemplo:

(RTOS 33.4)	devuelve "33.4"
(RTOS -12)	devuelve "-12"

El argumento *modo* se corresponde con la variable de **AutoCAD** LUNITS. Es decir, solamente puede ser un número entero entre 1 y 5 cuyo formato es el que se indica:

<i>modo</i>	Formato
1	Científico
2	Decimal
3	Pies y pulgadas I (fracción decimal)
4	Pies y pulgadas II (fracción propia)
5	Fraccionario

Si no se especifica se utiliza el formato actual de la variable en cuestión. Así:

(RTOS 34.1 1)	devuelve "3.4100E+01"
(RTOS 34.1 2)	devuelve "34.1"
(RTOS 34.1 3)	devuelve "2'-10.1''"
(RTOS 34.1 4)	devuelve "2'-10 1/8"
(RTOS 34.1 5)	devuelve "34 1/8"

El argumento *precisión* se corresponde con la variable `LUPREC` e indica la precisión en decimales para la cadena de texto que se desea obtener. Si no se indica, y al igual que con el argumento *modo*, se supone el valor de variable en la sesión actual de dibujo. Así:

<code>(RTOS (/ 1 3) 2 0)</code>	devuelve "0"
<code>(RTOS (/ 1 3) 2 1)</code>	devuelve "0.3"
<code>(RTOS (/ 1 3) 2 4)</code>	devuelve "0.3333"
<code>(RTOS (/ 1 3) 2 13)</code>	devuelve "0.33333333333333"
<code>(RTOS (/ 1 3) 2 16)</code>	devuelve "0.3333333333333333"

NOTA: Como deberíamos saber, **AutoCAD** internamente trabaja siempre con 16 decimales, indique lo que se le indique, otra cosa es la forma en que nos devuelva los resultados. Es por ello que a `RTOS` podemos indicarle una precisión superior a dieciséis, pero lo máximo que nos va a devolver serán esos dieciséis decimales.

Otros ejemplos:

<code>(RTOS 2.567 1 2)</code>	devuelve "2.57E+00"
<code>(RTOS -0.5679 5 3)</code>	devuelve "-5/8"
<code>(RTOS 12 3 12)</code>	devuelve "1' "

NOTA: La variable `UNITMODE` tiene efecto en los modos 3, 4 y 5.

`(ANGTOS valor_angular [modo [precisión]])`

Esta *subr* de AutoLISP toma el valor de un ángulo y lo devuelve como cadena de texto. Dicho valor habrá de ser un número en radianes.

El argumento *modo* se corresponde con la variable `AUNITS` de **AutoCAD**. Sus valores están en el intervalo de 0 a 4 según la siguiente tabla:

<i>modo</i>	Formato
0	Grados
1	Grados/minutos/segundo
2	Grados centesimales
3	Radianes
4	Unidades geodésicas

Por su lado, *precisión* se corresponde con la variable `AUPREC` de **AutoCAD**. Especifica el número de decimales de precisión. Veamos algunos ejemplos:

<code>(ANGTOS PI 0 2)</code>	devuelve "180"
<code>(ANGTOS 1.2 3 3)</code>	devuelve "1.2r"
<code>(ANGTOS (/ PI 2.0) 0 4)</code>	devuelve "90"
<code>(ANGTOS 0.34 2 10)</code>	devuelve "21.6450722605g"
<code>(ANGTOS -0.34 2 10)</code>	devuelve "378.3549277395g"

El ángulo indicado puede ser negativo, pero el valor es siempre convertido a positivo entre 0 y 2π .

NOTA: La variable `UNITMODE` afecta sólo al modo 4.

Veamos ahora las funciones inversas o complementarias a estas tres últimas explicadas.

(ATOI *cadena*)

ATOI convierte la cadena especificada en un número entero. Si la cadena contiene decimales la trunca. Ejemplos:

(ATOI "37.4")	devuelve 37
(ATOI "128")	devuelve 128
(ATOI "-128")	devuelve -128

Si ATOI encuentra algún carácter ASCII no numérico en la cadena, únicamente convierte a numérico hasta dicho carácter. Si no hay ningún carácter numérico y son todos no numéricos, ATOI devuelve 0. Por ejemplo:

(ATOI "-12j4")	devuelve -12
(ATOI "casita")	devuelve 0

(ATOF *cadena*)

Convierte cadenas en valores reales. Admite el guión que convertirá en signo negativo. Las mismas consideraciones con respecto a caracteres no numéricos que para ATOI. Ejemplos:

(ATOF "35.78")	devuelve 35.78
(ATOF "-56")	devuelve -56.0
(ATOF "35,72")	devuelve 35.0
(ATOF "23.3h23")	devuelve 23.3
(ATOF "pescado")	devuelve 0.0

(DISTOF *cadena* [*modo*])

DISTOF convierte una cadena en número real. El argumento *modo* especifica el formato del número real y sus valores son los mismos que los explicados para RTOS. Si se omite *modo* se toma el valor actual de LUNITS. Se pueden probar los ejemplos inversos a RTOS, son complementarios.

(ANGTOF *cadena* [*modo*])

Convierte una cadena de texto, que representa un ángulo en el formato especificado en *modo*, en un valor numérico real. *modo* admite los mismo valores que ANGROS. Si se omite *modo* se toma el valor actual de la variable AUNITS. Se pueden probar los ejemplos inversos a ANGROS, son complementarios.

ONCE.11.1. Conversión de unidades

Veamos ahora una última función un poco diferente. CVUNIT convierte un valor indicado de un tipo de unidades a otro, ambos también especificado en la sintaxis. Dicha sintaxis es la que sigue:

(CVUNIT *valor* *unidad_origen* *unidad_destino*)

valor representa el valor numérico que se desea convertir. *unidad_origen* y *unidad_destino* son, respectivamente, la unidades actuales del valor y la unidades a las que se quiere convertir.

Estos dos argumentos últimos hay que especificarlos como cadenas (entre comillas dobles). Los nombres que contengan dichas cadenas deberán existir en el archivo `ACAD.UNT`, archivo de conversión de unidades suministrado con **AutoCAD** precisamente para el buen funcionamiento de esta función de AutoLISP. Este archivo es ASCII y puede ser editado y personalizado, por ello, vamos a estudiar aquí y ahora cómo crear nuestras propias definiciones de conversión de unidades.

ONCE.11.1.1. Personalizar el archivo `ACAD.UNT`

Al abrir este archivo mediante un editor ASCII podremos observar que se asemeja completamente a muchos de los archivos personalizables de **AutoCAD** que ya hemos aprendido a modificar y crear en otros **MÓDULOS** de este curso, como por ejemplo a los de definiciones de tipos de línea, patrones de sombreado o formas. El motivo de que se haya dejado esta explicación para este punto es la relación entre este archivo `ACAD.UNT` y la función `CVUNIT` de AutoLISP.

El archivo de definición de unidades de **AutoCAD**, `ACAD.UNT`, permite definir factores para convertir datos de un sistema de unidades a otro. Estas definiciones, que son utilizadas por la función de conversión de unidades `CVUNIT` de AutoLISP, deben incluirse en este archivo en formato ASCII.

Cada definición consta de dos líneas en el archivo: el nombre de la unidad y su definición. La primera línea debe llevar un asterisco (*) en la primera columna, seguido del nombre de la unidad. Este nombre puede llevar varias abreviaturas o formas de escritura alternativas separadas por comas. El siguiente formato permite incluir un nombre de unidad en singular y en plural:

```
*[ [común] [ ( [singular.] plural) ] ]...
```

Pueden especificarse varias expresiones (singular y plural). No es necesario que vayan situadas al final de la palabra, y tampoco es necesario incluir la forma en plural. Ejemplos:

```
*pulgada(s)
*mileni(o.os)
*pi(e.es)
*metro(s),meter(s),metre(s),m
```

En esta última línea por ejemplo, la unidad definida permite llamarla después como argumento de `CVUNIT` de las formas siguientes: `metro`, `metros`, `meter`, `meters`, `metre`, `metres` o `m`. En el caso de la unidad de medida en pies del ejemplo: `pie` o `pies`.

La línea que sigue a esta primera define la unidad como fundamental o derivada. Una unidad fundamental es una expresión formada por constantes. Toda línea que siga a la del nombre de la unidad y no empiece por un signo igual, define una unidad fundamental. Consta de cinco enteros y dos números reales, de la siguiente forma:

```
c, e, h, k, m, r1, r2
```

Los cinco enteros corresponden a los exponentes de estas cinco constantes:

Constante	Significado
-----------	-------------

<i>C</i>	Velocidad de la luz en el vacío
<i>E</i>	Carga del electrón
<i>H</i>	Constante de Planck
<i>K</i>	Constante de Boltzman
<i>M</i>	Masa del electrón en reposo

Todos estos exponentes juntos definen la magnitud medida por la unidad: longitud, masa, tiempo, volumen, etcétera.

El primer número real (*x1*) es un multiplicador, mientras que el segundo (*x2*) es un desplazamiento de escala aditivo que sólo se utiliza para conversiones de temperatura. La definición de una unidad fundamental permite escribir el nombre de distintas formas (por ejemplo, metro y m) y no importa que esté en mayúsculas o en minúsculas. A continuación se define una unidad fundamental a modo de ejemplo:

```
*metro(s),metro(s),m
-1,0,1,0,-1,4.1214856408e11,0
```

En este ejemplo, las constantes que forman un metro son

$$((1 / c) * h * (1 / m)) * (4.1214856 * (10 ^ 11))$$

Las unidades derivadas se definen en función de otras unidades. Si la línea que sigue a la del nombre de la unidad comienza con un signo igual (=) se trata de una unidad derivada. Los operadores válidos para estas definiciones son * (multiplicación), / (división), + (suma), - (resta) y ^ (exponenciación). Puede hacerse referencia a las unidades predefinidas bien por su nombre o bien por sus abreviaturas (si tienen). Los elementos que componen la fórmula se multiplican todos, a menos que se especifique lo contrario mediante el operador correspondiente. Por ejemplo, la base de datos de unidades define los nombres de múltiplos y submúltiplos sin magnitudes, por lo que pueden especificarse unidades como micropulgadas introduciendo micropulgada. A continuación ofrecemos algunas definiciones de unidades derivadas a modo de ejemplo.

```
; Unidades de superficie
```

```
*township(s)
=93239571.456 meter^2
```

Se define una ciudad (*township*) como 93.239.571,456 metros cuadrados. Como vemos, las unidades cuadradas o cúbicas se indican mediante un nombre de unidad definido, el signo de exponenciación y el exponente. Metros cuadrados podría ser: meter^2, m^2, metros^2 y todas las demás combinaciones posibles; metros cúbicos: m^3, metro^3,...

```
; Unidades electromagnéticas
```

```
*voltio(s),v
=vatio/amperio
```

En este ejemplo se define un voltio como el resultado de dividir un vatio por un amperio. En el archivo ACAD.UNT, tanto los vatios como los amperios están definidos como unidades fundamentales.

Como podemos observar, para incluir comentarios basta con colocar al principio de la línea un punto y coma. El comentario continúa hasta el final de la línea.

ONCE.11.1.2. Ejemplos de CVUNIT

Se pueden convertir de unas unidades a otras no sólo valores numéricos sino también valores de punto (listas de dos o tres coordenadas). Veamos algunos ejemplos:

(CVUNIT 180 "degree" "radian")	devuelve 3.14159
(CVUNIT 10 "cm" "inch")	devuelve 3.93701
(CVUNIT 25 "celsius" "kelvin")	devuelve 298.15
(CVUNIT 1.25 "horas" "segundos")	devuelve 4500
(CVUNIT 2500 "m^2" "acre")	devuelve 0.617763
(CVUNIT 15 "kg" "libras")	devuelve 33.0693
(CVUNIT '(2 5 7) "mm" "pulgadas")	devuelve (0.0787432 0.19685 0.275591)
(CVUNIT 760 "grados" "círculo")	devuelve 2.11111

Para la conversión de unidades, AutoLISP necesita acceder cada vez al archivo ACAD.UNT y leer su contenido. Esto resulta asaz lento, por eso, si un programa requiere efectuar una conversión de diversos valores a las mismas unidades, es preferible calcular un factor con CVUNIT para un valor 1 y, después emplear este factor con los demás valores numéricos.

NOTA: Si alguna de las unidades no existe o la conversión resulta incoherente, AutoLISP devuelve nil.

9ª fase intermedia de ejercicios

- Realícense diversos ejercicios de conversión de datos y unidades.
- Desarrollar un programa que dibuje una curva helicoidal tridimensional sin grosor mediante una spline. Se indicará el radio inicial, radio final, precisión en puntos en cada vuelta, número de vueltas y paso o altura (se dará para elegir). La curva se generará en el plano XY del SCP actual y alineada con el eje Z.

ONCE.12. MANIPULACIÓN DE CADENAS DE TEXTO

Explicaremos a continuación todo lo referente a las funciones de AutoLISP para el manejo de cadenas de texto. Es frecuente en un programa la aparición de mensajes en la línea de comandos, para la solicitud de datos por ejemplo. Pues bien, muchas veces nos interesará utilizar las funciones que aprenderemos a continuación para que dichos mensajes sean más interesantes o prácticos. Además, determinadas especificaciones de un dibujo en la Base de Datos de **AutoCAD** se encuentran almacenadas como cadenas de texto, léase nombres de capa, estilos de texto, variables de sistema, etcétera. Por todo ello, será muy interesante asimilar bien los conocimientos sobre cadenas de texto para ascender un escalafón más en la programación en AutoLISP para **AutoCAD**.

Comencemos pues, sin más dilación, con una función sencilla:

```
(STRCASE cadena [opción])
```

STRCASE toma la cadena de texto especificada en *cadena* y la convierte a mayúsculas o minúsculas según *opción*. Al final se devuelve el resultado de la conversión.

Si opción no existe o es nil, la cadena se convierte a mayúsculas. Si opción es T, la cadena se convierte a minúsculas. Veamos unos ejemplos:

(STRCASE "Esto es un ejemplo")	devuelve "ESTO ES UN EJEMPLO"
(STRCASE "Esto es un ejemplo" nil)	devuelve "ESTO ES UN EJEMPLO"
(STRCASE "Esto es un ejemplo" T)	devuelve "esto es un ejemplo"
(STRCASE "Esto es un ejemplo" (= 3 3))	devuelve "esto es un ejemplo"
(STRCASE "Esto es un ejemplo" (/= 3 3))	devuelve "ESTO ES UN EJEMPLO"
(STRCASE "MINÚSCULAS" T)	devuelve "minúsculas"
(STRCASE "mayúsculas")	devuelve "MAYÚSCULAS"

La siguiente función es muy usada a la hora de programar, como veremos. STRCAT, que así se llama, devuelve una cadena que es la suma o concatenación de todas las cadenas especificadas. Veamos su sintaxis:

```
(STRCAT cadena1 [cadena2...])
```

Un ejemplo puede ser el siguiente:

```
(SETQ cad1 "Esto es un ")
(SETQ cad2 "ejemplo de")
(SETQ cad3 " concatenación ")
(SETQ cad4 "de cadenas.")
(STRCAT cad1 cad2 cad3)
```

Esto devuelve lo siguiente:

```
"Esto es un ejemplo de concatenación de cadenas."
```

Como vemos, ya sea en un lado o en otro, hemos de dejar los espacios blancos convenientes para que la oración sea legible. Un espacio es un carácter ASCII más, por lo que se trata igual que los demás.

Los argumentos de STRCAT han de ser cadenas forzosamente, de lo contrario AutoLISP mostrará un mensaje de error.

NOTA: Recordamos que al final de este **MÓDULO** existe una sección en la que se muestran todos los mensajes de error de AutoLISP con sus significados correspondientes.

Cada cadena únicamente puede contener 132 caracteres, sin embargo es posible concatenar varios textos hasta formar cadenas más largas.

Una utilidad muy interesante de esta función es la de visualizar mensajes que dependen del contenido de ciertas variables, por ejemplo:

```
(SETQ NombreBloque (GETSTRING "Nombre del bloque: "))
(SETQ PuntoIns (GETPOINT (STRCAT "Punto de inserción del
                               bloque " NombreBloque ": ")))
```

Y también con variables de tipo numérico, que deberemos convertir antes en un cadena con alguna de las funciones aprendidas en la sección anterior:

```
(SETQ Var1 (GETINT "Radio del círculo base: "))
(SETQ Var2 (GETINT (STRCAT "Número de círculos de radio " (ITOA Var1)
                          " que se dibujarán en una línea")))
```


De esta manera, pensemos que podemos introducir, en esa pequeña cuña que es la variable dentro del texto, el último dato introducido por el usuario como valor por defecto, por ejemplo. Lo veremos en algún ejemplo o ejercicio a lo largo de este **MÓDULO**.

`(SUBSTR cadena posición [longitud...])`

Esta función extrae *longitud* caracteres de *cadena* desde *posición* inclusive. Esto es, devuelve una subcadena, que extrae de la cadena principal, a partir de la posición indicada y hacia la derecha, y que tendrá tantos caracteres de longitud como se indique.

Tanto la posición de inicio como la longitud han de ser valores enteros y positivos. Veamos unos ejemplos:

```
(SETQ Cadena "Buenos días")

(SUBSTR Cadena 2 3)      devuelve "uen"
(SUBSTR Cadena 1 7)      devuelve "Buenos "
(SUBSTR Cadena 7 1)      devuelve " "
(SUBSTR Cadena 11 1)     devuelve "s"
(SUBSTR Cadena 11 17)    devuelve "s"
(SUBSTR Cadena 1 77)     devuelve "Buenos días"
```

`(STRLEN [cadena1 cadena2...])`

STRLEN devuelve la longitud de la cadena indicada. Si no se indica ninguna o se indica una cadena vacía (""), STRLEN devuelve 0. El valor de la longitud es un número entero que expresa el total de caracteres de la cadena. Si se indican varias cadenas devuelve la suma total de caracteres. Ejemplos:

```
(STRLEN "Buenos días")      devuelve 11
(STRLEN "Hola" "Buenos días") devuelve 15
(STRLEN)                   devuelve 0

(SETQ C1 "Hola, " C2 "buenos días.")
(STRLEN (STRCAT C1 C2))     devuelve 18
```

`(ASCII cadena)`

ASCII devuelve un valor entero que es el código decimal ASCII del primer carácter de la cadena indicada. Veamos unos ejemplos:

```
(ASCII "d")      devuelve 100
(ASCII "7")      devuelve 55
(ASCII "+")      devuelve 43
(ASCII "AutoLISP") devuelve 65
(ASCII "Programación") devuelve 80
```

Esta función puede ser interesante a la hora de capturar pulsaciones de teclas. Veamos el siguiente ejemplo:

```
(SETQ Tecla (GETSTRING "Teclee un radio o INTRO para terminar: "))
(while (/= (ASCII Tecla) 0)
  (prompt "Aún no terminamos...")
  (SETQ Tecla (GETSTRING "\nTeclee un radio o INTRO para terminar: "))
)
(prompt "FIN.")
```

En el momento en que pulsemos `INTRO`, *Tecla* guardará una respuesta nula cuyo código ASCII es 0. En ese momento el programa acabará. No confundir con el código ASCII del `INTRO` que es el 13, que no podríamos utilizar porque lo que se guarda en *Tecla* —que es lo que se compara— al pulsar `INTRO` es una cadena vacía `" "`.

`(CHR código_ASCII)`

`CHR` funciona complementariamente a `ASCII`, es decir, devuelve el carácter cuyo código ASCII coincide con el valor especificado. Ejemplos:

```
(CHR 54)      devuelve "6"
(CHR 104)     devuelve "h"
(CHR 0)       devuelve " "
```

NOTA: Apréciase que `CHR` devuelve cadenas de texto entrecomilladas.

`(WCMATCH cadena filtro)`

Esta función aplica un filtro o patrón a la cadena de texto. Se compara pues la cadena con dicho patrón indicado y se devuelve `T` si lo cumple; si no se devuelve `nil`.

La manera de formar filtros es mediante un conjunto de caracteres globales o comodín, que algunos recuerdan a la forma de trabajo al más puro estilo MS-DOS. La relación y significado de los posibles filtros utilizables se muestra en la siguiente tabla:

Carácter	Nombre	Definición
#	Almohadilla	Cualquier dígito numérico.
@	A de arroba	Cualquier carácter alfabético.
.	Punto	Cualquier carácter no alfanumérico.
*	Asterisco	Cualquier secuencia de caracteres, incluida una vacía.
?	Signo de interrogación	Cualquier carácter.
~	Tilde (ALT+126)	Si es el primer carácter del patrón, cualquier elemento excepto el patrón.
[...]	Corchetes quebrados	Cualquiera de los caracteres encerrados.
[~...]	~ + []	Cualquiera de los caracteres no encerrados.
-	Guión	Entre corchetes siempre para especificar un rango para un carácter único.
,	Coma	Separa dos patrones.
`	Apóstrofo invertido	Omite caracteres especiales (lee el siguiente carácter de forma literal).

Nota: Si la cadena es muy larga se comparan sólo del orden de 500 caracteres.

Veamos una serie de ejemplos:

— Detectar si una cadena comienza con la letra "B":

```
(WCMATCH "Bloques" "B*")      devuelve T
```

— Detectar si una cadena tiene cinco caracteres:

(WCMATCH "Bloques" "?????") devuelve nil

— Detectar si una cadena contiene la letra "q":

(WCMATCH "Bloques" "*q*") devuelve T

— Detectar si una cadena no contiene ninguna letra "q":

(WCMATCH "Bloques" "~*q*") devuelve nil

— Detectar si una cadena contiene una coma (hay que indicar el literal de la coma):

(WCMATCH "Bloques,armario" "',',*") devuelve T

— Detectar si una cadena comienza con la letra "B" o "b":

(WCMATCH "Bloques" "B*,b*") devuelve T

— Detectar si una cadena comienza por un carácter en mayúscula (cualquiera):

(WCMATCH "Bloques" "[A-Z]*") devuelve T

(READ [cadena])

Veamos una función muy útil. READ devuelve la primera expresión de la cadena indicada. Si la cadena no contiene ningún paréntesis y es un texto con espacios en blanco, READ devuelve el trozo de texto hasta el primer espacio (en general será la primera palabra del texto).

Si la cadena contiene paréntesis, se considera su contenido como expresiones en AutoLISP, por lo que devuelve la primera expresión. Se recuerda que los caracteres especiales que separan expresiones en AutoLISP son: *espacio blanco*, (,), ', " y ;. A continuación se ofrecen unos ejemplos:

(READ "Buenos días")	devuelve BUENOS
(READ "Hola;buenas")	devuelve HOLA
(READ "Estoy(más o menos)bien")	devuelve ESTOY

Hay un aspecto muy importante que no debemos pasar por alto, y es que READ examina la cadena de texto pero analiza su contenido como si fueran expresiones AutoLISP. Por ello devuelve no una cadena de texto, sino una expresión de AutoLISP. De ahí que los ejemplos anteriores devuelvan un resultado que está en mayúsculas.

Y es que la utilidad real de READ no es analizar contenidos textuales, sino expresiones de AutoLISP almacenadas en cadenas de texto. Por ejemplo:

(READ "(setq x 5)")	devuelve (SETQ X 5)
(READ "(SetQ Y (* 5 3)) (SetQ Z 2)")	devuelve (SETQ Y (* 5 3))

Es decir que devuelve siempre la primera expresión AutoLISP contenida en la cadena de texto. Si sólo hay una devolverá esa misma.

Estas expresiones pueden ser posteriormente evaluadas mediante la función EVAL cuya sintaxis es:

`(EVAL expresión)`

Esta función evalúa la expresión indicada y devuelve el resultado de dicha evaluación. Así por ejemplo:

```
(EVAL (SETQ x 15))
```

devuelve

```
15
```

Esto equivale a hacer directamente `(SETQ x 15)`, por lo que parece que en principio no tiene mucho sentido. Y es que la función `EVAL` únicamente cobra sentido al utilizarla junto con la función `READ`.

Vamos a ver un ejemplo que ilustra perfectamente el funcionamiento de `READ` y `EVAL` juntos. Aunque la verdad es que no es un ejemplo muy práctico, ya que requiere conocimientos de AutoLISP por parte del usuario del programa, pero examinémoslo (más adelante se mostrará otro ejemplo mejor). Además este programa nos ayudará a afianzar conocimientos ya aprehendidos:

```
(DEFUN datos_curva ( / mens fun fundef pini pfin y1)
  (IF fun0 () (SETQ fun0 ""))
  (SETQ mens
    (STRCAT "Expresión de la función en X <" fun0 ">: "))
  (IF (= "" (SETQ fun (GETSTRING T mens))) (SETQ fun fun0))(TERPRI)
  (SETQ fundef (STRCAT "(defun curvaf (x)" fun "))")
  (EVAL (READ fundef))
  (INITGET 1)
  (SETQ pini (GETPOINT "Inicio de curva en X: "))(TERPRI)
  (SETQ x1 (CAR pini) yb (CADR pini))
  (SETQ y1 (+ yb (curvaf x1)))
  (SETQ p1 (LIST x1 y1))
  (SETQ fun0 fun)
  (SETQ orto0 (GETVAR "orthomode")) (SETVAR "orthomode" 1)
  (INITGET 1)
  (SETQ pfin (GETPOINT pini "Final de curva en X: "))(TERPRI)
  (SETQ xf (CAR pfin))
  (WHILE (= xf x1)
    (PROMPT "Deben ser dos puntos diferentes.")(TERPRI)
    (INITGET 1)
    (SETQ pfin (GETPOINT pini "Final de curva en X: "))(TERPRI)
    (SETQ xf (CAR pfin))
  )
  (INITGET 7)
  (SETQ prx (GETREAL "Precisión en X: "))(TERPRI)
  (IF (< xf x1) (SETQ prx (- prx)))
  (SETQ n (ABS (FIX (/ (- xf x1) prx))))
)

(DEFUN curva (/ x2 y2 p2)
  (COMMAND "_pline" p1)
  (REPEAT n
    (SETQ x2 (+ x1 prx))
    (SETQ y2 (+ yb (curvaf x2)))
    (SETQ p2 (LIST x2 y2))
    (COMMAND p2)
    (SETQ x1 x2 p1 p2)
  )
)
```

```
)
)

(DEFUN ult_curva (/ p2 yf)
  (SETQ yf (+ yb (curvaf xf)))
  (SETQ p2 (list xf yf))
  (COMMAND p2 "")
)

(DEFUN C:Curva (/ xf yb prx p1 n x1)
  (SETVAR "cmdecho" 0)
  (SETQ refnt0 (GETVAR "osmode"))(SETVAR "osmode" 0)
  (COMMAND "_undo" "_begin")
  (datos_curva)
  (curva)
  (ult_curva)
  (COMMAND "_undo" "_end")
  (SETVAR "osmode" refnt0)(SETVAR "cmdecho" 1)(PRIN1)
)

(PROMPT "Nuevo comando CURVA definido.")(PRIN1)
```

El programa dibuja el trazado de la curva de una función cualquiera del tipo $y = f(x)$. Para ello se solicita al usuario la expresión de la curva, que habrá de introducir con el formato de una expresión de AutoLISP; por ejemplo $(+ (* 5 x x) (- (* 7 x)) 3)$ se correspondería con la función $y = 5x^2 - 7x + 3$. El programa también solicita el punto inicial y final de la curva, así como el grado de precisión de la misma, ya que se dibujará con tramos rectos de polilínea. Esta precisión viene a ser la distancia entre puntos de la polilínea.

El primer paso del programa consiste en desactivar el eco de los mensajes, guardar en `refnt0` el valor de los modos de referencia activados (variable `OSMODE`) para luego restaurarlo, y poner dicha variable a 0, y colocar una señal de *inicio* del comando `DESHACER`. Tras ejecutar todas las funciones, se coloca una señal de *fin* y todo esto para que se puedan deshacer todas las operaciones del programa con un solo `H` o un solo `DESHACER`.

Esto es una práctica normal en los programas AutoLISP. Lo que ocurre es que los programas realizan una serie de ejecuciones de comandos de **AutoCAD** pero en el fondo, todo se encuentra soterrado transparentemente bajo un único comando. Si no estuviéramos conformes con el resultado de una ejecución de un programa, al utilizar el comando `H` sólo se deshacería el último comando de la serie de comandos de la rutina. De la forma explicada se deshace todo el programa.

Lo primero que realiza el programa, tras lo explicado, es comprobar, con una función `IF`, si la variable `fun0` contiene alguna expresión o no. La forma de realizarlo es similar a un ejemplo de `WHILE` que ya se explico. En esta variable se guardará la última expresión introducida por el usuario y se utilizará como valor por defecto en la solicitud (siguientes líneas).

Lo que hace el `IF` es comprobar si `fun0` devuelve `T` o `nil`. Si devuelve `T` es que contiene algo, por lo que no hace nada (lista vacía `()`). Por el contrario, si devuelve `nil` es que está vacía, es decir, es la primera vez que se ejecuta el programa, por lo que la hace igual a una cadena vacía `""`. Esto se realiza así para que al imprimir el valor por defecto en pantalla, no se produzca ningún error al ser dicha variable `nil`. Es un método muy manido en la programación en AutoLISP.

NOTA: Nótese que la variable `fun0` no ha sido declarada como local en los argumentos de `DEFUN`. Esto es debido a que necesita ser global para guardarse en memoria y utilizarse en todas las ejecuciones del programa.

A continuación, el programa presenta el mensaje de solicitud de la función en la línea de comandos. Por defecto se presentará la última función introducida (`fun0`) si existe, si no los corchetes angulares estarán vacíos, pero no habrá ningún error. La manera de presentar este mensaje es mediante una concatenación de cadenas y un posterior `GETSTRING` sin texto. La función introducida por el usuario se guarda en `fun`. Luego, con el `IF` siguiente nos aseguramos de darle a `fun` el valor de `fun0` si `fun` es igual a una cadena vacía, es decir si se ha pulsado `INTRO` para aceptar la función por defecto.

Seguidamente se forma, mediante una concatenación de cadenas, la función completa, añadiéndole un `(DEFUN CURVAF (X)` por delante y un `)` por detrás. De esta manera tendremos una función de usuario evaluable por AutoLISP.

NOTA: Esta manera de definir funciones con una variable asociada se expone al final de esta explicación del ejercicio.

A continuación se evalúa mediante `EVAL` la función contenida en la cadena `fundef` que se lee con la función de AutoLISP `READ`. El resultado es que se ejecuta el `DEFUN` y la función `curvaf` queda cargada en memoria para su posterior utilización.

Ahora se pide el punto de inicio de la curva en X, y se capturan sus coordenadas X e Y en las variables `x1` e `y1` mediante las funciones `CAR` y `CADR`. Inmediatamente se calcula el inicio en Y (`y1`) llamando a la recién creada función `curvaf` y se guarda el punto como una lista de sus coordenadas (`LIST`) en `p1`. Después se guarda en `fun0` el valor de `fun` para que en próximas ejecuciones del programa aparezca como opción por defecto.

A continuación se guarda en `orto0` el valor de `ORTHOMODE` —para después restaurar— y se establece a 1 para activarlo. De esta forma se indica que la curva se trazará con una base horizontal. Se pregunta por la coordenada X final y se introduce el control del `WHILE` para que las coordenadas X inicial y final sean diferentes. Se restablece el valor de `ORTHOMODE`.

Por último en cuestión de solicitud de datos se solicita la precisión en X. Si el punto final está a la izquierda del inicial se establece la precisión negativa. El programa calcula el número de tramos de polilínea que almacena en `n`. `FIX` toma el valor entero del cociente; este valor es el número de tramos completos. Para dibujar el último tramo con intervalo incompleto se utiliza la función `ult-curva`.

A continuación, y ya en la función `curva`, se produce el dibujo de los tramos completos de la curva y, en la función `ult-curva`, del último tramo incompleto. Fin de la aplicación.

Llegados a este punto toca explicar la nueva manera de definir funciones de usuario con `DEFUN`. Veamos el siguiente ejemplo:

```
(DEFUN Seno (x)
  (SETQ xr (* PI (/ x 180.0)))
  (SETQ s (SIN xr))
)
```

Como vemos, este ejemplo utiliza una variable global, pero que luego es utilizada como argumento de la operación cociente sin estar definida. Esto no es del todo cierto, la variable está definida en el `DEFUN`, lo que ocurre es que no tiene valor. Este tipo de variables se denominan asociadas, ya que se asocian a una expresión.

Así, al ejecutar este programa desde **AutoCAD** no podríamos hacer simplemente:

```
(seno)
```

ya que produciría un mensaje de error, sino que habría que introducir un valor directamente a su variable asociada, por ejemplo:

```
(seno 90)
```

lo que calcularía el seno de 90 grados sexagesimales. Veamos otro ejemplo:

```
(DEFUN Suma (x y z)
  (SETQ Sum (+ x y z))
)
```

Con este ejemplo habríamos de escribir en la línea de comandos, por ejemplo:

```
(suma 13 56.6 78)
```

10ª fase intermedia de ejercicios

- Realizar un programa que facilite la edición de los archivos ASCII de AutoLISP. La rutina solicitará el nombre del archivo que se desea editar, proponiendo por defecto el último editado que se aceptará pulsando **INTRO**. Tras esto, el programa correrá el Bloc de notas de Microsoft (suministrado con Microsoft Windows) con el fichero especificado abierto.

- Realizar un programa que sea capaz de distribuir un texto introducido por el usuario en forma de arco: alrededor de un centro y con un radio especificado. El texto se generará en sentido horario.

ONCE.13. ÁNGULOS Y DISTANCIAS

Tras el estudio de cadenas vamos a estudiar un pequeño grupo de funciones que nos permiten manejar dos de los tipos de datos más utilizados por **AutoCAD**: los ángulos y las distancias. Recordemos aquí que dentro de AutoLISP los ángulos se miden siempre en radianes (como en casi todos los lenguajes de programación existentes).

Comenzamos por una función que se encarga de medir ángulos. Esta función es:

```
(ANGLE punto1 punto2)
```

ANGLE devuelve el ángulo determinado por la línea que une los dos puntos especificados (**punto1** y **punto2**) y la dirección positiva del actual eje X en el dibujo. Así pues, entre **punto1** y **punto2** se traza una línea imaginaria y, el ángulo es el formado por esa línea con respecto al eje X positivo.

Como sabemos, el ángulo se mide en radianes y su sentido positivo es el antihorario o trigonométrico. Veamos un pequeño ejemplo:

```
(SETQ Inicio (GETPOINT "Primer punto: "))
(SETQ Final (GETPOINT Inicio "Segundo punto: "))
(SETQ Ang (ANGLE Inicio Final))
```

Para pasar este valor a grados sexagesimales, como comentamos ya, habría que hacer:

```
(SETQ AngSex (/ (* 180 Ang) PI))
```

Vemos que es una función muy similar a GETANGLE o GETORIENT. La diferencia estriba en que estas dos solicitan un ángulo, ya sea marcando dos puntos para calcularlo o por teclado, y ANGLE calcula el ángulo entre dos puntos. Si se indican dos puntos en pantalla con GETORIENT o GETANGLE el resultado es el mismo que con dos GETPOINT y la función ANGLE.

Es importante tener cuidado a la hora de introducir ambos puntos, argumentos de la función ANGLE. El orden en que sean introducidos determina la medida de un ángulo que no coincidirá en absoluto si se indica su posición de manera inversa. Así por ejemplo, si en el caso anterior se hubiera escrito (SETQ Ang (ANGLE Final Inicio)), el ángulo devuelto no se correspondería con el que se devuelve al escribir (SETQ Ang (ANGLE Inicio Final)). Esto mismo ocurriría con GETANGLE y GETORIENT.

NOTA: Si los puntos introducidos son en 3D, se proyectan ortogonalmente en el plano XY actual.

```
(DISTANCE punto1 punto2)
```

Esta función devuelve la distancia 3D entre los dos puntos especificados. Lógicamente, con DISTANCE es indiferente el orden de introducción de puntos. Funciona de la misma manera, con dos GETPOINT, que GETDIST. Pero DISTANCE se puede utilizar para calcular la distancia entre dos puntos cualesquiera del proceso de un programa, es decir, que no hayan sido solicitados directamente al usuario. Veamos un ejemplo:

```
(DISTANCE (GETPOINT "Primer punto: ") (GETPOINT "Segundo punto: "))
```

El valor devuelto por DISTANCE es un número real, distancia 3D entre ambos puntos de acuerdo a sus coordenadas en el SCP actual.

Las unidades son siempre decimales, independientemente de la configuración de unidades actual en el dibujo. Si uno de los puntos especificado es 2D (no se indica su coordenada Z), se ignorará la coordenada Z del otro punto y se devolverá una distancia 2D. Evidentemente si los dos puntos son 2D, la distancia es también 2D.

```
(POLAR punto ángulo distancia)
```

La función POLAR devuelve un punto obtenido mediante coordenadas relativas polares a partir del punto especificado, es decir, se devuelven las coordenadas de un punto. Desde *punto* se lleva *distancia* en la dirección marcada por *ángulo*. Como siempre, el ángulo introducido se considera en radianes y positivo en sentido trigonométrico. Aunque el punto introducido como argumento pueda ser 3D, el valor del ángulo (argumento también) se toma siempre respecto al plano XY actual.

Veamos un pequeño programa ejemplo de POLAR:

```
(DEFUN C:Balda (/ Punto1 Punto2 Ortho0 Dist)
  (SETQ Ortho0 (GETVAR "orthomode")) (SETVAR "orthomode" 1)
  (SETQ Punto1 (GETPOINT "Primer punto de la balda: ")) (TERPRI)
  (SETQ Punto2 (GETCORNER Punto1 "Segundo punto de la balda: ")) (TERPRI)
  (COMMAND "_rectang" Punto1 Punto2)
  (SETQ Dist (GETDIST Punto1 "Distancia a la siguiente balda: ")) (TERPRI)
  (COMMAND "_select" "_l" "")
  (WHILE (/= Dist nil)
    (COMMAND "_copy" "_p" "" Punto1 (POLAR Punto1 (/ PI 2) Dist))
    (SETQ Dist (GETDIST Punto1 "Distancia a la siguiente balda: "))
  )
  (SETVAR "orthomode" Ortho0)
)
```


Este programa dibuja baldas a distancias perpendiculares a la horizontal indicadas por el usuario. Tras el comienzo de la función se guarda el valor del modo Orto para restaurarlo posteriormente y se establece como activado. Se pregunta por el primer y segundo punto de la diagonal del rectángulo que formará la primera balda. Una vez hecho esto, se dibuja la balda.

La siguiente fase consiste en copiar dicha balda las veces que se necesite en perpendicular. Para ello se pregunta por la distancia a la siguiente balda; el punto de base siempre será la primera esquina dibujada de la primera balda. A continuación se establece el último objeto dibujado (la primera balda) como conjunto de selección para recurrir a él después como previo.

Ya dentro del bucle se van copiando baldas a los puntos designados por el usuario cada vez. Para ello se utiliza la función `POLAR`. Como punto de inicio se utiliza siempre el de la esquina primera de la primera balda —como ya se ha dicho—, como ángulo $\pi / 2$, es decir, 90 grados sexagesimales, y como distancia la que cada vez indique el usuario (variable `Dist`).

De este programa se sale pulsando `INTRO` cuando se nos pregunte por una distancia. Esto lo controla el bucle `WHILE` de la forma que ya se ha explicado alguna vez. En el momento en que se pulse `INTRO`, `Dist` será igual a `nil` y `WHILE` no continuará. Se saldrá del bucle y se restablecerá el valor original de Orto para acabar.

Veamos la última de este tipo de funciones. Es `INTERS` y se utiliza para obtener puntos por intersección entre dos líneas. No es exactamente una función que calcule ángulos o distancias, pero por su similitud de funcionamiento con ellas se ha incluido aquí. Su sintaxis es:

`(INTSERS punto1 punto2 punto3 punto4 [prolongación])`

Esta función toma los puntos `punto1` y `punto2` como extremos de una línea (aunque no lo sean), los puntos `punto3` y `punto4` como extremos de otra, y calcula el punto intersección de ambas, el cual devuelve. Veamos un ejemplo:

```
(INTERS '(10 10) '(20 20) '(15 10) '(0 50))
```

esto devuelve

```
(13.6364 13.6364)
```

que es el punto intersección.

El argumento *prolongación* es optativo. Si su valor es `nil`, la función `INTERS` considera las líneas como infinitas y devuelve su punto de intersección no sólo entre los límites indicados, sino también en su prolongación (si se cortan evidentemente). En este caso todas las líneas 2D tendrían intersección, salvo que fueran paralelas.

Si *prolongación* no se indica o su valor es diferente de `nil`, entonces el punto de intersección sólo se devuelve si se encuentra entre los límites indicados.

Si las rectas se cortan en su prolongación pero no está indicado el parámetro necesario, o si no se cortan de ninguna manera, `INTERS` devuelve `nil`. Veamos unos ejemplos:

<code>(INTERS '(10 10) '(20 20) '(15 10) '(20 0))</code>	devuelve <code>nil</code>
<code>(INTERS '(10 10) '(20 20) '(15 10) '(20 0) nil)</code>	devuelve <code>(13.3333 13.3333)</code>
<code>(INTERS '(10 10) '(20 20) '(15 10) '(20 0) T)</code>	devuelve <code>nil</code>

```
(INTERS '(10 10) '(20 20) '(15 10) '(20 0) (/= 2 2)) devuelve (13.3333 13.3333)
```

Hay que tener cuidado en indicar los cuatro puntos en el orden correcto, pues en caso contrario, las líneas cuya intersección calcula INTERS serán diferentes. Evidente.

11ª fase intermedia de ejercicios

- Créese un nuevo Modo de Referencia a Objetos que se “enganche” de puntos medios virtuales, es decir de puntos medios cualesquiera entre otros dos puntos que se habrán de señalar.
- Realizar un programa que dibuje números de marca de despieces. Se solicitará el diámetro del punto de señalización, el diámetro del círculo de marca, los puntos de situación y el número de marca. Tras esto, se dibujará respetando dichos datos.

ONCE.14. RUTINAS DE CONTROL DE ERRORES

Se va a explicar bajo esta sección una parte muy importante de la programación en cualquier lenguaje, sólo que aquí orientada al lenguaje que nos ocupa, orientada a AutoLISP. Esta parte a la cual nos referimos es el tratamiento de errores en un programa.

Con todo lenguaje de programación existe una función muy importante que debe realizar un programador a la hora de diseñar su aplicación. Esta función es la que se refiere al depuramiento o depuración de un programa, es decir, a la total eliminación de los errores de sintaxis o programación que pudiera contener un programa. Una vez hecho esto se puede decir que el programa funciona correctamente, pero ¿es esto cierto? Una depuración exhaustiva no consiste únicamente en revisar línea por línea de código buscando errores de sintaxis, variables mal declaradas, etc. Al programa hay que lograr ponerlo bajo las condiciones más duras de trabajo, hay que hacer que ejecute su listado al borde de los límites y más allá. En definitiva, hay que suponer todos los casos con los que se va encontrar un futuro usuario al ejecutar la aplicación.

Entre todos estos casos a los que nos referimos existen multitud de situaciones inestables que producen error en un programa y hacen que éste aborte su ejecución inesperadamente. Estas situaciones son las que debe prever el programador.

Un programa puede llegar a abortar no por un mal diseño de su código, sino simplemente porque se intente buscar un archivo para ser abierto en la unidad de disco flexible, por ejemplo, y el disco no esté introducido. El programador no puede adivinar la voluntad “maliciosa” de un usuario que no desea introducir el disco en la disquetera para que el programa “casque”, o el olvido, o simplemente el despiste del manejador del programa. Sin embargo puede suponer que el caso puede producirse e introducir una rutina que controle dicho error y no deje abortar al programa.

En AutoLISP el problema es un poco diferente; no disponemos de las instrucciones o funciones potentes de control de errores que disponen otros lenguajes de programación. Y es que con AutoLISP siempre se diseñarán aplicaciones, rutinas, comandos, etcétera que corran bajo **AutoCAD**, y que un programa nuestro aborte no quiere decir que vaya a abortar **AutoCAD**. Simplemente habrá sido un comando fallido.

Examinemos, por ejemplo, qué ocurre cuando intentamos extruir una polilínea abierta. Tras solicitarnos todos los datos pertinentes, **AutoCAD** intenta extruir el objeto y, cuando llega a la conclusión de que no puede, nos muestra un mensaje de error que nos indica la imposibilidad de realizar esa operación con ese objeto. Podría habernos dicho que no podía ser

al designar la polilínea, pero no, ha ocurrido al final. ¿Qué es lo que ha sucedido internamente? Pensemos en ello.

AutoCAD contiene un comando interno que posibilita la extrusión de objetos planos con superficie para convertirlos en sólidos. Este comando responde a un programa que ejecuta una serie de operaciones hasta llegar al resultado final. Si nosotros designamos, como en este caso, un objeto que no puede ser extruído, **AutoCAD** "se lo traga" en un principio, es decir, no comprueba si el objeto designado es factible de ser extruído o no. Pero al realizar las operaciones pertinentes un error (evidente) se produce en su código interno. El caso es que este error no se nos materializa de forma escandalosa haciendo que aborte **AutoCAD** o que el sistema quede inoperante —que así podría haber sido si no estuviera controlado—, sino que simplemente nos advierte "amablemente" que a susodicho objeto no se le puede aplicar una extrusión. Y acaba la orden.

Este es un buen ejemplo del control de errores que deberemos realizar nosotros. En AutoLISP no pretendemos que un comando creado no aborte, porque siempre lo hará si encuentra algún problema, sino que procuraremos canalizar ese error para que no parezca tan grave o aparatoso y simplemente acabe de manera "limpia" el programa o, por ejemplo, nos ofrezca la posibilidad de volver a ejecutar el comando (que no es lo normal).

Veremos por ejemplo que con Visual Basic el control de errores será mucho más exhaustivo y real.

NOTA: Los comandos y aplicaciones que realicemos para **AutoCAD** en AutoLISP siempre deben ser lo más parecido a los propios comandos y aplicaciones nativos del programa. Tanto en solicitud de opciones, como en control de errores, como en diseño de cuadros de diálogo, debemos guardar esa línea tan característica que distingue a **AutoCAD** de todos los demás.

Veamos pues la manera de aplicar esto con AutoLISP.

ONCE.14.1. Definir una función de error

La manera de definir una función de error en AutoLISP es, precisamente, con la función para definir funciones de usuario, con `DEFUN`. La función que concretamente hemos de definir no puede ser cualquiera, sino que ha de ser una concreta llamada `*error*`. La sintaxis para esta función especial se corresponde con cualquier definición de función, únicamente hemos de saber que el argumento siempre será la declaración de una variable global que guardará el texto del mensaje devuelto por AutoLISP en un evento de error.

La función `*error*` es una función predefinida de manipulación de errores. Al comenzar una sesión de dibujo, esta función tiene un tratamiento de errores por defecto. Cada vez que ocurre un error al ejecutarse un programa AutoLISP, se realiza el tratamiento incluido en ella. Lo que vamos a hacer nosotros simplemente es redefinirla para personalizar dicho tratamiento a nuestro gusto. Veamos un ejemplo:

```
(DEFUN *error* (cader)
  (IF (= cader "tipo de argumento erróneo") (err_arg))
)
```

En este ejemplo, al producirse un error, AutoLISP suministra el texto con su descripción como argumento a la función `*error*`. En este caso la variable `cader` almacena el valor de ese texto. Se examina haber si esa cadena es "tipo de argumento erróneo" y, en caso afirmativo se llama a una determinada función de tratamiento, en este caso `err_arg`, que hará lo que tenga que hacer.

Un tratamiento estándar sería el siguiente código:

```
(DEFUN *error* (Cadena)
  (PRINC "Error: ") (PRINC Cadena) (TERPRI)
  (PROMPT "*No válido.*") (PRIN1)
)
```

Cuando se produce un error de AutoLISP en un programa, éste queda abortado y se nos devuelve la parte del listado del programa donde se ha producido dicho error, además de todas las funciones que engloban esa parte, es decir, hasta el paréntesis más externo que es normalmente un DEFUN. Este listado puede resultar molesto y desagradable a los ojos del usuario. Una rutina sencilla para evitarlo es ésta expuesta aquí.

Al producirse un error, el texto de ese error se escribe con PRINC (que ya veremos en su momento) y, en lugar de ofrecer el listado, se ofrece un mensaje *No válido*. En general será necesario realizar un control más profundo como el que vamos a explicar ahora.

Sea el ejemplo siguiente, ya explicado anteriormente pero un poco mejorado con detalles que ya conocemos:

```
(DEFUN Aro (/ Centro Radio Grosor Rint Rext Dint Dext)
  (INITGET 1)
  (SETQ Centro (GETPOINT "Centro del aro: ")) (TERPRI)
  (INITGET 7)
  (SETQ Radio (GETDIST Centro "Radio intermedio: ")) (TERPRI)
  (INITGET 7)
  (SETQ Grosor (GETDIST "Grosor del aro: ")) (TERPRI)
  (INITGET "Hueco Relleno")
  (SETQ Op (GETKEYWORD "Aro Hueco o Relleno (<H>/R): ")) (TERPRI)
  (IF (OR (= Op "Hueco") (= Op \n))
    (PROGN
      (SETQ Rint (- Radio (/ Grosor 2)))
      (SETQ Rext (+ Radio (/ Grosor 2)))
      (COMMAND "_circle" Centro Rext)
      (COMMAND "_circle" Centro Rint)
    )
    (PROGN
      (SETQ Dint (* (- Radio (/ Grosor 2))2))
      (SETQ Dext (* (+ Radio (/ Grosor 2))2))
      (COMMAND "_donut" Dint Dext Centro "")
    )
  )
)

(DEFUN C:Aro ()
  (SETVAR "cmdecho" 0)
  (COMMAND "_undo" "_begin")
  (Aro)
  (COMMAND "_undo" "_end")
  (SETVAR "cmdecho" 1)
  (PRIN1)
)

(TERPRI)
(PROMPT "Nuevo comando Aro definido.") (PRIN1)
```

Este programa dibuja aros huecos o rellenos, según se indique. Mientras todo funcione correctamente y los datos introducidos no sean descabellados, no habrá problema alguno.

Pero probemos a pulsar ESC cuando se nos pida un dato. El resultado en línea de comandos será el siguiente, por ejemplo:

```
error: Function cancelled
(COMMAND "_circle" CENTRO REXT)
(PROGN (SETQ RINT (- RADIO (/ GROSOR 2))) (SETQ REXT (+ RADIO (/ GROSOR 2)))
(COMMAND "_circle" CENTRO REXT) (COMMAND "_circle" CENTRO RINT))
(IF (OR (= OP "Hueco") (= OP \N)) (PROGN (SETQ RINT (- RADIO (/ GROSOR 2)))
(SETQ REXT (+ RADIO (/ GROSOR 2))) (COMMAND "_circle" CENTRO REXT) (COMMAND
"_circle" CENTRO RINT)) (PROGN (SETQ DINT (* (- RADIO (/ GROSOR 2)) 2)) (SETQ
DEXT (* (+ RADIO (/ GROSOR 2)) 2)) (COMMAND "_donut" DINT DEXT CENTRO ")))
(ARO)
(C:ARO)
Diameter/<Radius> <19.6581>:
```

NOTA: Aquí se trabaja con los mensajes en inglés, pero con la versión castellana el modo de hacer es el mismo.

Vemos que el programa ha sido abortado. El texto de error que lanza AutoLISP es `Function cancelled` (detrás de `error:`). Estos son los mensajes que guardará la variable asociada a la función de control de errores. Vamos pues a intentar controlar este error.

A la función que define el comando de **AutoCAD** (`C:Aro`) le vamos a añadir la línea siguiente: `(SETQ error0 *error* *error* ControlErrores)`, justo detrás de la desactivación del eco de la línea de comandos (variable `CMDECHO`) por razones obvias. Con esto lo que hacemos es, primero guardar la definición predeterminada de la función predefinida `*error*` (en `error0`) para volver a restaurarla después; así nos aseguramos de dejarlo todo como estaba. Y segundo, una vez salvaguardado el contenido de esta función, asignarle el valor de la nueva función de control errores `ControlErrores`, que enseguida definiremos. Todo ello lo hacemos en un solo `SETQ`, que no despiste, lo podríamos haber hecho en dos.

Como siguiente paso evidente definiremos la nueva función `ControlErrores` así:

```
(DEFUN ControlErrores (CadenaError)
  (SETQ *error* error0)
  (IF (= CadenaError "Function cancelled")
    (PROMPT "La función ha sido interrumpida por el usuario.")
  )
  (TERPRI)
  (PRIN1)
)
```

De esta manera asociamos la variable `CadenaError` a la función, lo que hará que se guarde en dicha variable el contenido de la cadena de error lanzada por AutoLISP (si un error se produjera). A continuación restauramos el valor de la función predefinida `*error*`. Después comparamos el valor de `CadenaError` con el texto que proporciona AutoLISP si se presiona ESC a una petición y, si el valor es cierto, se muestra el mensaje del `PROMPT`. Por fin, realizamos un salto de línea con `TERPRI` para que el `*Cancel*` que devuelve AutoLISP no quede pegado a nuestro mensaje y acabamos con `PRIN1`.

Con esta función habremos controlado la salida de una pulsación de `ESC`. Podremos deducir fácilmente cómo hacer para que la pulsación de `ESC` devuelva únicamente el mensaje por defecto de **AutoCAD**, es decir el `*Cancel*`, sin nada más (como ocurre con los comandos del programa).

Si quisiéramos controlar así todos los errores, la rutina sería un tanto extensa. Por ello, lo que siempre se suele hacer es definir una rutina de tratamiento como la que sigue, por ejemplo:

```
(DEFUN ControlErrores (CadenaError)
  (SETQ *error* error0)
  (IF (= CadenaError "quit / exit abort")
    (PRINC "\nDatos no válidos. Fin del programa.")
    (PRINC (STRCAT "\nError: " CadenaError ".")))
  )
  (TERPRI)
  (PRIN1)
)
```

De esta forma controlamos si el error ha sido producido por una función `QUIT` o `EXIT` de AutoLISP (que enseguida veremos) o por cualquier otra circunstancia. Si el caso es el primero se muestra un mensaje fijo y, si es el segundo se muestra el mensaje `Error:` y a continuación el propio error de AutoLISP. De esta manera tenemos cubiertas todas las posibilidades.

Estos mensajes se pueden personalizar para cada tipo de error o incluso definir diversas funciones de tratamiento de errores para cada función del programa, por ejemplo. Así, si sabemos que el usuario habrá de introducir una serie de datos que, al final, pueden producir una división entre 0, si este error ocurre el mensaje podría indicar específicamente que el programa no ha podido continuar porque se ha intentado realizar una división entre cero no permitida (algo parecido a lo que explicábamos que hacía `EXTRUSION`).

Pero aún nos queda algún problema por resolver. Si nos fijamos en el ejemplo del listado devuelto tras un mensaje de error (antes de controlar) que hemos proporcionado anteriormente, al final del listado vemos que **AutoCAD** reanuda el comando `CIRCULO` pidiendo nuevos datos. Con el control que hemos proporcionado ya no aparecería, pero fijémonos en el ejemplo siguiente:

```
(DEFUN C:EscribeHola ()
  (COMMAND "_text" "_s" "fantastic" "\\ " "\\ " "\\ " "Hola")
)
```

Este pequeño ejemplo escribe `Hola` con el punto de inserción, la altura de texto y el ángulo de rotación proporcionados por el usuario (recordemos que `"\\"` es igual que `pause`, y espera en medio de un comando una introducción de un dato por el usuario). El problema es que si el estilo de texto `FANTASTIC` no existe el programa queda abortado pero **AutoCAD** sigue solicitando datos del comando `TEXT`.

La manera de solucionar esto es incluyendo en la función de tratamiento de errores una llamada a la función `COMMAND` sin argumentos. De esta forma se cancela cualquier comando en curso (algo parecido a los caracteres `^C^C` que incluíamos en macros de menús y botones de barras de herramientas). Así nuestra rutina de control de errores quedaría:

```
(DEFUN ControlErrores (CadenaError)
  (SETQ *error* error0)
  (IF (= CadenaError "quit / exit abort")
    (PRINC "\nDatos no válidos. Fin del programa.")
    (PRINC (STRCAT "\nError: " CadenaError ".")))
  )
  (COMMAND)
  (TERPRI)
  (PRIN1)
)
```

Y otro problema que queda en el aire es la restauración de las variables cambiadas, así como la colocación de la marca de *final* del comando `DESHACER`. Cuando un programa queda abortado se cede el control a la función de tratamiento de errores y ya no vuelve a pasar por el resto de las funciones. Por eso todo aquello que hayamos variado habremos de restaurarlo antes de terminar. Una forma, y siempre dependiendo del programa, podría ser:

```
(DEFUN ControlErrores (CadenaError)
  (SETQ *error* error0)
  (IF (= CadenaError "quit / exit abort")
    (PRINC "\nDatos no válidos. Fin del programa.")
    (PRINC (STRCAT "\nError: " CadenaError ".")))
  )
  (COMMAND "_undo" "_end")
  (SETVAR "blipmode" blip0) (SETVAR "osmode" refnt0)
  (SETVAR "cmdecho" 1)
  (COMMAND)
  (TERPRI)
  (PRIN1)
)
```

Así también, si el programa hubiera dibujado ya algo en pantalla antes de producirse el error, se podría hacer que lo deshiciera todo para no dejar rutinas a medias.

Nótese que hemos dicho que un el programa cede el control a la rutina de tratamiento de errores al producirse un error, no que aborte inesperadamente. Por lógica se puede deducir que podemos realizar un control tal que el programa, por ejemplo, vuelva a reiniciarse por completo o ceda el control a la función fallida para volver a introducir datos o lo que sea.

Como sabemos y ya hemos dicho, los comandos de **AutoCAD** no vuelven a repetirse si se da un error de estas magnitudes. Sí lo hacen a la hora de introducir datos, porque los vuelven a pedir si son erróneos, pero eso ya lo controlamos con los `INITGET`. Por lo tanto, lo más lógico será coincidir con la línea de **AutoCAD** y salir de la rutina, eso sí, de una forma controlada y vistosa.

Además de todo lo expuesto decir que la variable de sistema `ERRNO` almacena un valor cuando una llamada de función de AutoLISP provoca un error que **AutoCAD** detecta después. Las aplicaciones de AutoLISP pueden consultar el valor actual de `ERRNO` mediante `GETVAR` para obtener información de las causas de un error. A no ser que se consulte inmediatamente después de que una función AutoLISP informe de un error, el error que su valor indica puede ser engañoso. Esta variable siempre se inicializa a cero al empezar o abrir un dibujo.

NOTA: Si siempre utilizamos la misma rutina de control de errores, podemos escoger incluirla en el archivo `ACAD.LSP` para que siempre esté definida de la misma forma. La función de este archivo se estudiará en la sección **ONCE.15.**

NOTA: Al final de este **MÓDULO** existe una relación completa de todos los códigos de error existentes (variable `ERRNO`) y de todos los mensajes devueltos por AutoLISP (función `*error*`).

ONCE.14.2. Otras características del control de errores

Existen tres funciones en AutoLISP que también se utilizan a la hora de detectar y procesar un error. La primera de ellas es `ALERT`, cuya sintaxis es:

`(ALERT mensaje)`

ALERT presenta un sencillo letrero de diálogo con el mensaje de advertencia especificado en el argumento *mensaje*. El letrero sólo contiene un botón *Aceptar* que, al pulsarlo hará que desaparezca el cuadro. Ejemplo:

```
(ALERT "Procedimiento no permitido")
```

Incluyendo el código de control `\n` podemos separar el mensaje en varias líneas:

```
(ALERT "Procedimiento no\npermitido")
```

El número de líneas de estos cuadros de advertencia y su longitud dependen de la plataforma, el dispositivo y la ventana utilizada. **AutoCAD** trunca las cadenas cuya longitud supere el tamaño del cuadro advertencia dependiendo de la plataforma.

ALERT puede ser utilizado en el momento que se detecta un error para presentar una advertencia en pantalla y, por ejemplo, solicitar de nuevo los datos. Veamos un ejemplo:

```
(DEFUN Datos1 ()
  (SETQ PuntoInicio (GETPOINT "Punto de inicio: ")) (TERPRI)
  (SETQ Radio (GETDIST PuntoInicio "Radio:" )) (TERPRI)
)

(DEFUN Datos_Vuelt ()
  (IF (SETQ Vueltas (GETINT "Número de vueltas: "))
    ()
    (PROGN
      (ALERT "Debe introducir un número de vueltas")
      (Datos_Vuelt)
    )
  )
)

...

(DEFUN C:Vuelt ()
  (Datos1)
  (Datos_Vuelt)

  ...
)
```

Las otras dos funciones que comentaremos son `EXIT` y `QUIT`. Sus respectivas sintaxis son:

`(EXIT)`

y

`(QUIT)`

Ambas producen el mismo efecto, esto es, interrumpen la aplicación actual en el momento en el que son leídas, devolviendo el mensaje de error `quitar / salir abandonar` (en inglés es `quit / exit abort`, como ya hemos visto).

Se pueden utilizar para acabar el programa directamente en el momento en que se detecte un error. Podremos utilizarlos de la manera siguiente, por ejemplo:

```
(DEFUN Datos1 ()
  (SETQ PuntoInicio (GETPOINT "Punto de inicio: ")) (TERPRI)
  (SETQ Radio (GETDIST PuntoInicio "Radio:" )) (TERPRI)
)

(DEFUN Datos_Vuelt ()
  (SETQ Vueltas (GETINT "Número de vueltas: "))
  (IF (< (SETQ Vueltas (GETINT "Número de vueltas: ")) 1)
    (QUIT)
  )
)

...

(DEFUN C:Vuelt ()
  (Datos1)
  (Datos_Vuelt)
)

...

)
```

Claro que habría que controlar la salida como ya hemos explicado.

NOTA: Normalmente estas funciones se utilizan también en la depuración de un programa. Detenemos la ejecución con `QUIT` o `EXIT` para evaluar distintas variables y demás.

12ª fase intermedia de ejercicios

- Programar una rutina AutoLISP que dibuje remates de tubos cortados en 2D. Se solicitarán los datos convenientes y se añadirá una rutina de control de errores.
- Realizar un programa con control de errores que dibuje puertas en planta cortando tabiques y muros. Solicitar los datos pertinentes.

ONCE.15. CARGA Y DESCARGA DE APLICACIONES

Hasta ahora hemos hablado de la forma de cargar nuestras aplicaciones o rutinas AutoLISP desde **AutoCAD**. Tras haber escrito el archivo de texto correspondiente, accedíamos a *Herr.>Cargar Aplicación...* y desde allí cargábamos nuestro programa. Esto equivale a ejecutar el comando `APPLOAD` en línea de comandos. Pues bien, existe una serie de funciones de AutoLISP para manejar esta carga de programas desde otros programas AutoLISP. Esas funciones son las que vamos a estudiar seguidamente.

La primera función que veremos es `LOAD`. Su sintaxis es:

<code>(LOAD nombre_archivo [fallo])</code>

Esta función carga en memoria el archivo AutoLISP especificado y devuelve su nombre. Como sabemos, estos archivos podrán o no estar en uno de los caminos de soporte del programa. Si sí estuvieran, con indicar el nombre es suficiente, si no habría que escribir la ruta de acceso o camino completo.

NOTA: Recordemos que los archivos de soporte los configuramos desde *Herr.>Preferencias....*

El nombre del archivo en cuestión no hace falta indicarlo con la extensión .LSP — aunque se aconseja por claridad en programas grandes que llamen a archivos .LSP, .DCL y/o a otros— si tiene esta extensión; si tuviera otra hay que especificarla obligatoriamente, ya que **AutoCAD** le coloca el .LSP detrás en el momento en que no existe. Veamos algún ejemplo:

```
(LOAD "circul")
(LOAD "escalera.lsp")
(LOAD "ventana.mio")
(LOAD "circuit.txt")
```

NOTA: Como se ve, los nombres de archivo han de ir entrecomillados por ser cadenas.

La ruta de acceso indicada se establece al estilo MS-DOS, pero con la particularidad de que no podemos utilizar caracteres contrabarra (\) como separadores de directorios o carpetas. Esto es debido a que, como sabemos, la contrabarra es un carácter de control en AutoLISP. Y si revisáramos el comienzo de este **MÓDULO**, cuando hablábamos de los caracteres de control, veríamos que la contrabarra hemos de indicarla con dos caracteres contrabarra (\\). Precisamente el primero es el carácter de control y el segundo la contrabarra en sí. Por ejemplo:

```
(LOAD "c:\\autocad\\program\\rutin\\caldera.lsp")
```

Por compatibilidad con sistemas UNIX, las rutas o caminos de acceso se pueden indicar con el carácter de barra inclinada normal (/). Esto casi siempre resulta más cómodo que el método anterior:

```
(LOAD "c:/autocad/program/rutin/caldera.lsp")
```

El argumento *fallo* será devuelto si el programa fracasa a la hora de ser cargado. Si este argumento no existe y el programa falla al ser cargado, AutoLISP devolverá un mensaje de error propio. Por ejemplo:

```
(LOAD "a:/miprogram.lsp" "El archivo no se encuentra o el disco no está en la
unidad")
```

Este argumento también ha de ser indicado como cadena y se devuelve como tal, es decir, entre comillas dobles.

La función `LOAD` la podemos usar para llamar a programas de AutoLISP desde otro programa, cargándolo antes con ella. Por ejemplo:

```
(DEFUN Prog1 ()
  (SETQ Pto1 (GETPOINT "Punto 1: "))
  (SETQ Pto2 (GETPOINT "Punto 2: "))
  (LOAD "c:/lisp/prog2.lsp")
  (prog2)
  (SETQ Result (/ (- NewPto1 NewPto2) 2.0))
)
```

En este caso, el programa llega a la función `LOAD`, carga el programa y lo ejecuta a continuación. Tras haber acabado el otro programa se devuelve el control a este primero.

NOTA: Mucho cuidado al declarar variables en este tipo de estructuras de llamadas a programas. Si las variables son locales y las necesita el otro programa, éste no funcionará.

Declararemos pues como locales únicamente aquellas que no vayan a ser utilizadas más que en la rutina actual.

NOTA: Las aplicaciones AutoLISP hoy por hoy no se pueden descargar de memoria.

`(AUTOLOAD nombre_archivo lista_comandos)`

La función `AUTOLOAD` por su lado hará que ahorremos memoria mientras no estemos utilizando los programas o rutinas AutoLISP. Esta función efectúa la carga del archivo especificado en `nombre_archivo` al igual que `LOAD`, pero únicamente la primera vez que se utilice uno de los comandos indicados en `lista_comandos`. Así por ejemplo:

```
(AUTOLOAD "c:/lisp/rosca.lsp" '("rosca" "rc"))
```

Al ser una lista el segundo argumento debe ir indicado como tal. Y los términos deben ir entrecomillados.

Esta línea hará que se cargue y ejecute automáticamente el programa guardado en `rosca.lsp` al escribir en línea de comandos `rosca` o `rc`. Si no se hace esto el programa no se cargará, con el consiguiente ahorro de memoria.

Deberemos tener un par de consideraciones. La primera es que las palabras con las cuales podemos llamar al programa indicado deben ser el nombre de funciones contenidas en dicho archivo. Y la segunda es que todas las funciones implicadas deberán estar definidas como comandos de **AutoCAD**, es decir con `C:` delante del nombre. Así por ejemplo, para que la línea anterior funcionara, el archivo `ROSCA.LSP` debería contener lo siguiente:

```
(DEFUN C:Rosca ( )
```

```
...
```

```
)
```

```
(DEFUN C:Rc ( )
```

```
  (c:rosca)
```

```
)
```

Es decir, los comandos con los que se puede acceder, uno de ellos definido como abreviatura, pero definido en el propio archivo.

Con respecto a los directorios de soporte, las mismas consideraciones que para la función `LOAD`.

NOTA: Desactivando la casilla *Volver a cargar aplicaciones de AutoLISP entre dibujos*, sita en la pestaña *Compatibilidad* del cuadro de diálogo *Preferencias*, al que se accede bajo *Herr.>Preferencias...*, haremos que cada vez que entremos en un dibujo nuevo no se pierdan los programas y variables globales de AutoLISP cargados en memoria. El texto de la casilla es confuso sobremanera, así que cuidado: la casilla activada hace que no se guarden estos programas en memoria. Esta misma característica, denominada de AutoLISP persistente, se controla mediante la variable `LISPINIT` de **AutoCAD**.

Pero tener que teclear líneas de estas cada vez que entramos en **AutoCAD** es un poco pesado. Si lo que queremos es que nada más cargar **AutoCAD**, o nada más entrar en un dibujo o abrir uno nuevo, se carguen en memoria determinadas rutinas o funciones, utilizaremos los archivos que vamos a explicar a continuación.

ONCE.15.1. ACADR14.LSP, ACAD.LSP y *.MNL

Existen dos archivos, entre otros muchos, que se cargan nada más arrancar **AutoCAD**, y también cada vez que abrimos un dibujo o comenzamos uno nuevo (si está activada la casilla de AutoLISP persistente antes mencionada). Estos dos archivos son el **ACADR14.LSP** y el **ACAD.LSP**. **ACADR14.LSP** es un archivo que se suministra con **AutoCAD** y en el que podemos encontrar una serie de rutinas que los desarrolladores de Autodesk han incluido ahí para que se carguen nada más comenzar un dibujo. Por normal general este archivo no lo modificaremos, ni tampoco añadiremos líneas de código a él.

Para la misma característica disponemos de un archivo **ACAD.LSP**. Este archivo se carga de manera igual al comenzar cualquier dibujo, pero está pensado para que los usuarios introduzcamos en él nuestras modificaciones. Probablemente no exista en un equipo en el que nunca haya sido creado, sea por el usuario o sea por aplicaciones verticales para **AutoCAD**. Si no existe únicamente deberemos crearlo, eso sí, con dicho nombre y extensión.

NOTA: El archivo **ACAD.LSP** ha de estar en un directorio de soporte para que sea cargado. Si existe más de un **ACAD.LSP** en uno o varios directorios de soporte, **AutoCAD** sólo carga el primero que encuentra.

Podemos suponer pues que todas las definiciones de código AutoLISP incluidas en estos archivos se cargarán nada más entrar en **AutoCAD** y al comenzar nuevos dibujos o abrir dibujos existentes. Por ello, si en ellos incluimos funciones **LOAD**, los programas a los que llamen serán cargados nada más abrir el editor de dibujo.

Mucho más lógico, y en la práctica es lo que se hace, es introducir funciones **AUTOLOAD**, que guarden una o varias palabras que, al ser escritas, cargarán programas AutoLISP y los ejecutarán automáticamente.

NOTA: Si la casilla de AutoLISP persistente está desactivada, como hemos dicho los programas y variables globales permanecerán en memoria, pero no se cargarán los archivos **ACAD.LSP** y **ACADR14.LSP** al entrar en un nuevo dibujo o en uno existente, sino sólo al arrancar **AutoCAD**. Sin embargo si la casilla está activada, estos archivos se cargan siempre al cambiar de dibujo, pero los programas y variables globales no permanecerán en memoria. Dios lo entiende... Parece que la práctica más habitual consiste en activar dicha casilla e incluir funciones **LOAD** o **AUTOLOAD** en el **ACAD.LSP**. De esta manera dispondremos de las rutinas y programas en memoria (o preparados) en cualquier momento y sesión de dibujo.

Los archivos **ACAD.LSP** y **ACADR14.LSP** funcionan como cualquier otro archivo de AutoLISP normal, es decir, lo que no está dentro de ningún **DEFUN** se evalúa automáticamente al ser cargado el programa; lo que esté dentro de algún **DEFUN** se evaluará cuando se llame a la función en cuestión.

Por último hemos de recordar en este punto la utilidad de los archivos **.MNL** de menús. Estos archivos contienen normalmente las rutinas necesarias para el funcionamiento de opciones de un nuevo menú creado (si las necesita). Es decir, si un menú llama a comandos programados en AutoLISP, el código de estos comandos se puede introducir en un archivo que tenga el mismo nombre que el menú y la extensión **.MNL**.

Si el código es muy extenso, o por mayor organización, se pueden incluir en este archivo llamadas a los programas necesarios con **LOAD** o **AUTOLOAD**. Los **.MNL** funcionan de la misma forma que el **ACAD.LSP** y el **ACADR14.LSP**, y se cargan en memoria al cargar el menú correspondiente o al abrir un dibujo que lleve implícitamente cargado ese menú.

ONCE.15.1.1. Configuraciones múltiples

Si un usuario trabaja con múltiples configuraciones de **AutoCAD**, o hay varios usuarios utilizando **AutoCAD** en un mismo puesto de trabajo, puede ser necesario cargar diferentes rutinas en cada caso. Esto se consigue fácilmente estableciendo un directorio o carpeta de inicio diferente para cada configuración y definiendo en ella un archivo `ACAD.LSP` con las instrucciones requeridas.

La carpeta de inicio se establece desde el acceso directo a **AutoCAD** en Windows, como sabemos. Si en el momento de iniciar la sesión, **AutoCAD** detecta un archivo `ACAD.LSP` en dicha carpeta, carga automáticamente su contenido. De esta manera, pueden coexistir en disco duro varios archivos `ACAD.LSP` situados en diferentes carpetas.

De hecho, éste es el procedimiento más sencillo que utilizan los desarrolladores de aplicaciones. Al instalarse una aplicación que funciona sobre **AutoCAD**, se crea una carpeta propia con un archivo `ACAD.LSP` y un acceso directo que inicia **AutoCAD** desde esa carpeta.

ONCE.15.1.2. Definir función como `S::STARTUP`

Llegado este momento vamos a explicar esta forma un poco especial de definir una función. Como sabemos, dentro de los archivos comentados anteriormente las funciones `LOAD` y `AUTOLOAD` hacen que se carguen programas automáticamente al arrancar **AutoCAD**. Pero si también queremos que algunas funciones se ejecuten automáticamente al abrir el programa deberemos incluirlas bajo una función definida como `S::STARTUP` de la siguiente forma:

```
(DEFUN S::STARTUP ()  
  ...  
)
```

Todo lo que haya entre los dos paréntesis del `DEFUN` se ejecutará automáticamente. Viene a ser lo mismo que dejarlo fuera de cualquier `DEFUN`, pero de manera más ordenadas y clara. Veamos un ejemplo:

```
(DEFUN S::STARTUP ()  
  (COMMAND "_purge" "_all" "" "_n")  
)
```

De esta manera, al entrar en cualquier dibujo se limpiará automáticamente todo lo no utilizado: capas, estilos de texto, tipos de línea...

El prefijo `S::` de esta función especial debe considerarse como reservado y es mejor no utilizarlo en ninguna función de usuario.

Veamos otro ejemplo más utilizado:

```
(DEFUN c:guardarr ()  
  ...  
)  
  
(DEFUN c:guardarcomo ()
```

```
...  
  
)  
  
(DEFUN S::STARTUP ()  
  (COMMAND "anuladef" "guardarr")  
  (COMMAND "anuladef" "guardarcomo")  
)
```

Si quisiéramos redefinir estos comandos de **AutoCAD**, de la versión castellana del producto, podríamos haber escrito esto en el `ACAD.LSP`. Así cargarían en memoria las nuevas definiciones de los comandos `GUARDARR` y `GUARDARCOMO` y, al ejecutarse automáticamente `S::STARTUP`, con el comando de **AutoCAD** `ANULADEF` se anularían las actuales definiciones de esos dos comandos. Para recuperar las definiciones normales de ambos comandos, basta emplear el comando de **AutoCAD** `REDEFINE`. Se recuerda aquí que para utilizar la definición habitual de comandos de **AutoCAD** redefinidos, basta preceder su nombre de un punto (para más información debemos dirigirnos al **MÓDULO SIETE** de este curso).

ONCE.15.2. Aplicaciones ADS

Hay una serie de funciones de AutoLISP para gestionar las aplicaciones ADS (ya obsoletas). En esta sección las vamos a explicar.

```
(XLOAD nombre_archivo [fallo])
```

Funciona de la misma manera que `LOAD` para las aplicaciones AutoLISP. En este caso de aplicaciones ADS, al mismo tiempo que se cargan se comprueban si son compatibles con AutoLISP. En el caso de detectarse incorrecciones el proceso quedaría abortado.

```
(AUTOXLOAD nombre_archivo lista_comandos)
```

Funciona de la misma manera de `AUTOLOAD` para las aplicaciones AutoLISP.

```
(XUNLOAD nombre_archivo [fallo])
```

Descarga aplicaciones ADS de la memoria. El nombre en `nombre_archivo` debe ser el mismo que el que se utilizó al cargar la aplicación con `XLOAD`. Si se indicó un camino de directorios al cargar con `XLOAD`, es posible omitirlo al descargar la misma aplicación con `XUNLOAD`. Si la aplicación se descarga correctamente se devuelve el nombre de la misma, si no un mensaje de error; a no ser que se indique algo en el argumento optativo `fallo`.

```
(ADS)
```

Devuelve una lista con los nombres y rutas de acceso (si hiciera falta) de las aplicaciones ADS actualmente cargadas. Por ejemplo:

```
(ADS)          podría devolver ("c:\\lsp\\prg.exp" "rut.exe" "ventn.exe")
```

```
(GETCFG nombre_parámetro)
```

El archivo `ACAD14.CFG` almacena la configuración de **AutoCAD**. Se trata de un archivo de texto al cual se puede acceder para añadir especificaciones o modificar las existentes. Contiene una sección denominada `AppData` que permite a los usuarios y programadores almacenar información relativa a la configuración de sus aplicaciones.

La función GETCFG recupera el valor del parámetro cuyo nombre se indique. Este nombre debe ser una cadena de texto en la forma:

`"AppData/nombre_aplicación/nombre_sección/.../nombre_parámetro"`

`(SETCFG nombre_parámetro valor)`

Esta función es complementaria de la anterior y permite escribir un valor concreto en la sección AppData del archivo de configuración ACAD14.CFG para el parámetro cuyo nombre se indique. Este parámetro debe especificarse en una cadena de texto cuyo formato es el mismo que para la función GETCFG.

ONCE.15.3. Aplicaciones ARX

Existe también una serie de funciones de AutoLISP para gestionar las aplicaciones ARX. Vamos a verlas.

`(ARXLOAD nombre_archivo [fallo])`

Funciona de la misma manera que XLOAD para las aplicaciones ADS y LOAD para aplicaciones AutoLISP. También se comprueba la compatibilidad del archivo con AutoLISP como con XLOAD.

`(AUTOARXLOAD nombre_archivo lista_comandos)`

Funciona de la misma manera de AUTOXLOAD para aplicaciones ADS y que AUTOLOAD para las aplicaciones AutoLISP.

`(ARXUNLOAD nombre_archivo [fallo])`

Funciona de la misma manera de XUNLOAD para las aplicaciones ADS.

`(ARX)`

Devuelve una lista con los nombres y rutas de acceso (si hiciera falta) de las aplicaciones ARX actualmente cargadas. Por ejemplo:

`(ARX)` podría devolver `("acadapp.arx" "oleaprot.arx")`

ONCE.15.4. Acceso a comandos externos

Los comandos y variables de **AutoCAD** que se definen por medio de aplicaciones ADS, ARX o AutoLISP se denominan *subrs* externas o comandos de definición externa. Los programas en AutoLISP acceden a estos comandos de manera distinta a los comandos específicos de **AutoCAD**. La función de AutoLISP `COMMAND` accede en general a todos los comandos internos de **AutoCAD**. Los comandos externos en cambio tienen su propia interfaz de comunicación. En esta sección se describe la manera de acceder a esos comandos externos, desde los programas en AutoLISP.

ONCE.15.4.1. Comandos programados en AutoLISP

Se trata de una serie de comandos de **AutoCAD** definidos mediante archivos de AutoLISP. Las instrucciones de autocarga se encuentran en el archivo ACADR14.LSP. A continuación se expone la manera de llamarlos desde AutoLISP, así como el archivo .LSP que los contiene.

(C:APpload)	<i>Comando APPLOAD en archivo APPLOAD.LSP</i>
(C:EDGE)	<i>Comando EDGE en archivo EDGE.LSP</i>
(C:FILTER)	<i>Comando FILTER en archivo FILTER.LSP</i>
(C:3D)	<i>Comando 3D en archivo 3D.LSP</i>
(C:AI_BOX)	<i>Comando AI_BOX en archivo 3D.LSP</i>
(C:AI_PYRAMID)	<i>Comando AI_PYRAMID en archivo 3D.LSP</i>
(C:AI_BOX)	<i>Comando AI_BOX en archivo 3D.LSP</i>
(C:AI_WEDGE)	<i>Comando AI_WEDGE en archivo 3D.LSP</i>
(C:AI_DOME)	<i>Comando AI_DOME en archivo 3D.LSP</i>
(C:AI_MESH)	<i>Comando AI_MESH en archivo 3D.LSP</i>
(C:AI_SPHERE)	<i>Comando AI_SPHERE en archivo 3D.LSP</i>
(C:AI_CONE)	<i>Comando AI_CONE en archivo 3D.LSP</i>
(C:AI_TORUS)	<i>Comando AI_TORUS en archivo 3D.LSP</i>
(C:AI_DISH)	<i>Comando AI_DISH en archivo 3D.LSP</i>
(C:3DARRAY)	<i>Comando 3DARRAY en archivo 3DARRAY.LSP</i>
(C:MVSETUP)	<i>Comando MVSETUP en archivo MVSETUP.LSP</i>
(C:DDINSERT)	<i>Comando DDINSERT en archivo DDINSERT.LSP</i>
(C:DDATTDEF)	<i>Comando DDATTDEF en archivo DDATTDEF.LSP</i>
(C:DDATTEXT)	<i>Comando DDATTEXT en archivo DDATTEXT.LSP</i>
(C:DDMODIFY)	<i>Comando DDMODIFY en archivo DDMODIFY.LSP</i>
(C:DDCHPROP)	<i>Comando DDCHPROP en archivo DDCHPROP.LSP</i>

Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en AutoLISP

(C:DDVIEW)	Comando DDVIEW en archivo DDVIEW.LSP
(C:DDVPOINT)	Comando DDVPOINT en archivo DDVPOINT.LSP
(C:DDOSNAP)	Comando DDOSNAP en archivo DDOSNAP.LSP
(C:DDPTYPE)	Comando DDPTYPE en archivo DDPTYPE.LSP
(C:DDUCSP)	Comando DDUCSP en archivo DDUCSP.LSP
(C:DDUNITS)	Comando DDUNITS en archivo DDUNITS.LSP
(C:DDGRIPS)	Comando DDGRIPS en archivo DDGRIPS.LSP
(C:DDSELECT)	Comando DDSELECT en archivo DDSELECT.LSP
(C:DDRENAME)	Comando DDRENAME en archivo DDRENAME.LSP
(C:DDCOLOR)	Comando DDCOLOR en archivo DDCOLOR.LSP
(C:BMAKE)	Comando BMAKE en archivo BMAKE.LSP
(C:ATTREDEF)	Comando ATTREDEF en archivo ATTREDEF.LSP

Esto es debido a que así se llaman las funciones definidas en los respectivos archivos. Téngase en cuenta que es lo mismo que hacíamos para llamar a funciones propias ya definidas.

ONCE.15.4.2. Comandos de transformaciones 3D

Se trata de comandos externos incorporados en **AutoCAD** a través de una aplicación ARX.

(ALINEAR argumentos...)	Comando ALINEAR en archivo GEOM3D.ARX
(GIRA3D argumentos...)	Comando GIRA3D en archivo GEOM3D.ARX
(SIMETRIA3D argumentos...)	Comando SIMETRIA3D en archivo GEOM3D.ARX

El orden y tipo de argumentos para los tres comandos externos son los mismos que se escriben en la línea de comando de **AutoCAD**.

ONCE.15.4.3. Calculadora de geometrías

Se trata de una utilidad incorporada en **AutoCAD** a través de una aplicación ARX.

(CAL [expr])	Comando CAL en archivo GEOMCAL.ARX
--------------	------------------------------------

Se introduce la expresión de la calculadora, como una cadena entre comillas.

ONCE.15.4.4. Intercambios en formato *PostScript*

Se trata de tres comandos externos incorporados en **AutoCAD** a través de una aplicación ARX.

(C:CARGAPS [nom ins esc])	Comando CARGAPS en archivo ACADPS.ARX
---------------------------	---------------------------------------

Se puede suministrar el nombre de la imagen de extensión .EPS para cargar, el punto de inserción y el factor de escala.

(C:ARRASTRAPs [mod])	Comando ARRASTRAPS en archivo ACADPS.ARX
----------------------	------------------------------------------

Los valores posibles para el modo *mod* son 0 ó 1, para desactivar o activar el arrastre de la imagen.

(C:RELLENAPS [pol pat [argumentos...]])	Comando RELLENAPS en archivo ACADPS.ARX
-----------------------------------------	-----------------------------------------

Se puede suministrar el nombre (*entity name*) de la polilínea 2D cuyo interior se quiere rellenar, y después el nombre del patrón *PostScript*. Los argumentos adicionales son las respuestas a los requerimientos de cada tipo de patrón.

ONCE.15.4.5. Proyección de sólidos en ventanas

Se trata de tres comandos externos añadidos y definidos como aplicaciones ARX.

(C:SOLVIEW argumentos...)	Comando SOLVIEW en archivo SOLIDS.ARX
---------------------------	---------------------------------------

(C:SOLDRAW argumentos...)	Comando SOLDRAW en archivo SOLIDS.ARX
---------------------------	---------------------------------------

(C:SOLPERFIL argumentos...)	Comando SOLPERFIL en archivo SOLIDS.ARX
-----------------------------	-----------------------------------------

El orden y tipo de argumentos para los tres comandos son los mismos que se escriben en la línea de comandos de **AutoCAD**.

ONCE.15.4.6. Comandos de *Render*

Se trata de una serie de comandos externos incorporadas en **AutoCAD** a través de una aplicación ARX.

(C:RENDER [arch [pto1 pto2]])	Comando RENDER en archivo RENDER.ARX
-------------------------------	--------------------------------------

Se puede especificar un nombre de archivo para almacenar la imagen obtenida y dos puntos para la ventana de ajuste de la imagen.

(C:RFILEOPT [frm X Y aspt [argumentos...]])	Comando RFILEOPT en archivo RENDER.ARX
---------------------------------------------	----------------------------------------

La siguiente tabla resume los diferentes argumentos y sus valores posibles. Los argumentos disponibles en cada formato hay que proporcionarlos en el orden especificado:

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

Argumento	Valores	Descripción
<i>frm</i>	"TGA" "PCX" "BMP" "PS" "TIFF"	Formato Targa. Formato Z-Soft. Formato de Mapa de Bits. Formato PostScript. Formato TIFF.
<i>X</i>	1 a 4096	Resolución X en pixeles.
<i>Y</i>	1 a 4096	Resolución Y en pixeles.
<i>aspt</i>		Proporción de aspecto de pixel.
<i>argumentos</i>	<i>modocolor</i>	Disponible para todos los formatos. Valores posibles: "MONO" Monocromo. "G8" Gris 256 niveles. "C8" Color 256. "C16" Color de 16 bits. "C24" Color de 24 bits (TGA, PS y TIFF). "C32" Color de 32 bits (sólo TGA).
	<i>entrelazado</i>	Disponible sólo para el formato TGA. Valores posibles: 1 Sin entrelazar. 2 Entrelazado 2:1. 4 Entrelazado 4:1.
	<i>compresión</i>	Disponible para los formatos TGA y TIFF. Valores: "COMP" Compresión activada. <i>vacío</i> No hay compresión.
	<i>orientación</i>	Disponible sólo para el formato TGA. Valores posibles: "UP" Hacia arriba. <i>vacío</i> Hacia abajo.
	<i>disposición</i>	Disponible sólo para el formato PS. Valores posibles: "P" Vertical. "L" Horizontal (por defecto).
	<i>tipo_tamaño</i>	Disponible sólo para el formato PS. Valores posibles: "A" Auto (por defecto). "I" Imagen. "C" Personalizado.
	<i>tamaño_imagen</i>	Disponible sólo para el formato PS. Valor del tamaño.

(C:ESCENA [mod [argumentos...]])

Comando ESCENA en archivo RENDER.ARX

Los modos de *mod* posibles y los argumentos para cada uno se indican en la siguiente tabla:

Modo	Argumentos	Descripción
"D"	<i>nombre</i>	Suprimir escena cuyo nombre se indica.
"S"	[<i>nombre</i>]	Definir escena cuyo nombre se indica. Si se omite

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

Modo	Argumentos	Descripción
"R"	<i>nombre_actual</i> <i>nombre_nuevo</i>	<i>nombre</i> , se devuelve el nombre de la escena actual. Renombrar escena cuyo nombre se indica. Nuevo nombre para la escena.
"L"	[<i>nombre</i>]	Listar escena cuyo nombre se indica. Si se omite, se listan todas las escenas del dibujo. Los elementos de la lista son los mismos que para crear nueva escena.
"N"	<i>nombre</i> [<i>vista</i> [<i>luces</i>]]	Crear escena nueva con el nombre indicado. Nombre de vista de AutoCAD . Si se indica T es la actual Lista de nombres de luz. Si se indica T es *TODAS*. Si se indica nil no se utiliza ninguna luz.
"M"	<i>nombre</i>	Modificar escena cuyo nombre se indica. Las opciones son las mismas que para crear escena nueva.

(C:LUZ [mod [argumentos...]])

Comando LUZ en archivo RENDER.ARX

Los modos posibles de *mod* y sus argumentos se indican en la siguiente tabla:

Modo	Argumentos	Descripción
"D"	<i>nombre</i>	Suprimir luz cuyo nombre se indica.
"L"	[<i>nombre</i>]	Listar luz (todas si no se indica un nombre). Los elementos de la lista son los mismos que para crear cada tipo de luz.
"R"	<i>nombre_antiguo</i> <i>nombre_nuevo</i>	Renombrar luz cuyo nombre se indica. Nuevo nombre para la luz.
"A"	[<i>intensidad</i> [<i>color</i>]]	Intensidad de luz ambiente. Número entre 0 y 1 (por defecto es 1). Lista de tres componentes RGB.
"NP"	<i>nombre</i> [<i>intensidad</i> [<i>desde</i> [<i>color</i> [<i>tam_mapa_sombra</i> [<i>suavidad_sombra</i> [<i>sombra</i> [<i>atenuación</i> [<i>objetos_sombra</i>]]]]]]]]	Crear nueva luz puntual cuyo nombre se indica. Número real entre 0 y un máximo según atenuación Lista de tres valores de posición de la luz. Lista de tres valores de componentes RGB. Entero entre 0 y 4096 con el tamaño en pixeles. Número real entre 0 y 10. Valor "OFF" sin sombra y "ON" con sombra. Los valores posibles son: <div style="margin-left: 40px;"> 0 Sin atenuación. 1 Atenuación lineal inversa. 2 Atenuación cuadrada inversa. </div> Selección de objetos delimitadores de mapas de sombra.
"ND"	<i>nombre</i> [<i>intensidad</i> [<i>desde</i> [<i>a</i> [<i>color</i> [<i>tam_mapa_sombra</i> [<i>haz_de_luz</i>	Crear nueva luz distante cuyo nombre se indica. Número real entre 0 y un máximo. Lista de tres valores de posición de la luz. Lista de tres valores de destino de la luz. Lista de tres valores de componentes RGB. Entero entre 0 y 4096 con el tamaño en pixeles. Ángulo en grados entre 1 y 160.

Modo	Argumentos	Descripción
	[disminución	Ángulo en grados entre 1 y 160.
	[suavidad_sombra	Número real entre 0 y 10.
	[sombra	Valor "OFF" sin sombra y "ON" con sombra.
	[objetos_sombra	Objetos delimitadores de mapas de sombra.
	[mes	Número entero de 1 a 12.
	[día	Número entero de 1 a 31.
	[hora	Número entero de 0 a 24.
	[minuto	Número entero de 0 a 59.
	[luz_diurna	Ahorro de luz diurna. Valores "OFF" y "ON".
	[latitud	Número real entre 0 y 90 grados (+ y -).
	[longitud	Número real entre 0 y 180 grados (+ y -).
	[zona_horaria	Número entero entre -12 y 12.
"NS"]]]]]]]]]]]]]]]]] nombre	Crear nueva luz de foco cuyo nombre se indica.
	[intensidad	Número real entre 0 y un máximo.
	[desde	Lista de tres valores de posición de la luz.
	[a	Lista de tres valores de destino de la luz.
	[color	Lista de tres valores de componentes RGB.
	[tam_mapa_sombra	Entero entre 0 y 4096 con el tamaño en pixeles.
	[haz_de_luz	Ángulo en grados entre 1 y 160.
	[disminución	Ángulo en grados entre 1 y 160.
	[suavidad_sombra	Número real entre 0 y 10.
	[sombra	Valor "OFF" sin sombra y "ON" con sombra.
	[atenuación	Los mismos valores posibles que en "NP".
	[objetos_sombra	Objetos delimitadores de mapas de sombra.
]]]]]]]]]]]	
"M"	nombre	Modificar luz cuyo nombre se indica. Las opciones son las mismas y en el mismo orden que para crear cada tipo de luz.

(C:BIBLIOMAT [mod nom [arch]])

Comando BIBLIOMAT en archivo RENDER.ARX

Los modos posibles para mod se proporcionan en la siguiente tabla:

Modo	Descripción
"I"	Importa el material <i>nom</i> del archivo <i>arch</i> de la biblioteca .MLI indicada.
"E"	Exporta el material <i>nom</i> al archivo <i>arch</i> de la biblioteca .MLI indicada.
"D"	Suprime el material <i>nom</i> (si no se indica <i>arch</i> es en la escena actual).
"C"	Suprime el material no enlazado <i>nom</i> de la biblioteca .MLI indicada.
"L"	Lista de materiales.

```
(C:MATERIALR [mod [argumentos...]])
```

Comando MATERIALR en archivo RENDER.ARX

Los modos para *mod* posibles y los argumentos para cada uno se indican en la siguiente tabla:

Modo	Argumentos	Descripción
"A"	<i>nombre</i> <i>[entidades]</i>	Enlazar material cuyo nombre se indica. Objetos a los que enlazará el material. Si se omite,

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

Modo	Argumentos	Descripción
		se devuelve una lista con los enlaces existentes. Opciones:
	<i>aci</i>	Número de color ACI entre 0 y 255.
	<i>conjunto</i>	Conjunto de objetos seleccionadas.
	<i>capa</i>	Nombre de capa.
"D"	<i>nombre</i>	Desenlazar material cuyo nombre se indica. Las
	<i>[entidades]</i>	mismas opciones que para "A".
"C"	<i>nombre_actual</i>	Crear material nuevo copiando el del nombre indicado.
	<i>nombre_nuevo</i>	Nombre para el material nuevo creado.
"L"	<i>[nombre]</i>	Listar material cuyo nombre se indica. Si se omite, se listan todos los materiales del dibujo. Los elementos de la lista son los mismos que para crear cada tipo de material.
"N"	<i>nombre</i>	Crear nuevo material con el nombre indicado.
	<i>tipo</i>	Tipo de material que crear. Los tipos posibles se indican a continuación, con las opciones de descripción de cada tipo. "STANDARD" tipo de material estándar.
	<i>[color]</i>	Lista de tres valores de componentes RGB de color del material.
	<i>[intensidad_color]</i>	Factor de intensidad especular del color del material.
	<i>[patrón]</i>	Lista de argumentos de correspondencia patrón/textura. Son:
	<i>nombre</i>	Nombre del archivo de mapa de bits.
	<i>[combinac.</i>	Cantidad de color de correspondencia que utilizar.
	<i>[mosaico</i>	Valor 0 (sin mosaico); valor 1 (mosaico).
	<i>[escala</i>	Lista con factores de escala U y V.
	<i>[desf.]]]]</i>	Lista con valores de desfase U y V.
	<i>[ambiente</i>	Valores RGB de color de ambiente (sombreado) del material.
	<i>[intensidad_amb.</i>	Factor de intensidad ambiente.
	<i>[reflexión</i>	Valores RGB de color de reflexión del material.
	<i>[intensidad_refl.</i>	Factor de intensidad de reflexión.
	<i>[corresp._reflex.</i>	Lista de argumentos de correspondencia reflexión/entorno. Son:
	<i>nombre</i>	Nombre del archivo de mapa de bits
	<i>[combin.</i>	Cantidad de color de correspondencia que utilizar
	<i>[simetría]]</i>	Valor 0 (sin simetría); valor 1 (simetría).
	<i>[aspereza</i>	Factor de aspereza.
	<i>[transparencia</i>	Grado de transparencia.

Modo	Argumentos	Descripción
	<i>[corresp._opac.</i>	Lista de argumentos de correspondencia de opacidad. Son los mismos que para la correspondencia patrón/textura.
	<i>[refracción</i>	Índice de refracción.
	<i>[corresp._relieve</i>	Lista de argumentos de correspondencia de relieve.
		Son:
	<i>nombre</i>	Nombre del archivo de mapa de bits.
	<i>[relieve</i>	Factor de grado de relieve
	<i>[mosaico</i>	Valor 0 (sin mosaico); valor 1 (mosaico).
	<i>[escala</i>	Lista con factores de escala U y V.
	<i>[desf.]]]]</i>	Lista con valores de desfase U y V.
	<i>]]]]]]]]]]]]]]</i>	
	<i>"MARBLE"</i>	Tipo de material mármol.
	<i>[color_piedra</i>	Valores RGB de color de la matriz principal del mármol.
	<i>[color_veta</i>	Valores RGB de color de la veta del mármol.
	<i>[reflexión</i>	Valores RGB de color de reflexión.
	<i>[intensidad_refl.</i>	Factor de intensidad de reflexión.
	<i>[corresp._reflex.</i>	Lista de argumentos de correspondencia reflexión/entorno.
	<i>[aspereza</i>	Factor de aspereza.
	<i>[turbulencia</i>	Factor de ondulación de las vetas.
	<i>[nitidez</i>	Factor de nitidez de precisión del contorno de las vetas.
	<i>[corresp._relieve</i>	Lista de argumentos de correspondencia de relieve.
	<i>]]]]]]]]]]]]</i>	
	<i>"GRANITE"</i>	Tipo de material granito.
	<i>[primer_color</i>	Valores RGB del primer color del granito.
	<i>[intensidad1</i>	Factor de intensidad del primer color.
	<i>[segundo_color</i>	Valores RGB del segundo color del granito.
	<i>[intensidad2</i>	Factor de intensidad del segundo color.
	<i>[tercer_color</i>	Valores RGB del tercer color del granito.
	<i>[intensidad3</i>	Factor de intensidad del tercer color.
	<i>[cuarto_color</i>	Valores RGB del cuarto color del granito.
	<i>[intensidad4</i>	Factor de intensidad del cuarto color.
	<i>[reflexión</i>	Valores RGB de color de reflexión.
	<i>[intensidad_refl.</i>	Factor de intensidad de reflexión.
	<i>[corresp._reflex.</i>	Lista de argumentos de correspondencia reflexión/entorno.
	<i>[aspereza</i>	Factor de aspereza.
	<i>[nitidez</i>	Factor de nitidez del contorno del granulado del granito.
	<i>[escala</i>	Factor de escala global del granulado.
	<i>[corresp._relieve</i>	Lista de argumentos de correspondencia de relieve.
	<i>]]]]]]]]]]]]]]]]</i>	
	<i>"WOOD"</i>	Tipo de material madera.
	<i>[color_claro</i>	Valores RGB del color de los anillos claros de la madera.
	<i>[color oscuro</i>	Valores RGB del color de los anillos oscuros de la

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

Modo	Argumentos	Descripción
		madera.
	[<i>reflexión</i>	Valores RGB de color de reflexión.
	[<i>intensidad_refl.</i>	Factor de intensidad de reflexión.
	[<i>corresp._reflex.</i>	Lista de argumentos de correspondencia reflexión/entorno.
	[<i>aspereza</i>	Factor de aspereza.
	[<i>relación</i>	Factor de relación entre anillos claros y oscuros.
	[<i>densidad</i>	Factor de densidad de los anillos.
	[<i>grosor</i>	Factor de variación del grosor de los anillos.
	[<i>forma</i>	Factor de variación de la forma de los anillos.
	[<i>escala</i>	Factor de escala global de los anillos.
	[<i>corresp._relieve</i>	Lista de argumentos de correspondencia de relieve.
]]]]]]]]]]]]	
"M"	<i>nombre</i>	Modificar material cuyo nombre se indica. Las opciones son las mismas y en el mismo orden que para crear cada tipo de material.

(C:MOSTRMAT [<i>obj1 obj2...</i>])	Comando MOSTRMAT en archivo RENDER.ARX
--------------------------------------	----------------------------------------

Se indican los objetos cuyo material enlazado se desea mostrar. Como argumento se puede indicar un nombre de objeto o varios, un número entero de color ACI de **AutoCAD** o un nombre de capa.

(C:MAPEADO [<i>mod [argumentos...]</i>])	Comando MAPEADO en archivo RENDER.ARX
--------------------------------------------	---------------------------------------

Los dos modos posibles en *mod* y los argumentos para cada uno se indican en la siguiente tabla:

Modo	Argumentos	Descripción
"A"	<i>conjunto_selec.</i>	Asignar mapa de coordenadas al conjunto de objetos indicado.
	<i>tipo_referencia</i>	Tipo de referencia de proyección del mapa. Valores posibles: <div style="margin-left: 40px;"> "p" Proyección plana. "C" Proyección cilíndrica. "S" Proyección esférica. "R" Proyección sólida. </div>
	<i>punto1</i>	Tres listas de puntos que definen la geometría de la referencia:
	<i>punto2</i>	Plana: Esquina inferior izquierda, inferior derecha y superior izquierda.
	<i>punto3</i>	Cilíndrica: Centro inferior, centro superior y dirección de línea de pliegue.
		Esférica: Centro, punto radio norte y dirección de línea de pliegue.
	[<i>punto4</i>]	Sólida: Origen, punto definición eje U, punto definición eje V y se añade un cuarto punto de definición del eje W.
	[<i>mosaico</i>	Valor 0 (sin mosaico); valor 1 (mosaico).
	[<i>escala</i>	Lista con factores de escala U y V.
	[<i>desfase</i>]]]	Lista con valores de desfase U y V.
"D"	<i>conjunto_selec.</i>	Desenlazar mapa de coordenadas del conjunto

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

Modo	Argumentos	Descripción
------	------------	-------------

indicado

(C:FONDO [mod [argumentos...]])

Comando FONDO en archivo RENDER.ARX

Los modos de *mod* posibles y los argumentos para cada uno se indican en la siguiente tabla:

Modo	Argumentos	Descripción
"SOLID"		Utilizar un color sólido de fondo. Opciones: "ACAD" Color de fondo de AutoCAD . <i>color</i> Lista de componentes RGB.
"GRADIENT"	<i>color1</i> <i>color2</i> <i>color3</i> <i>[ángulo]</i> <i>[centro]</i> <i>[altura]]]</i>	Utilizar un gradiente de dos o tres colores. Lista de valores RGB del color de la banda superior. Lista de valores RGB del color de la banda intermedia. Lista de valores RGB del color de la banda inferior. Ángulo del gradiente. Centro del gradiente en porcentaje de altura de pantalla. Altura de la banda intermedia en porcentaje de pantalla.
"IMAGE"	<i>archivo</i> <i>[ajuste]</i> <i>[ángulo]</i> <i>[escalax]</i> <i>[escalay]</i> <i>[desfasey]</i> <i>[desfasey]</i> <i>[mosaico]]]]]]]</i>	Utilizar una imagen de fondo desde el archivo indicado. Modo de ajuste de la imagen. Valores posibles: "FIT" Ajuste independiente de escalas X e Y. "FIT ASPECT" Ajuste uniforme de escalas X e Y. (Actualmente se ignora este valor.) Escala X de la imagen. Escala Y. Desfase X entre centro de la imagen y centro de la salida. Desfase Y. Valor 0 (sin mosaico); valor 1 (mosaico).
"MERGE"		Fusionar fondo de vista actual con fondo de imagen.
"ENVIRONMENT"		Establecer mapa de reflexión global. Valores: "BLOQUEO" Mapa global bloqueado en el fondo. <i>archivo</i> Mapa global procede de un archivo.

(C:NIEBLA [actv [argumentos...]])

Comando NIEBLA en archivo RENDER.ARX

La siguiente tabla resume los diferentes argumentos y su significado.

Argumento	Descripción
<i>actv</i>	El valor "ON" activa la niebla y el valor "OFF" la desactiva.
<i>[color]</i>	Lista de componentes RGB del color de la niebla.
<i>[distancia_cerca]</i>	Define dónde comienza la niebla.

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

Argumento	Descripción
<i>[distancia_lejos</i>	Define dónde termina la niebla.
<i>[porcentaje_cerca</i>	Porcentaje de niebla al comienzo.
<i>[porcentaje_lejos</i>	Porcentaje de niebla al final.
<i>[fondo]]]]]]</i>	El valor "ON" aplica la niebla al fondo y el valor "OFF" no.

(C:NVPAISAJE [<i>argumentos...</i>])	Comando NVPAISAJE en archivo RENDER.ARX
----------------------------------------	-----------------------------------------

La siguiente tabla resume los diferentes argumentos y su significado.

Argumento	Descripción
<i>tipo</i>	Nombre del elemento en la biblioteca de paisajes.
<i>altura</i>	Altura del elemento en unidades de dibujo.
<i>posición</i>	Punto 3D de posición del elemento paisajístico.
<i>alineación</i>	Datos de alineación del elemento paisajístico. Valores:
	0 Cara única alineada con la cámara.
	1 Cara única no alineada con la cámara.
	2 Caras que intersectan no alineadas con la cámara.
	3 Caras que intersectan alineadas con la cámara.

(C:EDPAISAJE [<i>mod</i> [<i>argumentos...</i>]])	Comando EDPAISAJE en archivo RENDER.ARX
------------------------------------------------------	-----------------------------------------

La siguiente tabla resume los dos modos *mod* existentes y los diferentes argumentos con su significado:

Modo	Argumentos	Descripción
"LIST"	<i>nombre</i>	Listar datos del elemento cuyo nombre se indica.
<i>nombre</i>		Nombre de objeto (<i>entity name</i>) del elemento que modificar.
	<i>altura</i>	Nueva altura del elemento paisajístico.
	<i>[posición</i>	Nueva posición del elemento paisajístico.
	<i>[alineación]]</i>	Nueva alineación del elemento. Los valores son los mismos que en NVPAISAJE.

(C:BIBPAISAJE [<i>mod</i> [<i>argumentos...</i>]])	Comando BIBPAISAJE en archivo RENDER.ARX
-------------------------------------------------------	------------------------------------------

La siguiente tabla resume los dos modos *mod* y los diferentes argumentos con su significado:

Modo	Argumentos	Descripción
"ADD"	<i>nombre</i>	Añadir a la biblioteca actual el elemento <i>nombre</i> .
	<i>textura</i>	Nombre del archivo de imagen de textura.
	<i>opacidad</i>	Nombre del archivo de imagen de opacidad.
	<i>alineación</i>	Alineación del elemento. Valores los de NVPAISAJE.
"DEL"	<i>nombre</i>	Suprimir de la biblioteca actual el elemento <i>nombre</i> .
"MODIFY"	<i>nombre</i>	Modificar de la biblioteca actual el elemento <i>nombre</i> . Sus argumentos son los mismos que para "ADD".
"OPEN"	<i>archivo</i>	Abre la biblioteca .LLI indicada y la convierte en la actual.
"SAVE"	<i>archivo</i>	Guarda la biblioteca actual con el nombre .LLI

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

Modo	Argumentos	Descripción
"LIST"		indicado. Lista todos los elementos de la biblioteca actual.

(C:PREFR [mod op [parm]])

Comando PREFR en archivo RENDER.ARX

Los valores combinados de *mod* y *op* se indican en la siguiente tabla:

Modo	Opción	Descripción
"DEST"		Destino de la imagen modelizada.
	"FRAMEBUFFER"	Modelizar en pantalla.
	"HARDCOPY"	Modelizar en ventana de <i>Render</i> .
	"FILE"	Modelizar en un archivo.
"ICON"	<i>tamaño</i>	Escala de iconos: <i>tamaño</i> de porcentaje de pantalla.
"ROPT"		Más opciones de modelizado.
"SELECT"		Solicitud de designación de objetos para modelizar.
	"ALL"	Modelizar todos los objetos en ventana gráfica.
	"ASK"	Solicitar designación de objetos.
"STYPE"		Tipo de modelizado.
	"ARENDER"	Modelizado básico.
	"ASCAN"	Modelizado fotorrealístico.
	"ARRAY"	Modelizado de tipo <i>Raytrace</i> .
"TOGGLE"		Opciones de conmutación.
	"CACHE"	Modelizar en archivo de caché. Valores: "ON" y "OFF".
	"SHADOW"	Modelizar con sombras. Valores: "ON" y "OFF".
	"SMOOTH"	Modelizar con suavizado. Valores: "ON" y "OFF".
	"FUSION"	Fusionar objetos con el fondo. Valores: "ON" y "OFF".
	"FINISH"	Aplicar materiales. Valores: "ON" y "OFF".
	"SKIPRDLG"	No mostrar cuadro de <i>Render</i> . Valores: "ON" y "OFF".

(C:GUARDARIMG [arch tip [arg...][cmp]])

Comando GUARDARIMG en archivo RENDER.ARX

La siguiente tabla resume los diferentes argumentos y sus valores posibles:

Argumento	Valores	Descripción
<i>archivo</i>		Nombre del archivo de imagen para guardar.
<i>tipo</i>	"BMP"	Formato de Mapa de Bits.
	"TGA"	Formato Targa.
	"TIFF"	Formato TIFF.
<i>argumentos</i>	<i>[parte</i>	Parte de la pantalla que se va a guardar. Valores:
		"A" Ventana gráfica activa.
		"D" Área de dibujo.
		"F" Pantalla completa.
	<i>[desplaz._X</i>	Desplazamiento X en pixeles.
	<i>desplaz._Y]</i>	Desplazamiento Y en pixeles.
	<i>[tamaño_X</i>	Tamaño X en pixeles.
	<i>tamaño_Y]]</i>	Tamaño Y en pixeles.

Argumento	Valores	Descripción
	[<i>compresión</i>]	Sistema de compresión. Los valores posibles son:
	"NONE"	Ninguna compresión.
	"PAK"	Compresión de archivos TIFF.
	"RLE"	Compresión de archivos Targa.

(C:REPRODUCIR [*arch tip [arg...][cmp]]*) *Comando REPRODUCIR en archivo RENDER.ARX*

Los argumentos y sus valores posibles, son los mismos que en GUARDARIMG.

(C:ESTADIST [*arch*]) *Comando ESTADIST en archivo RENDER.ARX*

Si no se indica un nombre de archivo, la llamada al comando externo ESTADIST muestra el cuadro de diálogo con las estadísticas del último modelizado. Si se indica un nombre de archivo, guarda en él las estadísticas, sin mostrar cuadro de diálogo. Si se introduce (C:ESTADIST nil) se indica a *Render* que detenga el guardado de estadísticas.

(C:RENDERUPDATE [*op*]) *Comando RENDERUPDATE en archivo RENDER.ARX*

Utilizado sin ningún argumento regenera el archivo ENT2FACE que controla la geometría de facetas de superficies para el *Render*. La opción de regeneración puede ser "ALWAYS" para generar un nuevo archivo de geometría en cada modelizado, u "OFF" para desactivar la opción anterior.

ONCE.15.4.7. Intercambio con 3D Studio

Son dos comandos externos cuya definición es aplicación ARX.

(C:CARGAR3DS [*mod [mlt crea] arch*]) *Comando CARGAR3DS en archivo RENDER.ARX*

El modo *mod* puede ser 0 (modo no interactivo), en cuyo caso hay que suministrar los dos argumentos siguientes, o 1 (modo interactivo), en cuyo caso basta con indicar el nombre de archivo que se importa, que debe contener la extensión .3DS. El argumento *mlt* especifica cómo tratar objetos con materiales múltiples. Sus valores son 0 para crear un objeto nuevo por cada material y 1 para asignar el primer material al objeto nuevo. El argumento *crea* especifica cómo crear objetos nuevos. Sus valores son 0 para crear una capa por cada objeto 3D Studio, 1 para crear una capa por cada color, 2 para crear una capa por cada material y 3 para situar todos los objetos nuevos en una misma capa.

(C:SALVAR3DS [*sel sal div amol sol arch*]) *Comando SALAR3DS en archivo RENDER.ARX*

En primer lugar se indica el grupo de selección con los objetos que exportar, después el tipo de salida, 0 para formato DXF ó 1 (actualmente también es formato DXF). Luego el tipo de división, 0 para crear un objeto 3D Studio por cada capa de **AutoCAD**, 1 por cada color o 2 por cada objeto de **AutoCAD**, después el umbral de ángulo de amoldamiento (si el valor es -1, no se lleva a cabo el amoldamiento automático). Después el umbral de distancia para soldar vértices cercanos (si el valor es negativo, se desactiva la soldadura), y por último el archivo que se creará con la extensión incluida .3DS.

ONCE.15.4.8. Comandos de ASE

Se trata de seis comandos externos incorporados en **AutoCAD** a través de una aplicación ARX.

<code>(COMMAND "_aseadmin" argumentos...)</code>	<i>Archivo ASE.ARX</i>
--------------------------------------------------	------------------------

<code>(COMMAND "_aseexport" argumentos...)</code>	<i>Archivo ASE.ARX</i>
---------------------------------------------------	------------------------

<code>(COMMAND "_aselinks" argumentos...)</code>	<i>Archivo ASE.ARX</i>
--------------------------------------------------	------------------------

<code>(COMMAND "_aserows" argumentos...)</code>	<i>Archivo ASE.ARX</i>
-------------------------------------------------	------------------------

<code>(COMMAND "_aseselect" argumentos...)</code>	<i>Archivo ASE.ARX</i>
---------------------------------------------------	------------------------

<code>(COMMAND "_asesqled" argumentos...)</code>	<i>Archivo ASE.ARX</i>
--------------------------------------------------	------------------------

A todos ellos se puede acceder mediante `COMMAND` y el nombre del comando, suministrando los argumentos tal y como los solicita **AutoCAD**.

ONCE.15.5. Inicio de aplicaciones Windows

Existe una función que nos permite iniciar aplicaciones basadas en plataforma Windows desde AutoLISP. Esta función es:

<code>(STARTAPP aplicación [archivo])</code>

aplicación dice referencia al programa que queremos iniciar y *archivo* al archivo que queremos abrir directamente con dicho programa. Si no se indica una ruta de acceso completa, la aplicación se busca en la ruta de búsqueda especificada en la variable de entorno *PATH* del sistema operativo. Veamos tres de ejemplos:

```
(STARTAPP "notepad" "ejemplo.txt")  
(STARTAPP "c:/windows/app/miprogram.exe")  
(STARTAPP "wordpad" "a:\\lisp\\tuerca.lsp")
```

Si `STARTAPP` se evalúa correctamente devuelve un número entero, en caso contrario devuelve `nil`.

13ª fase intermedia de ejercicios

- Probar las diferentes funciones estudiadas en estas secciones anteriores con rutinas que llamen a otras rutinas, programas que utilicen los comandos externos comentados y demás. Personalícese también un archivo `ACAD.LSP` como práctica.

ONCE.16. INTERACCIÓN CON LETREROS EN DCL

Desde la versión 12 de **AutoCAD** el usuario tiene la posibilidad de elaborar sus propios cuadros o letreros de diálogo en lenguaje DCL para formar parte de aplicaciones personales.

Todo lo referente al diseño de estos cuadros ya ha sido estudiado en el **MÓDULO DIEZ** de este curso. Pero como se dijo entonces, con DCL elaboramos la parte visible del cuadro, es decir su diseño gráfico aparente únicamente. Para hacer que funcionen y para controlarlos debemos recurrir, por ejemplo, a aplicaciones programadas en AutoLISP.

En la presente sección vamos a explicar todas las funciones del lenguaje AutoLISP referentes al control de letreros de diálogo programados en DCL. Se estudiarán todos los métodos para controlar un cuadro y se dispondrán diversos ejemplos que harán referencia al funcionamiento de los cuadros diseñados en el **MÓDULO DIEZ**. Empecemos sin más dilación.

ONCE.16.1. Carga, muestra, inicio, fin y descarga

Las primeras funciones que vamos a estudiar las hemos comentado ya a la hora de hablar del diseño en DCL. Son las utilizadas para cargar, visualizar y activar un cuadro de diálogo.

Como sabemos, para que un programa DCL funcione hace falta utilizar como mínimo tres expresiones AutoLISP: una para cargar el diseño del cuadro en memoria `LOAD_DIALOG`, otra para mostrarlo en pantalla `NEW_DIALOG` y otra para activarlo `START_DIALOG`.

```
(LOAD_DIALOG archivo_DCL)
```

`LOAD_DIALOG` carga en memoria el archivo `.DCL` de definición de cuadros de diálogo especificado. Un archivo `.DCL` puede tener más de una definición de cuadros de diálogo, en este caso se cargarían todas ellas. De la misma forma, es posible utilizar varias veces `LOAD_DIALOG` para cargar varios archivos `.DCL`, cada uno con uno o más diseños de letreros. `archivo_DCL` es la ruta y nombre del archivo en cuestión; es una cadena. Ejemplos:

```
(LOAD_DIALOG "prueba")  
(LOAD_DIALOG "c:/archivs/lisp/cuadro.dcl")  
(LOAD_DIALOG "c:\\mislsp\\helic-pol.dcl")
```

NOTA: No es imprescindible indicar la extensión, pero es conveniente por claridad.

Si la carga del archivo es fructífera se devuelve un número entero positivo que es un índice de carga que luego utilizaremos. Este índice comienza en 1, la primera vez que se carga un archivo con esta función, y continúa con 2, 3, 4, 5... Si la carga resulta fallida, se devuelve cero (0) o un número negativo.

NOTA: Todas estas indicaciones, a lo largo de este curso, acerca de lo que devuelve una función (si falla, si no falla, si encuentra o no un archivo, etc.) no son puro capricho, sino muy útiles a la hora de programar. Nótese que, tras haber estudiado el control de errores, podemos utilizar estas respuestas de AutoLISP para controlar el normal flujo de un programa. En este caso que nos ocupa, el hecho de que `LOAD_DIALOG` devuelva 0 si el proceso de carga falla, nos ofrece la posibilidad de controlar desde el propio programa AutoLISP la correcta o incorrecta carga en memoria de un archivo `.DCL`; tomando las oportunas decisiones en consecuencia.

El hecho de que `LOAD_DIALOG` devuelva un índice de carga nos hace pensar la manera de capturarlo para utilizarlo luego con la siguiente función. La manera es bien sencilla, ya que se hace mediante un simple `SETQ`. Esta será la forma habitual de suministrar esta función, por ejemplo:

```
(SETQ Ind (LOAD_DIALOG "c:/lisp/torno.dcl"))
```

De esta forma guardamos en una variable el índice que devuelva `LOAD_DIALOG`.

`(NEW_DIALOG nombre_letrero índice_carga [acción [punto_pantalla]])`

Esta función muestra en pantalla el letrero cuyo nombre se indique. No hemos de confundir este nombre de letrero con el nombre del archivo `.DCL` que, si el archivo sólo contiene una definición de cuadro de diálogo coincidirá habitualmente. El nombre del letrero es el que precede a `:dialog` en el archivo `.DCL`. Tengamos en cuenta que, como ya hemos dicho, un archivo puede contener la definición de varios letreros, por eso hay que indicar el nombre del que queremos visualizar. Con `LOAD_DIALOG` suministrábamos el nombre del archivo `.DCL`.

También habremos de indicar obligatoriamente con `NEW_DIALOG` el índice de carga devuelto por `LOAD_DIALOG`. Así, una manera típica de utilizar esta función tras usar `LOAD_DIALOG` y haber guardado el índice de carga en una variable llamada `Ind` (como en el ejemplo anterior), sería:

```
(NEW_DIALOG "torno" Ind)
```

NOTA: Aquí vemos como puede coincidir el nombre del archivo `.DCL` con el nombre del letrero (ver ejemplo anterior).

El argumento *acción* especifica la acción por defecto, en forma de expresión de AutoLISP, que se asignará a todos los elementos del cuadro que no tengan una acción asignada en el programa. Volveremos sobre este argumento de `NEW_DIALOG` cuando expliquemos mejor esto de las acciones.

El argumento *punto_pantalla* es un punto en 2D que especifica la posición inicial del letrero en pantalla, generalmente por la esquina superior izquierda. La posición por defecto es el centro de la pantalla, aunque hablaremos un poco más de este tema cuando se estudie la función `DONE_DIALOG`.

Si `NEW_DIALOG` se evalúa correctamente devuelve `T`, si no devuelve `nil`.

NOTA: En Windows el letrero queda ya inicializado con `NEW_DIALOG`, antes de utilizar la siguiente función. Hay que tener en cuenta este aspecto a la hora de hacer llamadas interactivas de `NEW_DIALOG` cuando hay casillas de un letrero que llaman a otro.

`(START_DIALOG)`

`START_DIALOG` inicializa el letrero de diálogo anteriormente mostrado en pantalla con `NEW_DIALOG`.

Una vez utilizada esta función, la cual no tiene argumentos, el letrero permanece activo hasta que el usuario señale un elemento que lleve asociada una acción de final de letrero (`DONE_DIALOG`) que lo cierre. Esta acción suele estar asociada habitualmente a los botones *Aceptar* y *Cancelar*.

`START_DIALOG` devuelve el estado en el que se ha cerrado el letrero. Si se ha hecho mediante la casilla *Aceptar* devuelve 1; si se ha pulsado *Cancelar* devuelve 0; si había letreros superpuestos y se han cerrado todos a la vez mediante `TERM_DIALOG` devuelve -1.

Por último, decir que, como enseguida veremos, en cada llamada a `DONE_DIALOG` se puede indicar un estado de terminación. Si así se ha hecho, `START_DIALOG` devuelve ese número de estado.

NOTA: mientras un cuadro permanece activo no es posible llamar a funciones de AutoLISP que afectan a la pantalla o requieren entradas del usuario que no tienen que ver con el cuadro. Esto ocurre por ejemplo con `COMMAND` y con todas las funciones `GET...`. La función `SSGET` —ya veremos— se puede utilizar, siempre que no sea una de sus opciones interactivas que solicita la acción del usuario.

`(DONE_DIALOG [estado])`

Esta función cierra el letrero activo actual. Debe ser llamada desde una expresión de control de los componentes de un letrero. Generalmente será llamada desde la función que defina la acción del botón *Aceptar* (enseguida veremos todo esto).

La función `DONE_DIALOG` devuelve las coordenadas X e Y (generalmente de la esquina superior izquierda) correspondientes a la posición del letrero en el momento de cerrarse. Estas coordenadas pueden ser utilizadas para luego proporcionarlas en sucesivas llamadas a `NEW_DIALOG` que, como hemos comentado antes, admite como argumento unas coordenadas.

Entonces, si calculáramos un valor por defecto para las coordenadas de un cuadro a la hora de abrirse por primera vez (siempre que no exista ya un valor de coordenadas) y capturáramos la salida de `DONE_DIALOG` así, por ejemplo:

```
(SETQ CoorCuadro (DONE_DIALOG))
```

podríamos luego llamar al mismo cuadro utilizando dichas coordenadas. De esta manera el cuadro siempre aparecería en el último lugar en pantalla en el que se cerró. A veces esto puede resultar útil y a veces molesto.

Si se especifica un argumento en *estado*, debe ser un valor entero positivo. Este argumento indica un estado de terminación que será el que se devuelva en sucesivas llamadas a `START_DIALOG`. De esta forma se podrá asignar una acción diferente dependiendo desde qué casilla o botón se haya realizado el cierre.

`(TERM_DIALOG)`

Esta función cancela todos los letreros anidados en pantalla, si los hubiera. Siempre devuelve `nil`. `START_DIALOG`, que ha activado el primer letrero desde el que después han salido los demás, devuelve en este caso `-1` una vez cancelados todos.

`(UNLOAD_DIALOG índice_carga)`

Una vez utilizado el diseño del cuadro DCL, es decir una vez mostrado en pantalla, el cuadro puede ser descargado mediante la función `UNLOAD_DIALOG` (aunque se siga interactuando con él). O sea, el cuadro puede ser descargado inmediatamente después de utilizar `START_DIALOG` (para liberar espacio en memoria). `UNLOAD_DIALOG` siempre devuelve `nil`.

A modo de resumen podríamos decir que el epígrafe de esta sección **ONCE.16.1.** describe muy bien la manera básica de manejo de cuadros desde AutoLISP. Veamos un ejemplo simplificado:

```
(SETQ Ind (LOAD_DIALOG "archivo")); Carga
(NEW_DIALOG "cuadro" Ind); Muestra

...

(START_DIALOG); Inicio
```



```
(UNLOAD_DIALOG Ind); Descarga
```

```
...
```

```
(DONE_DIALOG); Fin
```

```
...
```

ONCE.16.2. Gestión de elementos del letrero

Una vez estudiada la manera de iniciar y terminar un cuadro en pantalla, vamos a ver cómo gestionar sus componentes incluidos, estableciendo así valores por defecto en el inicio, extrayendo del letrero dichos valores para su proceso, asignando acciones a botones, etcétera. Además afianzaremos lo aprendido anteriormente viendo ya ejemplos prácticos.

Para todo ello vamos a ir desarrollando un ejemplo al cual se referirán todas las explicaciones. Este ejemplo se corresponde con uno de los explicados en el **MÓDULO DIEZ**, concretamente el de control de variables de **AutoCAD**. Para mayor comodidad mostramos aquí el letrero de diálogo y su código en DCL (un poco variado al final):

NOTA: Los ejemplos que mostremos irán precedidos de epígrafes que estarán identificados por número de ejemplo y número de parte.

Letrero 1



Código DCL 1 —VARIABLES.DCL—

```
variables:dialog {label="Variables";
:row {
:boxed_column {label="Mallas";
:edit_box {label="SURFTAB&1";edit_width=3;edit_limit=3;key="Surf1";}
:edit_box {label="SURFTAB&2";edit_width=3;edit_limit=3;key="Surf2";}
spacer_1;
:button {label="De&fecto";fixed_width=true;alignment=centered;key="Def1";}
}
:boxed_column {label="Sólidos";
:edit_box {label="Isolíneas";edit_width=2;edit_limit=2;key="Iso";
```

```
mnemonic="s";}
:edit_box {label="Suavizado";edit_width=4;edit_limit=8;key="Suav";
mnemonic="v";}
spacer_1;
:toggle {label="Si&lueta";key="Sil";}
:button {label="Defe&cto";fixed_width=true;alignment=centered;key="Def2";}
}
}
spacer_1;
:row {:toggle {label="&Diálogo al imprimir";key="Dia";}}
:row {:toggle {label="&Gestión de archivos";key="Ges";}}
spacer_1;
:row {ok_cancel_help;}
}
```

`(SET_TILE clave valor)`

La función con la que comenzaremos es SET_TILE. SET_TILE asigna un valor al elemento cuya clave —atributo *key* de DCL— se indica. El atributo *valor* es una cadena de texto que especifica el valor que se asignará a determinado *tile* o elemento, como ya hemos dicho, identificado por su *key*.

Todo texto que aparece en un letrero de diálogo ha de ser una cadena, es decir, texto alfanumérico en formato ASCII, y no valores numéricos en formato numérico. Así por ejemplo:

```
(SET_TILE "Num" "23.32")
(SET_TILE "Telef" "944661234")
(SET_TILE "ValText" "Marca de moleteado")
```

Será muchas veces práctica habitual rellenar casillas de un cuadro con valores del sistema por defecto. En este caso, si el valor corresponde a una variable alfanumérica (de **AutoCAD** o del usuario) no habrá ningún problema en especificar, por ejemplo:

```
(SET_TILE "Estilo" (GETVAR "textstyle"))
```

lo que cogerá el valor de la variable de **AutoCAD** TEXTSTYLE y la introducirá en el elemento con *key*="Estilo", que podría ser una :edit_box. Pero si el valor es numérico, habremos de convertirlo antes a formato ASCII para después introducirlo en el elemento en cuestión. Esto lo realizaremos con cualquiera de las funciones a tal efecto ya estudiadas, por ejemplo:

```
(SET_TILE "TiempSal" (ITOA (GETVAR "savetime")))
```

lo que cogerá el valor de SAVETIME, que es numérico entero, lo convertirá a cadena ASCII con la función ITOA y lo asignará al *tile* identificado por TiempSal.

NOTA: En la relación de variables de **AutoCAD** que aparece en el **APÉNDICE B** de este curso, se especifica para cada una el tipo de formato de datos, así como los valores que admite. De toda maneras, para saber si una variable es numérica o alfanumérica únicamente deberemos teclear su nombre. Si al preguntarnos por el nuevo valor (si no es de sólo lectura) el valor por defecto aparece entrecomillado, es una cadena o variable alfanumérica; si no, es numérica.

Lo primero que debemos hacer en un programa AutoLISP que controla un letrero de diálogo, y después de cargarlo y mostrarlo en pantalla, es rellenar sus casillas con valores por defecto, activar las opciones de casillas de activación y botones excluyentes por defecto, etcétera. Como sabemos, esto lo podemos hacer en el propio archivo .DCL (atributo *value*), pero es preferible realizarlo desde AutoLISP.

Si inicializáramos elementos desde DCL y posteriormente desde AutoLISP, se daría una redundancia no deseada. Por ejemplo, si inicializamos una casilla de edición :edit_box en DCL, que contiene una valor de una variable de **AutoCAD**, como value="25" y luego, desde AutoLISP la volvemos a inicializar con el valor de la variable en el momento de mostrar el cuadro (que es lo lógico), se producirá un instantáneo "baile" de números (entre el de value y el actual, si no coinciden) que no es vistoso para el usuario. Por ello, normalmente desecharemos la asignación por defecto desde DCL y la realizaremos desde AutoLISP.

Todas la operaciones de relleno de casillas, asignación de acciones a botones, y en general todos los acceso al cuadro de diálogo (que ya veremos) se introducen antes de la función START_DIALOG y después de NEW_DIALOG, si no no funcionará. Veamos ahora entonces cómo comenzaríamos el ejemplo de programa que nos ocupa.

Código AutoLISP 1 —VARIABLES.LSP— (primera parte)

```
(DEFUN C:Var ()
  (SETQ Ind (LOAD_DIALOG "c:/misdoc~1/autocad/dcl/variables.dcl"))
  (NEW_DIALOG "variables" Ind)
  (SET_TILE "Surf1" (ITOA (GETVAR "surftab1")))
  (SET_TILE "Surf2" (ITOA (GETVAR "surftab2")))
  (SET_TILE "Iso" (ITOA (GETVAR "isolines")))
  (SET_TILE "Suav" (RTOS (GETVAR "facetres") 2 2))
  (SET_TILE "Sil" (ITOA (GETVAR "dispsilh")))
  (SET_TILE "Dia" (ITOA (GETVAR "cmddia")))
  (SET_TILE "Ges" (ITOA (GETVAR "filedia")))
```

Comentarios al código AutoLISP 1 —VARIABLES.LSP— (primera parte)

Lo primero que hacemos aquí es definir una función que es comando de **AutoCAD** (Var); como sabemos podríamos haber definido una función interna para luego llamarla desde otra que sea comando de **AutoCAD**.

Posteriormente cargamos el cuadro de diálogo con LOAD_DIALOG guardando en la variable Ind el índice de carga. A continuación, mostramos el cuadro en pantalla con NEW_DIALOG proporcionando el índice devuelto por la función anterior y guardado en Ind.

Una vez hecho esto tenemos que inicializar todas las casillas, introduciendo correlativamente en cada elemento, identificado por su key, el valor correspondiente. Nótese que este caso son valores de variables numéricas de **AutoCAD**, por lo que debemos utilizar funciones ITOA y RTOS (esta última en el caso del suavizado por ser valor real) para convertir los datos a cadenas de texto.

NOTA: Como ya dijimos en el **MÓDULO DIEZ**, en DCL se realiza distinción entre mayúsculas y minúsculas, por lo que al identificar las key hay que introducir el mismo nombre que se escribió en el archivo .DCL; no es lo mismo Suav que suav.

Como sabemos, en el caso de casillas de activación :toggle (y de otros elementos como botones excluyentes), un valor de 0 significa desactivado y un valor de 1 significa activado. Aún así, 1 y 0 siguen siendo valores numéricos enteros, por lo que se utiliza para su conversión la función ITOA.

(ACTION_TILE clave acción)

Esta es la siguiente *subr* de AutoLISP que vamos a explicar. ACTION_TILE asigna una expresión AutoLISP indicada en *acción* al elemento o *tile* expresado por la clave de su

atributo `key` en el diseño DCL. Ambos argumentos han de expresarse entre comillas dobles ("") por ser cadenas. Por ejemplo:

```
(ACTION_TILE "bot1" "(SETQ Var1 12) (SETQ Var2 0)")
```

Lo más lógico parece ser indicar bastantes más acciones que realizar en una asignación con `ACTION_TILE`. Por eso, la práctica habitual aconseja asignar como acción una llamada a una subrutina o función interna dentro del programa AutoLISP, así:

```
(ACTION_TILE "bot1" "(Acc1)")
```

De esta forma asignamos al *tile* que tenga como *key* `bot1` una secuencia de acciones contenidas en la función `Acc1`. Esta función deberá estar definida posteriormente, y ya detrás de `START_DIALOG`, con su `DEFUN` particular. Sigamos con nuestro ejemplo para entenderlo mejor.

Código AutoLISP 1 —VARIABLES.LSP— (segunda parte)

```
(DEFUN C:Var ()
  (SETQ Ind (LOAD_DIALOG "c:/misdoc~1/autocad/dcl/variables.dcl"))
  (NEW_DIALOG "variables" Ind)
  (SET_TILE "Surf1" (ITOA (GETVAR "surftab1")))
  (SET_TILE "Surf2" (ITOA (GETVAR "surftab2")))
  (SET_TILE "Iso" (ITOA (GETVAR "isolines")))
  (SET_TILE "Suav" (RTOS (GETVAR "facetres") 2 2))
  (SET_TILE "Sil" (ITOA (GETVAR "dispsilh")))
  (SET_TILE "Dia" (ITOA (GETVAR "cmddia")))
  (SET_TILE "Ges" (ITOA (GETVAR "filedia")))
  (ACTION_TILE "Def1" "(Defecto1)")
  (ACTION_TILE "Def2" "(Defecto2)")
  (ACTION_TILE "accept" "(Chequear) (IF errores () (Aceptar))")
  (START_DIALOG)
  (UNLOAD_DIALOG Ind)
)
```

Comentarios al código AutoLISP 1 —VARIABLES.LSP— (segunda parte)

Los dos primeros `ACTION_TILE` que se definen asignan a cada uno de los botones de valores por defecto del letrero un par de acciones, englobadas ellas en sendas subrutinas que luego se definirán. El tercer y último `ACTION_TILE` asigna al botón *Aceptar* (recordemos que se le da predeterminadamente `accept` como *key*) otra subrutina denominada *Chequear*. Ésta comprobará la existencia de errores en las entradas realizadas por el usuario en el cuadro y, cuando termine, cederá el control a la siguiente función, al `IF`. Si `errores`, que es una variable que definiremos después, tiene algún valor no se realizará nada (sólo se mostrará un mensaje de error que veremos luego en la función *Chequear*). Si `errores` es `nil`, es decir si no hay errores (como veremos), se pasa a la función *Aceptar*, que terminará el cuadro (como veremos).

NOTA: No sólo podemos asignar acciones a los botones, sino a casi cualquier elemento de un letrero de diálogo. Imaginemos, por ejemplo, la posibilidad de realizar una serie de acciones al activar una casilla `:toggle` (mostrar otro cuadro, comprobar valores...), aunque no sea muy usual.

A continuación, y tras la definición y asignación de valores y acciones en el cuadro, se activa con `START_DIALOG` y se descarga de memoria con `UNLOAD_DIALOG`, pues ya no interesa ocupar memoria con él.

Como ya se dijo, y ahora comprenderemos, con la función `NEW_DIALOG` se puede indicar una acción por defecto. Esta acción por defecto se aplicará a todos aquellos elementos que no tengan después una acción asignada. Así si deseamos asignar la misma acción a varios elementos de un cuadro, podemos hacerlos en `NEW_DIALOG`, ahorrándonos espacio en el archivo AutoLISP y su consiguiente memoria en el sistema.

NOTA: Si un elemento tiene asignada una acción con `ACTION_TILE`, ésta se efectúa ignorando la opción por defecto de `NEW_DIALOG`. Es posible una tercera manera de asignar una acción a un elemento: el atributo `action` dentro de un archivo en DCL. También en este caso `ACTION_TILE` tiene prioridad sobre `action`. Como norma general usaremos `ACTION_TILE` para asignar acciones a los diversos *tiles* de un letrero de diálogo.

La expresión de acción asignada a un elemento puede acceder a una serie de variables que informan sobre el estado del elemento o la manera en que el usuario ha modificado dicho elemento. Estas variables son de sólo lectura y pueden verse en la tabla siguiente:

Variable	Significado
<code>\$key</code>	Se ha seleccionado el atributo <code>key</code> del componente. Esta variable se aplica a todas las acciones.
<code>\$value</code>	Es el valor actual del componente expresado como cadena. Esta variable se aplica a todas las acciones. Si el componente es una casilla de edición <code>:edit_box</code> se mostrará su valor como cadena; si es una casilla de verificación <code>:toggle</code> o un botón excluyente <code>:radio_button</code> , se mostrará una cadena "0", para el componente desactivado, o una cadena "1", para el componente activado. Si el componente es un cuadro de lista <code>:list_box</code> o una lista desplegable <code>:popup_list</code> y ningún elemento se encuentra seleccionado, el valor de <code>\$value</code> será <code>nil</code> .
<code>\$data</code>	Son los datos gestionados por la aplicación (si los hay) establecidos justo después de <code>NEW_DIALOG</code> mediante <code>CLIENT_DATA_TILE</code> (se verá). Esta variables se aplica a todas las acciones, pero no tiene sentido a menos que la aplicación ya la haya inicializado llamando a <code>CLIENT_DATA_TILE</code> .
<code>\$reason</code>	Es el código de razón que indica la acción del usuario que ha provocado la acción. Se emplea con los <i>tiles</i> <code>:edit_box</code> , <code>:list_box</code> , <code>:image_button</code> y <code>:slider</code> . Esta variable indica por qué se ha producido la acción. Su valor se define para cualquier tipo de acción, pero sólo hace falta inspeccionarla cuando dicha acción está asociada a alguno de los componente comentados.
<code>\$x</code>	La coordenada X de una selección de <code>:image_button</code> . Es la cadena que contiene la coordenada X del punto designado por el usuario al escoger un componente <code>:image_button</code> . No tiene ningún significado para los demás <i>tiles</i> . La coordenada X forma parte del rango que devuelve <code>DIMX_TILE</code> para el componente, que ya estudiaremos.
<code>\$y</code>	Lo mismo que la anterior pero para la coordenada Y. Forma parte del rango de <code>DIMY_TILE</code> .

Por ejemplo, si se quiere detectar cuál es el botón concreto que ha seleccionado un usuario, se asigna la misma expresión (variable) a los botones, incluyendo en ella la variable `$key`:

```
(ACTION_TILE "botón1" "(SETQ elem $key)(DONE_DIALOG)")  
(ACTION_TILE "botón2" "(SETQ elem $key)(DONE_DIALOG)")
```

En cuanto el usuario señala un botón, `$key` devuelve la clave asociada a dicho elemento, que se almacena en la variable `elem` y se cierra el cuadro. Al salir, examinando la variable `elem` se puede averiguar qué botón ha sido pulsado.

Ejemplo con `$value`:

```
(ACTION_TILE "casilla" "(SETQ val $value)")
```

En este segundo ejemplo, al introducir el usuario un valor en una casilla de edición cuya clave `key` es `casilla` la variable `$value` devuelve ese valor y la expresión de acción lo almacena en `val`.

La variable `$reason` devuelve el código de razón que indica el tipo de acción que el usuario ha realizado sobre el elemento. La mayoría de las veces el valor devuelto es 1, que indica que se ha designado dicho elemento. Pero hay cuatro tipos de elementos que pueden expresar otras acciones del usuario que la simple designación: casillas de edición, listas, imágenes y deslizadores. A continuación se explican de una manera más detallada los códigos de razón que puede devolver `$reason` (también denominados de *retorno de llamada*).

- Código 1: El usuario ha designado el elemento, o ha pulsado `INTRO` si se trataba del elemento por defecto. Se aplica a todos los elementos, pero sólo es necesario examinar este valor para los cuatro tipos enumerados más arriba, puesto que en los demás es el único valor posible.
- Código 2: Se aplica a casillas de edición. Significa que el usuario ha entrado en la casilla, ha realizado modificaciones en el texto contenido, pero ha salido de ella pulsando tabulador o seleccionando otro componente. Conviene comprobar en este caso la validez del valor de la casilla antes de realizar ninguna acción, pues el usuario podría haber dejado incompleta su modificación del texto.
- Código 3: Se aplica a cursores deslizantes. Significa que el usuario ha arrastrado el cursor deslizante, pero no ha realizado una selección final. En este caso conviene mostrar el valor en que ha dejado el usuario el cursor, bien mediante un texto o en la casilla de edición que pueda haber junto al deslizador, pero sin realizar la acción asignada al elemento por si el usuario modifica posteriormente el cursor deslizante.
- Código 4: Se aplica a cuadros de lista y casilla de imagen. En el caso de cuadros de lista significa que el usuario ha realizado una doble pulsación sobre uno de los nombres de la lista. Si el cuadro de lista es el elemento principal de un cuadro, la doble pulsación debe tener asociada la selección del nombre y además el cierre del cuadro. En este caso, la acción asignada debe distinguir el código de razón 1 (una simple pulsación que selecciona el nombre de la lista pero no cierra el cuadro) del código 4 (selección del nombre y cierre). Si el cuadro de lista es un elemento que acompaña a otros en el cuadro, la doble y la simple pulsación se traducen en el mismo efecto de seleccionar un nombre, con lo que la acción asignada no tendría por qué distinguir en principio los dos códigos posibles. En el caso de casilla de imagen, también significa que el usuario ha realizado una doble pulsación sobre ella. La acción asignada puede así distinguir una simple de una doble pulsación y realizar operaciones diferentes si procede.

```
(GET_TILE clave)
```

La función `GET_TILE` extrae el valor actual del elemento del cuadro de diálogo identificado por su `key` en `clave`.

Suele ser habitual emplear `GET_TILE` a la hora de realizar la acción asignada al botón *Aceptar*. De esta forma extraemos los valores últimos introducimos por el usuario haciéndolos efectivos. También se puede utilizar en la función que controle los errores del cuadro para escribir un mensaje en la línea de errores (*key error*). Lo veremos enseguida en nuestro primer ejemplo, a continuación de `MODE_TILE`.

`(MODE_TILE clave modo)`

`MODE_TILE` establece un modo para el elemento cuya clave se indique. Los valores de *modo* se expresan en la siguiente tabla:

<i>modo</i>	Descripción
0	Habilita el elemento (lo pone en negro).
1	Inhabilita el elemento (lo pone en gris).
2	Ilumina, resalta o hace que parpadee el elemento.
3	Selecciona el contenido de una casilla de edición : <i>edit_box</i> .
4	Resalta una casilla de imagen o anula el resaltado.

Esta función podemos utilizarla para controlar el aspecto de determinados elementos con respecto a otros, por ejemplo si al seleccionar una casilla de selección nos interesa que otra determinada se inhabilite, porque son incompatibles. Lo vemos ahora en la siguiente parte de nuestro ejemplo.

Código AutoLISP 1 —VARIABLES.LSP— (última parte)

```
(DEFUN C:Var ()
  (SETQ Ind (LOAD_DIALOG "c:/misdoc~1/autocad/dcl/variables.dcl"))
  (NEW_DIALOG "variables" Ind)
  (SET_TILE "Surf1" (ITOA (GETVAR "surftab1")))
  (SET_TILE "Surf2" (ITOA (GETVAR "surftab2")))
  (SET_TILE "Iso" (ITOA (GETVAR "isolines")))
  (SET_TILE "Suav" (RTOS (GETVAR "facetres") 2 2))
  (SET_TILE "Sil" (ITOA (GETVAR "dispsilh")))
  (SET_TILE "Dia" (ITOA (GETVAR "cmddia")))
  (SET_TILE "Ges" (ITOA (GETVAR "filedia")))
  (ACTION_TILE "Def1" "(Defectol)")
  (ACTION_TILE "Def2" "(Defecto2)")
  (ACTION_TILE "accept" "(Chequear) (IF errores () (Aceptar))")
  (START_DIALOG)
  (UNLOAD_DIALOG Ind)
)

(DEFUN Defectol ()
  (SET_TILE "Surf1" "24")
  (SET_TILE "Surf2" "24")
  (SET_TILE "error" "")
)

(DEFUN Defecto2 ()
  (SET_TILE "Iso" "4")
  (SET_TILE "Suav" "2")
  (SET_TILE "Sil" "1")
  (SET_TILE "error" "")
)

(DEFUN Chequear (/ Surf1Ch Surf2Ch IsoCh SuavCh)
  (SETQ errores nil)
  (SETQ Surf1Ch (ATOI (GET_TILE "Surf1")))
```

```
(IF (< Surf1Ch 2)
  (PROGN
    (SETQ errores T)
    (SET_TILE "error" "Valor de SURFTAB1 debe ser 2 como mínimo.")
    (MODE_TILE "Surf1" 2)
  )
)
(SETQ Surf2Ch (ATOI (GET_TILE "Surf2")))
(IF (< Surf2Ch 2)
  (PROGN
    (SETQ errores T)
    (SET_TILE "error" "Valor de SURFTAB2 debe ser 2 como mínimo.")
    (MODE_TILE "Surf2" 2)
  )
)
(SETQ IsoCh (ATOI (GET_TILE "Iso")))
(IF (< IsoCh 0)
  (PROGN
    (SETQ errores T)
    (SET_TILE "error" "Valor de las Isolíneas debe ser 0 ó mayor.")
    (MODE_TILE "Iso" 2)
  )
)
(SETQ SuavCh (ATOF (GET_TILE "Suav")))
(IF (OR (< SuavCh 0) (> SuavCh 10))
  (PROGN
    (SETQ errores T)
    (SET_TILE "error" "Valor del Suavizado debe estar entre 0.01 y 10.00.")
    (MODE_TILE "Suav" 2)
  )
)
)

(DEFUN Aceptar ()
  (SETVAR "surftab1" (ATOI (GET_TILE "Surf1")))
  (SETVAR "surftab2" (ATOI (GET_TILE "Surf2")))
  (SETVAR "isolines" (ATOI (GET_TILE "Iso")))
  (SETVAR "facetres" (ATOF (GET_TILE "Suav")))
  (SETVAR "dispsilh" (ATOI (GET_TILE "Sil")))
  (SETVAR "cmddia" (ATOI (GET_TILE "Dia")))
  (SETVAR "filedia" (ATOI (GET_TILE "Ges")))
  (DONE_DIALOG)
)
```

Comentarios al código AutoLISP 1 —VARIABLES.LSP— (última parte)

A partir de donde lo habíamos dejado, el siguiente paso consiste en crear las funciones a las que llaman las diferentes acciones de los botones.

Las funciones Defecto1 y Defecto2, correspondientes a la pulsación de uno de los dos botones que rellenan su parte de cuadro con los valores por defecto, contienen funciones SET_TILE que colocan en cada casilla o cuadro de edición un valor que nosotros hemos considerado por defecto para cada variable. Además, contienen una última función SET_TILE que asigna una cadena vacía a la línea de error, identificada por la key error, como sabemos. De esto hablaremos ahora mismo al comentar la función Chequear.

La función Chequear lo primero que hace es iniciar una variable denominada errores como nula, sin valor (nil). Esto se hace por si el cuadro ya había sido utilizado y la variable

valiera T (haber cometido un error y salir con *Cancelar* sin arreglarlo), lo cual sería antiproducente para nuestros objetivos.

A continuación se extrae mediante `GET_TILE` el valor de cada uno de los elementos del cuadro de diálogo, guardándolos en variables de usuario (previa transformación en valores numéricos enteros o reales, con `ATOI` o `ATOF` respectivamente), y se comparan mediante `IF` con los valores permitidos por **AutoCAD** para cada variable de sistema. Es decir, si `SURFTAB1` es menor de 2, por ejemplo, sabemos que **AutoCAD** produce un error. Lo mismo con los valores o rangos de las demás variables.

NOTA: Recordar que en el **APÉNDICE B** de este curso se ofrecen todos los valores posibles a todas las variables de **AutoCAD**. Aún así, si introducimos un valor descabellado a la pregunta de una variable en la línea de comandos, normalmente se nos ofrece un mensaje de error acompañado del rango de valores aceptados.

NOTA: La razón de que en este ejemplo no se controlen los valores altos de variables como `SURFTAB1` o `SURFTAB2`, es porque desde un principio se limitó la entrada en las casillas de edición a un número máximo de 3 caracteres (argumento `edit_limit`). Por ello no es necesario un control tan exhaustivo. Para que el programa fuera completo, habría que permitir dichos valores y controlar sus entradas, aquí no se hizo porque valores de `SURFTAB1` o `SURFTAB2` mayores de 999 son realmente exagerados (ambos admiten hasta 32766). Si se quisieran introducir valores más altos habría que hacerlo desde línea de comandos.

Seguimos. Si se introduce un valor no válido en una casilla, se establece errores como T y se imprime un mensaje en la línea de errores. Esto se logra con `SET_TILE`, asignando a la key `error` un mensaje entre comillas. A continuación, se establece con `MODE_TILE` un modo 2 para el elemento que contiene el error, es decir, que se iluminará (generalmente en azul).

Todo esto se realiza al pulsar el botón *Aceptar*. Al terminar la función de chequeo de errores, el programa devuelve el control a la siguiente instrucción desde donde fue llamada dicha función, al `IF` del principio. Si errores tiene un valor no ocurre nada (lista vacía), se muestra el error pertinente (ya estaba mostrado) y no se sale del cuadro.

Al corregir el error y volver a pulsar *Aceptar* se repite todo el proceso. Si no hay ningún error se puede seguir adelante. También por ello al iniciar la rutina de control de errores hay que restablecer errores a `nil`.

Cuando todo está bien se pasa a la subrutina *Aceptar*, la cual introduce finalmente los valores válidos en las variables de sistema mediante `SETVAR`, extrayéndolas de los correspondientes *tiles* mediante `GET_TILE` (previa transformación a valores numéricos como anteriormente hemos explicado).

Al final es necesario acabar con `DONE_DIALOG`. Como sabemos, la key `accept` es predefinida y lleva una función predeterminada inherente. Al asociar nosotros `accept` con una rutina *Aceptar*, lo que hacemos es sobrescribir la rutina por defecto de `accept` —que lleva ya implícito un `DONE_DIALOG`—. Es por eso que debemos incluir `DONE_DIALOG`, porque si no el cuadro no se cerraría. El botón *Cancelar* cierra el cuadro sin más.

Existen un par de funciones más dentro de este grupo que no hemos explicado por no estar incluidas en el ejemplo, aunque una la hemos mencionado de pasada. Son `CLIENT_DATA_TILE` y `GET_ATTR`. La sintaxis de la primera (`CLIENT_DATA_TILE`) es la siguiente:

```
(CLIENT_DATA_TILE clave datos_cliente)
```

Esta función asocia datos procesados mediante una aplicación a un elemento de un letrero de diálogo. El argumento *datos*, así como el argumento *clave*, debe ser una cadena de

texto. Una acción asignada al elemento puede hacer referencia a estos datos a través de la variable \$data.

Una aplicación típica es la asignación de datos en una lista a una casilla desplegable:

```
(CLIENT_DATA_TILE  
  "mod_sombra"  
  "256_colores 256_col_resaltar 16_col_resaltar  
  16_col_rellenar"  
)
```

De esta manera se asignan a la casilla con clave `mod_sombra` los cuatro elementos especificados lista.

Posteriormente, y mediante `ACTION_TILE`, es posible referirse a la lista a través de la variable \$data, y/o al elemento o elementos seleccionados a través de la variables \$value.

`(GET_ATTR clave atributo)`

Cuando definimos un archivo en DCL utilizamos diversos atributos asignados a los diversos elementos o *tiles* del letrero de diálogo. Con `GET_ATTR` podemos capturar los valores de esos atributos (`label`, `value`, `width`, `edit_limit`, `key`...) de cada elemento, indicando el valor de su clave o atributo `key`. Tanto *clave* como *atributo* son cadenas.

Por ejemplo, para extraer el valor por defecto con el que fue inicializado el cuadro en determinado componente, podríamos hacer:

```
(GET_ATTR "casilla" "value")
```

ONCE.16.3. Gestión de componentes de imagen

En esta siguiente sección se van a explicar unas funciones especiales para el control de las imágenes de los letreros DCL. Para ello recurriremos a un segundo ejemplo que viene como el anterior de uno de los ejemplos del **MÓDULO DIEZ**. En este caso utilizaremos el la creación de una curva helicoidal o hélice 3D mediante un polilínea 3D.

NOTA: El código de este programa (sin letrero de diálogo) se puede examinar en uno de los ejercicios resueltos de este mismo **MÓDULO**. Aquí, evidentemente, variará un tanto para adaptarlo a la introducción de datos desde el letrero.

Pero antes de comenzar con el ejemplo mostraremos antes el código AutoLISP de la primera parte del programa, el cual se refiere al relleno de casillas y control de valores por defecto. Esta parte todavía no incluye control de imágenes, por lo que dejaremos este tema para después.

Para mayor comodidad, se muestra aquí también tanto el letrero de diálogo en sí como el código DCL que lo define. Comenzamos en la página siguiente.

Letrero 2



Código DCL 2 —HELICE-POL.DCL—

```
helice:dialog {label="Hélice con Polilínea 3D";
:row {
:image {width=20;aspect_ratio=0.8;color=0;fixed_height=true;key=img;}
:boxed_column {label="Radios";
:radio_row {
:radio_button {label="Iguales";key=igu;}
:radio_button {label="Diferentes";key=dif;}
}
:edit_box {label="Radio inicial:";edit_width=6;fixed_width=true;
key=radin;}
:edit_box {label="Radio final: ";edit_width=6;fixed_width=true;
is_enabled=false;key=radif;}
spacer_1;
}
}
:row {
:boxed_column {label="Vueltas";fixed_width=true;
:edit_box {label="Nº vueltas:";edit_width=2;edit_limit=2;key=nv;}
:popup_list {label="Precisión:";edit_width=8;list="8 ptos.\n16 ptos.\n24
ptos.\n32 ptos.";key=pre;}
spacer_1;
}
:boxed_column {label="Paso/Altura";
:radio_row {
:radio_button {label="Paso";value="1";key=bpas;}
:radio_button {label="Altura";key=balt;}
}
:edit_box {label="Paso:";edit_width=8;key=pas;}
:edit_box {label="Altura:";edit_width=8;is_enabled=false;key=alt;}
}
}
:row {ok_cancel;}
```

```
:row {errtile;}  
}
```

Código AutoLISP 2 —HELICE-POL.LSP— (primera parte)

```
(DEFUN Hélice ()  
  (SETQ Ind (LOAD_DIALOG "c:/misdoc~1/autocad/dcl/helice-pol.dcl"))  
  (NEW_DIALOG "helice" Ind)  
  
  (IF radin0 () (SETQ radin0 10))  
  (SET_TILE "radin" (RTOS radin0 2 2))  
  (IF radfin0 () (SETQ radfin0 10))  
  (SET_TILE "radif" (RTOS radfin0 2 2))  
  (IF nv0 () (SETQ nv0 1))  
  (SET_TILE "nv" (ITOA nv0))  
  (IF paso0 () (SETQ paso0 1))  
  (SET_TILE "pas" (RTOS paso0 2 2))  
  (IF alt0 () (SETQ alt0 10))  
  (SET_TILE "alt" (RTOS alt0 2 2))  
  
  (IF op1 () (SETQ op1 "1"))  
  (IF (= op1 "1")  
    (PROGN (SET_TILE "igu" "1")(Iguales))  
    (PROGN (SET_TILE "dif" "1")(Diferentes))  
  )  
  (IF op2 () (SETQ op2 "0"))  
  (SET_TILE "pre" op2)  
  (IF op3 () (SETQ op3 "1"))  
  (IF (= op3 "1")  
    (PROGN (SET_TILE "bpas" "1")(BPaso))  
    (PROGN (SET_TILE "balt" "1")(BAltura))  
  )  
  
  (ACTION_TILE "igu" "(Iguales)")  
  (ACTION_TILE "dif" "(Diferentes)")  
  (ACTION_TILE "bpas" "(BPaso)")  
  (ACTION_TILE "balt" "(BAltura)")  
  (ACTION_TILE "accept" "(Chequear)(IF errores () (Aceptar))")  
  
  (IF (= (START_DIALOG)1)(Dibujar_Hélice))  
)
```

Comentarios al código AutoLISP 2 —HELICE-POL.LSP— (primera parte)

Lo primero que hacemos, como ya hemos de saber, es cargar y mostrar el letrero en pantalla. Seguidamente, y entre `NEW_DIALOG` y `START_DIALOG` se introduce toda la batería de funciones AutoLISP para rellenar el cuadro, establecer los valores por defecto y declarar las funciones de acción. Vamos a ir paso por paso.

La manera que se ha utilizado en este programa para rellenar los elementos del cuadro es un tanto diferente a la usada en el ejemplo anterior. Recuérdese que en dicho ejemplo lo que hacíamos era leer el valor de cada variable del sistema y rellenar las casillas. En éste la técnica es diferente, porque no son variables de sistema y tenemos que darle un valor por defecto. Para ello, nos inventamos unas variables (las acabadas en 0) que guardarán los valores por defecto o los últimos introducidos al volver a abrir el cuadro. Esta técnica ya se ha utilizado en otros ejemplos de programas hasta ahora. Se comprueba si la variable existe y, si no se le da un valor por defecto. La próxima vez que se abra el cuadro, la variable ya tendrá un valor (el dado por el usuario la última vez).

Nos inventamos también otras tres variables (*op1*, *op2* y *op3*) que contendrán los valores por defecto de las dos *:radio_row* y de la *:popup_list*.

op1 se refiere a los botones excluyentes de *Igual* y *Diferente*. Entonces, si *op1* no tiene valor (no existe; igual a *nil*), se le da un valor por defecto de 1. Inmediatamente después se comprueba el valor de *op1*; si es 1 se activa la casilla de *Igual* y se llama a la subrutina *Igual*; si no es 1 se activa la casilla *Diferente* y se llama a la subrutina *Diferente*. Ambas subrutinas se mostrarán más adelante (contienen los *MODE_TILE* para que se inhabilite o no la casilla del radio final dependiendo si ambos son iguales o diferentes).

op2 se refiere a la lista desplegable *Precisión*. Si no tiene un valor se le da por defecto 0. A continuación se asigna dicho valor a la lista desplegable. En las listas desplegables y cajas de lista, el primer valor es identificado como 0, el segundo como 1, el tercero como 2, y así sucesivamente. De esta forma aprovechamos esta nomenclatura para declarar la variable.

Por su lado, *op3* funciona de manera igual que *op1*. Más adelante en el programa incluiremos las funciones necesarias que lean el cuadro a la hora de cerrarlo para actualizar todos estos valores por los indicados por el usuario.

A continuación se asignan las subrutinas de acción a los elementos que las necesitan, es decir, al botón *Aceptar* y a los elementos *:radio_button*, que deberán inhabilitar o no determinadas casillas según qué condiciones.

Por último se incluye el *START_DIALOG* y la llamada a la subrutina que realiza el dibujo final de la hélice. Veremos luego por qué se escribe aquí.

A continuación se explican las funciones para controlar componentes de imagen y se sigue después con el ejemplo.

```
(START_IMAGE clave)
```

Esta función inicia la creación de una imagen en un cuadro de imagen. Hay que suministrarle como argumento la clave o atributo *key* de la casilla de imagen correspondiente en el archivo en DCL.

Las llamadas al resto de funciones que gestionen el componente de imagen se realizarán entre *START_IMAGE* y *END_IMAGE*, la cual enseguida veremos.

```
(SLIDE_IMAGE X Y anchura altura archivo_foto)
```

SLIDE_IMAGE visualiza una foto de **AutoCAD** en la casilla de imagen que se ha inicializado con *START_IMAGE*.

Los argumentos *x* e *y* se refieren al vértice superior izquierdo de la imagen de la foto con respecto al origen de la casilla, es decir, especifican un desfase de la posición de la foto en referencia a la casilla que la contiene. Evidentemente, si se indican ambos como 0, el vértice superior izquierdo de la foto coincide con el de la casilla.

Los argumentos *anchura* y *altura* especifican ambos tamaños de la imagen de foto. Y por último se escribe el nombre de la foto de **AutoCAD** y la ruta de acceso si es necesario. Si la foto se encuentra en una biblioteca de fotos o fototeca, hay que indicar el nombre de esta y, después, el nombre de la foto entre paréntesis. Este último argumento irá entre comillas por ser una cadena. La extensión *.SLD* no es obligatoria, pero conveniente por claridad.

Veamos unos ejemplos:

```
(SLIDE_IMAGE 0 0 10 10 "planta")  
(SLIDE_IMAGE 1 0 12 45 "c:/dcl/fotos/planta.lsd")  
(SLIDE_IMAGE 0 0 100 100 "c:\\dcl\\fotos\\fotos1(planta)")
```

Con respecto a la altura y la anchura de la foto, se puede pensar que puede resultar un poco complicado calcularlas, ya que en DCL se miden las distancias en caracteres y nunca sabremos exactamente cuánto darle a cada valor, porque el resultado final es un tanto arbitrario. Para ahorrarnos dicho cálculo, podemos hacer que AutoLISP calcule por nosotros ambas medidas. Esto se realiza con las dos siguientes funciones que explicamos.

```
(DIMX_TILE X clave)
```

y

```
(DIMY_TILE Y clave)
```

Estas dos funciones de AutoLISP devuelven la anchura o dimensión X (DIMX_TILE) y la altura o dimensión Y (DIMY_TILE) del elemento DCL cuya clave se indica. Podemos utilizarlo con cualquier *tile*, aunque habitualmente se usan con los que implican imágenes.

El origen de la casilla siempre se considera el vértice superior izquierdo. Los valores devueltos son siempre relativos a ese origen, por eso hay que considerarlos hacia la derecha y hacia abajo. De esta manera podríamos escribir una muestra de imagen con SLIDE_IMAGE así:

```
(SLIDE_IMAGE 0 0 (DIMX_TILE "casilla") (DIMY_TILE "casilla") "foto1.sld")
```

De esta forma no habremos de preocuparnos por calcular el ancho y el alto que debe ocupar la foto.

Existe otra forma de utilizar estas dos funciones. Como sabemos también hay un elemento que no es sólo una imagen, sino un botón de imagen. Normalmente esta casilla de imagen se utiliza como un simple botón, es decir, su designación da lugar a una acción. Sin embargo, se pueden definir regiones o zonas de la casilla, de manera que la acción que haya que efectuar sea diferente según en qué zona señala el usuario. El retorno de llamada de una casilla de imagen, es la posición X e Y del punto en que ha señalado el usuario. Esta posición se puede obtener de las variables \$x y \$y ya vistas. Examinando las dimensiones de la casilla, se averigua en qué zona se ha señalado. Este mecanismo se utiliza por ejemplo en el cuadro de DDVPOINT para seleccionar los dos ángulos del punto de vista.

Por ejemplo, se dispone de una casilla de imagen cuya clave es ventanas y se desea considerar cuatro zonas diferenciadas, todas del mismo tamaño:

```
(ACTION_TILE "ventanas" "(sel_ventana $x $y)")  
  
...  
  
(DEFUN sel_ventana (x y)  
  (SETQ mitadx (/ (DIMX_TILE "ventanas") 2)  
    mitady (/ (DIMY_TILE "ventanas") 2))  
  (COND  
    (AND (< x mitadx)(> y mitady)) (SETQ acción "II"))  
    (AND (< x mitadx)(< y mitady)) (SETQ acción "SI"))  
    (AND (> x mitadx)(> y mitady)) (SETQ acción "ID"))  
    (AND (> x mitadx)(< y mitady)) (SETQ acción "SD"))  
  )  
)
```

La acción asignada a la casilla *ventanas* es la función `sel_ventana` que tiene dos variables dependientes `x` e `y`. Al señalar en la casilla, las variables `$x` y `$y` devuelven las coordenadas del punto de señalamiento, siempre referidas al origen en el vértice superior izquierdo de la casilla. La función `sel_ventana` se ejecuta con esos dos valores como variables dependientes o asociadas. Calcula en primer lugar la mitad de tamaño en `X` y en `Y` de la casilla. A continuación examina los valores de las variables `x` e `y` que son los devueltos por `$x` y `$y`. En función de que las coordenadas del punto señalado sean mayores o menores que las de la mitad de la casilla, tanto en `x` como en `y`, se averigua en qué cuadrante ha señalado el usuario. La variable `acción` almacenará el resultado como una cadena de texto con las iniciales combinadas de *inferior*, *superior*, *izquierda* y *derecha*.

`(FILL_IMAGE X Y anchura altura color)`

Esta función rellena con un color plano uniforme la casilla inicializada con `START_IMAGE`. Hemos de indicar también un origen en `X` y en `Y` y una altura y anchura (normalmente con las dos funciones vistas anteriormente).

El argumento `color` especifica el color de **AutoCAD** (0 a 255) con el que se realizará el relleno. Además es posible indicar un número de color lógico según se indica en la siguiente tabla:

Color lógico	Significado
-2	Color de fondo actual de la pantalla gráfica de AutoCAD .
-15	Color de fondo del letrero de diálogo actual.
-16	Color de primer plano (del texto) del letrero de diálogo actual.
-18	Color de líneas del letrero de diálogo actual.

Así podremos, por ejemplo, hacer que desaparezca una casilla de imagen rellenándola con el color de fondo del letrero, o eliminar una foto representada para representar otra (sino se superpondrían).

`(END_IMAGE)`

Simplemente finaliza la gestión de un cuadro de imagen.

Por lo tanto podríamos resumir el proceso principal de gestión de una imagen así, por ejemplo:

```
(START_IMAGE "claveimg")
(SLIDE_IMAGE 0 0 (DIMX_TILE "claveimg") (DIMY_TILE "claveimg") "foto")

...

(END_IMAGE)
```

Código AutoLISP 2 —HELICE-POL.LSP— (segunda parte)

```
(DEFUN Hélice ()
  (SETQ Ind (LOAD_DIALOG "c:/misdoc~1/autocad/dcl/helice-pol.dcl"))
  (NEW_DIALOG "helice" Ind)

  (IF radin0 () (SETQ radin0 10))
  (SET_TILE "radin" (RTOS radin0 2 2))
  (IF radfin0 () (SETQ radfin0 10))
  (SET_TILE "radif" (RTOS radfin0 2 2))
  (IF nv0 () (SETQ nv0 1))
```

```
(SET_TILE "nv" (ITOA nv0))
(IF paso0 () (SETQ paso0 1))
(SET_TILE "pas" (RTOS paso0 2 2))
(IF alt0 () (SETQ alt0 10))
(SET_TILE "alt" (RTOS alt0 2 2))
(IF op1 () (SETQ op1 "1"))
(IF (= op1 "1")
  (PROGN (SET_TILE "igu" "1")(Iguales))
  (PROGN (SET_TILE "dif" "1")(Diferentes))
)
(IF op2 () (SETQ op2 "0"))
(SET_TILE "pre" op2)
(IF op3 () (SETQ op3 "1"))
(IF (= op3 "1")
  (PROGN (SET_TILE "bpas" "1")(BPaso))
  (PROGN (SET_TILE "balt" "1")(BAltura))
)

(ACTION_TILE "igu" "(Iguales)")
(ACTION_TILE "dif" "(Diferentes)")
(ACTION_TILE "bpas" "(BPaso)")
(ACTION_TILE "balt" "(BAltura)")
(ACTION_TILE "accept" "(Chequear)(IF errores () (Aceptar))")

(IF (= (START_DIALOG) 1)(Dibujar_Hélice))
)

(DEFUN Iguales ()
  (START_IMAGE "img")
  (FILL_IMAGE 0 0 (DIMX_TILE "img") (DIMY_TILE "img") 0)
  (SLIDE_IMAGE 0 0 (DIMX_TILE "img") (DIMY_TILE "img")
    "c:/misdoc~1/autocad/dcl/helice1.sld")
  (END_IMAGE)
  (MODE_TILE "radif" 1)
)

(DEFUN Diferentes ()
  (START_IMAGE "img")
  (FILL_IMAGE 0 0 (DIMX_TILE "img") (DIMY_TILE "img") 0)
  (SLIDE_IMAGE 0 0 (DIMX_TILE "img") (DIMY_TILE "img")
    "c:/misdoc~1/autocad/dcl/helice2.sld")
  (END_IMAGE)
  (MODE_TILE "radif" 0)
)
```

Comentarios al código AutoLISP 2 —HÉLICE-POL.LSP— (segunda parte)

Tras lo ya comentado, aquí definimos las dos primeras funciones internas de acción. En cada una de ellas se inicializa la casilla de imagen definida en el archivo DCL. A continuación se rellena de negro (color con el que se definió). Esto se hace porque, al cambiar entre radios iguales y diferentes queremos que la foto varíe, presentándose así una hélice recta o una hélice cónica según el caso. Por eso debemos hacer un relleno de negro, para que al cambiar de foto no se superponga a la que anteriormente había.

Después se muestra la imagen correspondiente —una u otra— y se finaliza con `END_IMAGE`. Por último, se utiliza `MODE_TILE` para activar o desactivar la casilla del radio final según convenga.

NOTA: Apréciase la utilización de `DIMX_TILE` y `DIMY_TILE`.

NOTA: Un truco para sacar las fotos en **AutoCAD**, las cuales luego rellenarán casillas de imágenes en cuadros de diálogo, es minimizar la sesión y ajustar la ventana del programa lo más posible a una equivalencia al cuadro que contendrá la imagen (a escala mayor). Si no hacemos esto, la foto se sacará en un formato que nada tiene que ver con el de la casilla en cuestión y, probablemente, se verá pequeña o descentrada.

Código AutoLISP 2 —HELICE-POL.LSP— (tercera parte)

```
(DEFUN Hélice ()
  (SETQ Ind (LOAD_DIALOG "c:/misdloc~1/autocad/dcl/helice-pol.dcl"))
  (NEW_DIALOG "helice" Ind)

  (IF radin0 () (SETQ radin0 10))
  (SET_TILE "radin" (RTOS radin0 2 2))
  (IF radfin0 () (SETQ radfin0 10))
  (SET_TILE "radif" (RTOS radfin0 2 2))
  (IF nv0 () (SETQ nv0 1))
  (SET_TILE "nv" (ITOA nv0))
  (IF paso0 () (SETQ paso0 1))
  (SET_TILE "pas" (RTOS paso0 2 2))
  (IF alt0 () (SETQ alt0 10))
  (SET_TILE "alt" (RTOS alt0 2 2))

  (IF op1 () (SETQ op1 "1"))
  (IF (= op1 "1")
    (PROGN (SET_TILE "igu" "1")(Iguales))
    (PROGN (SET_TILE "dif" "1")(Diferentes))
  )
  (IF op2 () (SETQ op2 "0"))
  (SET_TILE "pre" op2)
  (IF op3 () (SETQ op3 "1"))
  (IF (= op3 "1")
    (PROGN (SET_TILE "bpas" "1")(BPaso))
    (PROGN (SET_TILE "balt" "1")(Baltura))
  )

  (ACTION_TILE "igu" "(Iguales)")
  (ACTION_TILE "dif" "(Diferentes)")
  (ACTION_TILE "bpas" "(BPaso)")
  (ACTION_TILE "balt" "(Baltura)")
  (ACTION_TILE "accept" "(Chequear)(IF errores () (Aceptar))")

  (IF (= (START_DIALOG) 1)(Dibujar_Hélice))
)

(DEFUN Iguales ()
  (START_IMAGE "img")
  (FILL_IMAGE 0 0 (DIMX_TILE "img") (DIMY_TILE "img") 0)
  (SLIDE_IMAGE 0 0 (DIMX_TILE "img") (DIMY_TILE "img")
    "c:/misdloc~1/autocad/dcl/helice1.sld")
  (END_IMAGE)
  (MODE_TILE "radif" 1)
)

(DEFUN Diferentes ()
  (START_IMAGE "img")
  (FILL_IMAGE 0 0 (DIMX_TILE "img") (DIMY_TILE "img") 0)
  (SLIDE_IMAGE 0 0 (DIMX_TILE "img") (DIMY_TILE "img")
    "c:/misdloc~1/autocad/dcl/helice2.sld")
  (END_IMAGE)
  (MODE_TILE "radif" 0)
)
```

```
(DEFUN BPaso ()
  (MODE_TILE "pas" 0)
  (MODE_TILE "alt" 1)
)
(DEFUN BAltura ()
  (MODE_TILE "pas" 1)
  (MODE_TILE "alt" 0)
)

(DEFUN Chequear ()
  (SETQ Errores nil)
  (SETQ radin0 (ATOF (GET_TILE "radin")))
  (IF (< radin0 0)
    (PROGN
      (SETQ errores T)
      (SET_TILE "error" "Radio inicial no puede ser negativo.")
      (MODE_TILE "radin" 2)
    )
  )

  (SETQ radfin0 (ATOF (GET_TILE "radif")))
  (IF (< radfin0 0)
    (PROGN
      (SETQ errores T)
      (SET_TILE "error" "Radio final no puede ser negativo.")
      (MODE_TILE "radif" 2)
    )
  )

  (SETQ nv0 (ATOI (GET_TILE "nv")))
  (IF (<= nv0 0)
    (PROGN
      (SETQ errores T)
      (SET_TILE "error" "Número de vueltas ha de ser mayor de 0.")
      (MODE_TILE "nv" 2)
    )
  )
)

(DEFUN Aceptar ()
  (SETQ paso0 (ATOF (GET_TILE "pas")))
  (SETQ alt0 (ATOF (GET_TILE "alt")))

  (SETQ op1 (GET_TILE "igu"))
  (SETQ op2 (GET_TILE "pre"))
  (SETQ op3 (GET_TILE "bpas"))

  (COND
    ((= op2 "0") (SETQ pv 8))
    ((= op2 "1") (SETQ pv 16))
    ((= op2 "2") (SETQ pv 24))
    ((= op2 "3") (SETQ pv 32))
  )

  (DONE_DIALOG 1)
  (SETQ radin radin0 radfin radfin0 nv nv0 paso paso0 alt alt0)
```

Comentarios al código AutoLISP 2 —HÉLICE-POL.LSP— (tercera parte)

Una vez establecidas las acciones para el cuadro de imagen, se definen las restantes, es decir la del paso y la de la altura. En este caso, únicamente se habilitan o inhabilitan las casillas correspondientes según el caso.

Después se define la función que controla los errores introducidos en el cuadro de diálogo al pulsar el botón *Aceptar*. Esto se hace de manera análoga al primer ejemplo explicado.

Por último, se define la función de aceptación. Lo que hace ésta es, primero asignar a las variables correspondientes los valores capturados de las casillas del paso y de la altura. Segundo, realiza lo mismo con las variables *op1*, *op2* y *op3*. Tercero, extrae el valor de *op2* (el de la precisión) y, según dicho valor, asigna una precisión u otra a la variable *pv* que se utilizará después en el dibujado de la hélice. Al final, se acaba con *DONE_DIALOG* y se asignan los valores pertinentes de las variables que se utilizan para guardar los valores por defecto a las variables que se usarán en la rutina de dibujo.

NOTA: Nótese que si los valores de la lista desplegable no hubieran sido cadenas con texto (*8 ptos.*, *16 ptos.*, etc.), sino cadenas con un valor simple (*8*, *16...*), podríamos haber capturado directamente el valor actual designado con la variable *\$value* de *ACTION_TILE* y asignárselo como tal (pasándolo a valor numérico) a la variable *pv*.

Como ya hemos comentado, mientras un cuadro de diálogo esté activo no se pueden realizar operaciones de dibujo en el área gráfica. Cuando se ejecuta un programa AutoLISP como el que estamos viendo, tras leerse el principio, se queda "esperando" en el *START_DIALOG* hasta que se produzca un *DONE_DIALOG* y un final de letrero, realizando las llamadas a las subrutinas necesarias en cada momento. Sin embargo, hasta que el cuadro no esté cerrado del todo no se puede realizar ningún proceso más. Esto es, cuando se produce el *DONE_DIALOG*, el control se devuelve a la siguiente instrucción de *START_DIALOG* y es entonces cuando el cuadro estará perfecta y completamente cerrado.

Si hubiéramos puesto la llamada a la subrutina de dibujo (*Dibujar_Hélice*) después del *DONE_DIALOG* (en la rutina *Aceptar*), el cuadro no habría estado cerrado por completo, por lo que no se dibujaría la hélice y se produciría un mensaje de error. Al colocar susodicha llamada inmediatamente después de *START_DIALOG*, se produce el *DONE_DIALOG*, se devuelve el control a *START_DIALOG* y, ahora, el cuadro ya está cerrado, pudiéndose realizar la llamada y el consiguiente proceso de dibujado de la hélice.

La razón para la inclusión de la comparación con el *IF* es que, si sale con el código 1 de *DONE_DIALOG* es que viene el control de la rutina de aceptación, por lo que proseguirá el programa; si sale con un código distinto (0 en este caso) es que se ha pulsado *Cancelar*, por lo que no se proseguirá el programa.

Código AutoLISP 2 —HELICE-POL.LSP— (última parte)

```
(DEFUN Hélice ()
  (SETQ Ind (LOAD_DIALOG "c:/misdpc~1/autocad/dcl/helice-pol.dcl"))
  (NEW_DIALOG "helice" Ind)

  (IF radin0 () (SETQ radin0 10))
  (SET_TILE "radin" (RTOS radin0 2 2))
  (IF radfin0 () (SETQ radfin0 10))
  (SET_TILE "radif" (RTOS radfin0 2 2))
  (IF nv0 () (SETQ nv0 1))
  (SET_TILE "nv" (ITOA nv0))
  (IF paso0 () (SETQ paso0 1))
  (SET_TILE "pas" (RTOS paso0 2 2))
  (IF alt0 () (SETQ alt0 10))
```

```
(SET_TILE "alt" (RTOS alt0 2 2))

(IF op1 () (SETQ op1 "1"))
(IF (= op1 "1")
  (PROGN (SET_TILE "igu" "1")(Iguales))
  (PROGN (SET_TILE "dif" "1")(Diferentes))
)
(IF op2 () (SETQ op2 "0"))
(SET_TILE "pre" op2)
(IF op3 () (SETQ op3 "1"))
(IF (= op3 "1")
  (PROGN (SET_TILE "bpas" "1")(BPaso))
  (PROGN (SET_TILE "balt" "1")(BAltura))
)

(ACTION_TILE "igu" "(Iguales)")
(ACTION_TILE "dif" "(Diferentes)")
(ACTION_TILE "bpas" "(BPaso)")
(ACTION_TILE "balt" "(BAltura)")
(ACTION_TILE "accept" "(Chequear)(IF errores () (Aceptar))")

(IF (= (START_DIALOG) 1)(Dibujar_Hélice))
)

(DEFUN Iguales ()
  (START_IMAGE "img")
  (FILL_IMAGE 0 0 (DIMX_TILE "img") (DIMY_TILE "img") 0)
  (SLIDE_IMAGE 0 0 (DIMX_TILE "img") (DIMY_TILE "img")
    "c:/misdod~1/autocad/dcl/helice1.sld")
  (END_IMAGE)
  (MODE_TILE "radif" 1)
)

(DEFUN Diferentes ()
  (START_IMAGE "img")
  (FILL_IMAGE 0 0 (DIMX_TILE "img") (DIMY_TILE "img") 0)
  (SLIDE_IMAGE 0 0 (DIMX_TILE "img") (DIMY_TILE "img")
    "c:/misdod~1/autocad/dcl/helice2.sld")
  (END_IMAGE)
  (MODE_TILE "radif" 0)
)

(DEFUN BPaso ()
  (MODE_TILE "pas" 0)
  (MODE_TILE "alt" 1)
)

(DEFUN BAltura ()
  (MODE_TILE "pas" 1)
  (MODE_TILE "alt" 0)
)

(DEFUN Chequear ()
  (SETQ Errores nil)
  (SETQ radin0 (ATOF (GET_TILE "radin")))
  (IF (< radin0 0)
    (PROGN
      (SETQ errores T)
      (SET_TILE "error" "Radio inicial no puede ser negativo.")
      (MODE_TILE "radin" 2)
    )
  )
)
```

```
)

(SETQ radfin0 (ATOF (GET_TILE "radif")))
(IF (< radfin0 0)
  (PROGN
    (SETQ errores T)
    (SET_TILE "error" "Radio final no puede ser negativo.")
    (MODE_TILE "radif" 2)
  )
)

(SETQ nv0 (ATOI (GET_TILE "nv")))
(IF (<= nv0 0)
  (PROGN
    (SETQ errores T)
    (SET_TILE "error" "Número de vueltas ha de ser mayor de 0.")
    (MODE_TILE "nv" 2)
  )
)
)

(DEFUN Aceptar ()
  (SETQ paso0 (ATOF (GET_TILE "pas")))
  (SETQ alt0 (ATOF (GET_TILE "alt")))

  (SETQ op1 (GET_TILE "igu"))
  (SETQ op2 (GET_TILE "pre"))
  (SETQ op3 (GET_TILE "bpas"))

  (COND
    ((= op2 "0")(SETQ pv 8))
    ((= op2 "1")(SETQ pv 16))
    ((= op2 "2")(SETQ pv 24))
    ((= op2 "3")(SETQ pv 32))
  )

  (DONE_DIALOG 1)
  (SETQ radin radin0 radfin radfin0 nv nv0 paso paso0 alt alt0)
)

(DEFUN Dibujar_Hélice (/ cen dz drad dang n z z0)
  (INITGET 1)
  (SETQ cen (GETPOINT "Centro de la hélice: "))(TERPRI)
  (IF (= op1 "1") (SETQ radfin radin))
  (IF (= op3 "0") (SETQ paso (/ alt nv)))
  (SETQ dang (/ (* 2 PI) pv))
  (SETQ dz (/ paso pv))
  (SETQ drad (/ (- radfin radin) (* pv nv )))

  (SETQ p0 (POLAR cen 0 radin))
  (SETQ z0 (CADDR p0))
  (COMMAND "_3dpoly" p0)
  (SETQ n 1)
  (REPEAT (* pv nv)
    (SETQ z (+ z0 (* dz n)))
    (SETQ xy (POLAR cen (* dang n) (+ radin (* drad n))))
    (SETQ pto (LIST (CAR xy) (CADR xy) z))
    (COMMAND pto)
    (SETQ n (1+ n))
  )
  (COMMAND)
```

```
)  
  
(DEFUN C:Hélice ()  
  (SETVAR "CMDECHO" 0)  
  (SETQ Error_Dibujo *error* *error* Control_Errores)  
  (Hélice)  
  (SETQ *error* Error_Dibujo)  
  (SETVAR "CMDECHO" 1)  
  (PRIN1)  
)  
  
(DEFUN Control_Errores (Mensaje)  
  (PRINC (STRCAT "Error: " Mensaje))(TERPRI)  
  (SETQ *error* Error_Dibujo)  
  (SETVAR "CMDECHO" 1)  
  (COMMAND)  
  (PRIN1)  
)
```

Comentarios al código AutoLISP 2 —HÉLICE-POL.LSP— (última parte)

Por último, se define la rutina de dibujo, la de control de errores en tiempo de corrida y la que define el nuevo comando HÉLICE de **AutoCAD**.

Existe una función más de gestión de imágenes de cuadros de diálogo que no hemos visto por no estar incluida en el ejemplo. Veámosla ahora:

```
(VECTOR_IMAGE X1 Y1 X2 Y2 color)
```

La función AutoLISP `VECTOR_IMAGE` dibuja un vector en la casilla de imagen cuyo proceso se haya iniciado mediante `START_IMAGE`.

Para el dibujo de dicho vector se indican unas coordenadas X e Y de origen (argumentos `x1` e `y1`), unas coordenadas X e Y de destino (argumentos `x2` e `y2`) y un color (argumento `color`). Las coordenadas siempre referidas al vértice superior izquierdo de la casilla de imagen; el color puede ser cualquiera de los de **AutoCAD** o cualquiera de los colores lógicos expuestos en la tabla de la función `FILL_IMAGE`.

Esta función se puede utilizar para representar dibujos sencillos en las casillas o para dividirlos en cuadrantes, por ejemplo.

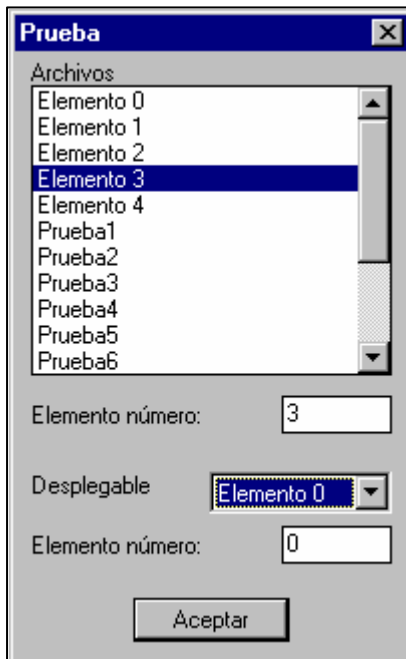
ONCE.16.4. Gestión de casillas de lista y listas desplegables

Al igual que lo visto hasta ahora, existen también tres funciones específicas para el manejo de casillas o cajas de listas y de listas desplegables. En este tipo de elementos del letrero, el programa de control AutoLISP debe tener acceso por un lado a la lista completa asignada al elemento y, por otro lado debe disponer de un retorno de llamada que especifique el elemento o elementos de la lista designados por el usuario.

La diferencia entre ambos elementos es que en las casillas con listados es posible permitir varias designaciones al tiempo, mientras que en las listas desplegables sólo es posible una.

Las funciones en concreto las veremos a enseguida y, para ello, vamos a utilizar un pequeño ejemplo de prueba cuyo código DCL se muestra a continuación del cuadro en sí.

Letrero 3



Código DCL 3 —PRUEBA.DCL—

```
Prueba:dialog {label="Prueba";
:column {
:row {
:list_box {label="Archivos";list="Elemento 0\nElemento 1\nElemento
2\nElemento 3\nElemento 4";key="ClaveLista";}
}
:row {
:edit_box {label="Elemento número:";key="ClaveElemenLista";}
}
spacer_1;
:row {
:popup_list {label="Desplegable";list="Elemento 0\nElemento 1\nElemento
2";key="ClaveDespleg";}
}
:row {
:edit_box {label="Elemento número:";key="ClaveElemenDespleg";}
}
spacer_1;
ok_only;
}
}
```

Antes de ver el ejemplo en AutoLISP vamos a explicar estas tres funciones de control de cuadros de lista y listas desplegables.

<code>(START_LIST clave [operación [índice]])</code>

Esta función `START_LIST` inicia el proceso de control de una lista de casilla o desplegable. Ha de encontrarse situada entre `NEW_DIALOG` y `START_DIALOG`.

El argumento *clave* suministra la clave o atributo *key* del elemento en cuestión. Por su lado, *operación* es un número entero que puede tener uno de los valores siguientes:

<i>operación</i>	Significado
1	Cambiar contenido seleccionado de la lista.
2	Añadir nueva entrada a la lista.
3	Suprimir lista actual y crear una nuevo (es el valor por defecto).

Estos valores de operación controlan la manera en que sucesivas funciones `ADD_LIST` (que ahora veremos) van a modificar la lista. El argumento *índice* sólo tiene efecto si se ha indicado un código 1 para *operación*. En este caso, *índice* especifica el número de orden (comenzando desde 0 como sabemos) del elemento de la lista que será cambiado en la posterior utilización de `ADD_LIST`. Veamos un ejemplo:

```
(START_LIST "listaobjetos" 1 4)
(ADD_LIST "Objeto1")
(END_LIST)
```

Este ejemplo cambia el elemento quinto de la casilla por uno nuevo denominado `Objeto1`.

Otro ejemplo:

```
(SETQ ListaObjs '("Objeto1" "Objeto2" "Objeto3"))
(START_LIST "listaobjetos" 2)
(MAPCAR 'ADD_LIST ListaObjs)
```

Este ejemplo añade varios elementos al final de la lista ya existente.

Último ejemplo:

```
(SETQ ListaObjs '("Objeto1" "Objeto2" "Objeto3"))
(START_LIST "listaobjetos")
(MAPCAR 'ADD_LIST ListaObjs)
```

Este ejemplo reemplaza todos los elementos de la lista existente por los de la lista proporcionada. Se podría haber indicado el índice 3.

NOTA: No es posible eliminar o insertar en medio un elemento nuevo en la lista. Para hacerlo habría que reconstruir la lista desde el primer elemento.

`(ADD_LIST cadena)`

Esta función realiza una u otra función según el código de operación explicado en `START_LIST`. Sólo se puede incluir una cadena por cada `ADD_LIST`. Para añadir más de un elemento a una lista, por ejemplo, habría que hacer sucesivas llamadas a `ADD_LIST`.

`(END_LIST)`

Finaliza el proceso de control de caja de lista o lista desplegable iniciado por `START_LIST`. Siempre irá la última tras ésta y `ADD_LIST`.

Por lo tanto, la manera genérica de utilizar estas tres funciones es:

```
(START_LIST "lista")
(ADD_LIST "cadena")
(END_LIST)
```

Veamos ahora pues el código AutoLISP de nuestro ejemplo:

Código AutoLISP 3 —PRUEBA.LSP— (primera parte)

```
(DEFUN Datos (/ Cont)
  (INITGET 7)
  (SETQ Num (GETINT "Número de términos que se añadirán al cuadro de lista:
    "))(TERPRI)
  (SETQ Cont 1)
  (SETQ Lista '())
  (REPEAT Num
    (INITGET 1)
    (SETQ Term (GETSTRING T (STRCAT "\nTérmino nuevo número " (ITOA Cont) " del
      cuadro: ")))
    (IF Lista
      (PROGN
        (SETQ Term (LIST Term))
        (SETQ Lista (APPEND Lista Term))
      )
      (SETQ Lista (LIST Term))
    )
    (SETQ Cont (1+ Cont))
  )

  (INITGET 7)
  (SETQ Num2 (GETINT "\nNúmero de términos que se añadirán a la lista
    desplegable: "))(TERPRI)
  (SETQ Cont 1)
  (SETQ Lista2 '())
  (REPEAT Num2
    (INITGET 1)
    (SETQ Term2 (GETSTRING T (STRCAT "\nTérmino nuevo número " (ITOA Cont) " de
      la lista: ")))
    (IF Lista2
      (PROGN
        (SETQ Term2 (LIST Term2))
        (SETQ Lista2 (APPEND Lista2 Term2))
      )
      (SETQ Lista2 (LIST Term2))
    )
    (SETQ Cont (1+ Cont))
  )
)
```

Comentarios al código AutoLISP 3 —PRUEBA.LSP— (primera parte)

En esta primera del código en AutoLISP todavía no se utilizan las funciones de cajas de listas y listas desplegables, sino que se preparan dos lista para luego añadir a las predefinidas en el archivo .DCL.

Se le pide al usuario el número de términos que va a añadir a la caja de lista y, posteriormente, el número de términos que va a añadir a la lista desplegable. Para cada término se va solicitando un valor con el que se irá formando cada lista. Si es el primer valor se forma una lista con él, en los sucesivos se van añadiendo términos a la primera lista.

NOTA: La función APPEND aún no la hemos estudiado, pero se puede deducir fácilmente se funcionamiento.

Código AutoLISP 3 —PRUEBA.LSP— (última parte)

```
(DEFUN Datos (/ Cont)
  (INITGET 7)
  (SETQ Num (GETINT "Número de términos que se añadirán al cuadro de lista:
    "))(TERPRI)
  (SETQ Cont 1)
  (SETQ Lista '())
  (REPEAT Num
    (INITGET 1)
    (SETQ Term (GETSTRING T (STRCAT "\nTérmino nuevo número " (ITOA Cont) " del
      cuadro: ")))
    (IF Lista
      (PROGN
        (SETQ Term (LIST Term))
        (SETQ Lista (APPEND Lista Term))
      )
      (SETQ Lista (LIST Term))
    )
    (SETQ Cont (1+ Cont))
  )

  (INITGET 7)
  (SETQ Num2 (GETINT "\nNúmero de términos que se añadirán a la lista
    desplegable: "))(TERPRI)
  (SETQ Cont 1)
  (SETQ Lista2 '())
  (REPEAT Num2
    (INITGET 1)
    (SETQ Term2 (GETSTRING T (STRCAT "\nTérmino nuevo número " (ITOA Cont) " de
      la lista: ")))
    (IF Lista2
      (PROGN
        (SETQ Term2 (LIST Term2))
        (SETQ Lista2 (APPEND Lista2 Term2))
      )
      (SETQ Lista2 (LIST Term2))
    )
    (SETQ Cont (1+ Cont))
  )
)

(DEFUN Prueba (/ Cont ElemAdd)
  (SETQ Ind (LOAD_DIALOG "c:/misdoc~1/autocad/dcl/prueba.dcl"))
  (NEW_DIALOG "Prueba" Ind)

  (IF Lista0 () (SETQ Lista0 "0"))
  (SET_TILE "ClaveLista" Lista0)

  (IF Despleg0 () (SETQ Despleg0 "1"))
  (SET_TILE "ClaveDespleg" Despleg0)

  (IF ElemLista () (SETQ ElemLista Lista0))
  (SET_TILE "ClaveElemenLista" ElemLista)

  (IF ElemDespleg () (SETQ ElemDespleg Despleg0))
  (SET_TILE "ClaveElemenDespleg" ElemDespleg)
```

```
(ACTION_TILE "ClaveLista" "(SETQ ElemLista $value) (CajaLista)")
(ACTION_TILE "ClaveDespleg" "(SETQ ElemDespleg $value) (Desplegable)")
(ACTION_TILE "accept" "(Aceptar)")
(START_LIST "ClaveLista" 2)
(SETQ Cont 0)
(REPEAT Num
  (SETQ ElemAdd (NTH Cont Lista))
  (ADD_LIST ElemAdd)
  (SETQ Cont (1+ Cont))
)
(END_LIST)

(START_LIST "ClaveDespleg" 2)
(SETQ Cont 0)
(REPEAT Num2
  (SETQ ElemAdd (NTH Cont Lista2))
  (ADD_LIST ElemAdd)
  (SETQ Cont (1+ Cont))
)
(END_LIST)

(START_DIALOG)
)

(DEFUN CajaLista ()
  (SET_TILE "ClaveElemenLista" ElemLista)
)

(DEFUN Desplegable ()
  (SET_TILE "ClaveElemenDespleg" ElemDespleg)
)

(DEFUN Aceptar ()
  (SETQ Lista0 "0" Despleg0 "1")
  (DONE_DIALOG)
)

(DEFUN C:Prueba ()
  (Datos)
  (Prueba)
)
```

Comentarios al código AutoLISP 3 —PRUEBA.LSP— (última parte)

A continuación de lo anterior, se inicializa el cuadro y los distintos elementos también. Además se declaran los ACTION_TILE necesarios para el funcionamiento de los elementos con acción asociada. Estos elementos son el botón *Aceptar*, el cuál inicializará de nuevo los elementos con sus valores originales y cerrará el cuadro; la caja de lista, en la que al seleccionar un elemento se escribirá su número de orden en la casilla de edición inferior; y la lista desplegable, que realiza lo mismo que la caja de lista en la casilla inferior a ella. Estas acciones las podemos ver definidas en las subrutinas correspondientes; véase el funcionamiento y utilización de la variable \$value.

La manera de añadir los elementos nuevos en la lista es la explicada en la teoría. Únicamente explicar que se utiliza la función NTH que va cogiendo cada elemento de la lista por su número de orden (empezando por el cero). Esta función se explicará en breve.

Por último, se define el nuevo comando de **AutoCAD** Prueba.

14ª fase intermedia de ejercicios

• Desarrollese un programa AutoLISP para controlar y hacer funcionar cada uno de los cuadros de diálogo de los ejercicios propuestos del **MÓDULO DIEZ**.

ONCE.17. OTRAS FUNCIONES DE MANEJO DE LISTAS

En la sección **ONCE.10.** ya se explicó una serie de funciones las cuales tenían como cometido manejar listas. Ahora veremos un complemento a lo ya aprendido.

Las funciones que vamos a estudiar manejan listas de la misma forma que las mencionadas en dicha sección **ONCE.10.**, pero éstas son mucho más completas y funcionales. Descubriremos maneras de realizar trabajos, que antes podían resultar tediosos, de forma sencilla, rápida y eficaz. Pero sobre todo, aprenderemos las funciones básicas que luego nos permitirán acceder a la Base de Datos interna de **AutoCAD** que, como ya se comentará, está estructurada en forma de listas y sublistas.

Todo esto no quiere decir que hemos de olvidar las otras funciones de manejo de listas estudiadas, pues dependiendo del ejercicio que se deba realizar pueden ser útiles. Además, ciertos problemas únicamente pueden resolverse con ellas.

Comencemos pues con la primera de estas nuevas funciones, llamada ASSOC; su sintaxis es la siguiente:

```
(ASSOC elemento_clave lista_asociaciones)
```

ASSOC busca el elemento especificado en *elemento_clave* en la lista especificada en *lista_asociaciones*; devuelve la lista asociada cuyo primer elemento es el especificado. Para ello, esta lista debe ser una lista de asociaciones, es decir, que contenga sublistas incluidas con elementos asociados, si no no funcionará.

Veamos un ejemplo sencillo. Supongamos una lista que contenga varias sublistas de asociaciones y definida de la siguiente forma:

```
(SETQ milista (LIST '(largo 10) '(ancho 20) '(alto 30)))
```

es decir, que el valor de la lista *milista* es un conjunto de listas de asociación. Las siguientes funciones ASSOC devolverían lo que se indica:

(ASSOC largo milista)	devuelve (LARGO 10)
(ASSOC ancho milista)	devuelve (ANCHO 20)
(ASSOC alto milista)	devuelve (ALTO 30)
(ASSOC volumen milista)	devuelve nil

Veremos ahora otra también muy utilizada cuya sintaxis es:

```
(CONS primer_elemento lista)
```

Esta función toma la lista indicada y le añade un nuevo primer elemento, devolviendo la lista resultante. Así, si tenemos una lista definida de la manera siguiente:

```
(SETQ milista '(10 20 30))
```

podemos hacer:

```
(CONS 5 milista) devuelve (5 10 20 30)
```

Pero atención; este comando añade el elemento y devuelve la lista completa, pero el cambio no es permanente. Si queremos actualizar la lista para futuros usos, la manera sería:

```
(SETQ milista (CONS 5 milista))
```

También, evidentemente podríamos hacer algo como lo siguiente (sin ningún problema):

```
(SETQ lista3 (CONS (CAR lista1) (CDR lista2)))
```

De esta forma, añadimos a la lista *lista2* (exceptuando el primer elemento) el primer elemento de la lista *lista1*. Todo ello lo guardamos en *lista3*.

Existe un modo particular de trabajar con la función *CONS*. Y es que si como argumento *lista* no especificamos una lista, sino un valor concreto o el nombre de una variable definida, la función construye un tipo especial de lista de dos elementos llamado *par punteado* (se denomina así por tener un punto de separación entre ambos elementos). Los pares punteados ocupan menos memoria que las listas normales y son muy utilizados en la Base de Datos de **AutoCAD**. Así:

(CONS 'clase 1)	devuelve (CLASE . 1)
(CONS 'nombre 'antonio)	devuelve (NOMBRE . ANTONIO)
(CONS 'nota 5.5)	devuelve (NOTA . 5.5)

NOTA: A estos elementos de par punteado se accede directamente con las funciones *CAR* (para el primero) y *CDR* (para el segundo), no *CADR*.

```
(SUBST elemento_nuevo elemento_antiguo lista)
```

El cometido de esta función es sustituir un elemento de una lista. Para ello, busca en la lista indicada como último argumento (*lista*) el elemento indicado como segundo argumento (*elemento_antiguo*) y lo sustituye por el elemento indicado como primer argumento (*elemento_nuevo*).

Por ejemplo, la variable *lin* contiene una lista que es el par punteado (8 . "0") y queremos cambiar su segundo elemento:

```
(SETQ lin (SUBST "pieza" "0" lin))
```

El nuevo par punteado será (8 . "PIEZA").

NOTA: Sin querer acabamos de ver un ejemplo en el que cambiamos a una línea de capa, ya que 8 es el código de la Base de Datos de **AutoCAD** para el nombre de la capa y se expresa como una sublista de asociaciones (porque asocia un valor a un código), que es un par punteado. La mecánica básica es ésta, entre otras operaciones, pero ya lo veremos ampliamente más adelante. Sirva de introducción.

La función siguiente *APPEND* reúne todos los elementos de las listas especificadas en una sola lista que los engloba. Su sintaxis es la siguiente:

```
(APPEND lista1 lista2...)
```

Veamos un ejemplo:

```
(APPEND '(e1 e2) '(e3 e4))           devuelve (E1 E2 E3 E4)
```

NOTA: Obsérvese que no se devuelve ((E1 E2)(E3 E4)). Es decir, lo que reúne son los elementos de las listas y no las listas mismas. Así:

```
(APPEND '(e1 (e2 e3)) '(e4 (e5)))     devuelve (E1 (E2 E3) E4 (E5))
```

ya que (e2 e3) y (e5) son sublistas que, en realidad, son elementos de las listas que los contienen.

Los argumentos de APPEND han de ser siempre listas. Esto es lo que diferencia a esta función de LIST —ya estudiada—, que reúne elementos sueltos y forma una lista. Es por ello que a APPEND no se le pueda indicar como lista elementos de una lista extraídos con CAR o CDR por ejemplo. Habríamos de formar listas previas con ellos para luego utilizarlas con esta función APPEND.

NOTA: APPEND no funciona con pares punteados directamente.

Estas cuatro funciones vistas hasta aquí (ASSOC, CONS, SUBST y APPEND) son las más habituales en la gestión de la Base de Datos de **AutoCAD**. Veremos a continuación otras también muy interesantes.

`(LENGTH lista)`

Esta función devuelve la longitud de la lista indicada, es decir, su número de elementos. La devolución de LENGTH será siempre un número entero, evidentemente. Veamos uno ejemplos:

```
(LENGTH '(10 n es 14 5 o))           devuelve 6
(LENGTH '(10 20 (10 20 30)))          devuelve 3
(LENGTH '())                          devuelve 0
```

NOTA: LENGTH no funciona con pares punteados directamente.

`(LAST lista)`

Esta función devuelve el último elemento de la lista especificada. Veamos unos ejemplos:

```
(LAST '(10 n es 14 5 o))              devuelve o
(LAST '(10 20 (10 20 30)))            devuelve (10 20 30)
(LAST '(nombre))                     devuelve NOMBRE
(LAST '())                            devuelve nil
```

Aunque parezca evidente, LAST no es nada aconsejable para devolver la coordenada Z de una lista que represente un punto. Y es que si el punto sólo tuviera las coordenadas X e Y (punto 2D sin Z=0), LAST devolvería la Y. En cambio, con CADDR —ya estudiado— tenemos la seguridad de obtener siempre la coordenada Z, pues devolvería nil en caso de no existir.

NOTA: LAST no funciona con pares punteados directamente.

`(MEMBER elemento lista)`

MEMBER busca el elemento especificado en la lista indicada y devuelve el resto de la lista a partir de ese elemento, incluido él mismo. Veamos algunos ejemplos:

```
(MEMBER 'x '(n h x s u w))      devuelve (X S U W)
(MEMBER 'd1 '(n d1 x d1 u))     devuelve (D1 X D1 U)
(MEMBER '(3 4) '((1 2) (3 4) 5)) devuelve ((3 4) 5)
```

Como vemos en el segundo ejemplo, si el elemento en cuestión se repite en la lista se toma la primera aparición.

Si el elemento buscado no existe, MEMBER devuelve nil.

NOTA: MEMBER no funciona con pares punteados directamente.

`(NTH número_orden lista)`

NTH devuelve el elemento de la lista *lista* que se encuentre en la posición *número_orden*.

Esta función puede resultar harto interesante a la hora de acceder a elementos “lejanos” en una lista, ya que evitaremos el uso de las combinaciones de CAR y CDR que pueden dar lugar a confusión.

Debemos tener en cuenta que el primer elemento de la lista para NTH es el 0, luego el 1, el 2, etcétera. Si la posición especificada es mayor que la posición del último elemento de la lista, NTH devuelve nil. Vamos a ver algún ejemplo:

```
(NTH 2 '(10 20 30))      devuelve 30
(NTH 0 '(10 20 30))      devuelve 10
(NTH 1 '(10 (10 20) 20)) devuelve (10 20)
(NHT 3 '(10 20 30))      devuelve nil
```

NOTA: NTH no funciona con pares punteados directamente.

`(REVERSE lista)`

REVERSE tiene como misión única y exclusiva devolver la lista indicada con todos sus elementos invertidos en el orden. Ejemplos:

```
(REVERSE '(10 20 30 40))      devuelve (40 30 20 10)
(REVERSE '(x y (10 20) z))     devuelve (Z (10 20) Y X)
(REVERSE '(nombre))           devuelve (NOMBRE)
(REVERSE '())                 devuelve nil
```

NOTA: REVERSE no funciona con pares punteados directamente.

`(ACAD_STRLSORT lista)`

Toma la lista especificada, ordena las cadenas de texto contenidas alfabéticamente y devuelve la lista resultante. La lista sólo puede contener cadenas entre comillas. Ejemplos:

```
(ACAD_STRLSORT '("z" "s" "a" "g" "p"))      devuelve ("a" "g" "p" "s" "z")
(ACAD_STRLSORT '("zar" "aire" "12" "4"))     devuelve ("12" "4" "aire" "zar")
(ACAD_STRLSORT '("sol" "sal" "s" "s"))       devuelve ("s" "s" "sal" "sol")
```

NOTA: Como se puede observar, los números se ordenan como las palabras, es decir, comenzando por el primer dígito, siguiendo con el segundo y así sucesivamente (tipo Windows).

NOTA: ACAD_STRLSORT no funciona con pares punteados directamente.

Para terminar vamos a ver un ejemplo de un programa que, aunque únicamente utiliza una de las funciones vistas en esta sección, puede resultar muy jugoso como ejemplo. A la hora de estudiar el acceso a la Base de Datos de **AutoCAD**, será cuando comencemos a sacar partido de estas funciones. Por ahora, sólo queda que el lector practique particularmente con ellas; son muy sencillas de comprender.

El ejemplo que veremos se corresponde con un programa que permite dibujar tornillos normalizados rápidamente. Este programa maneja un cuadro de diálogo, por lo tanto se proporciona también el mismo, así como su código DCL. El cuadro del programa es el siguiente:



A continuación se muestra el código DCL de este cuadro de diálogo:

```
tornillo:dialog {label="Tornillo normalizado";
  :boxed_row {label="Norma DIN";
    :column {
      :image {height=8;aspect_ratio=0.8;color=graphics_background;
        key="timagen";}
      spacer_1;
    }
    :column {
      spacer_1;
      :popup_list {label="&Norma DIN:";list="DIN 931";key="tnorma";}
      spacer_1;
    }
  }
  :row {
```



```
:boxed_column {label="Punto de inserción";
  spacer_1;
  :text {label="X:";key="tx";}
  :text {label="Y:";key="ty";}
  :text {label="Z:";key="tz";}
  spacer_1;
  :button {label="&Designar punto <";key="tpoints";}
  spacer_1;
}
:boxed_column {label="Dimensión";
  spacer_1;
  :popup_list {label="&Métrica ";key="tmetrica";}
  :popup_list {label="&Longitud";key="tlongitud";}
  spacer_1;
}
}
spacer_1;
ok_cancel;
errtile;
}
```

Como se puede apreciar, el cuadro posee varias áreas para solicitar los datos necesarios al usuario y así dibujar el tornillo. En el área de la norma únicamente hay una lista desplegable que sólo posee una sola norma invariable (DIN 931), no se ha diseñado para más. Es por ello que el contenido de esa lista lo añadimos en el código DCL; los componentes de las demás listas serán agregados en la rutina AutoLISP. Veámosla pues:

```
(DEFUN Datos (/ SD Ancho Alto)
  (SETQ SD nil)
  (IF (NOT PuntoIns)
    (SETQ Ind (LOAD_DIALOG "c:/misdoc~1/autocad/autolisp/tornillo.dcl"))
  )
  (NEW_DIALOG "tornillo" Ind)
  (START_IMAGE "timagen")
  (SETQ Ancho (DIMX_TILE "timagen"))
  (SETQ Alto (DIMY_TILE "timagen"))
  (SLIDE_IMAGE 0 0 Ancho Alto "c:/misdoc~1/autocad/autolisp/tornillo.sld")
  (END_IMAGE)

  (START_LIST "tmetrica")
  (SETQ lista1 '("8" "10" "12"))
  (MAPCAR 'ADD_LIST lista1)
  (MODE_TILE "tmetrica" 3)
  (END_LIST)
  (START_LIST "tlongitud")
  (SETQ lista2 '("35" "40" "50" "60" "70" "80" "90" "100" "110"))
  (MAPCAR 'ADD_LIST lista2)
  (END_LIST)

  (IF Métrica0 () (SETQ Métrica0 "0"))
  (SET_TILE "tmetrica" Métrica0)
  (IF Longitud0 () (SETQ Longitud0 "0"))
  (SET_TILE "tlongitud" Longitud0)
  (IF PuntoIns
    (PROGN
      (SET_TILE "tx" (STRCAT "X: " (RTOS (CAR PuntoIns))))
      (SET_TILE "ty" (STRCAT "Y: " (RTOS (CADR PuntoIns))))
      (SET_TILE "tz" (STRCAT "Z: " (RTOS (CADDR PuntoIns))))
    )
  )
)
```

```
(ACTION_TILE "tpoints" "(SETQ Métrica0 (GET_TILE \"tmetrica\") Longitud0
                                (GET_TILE \"tlongitud\"))
                                (DONE_DIALOG 2)\"
)
(ACTION_TILE "accept" "(ControlDCL)
    (IF PuntoIns
        (PROGN
            (SETQ Métrica0 (GET_TILE \"tmetrica\") Longitud0
                (GET_TILE \"tlongitud\"))
            (DONE_DIALOG 1)
        )
    )"
)
(ACTION_TILE "cancel" "(DONE_DIALOG 0)\"")

(SETQ SD (START_DIALOG))
(COND
    ((= 2 SD) (Designar))
    ((= 1 SD) (IF ErrorDCL () (Aceptar)))
    ((= 0 SD) ())
)
)

(DEFUN Designar ()

    (SETQ PuntoIns (GETPOINT "\nPunto para la inserción: "))
    (Datos)

)

(DEFUN Aceptar ()
    (COND
        ((= Métrica0 "0") (SETQ Métrica (ATOI (NTH 0 lista1))))
        ((= Métrica0 "1") (SETQ Métrica (ATOI (NTH 1 lista1))))
        ((= Métrica0 "2") (SETQ Métrica (ATOI (NTH 2 lista1))))
    )
    (COND
        ((= Longitud0 "0") (SETQ Longitud (ATOI (NTH 0 lista2))))
        ((= Longitud0 "1") (SETQ Longitud (ATOI (NTH 1 lista2))))
        ((= Longitud0 "2") (SETQ Longitud (ATOI (NTH 2 lista2))))
        ((= Longitud0 "3") (SETQ Longitud (ATOI (NTH 3 lista2))))
        ((= Longitud0 "4") (SETQ Longitud (ATOI (NTH 4 lista2))))
        ((= Longitud0 "5") (SETQ Longitud (ATOI (NTH 5 lista2))))
        ((= Longitud0 "6") (SETQ Longitud (ATOI (NTH 6 lista2))))
        ((= Longitud0 "7") (SETQ Longitud (ATOI (NTH 7 lista2))))
        ((= Longitud0 "8") (SETQ Longitud (ATOI (NTH 8 lista2))))
    )
    (COND
        ((= Métrica 8) (Métrica_8))
        ((= Métrica 10) (Métrica_10))
        ((= Métrica 12) (Métrica_12))
    )
)

(DEFUN Métrica_8 ()
    (SETQ K 22 EA 14.38 H 5.5 EC 13)
    (Dibujo)
)

(DEFUN Métrica_10 ()
    (SETQ K 26 EA 18.90 H 7 EC 17)
```

```
(Dibujo)
)

(DEFUN Métrica_12 ()
  (SETQ K 30 EA 18.10 H 8 EC 19)
  (Dibujo)
)

(DEFUN Dibujo (/ XPTO PTO YPTO PTO2 PTO3 PTO4 PTO5 PTO6 PTO7 PTO8 PTO9 PTO10
                PTO11 PTO12 PTO13 PTO14)
  (SETQ XPTO (+ (CAR PuntoIns) (/ EA 2)))
  (SETQ PTO (LIST XPTO (CADR PuntoIns)))
  (SETQ YPTO (+ (CADR PTO) H))
  (SETQ PTO2 (LIST (CAR PTO) YPTO))
  (SETQ PTO3 (LIST (CAR PuntoIns) (CADR PTO2)))
  (SETQ PTO4 (LIST (+ (CAR PuntoIns) (/ Métrica 2)) (CADR PuntoIns)))
  (SETQ PTO5 (LIST (CAR PTO4) (- (CADR PTO4) Longitud)))
  (SETQ PTO7 (LIST (CAR PTO4) (CADR PTO2)))
  (SETQ PTO8 (LIST (- (CAR PTO5) 1) (- (CADR PTO5) 1)))
  (SETQ PTO6 (LIST (- (CAR PTO8) (- (/ Métrica 2) 1)) (CADR PTO8)))
  (SETQ PTO9 (LIST (CAR PTO8) (+ (CADR PTO8) K)))
  (SETQ PTO10 (LIST (CAR PTO5) (CADR PTO9)))
  (SETQ PTO11 (LIST (CAR PTO8) (+ (CADR PTO8) 1)))
  (SETQ PTO12 (LIST (CAR PTO6) (CADR PTO5)))
  (SETQ PTO13 (LIST (CAR PTO12) (CADR PTO10)))
  (SETQ PTO14 (LIST (CAR PTO10) (+ (CADR PTO10) 1)))

  (COMMAND "_.pline" PuntoIns PTO PTO2 PTO3 PTO2 PTO PTO4 PTO7 PTO5 PTO8 PTO6
    PTO8 PTO11 PTO12 PTO5 PTO11 PTO8 PTO9 PTO10 PTO13 PTO9 PTO14 "")
  (COMMAND "_.mirror" "_l" "" PuntoIns PTO6 "")
)

(DEFUN C:Tornillo (/ PuntoIns ErrorLSP ErrorDCL Métrica Longitud K EA H EC)
  (SETVAR "cmdecho" 0)
  (GRAPHSCR)
  (SETQ PuntoIns nil)
  (COMMAND "_.undo" "_be")
  (SETQ ErrorLSP *error* *error* ControllSP)
  (Datos)
  (COMMAND "_.undo" "_e")
  (SETQ *error* ErrorLSP)
  (SETVAR "cmdecho" 1)
  (PRIN1)
)

(DEFUN ControlDCL ()
  (SETQ ErrorDCL nil)
  (IF PuntoIns ()
    (PROGN
      (SETQ ErrorDCL T)
      (SET_TILE "error" "Faltan coordenadas del punto de inserción.")
      (MODE_TILE "tpoints" 2)
    )
  )
)

(DEFUN ControllSP (Mensaje)
  (SETQ *error* ErrorLSP)
  (PRINC Mensaje)(TERPRI)
  (COMMAND "_.undo" "_e")
  (SETVAR "cmdecho" 1)
)
```

```
(PRIN1)  
)
```

Como vemos, el programa carga el cuadro de diálogo (tras las típicas operaciones), así como la foto (que evidentemente deberemos tener). Después inicializa las listas y las rellena y rellena los demás elementos con valores por defecto.

Es interesante ver cómo maneja el cuadro las salidas con `DONE_DIALOG` y `START_DIALOG`, sobre todo en el botón de designar un punto de inserción, que sale del letrero y vuela a entrar (conservando valores). El truco está en asignar a una variable la devolución de `START_DIALOG` (en este caso `SD`) y controlar la pulsación de cada botón mediante dicha variable (asignando un número de salida a cada `DONE_DIALOG`). Evidentemente, al volver a entrar en el cuadro tras designar un punto de inserción, la variable ha de hacerse `nil` (véase).

Otro tema interesante es el control de la carga o no del cuadro en memoria. Al volver a entrar en el letrero después de designar un punto de inserción, el cuadro no ha de cargarse en memoria de nuevo —ya está cargado—, sino simplemente mostrarse. Esto se controla averiguando si existe ya o no un punto de inserción.

De la misma manera se controlan y se escriben las coordenadas de dicho punto de inserción encima del botón para designar.

La manera de asignar valores de métrica y longitud se realiza recurriendo a la función `NTH`, vista en esta sección. Con ella se extrae cada valor de cada lista y, previa conversión a valor numérico entero, se asigna a la variable correspondiente. De esta forma tendremos la posibilidad de variar el ámbito de valores de rango del programa simplemente cambiando unas cuantas líneas.

Lo restante dice relación al propio dibujo del tornillo, así como al control de errores típico, tanto de DCL como del programa AutoLISP en sí, y a funciones de “embellecimiento” del programa. Entre esta últimas existe una que aún no hemos estudiado, es `GRAPHSCR`. Se ha incluido porque la veremos enseguida, en la siguiente sección, con otras análogas.

15ª fase intermedia de ejercicios

- Indicar el resultado de AutoLISP ante las siguientes proposiciones:

```
— (ASSOC 'x '((t y u) (r s e) (x g f) (12 34)))  
— (CONS (CADR '(14 56 45)) (CDR '(34 56 78 65.6)))  
— (CONS (NTH 1 '(vivo resido)) '(aquí mismo))  
— (CONS 'tipo 32)  
— (SUBST '5 '78 '(45 32 78 2 4))  
— (APPEND '(12 er 45 fg) '(dr fr 54 3.45) '((12 34) (df fr)))  
— (LENGTH '(23 hg 78 (ty gh) 89 (ju hg gft 89) ed 34.56 t))  
— (LAST '(yh yh hy yh))  
— (LAST '(12.3 78.87 (23.45 54.43)))  
— (MEMBER 't1 '(n 1 (t1 t2) 1))  
— (NTH 4 '(c1 cd c4 cf c5 g6 hy7 (fr 45)))  
— (NTH 3 '(12 34.43 56))  
— (REVERSE '(mismo aquí resido))  
— (ACAD_STRLSORT '("as" "aw" "h" "perro" "perra" "12" "01" "02"))
```

ONCE.18. MISCELÁNEA DE FUNCIONES ÚTILES

Se ha querido incluir esta sección aquí por aquello de que vamos a estudiar una serie de funciones que, sin bien no pueden englobarse en un grupo general, son muy útiles a la hora desarrollar programas mínimamente presentables. Es tiempo ahora, una vez aprendida la base —y algo más— de la programación en AutoLISP, de que comencemos a hurgar un poco más profundo en el acceso a las características de **AutoCAD**.

Con estas nuevas funciones podremos forzar la aparición de la ventana de texto de **AutoCAD**, visualizar la versión numérica e idiomática del programa y etcétera. Comenzamos.

ONCE.18.1. Asegurándonos de ciertos datos

`(GRAPHSCR)`

Esta función no tiene argumentos y su utilidad es conmutar a pantalla gráfica. Tiene el mismo efecto que pulsar **F2** mientras se está en el modo de pantalla de texto. Si ya estamos en pantalla gráfica, simplemente se queda como está.

Se utiliza para asegurarnos de que **AutoCAD** se encuentra en pantalla gráfica cuando es preciso procesar objetos de dibujo. Como hemos visto la hemos utilizado en el último ejemplo, aunque en ese caso no haría falta, pues al arrancar un cuadro de diálogo siempre se conmuta a pantalla gráfica automáticamente.

En configuraciones con dos pantallas, una gráfica y otra de texto, esta función no tiene efecto.

`(TEXTSCR)`

Esta función no tiene argumentos y su utilidad es conmutar a pantalla de texto. Tiene el mismo efecto que pulsar **F2** mientras se está en el modo de pantalla gráfica. Si ya estamos en pantalla de texto, simplemente se queda como está.

Se utiliza para asegurarnos de que **AutoCAD** se encuentra en pantalla de texto cuando es preciso mostrar listados de capas, objetos, propiedades, etc. De esta forma, por ejemplo, para mostrar una lista de todos los objetos designados deberíamos utilizarla, ya que con una configuración de dos o tres líneas en línea de comandos (que viene a ser lo típico) no se vería nada. Y así, además, evitamos que el usuario tenga que pulsar **F2** al aparecer la lista. Algunos comandos de **AutoCAD** (**LIST** por ejemplo) así lo hacen.

En configuraciones con dos pantallas, una gráfica y otra de texto, esta función no tiene efecto.

`(TEXTPAGE)`

TEXTPAGE conmuta a pantalla de texto, de manera similar a **TEXTSCR**. La diferencia radica en que **TEXTPAGE** efectúa además un borrado o limpiado de pantalla, es decir, el cursor se sitúa al principio de la pantalla de texto.

Esta función resulta útil cuando se escriben listados y no se desea el efecto de *scroll* o persiana en la pantalla.

`(VER)`

`VER` es una función de AutoLISP sin argumentos y meramente informativa. Devuelve una cadena de texto que contiene el número de versión actual y dos letras entre paréntesis que indican el idioma de la misma, además de un texto fijo.

Por ejemplo, en una versión en castellano de **AutoCAD**, `VER` devolvería:

```
"AutoLISP Versión 14.0 (es)"
```

y en una versión inglesa devolvería:

```
"AutoLISP Release 14.0 (en)"
```

`VER` se suele utilizar para comparar la compatibilidad entre programas. Por ejemplo, imaginemos que hemos diseñado un programa en AutoLISP 14 que utiliza funciones inherentes que no existían en la versión 13. Lo primero que podría hacer el programa es comprobar el texto que muestra la función `VER` para obtener el número de versión. Si no es la 14 mostraría un mensaje de error y acabaría.

Otra utilidad sería capturar el idioma y, dependiendo de que sea uno u otro, escribir determinados textos en dicho idioma.

ONCE.18.2. Acceso a pantalla gráfica

`(GRCLEAR)`

NOTA: Esta función se encuentra obsoleta en la versión 14 de **AutoCAD**, esto es, no realiza su función. Aún así, se mantiene por compatibilidad con programas escritos para versiones anteriores. En **AutoCAD** simplemente devuelve `nil`, sin efecto alguno aparente. En versiones anteriores realizaba la función que aquí se expone a continuación.

`GRCLEAR` despeja la pantalla gráfica actual. El efecto es el mismo que utilizar, por ejemplo, el comando `MIRAFOTO` de **AutoCAD** para mostrar una foto sin objeto alguno, es decir, "en blanco". Se puede volver luego a la situación actual con un simple redibujado.

Esta función sólo afecta al área gráfica, no a la línea de estado, línea de comandos o área de menú de pantalla.

`(GRDRAW inicio fin color [resaltado])`

Esta función dibuja un vector virtual en el área gráfica de la pantalla, entre los dos puntos indicados en *inicio* y *fin*. Los puntos, como siempre, son listas de dos o tres números reales; las coordenadas se refieren al SCP actual del dibujo.

El vector se visualiza con el color especificado —número entero— por el tercer argumento (*color*). Si se indica `-1` como número de color, el vector se visualiza como *tinta XOR* ("O exclusivo"), es decir, se complementa al dibujarse por encima de algo y se suprime si se dibuja algo nuevo por encima de él.

Si se especifica un argumento *resaltado* diferente a `nil`, el vector se visualiza destacado como vídeo inverso, brillo, línea de trazos u otro sistema.

Los vectores dibujados con esta función son virtuales —como ya se ha dicho— es decir, no forman parte del dibujo y desaparecerán con un redibujado o una regeneración. Ejemplo:

```
(GRDRAW '(50 50) '(200 200) 1 T)
```

`(GRVECS lista_vectores [transformación])`

Permite dibujar en pantalla una serie de vectores virtuales. El primer argumento *lista_vectores* es una lista con el color para los vectores y sus coordenadas inicial y final. El ejemplo siguiente dibuja un cuadrado de 10 × 10 con cada línea de un color diferente:

```
(GRVECS '( 2 (10 10) (20 20)
           4 (20 10) (20 20)
           1 (20 20) (10 20)
           3 (10 20) (10 10)
           )
)
```

El primer color especificado (2 en el ejemplo) se aplicará a los vectores siguientes dentro de la lista hasta que se indique otro color. Si el valor del color es mayor de 255, se dibujarán en *tinta XOR*; si es menor de 0, el vector quedará resaltado según el dispositivo de visualización, normalmente en línea discontinua.

El segundo argumento, optativo, es una matriz de transformación que aplicada a la lista de vectores, permite cambiar el emplazamiento y la escala de los vectores que se generarán. Por ejemplo:

```
('( (2.0 0.0 0.0 30.0)
    (0.0 2.0 0.0 45.0)
    (0.0 0.0 2.0 0.0)
    (0.0 0.0 0.0 1.0)
)
```

Esta matriz, aplicada al ejemplo anterior, dibujaría un cuadrado al doble de su tamaño y a 30 unidades de desplazamiento en X y 45 en Y.

`(GRTEXT [rectángulo texto [resaltado]])`

Esta función se utiliza para escribir textos virtuales en las áreas de texto de la pantalla gráfica de **AutoCAD**. Según en cuál de las tres áreas de texto se quiera escribir, se utilizará la función de manera diferente.

Área del menú de pantalla.

Se indica el número de casilla del área de menú. Este número debe ser un entero positivo ó 0. Las casillas se numeran de arriba abajo empezando por el 0, hasta el número máximo de líneas permitidas por la interfaz gráfica. Por ejemplo una tarjeta gráfica VGA permite hasta 26 líneas en esta área del menú; por lo tanto las casillas se numeran de 0 a 25.

Una vez indicado el número de casilla, se especifica el texto (entre comillas) que se desea visualizar en esa casilla. El texto se truncará si no cabe entero en la casilla, o se completará con blancos en el caso de que sobren caracteres.

Si se indica el argumento *resaltado* (debe ser un número entero) y su valor es diferente de 0, el texto se pondrá de relieve en la casilla correspondiente. Si su valor es 0, el texto vuelve a su visualización normal. Al indicar un argumento de resaltado, no cambia el texto de la casilla sino sólo su visualización. Por ejemplo:

```
(GRTEXT 8 "HOLA")  
(GRTEXT 8 "QUÉ TAL" 1)
```

La primera utilización de GRTEXT escribe el texto HOLA en la casilla 8. En la segunda utilización se produce un resaltado en esa casilla, pero el texto sigue siendo HOLA, no ha cambiado. Por eso hay que escribir primero el texto que se desee y después resaltarlo:

```
(GRTEXT 8 "HOLA")  
(GRTEXT 8 "HOLA" 1)
```

En este caso, para poner de relieve el texto de la casilla 8, se ha indicado el mismo texto que ya tiene escrito. Si se suministra un valor de texto diferente, se pueden producir comportamientos anómalos del programa en AutoLISP.

El texto escrito en la casilla indicada es virtual, no modifica la opción de menú contenida debajo. En cuanto se cambie de submenú o se produzca un redibujado del área de menú, desaparecerán los textos virtuales escritos con GRTEXT. Como el menú de pantalla suministrado por **AutoCAD** utiliza hasta 26 líneas, todo lo que se escriba más abajo con GRTEXT si hay sitio, permanecerá normalmente en pantalla. La variable de **AutoCAD** SCREENBOXES almacena el número de casillas disponibles.

Línea de estado (área de modos)

Para visualizar un texto en la línea de estado, en el área donde se escriben los modos activados, rejilla, forzado de cursor, etc., hay que especificar un número de casilla -1. La longitud máxima del texto depende de la tarjeta gráfica (generalmente se admiten más de 40 caracteres). El argumento de resaltado no tiene efecto. Normalmente el texto se situará a la izquierda del área de coordenadas. Cualquier actuación sobre los modos eliminará el texto.

```
(GRTEXT -1 "DISEÑO ASISTIDO POR ORDENADOR")
```

Línea de estado (área de coordenadas)

Para escribir el texto en la línea de estado, en la zona de visualización de coordenadas, hay que indicar un número de casilla -2. El argumento de resaltado no tiene efecto. Para que se vea el texto, debe estar desactivado el seguimiento de coordenadas. En cuanto se active o se actúe sobre los modos, se eliminará el texto.

```
(GRTEXT -2 "EJECUTANDO AutoLISP")
```

Por último, si se llama a GRTEXT sin argumentos, se restablecerán todas las áreas a su estado original, desapareciendo todos los textos virtuales que se hayan escrito.

ONCE.18.3. Lectura de dispositivos de entrada

<pre>(GRREAD [seguimiento] [claves [tipo_cursor]])</pre>

Esta función permite la lectura directa de dispositivos de entrada. Si se suministra el argumento *seguimiento* con un valor diferente de nil, se activa la lectura continua de dispositivos señaladores en movimiento. En este caso GRREAD acepta el primer valor del dispositivo, sin esperar a que se pulsen botones selectores.

El segundo argumento *claves* deberá ser un número entero con los siguientes valores posibles:

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

Clave	Significado
1	Devuelve las coordenadas en modo arrastre, según la posición del cursor.
Clave	Significado
2	Devuelve todos los valores de las teclas y no desplaza el cursor cuando se pulse una tecla de cursor.
4	Utiliza el valor del tercer argumento <i>tipo_cursor</i> .
8	Omite el mensaje de <i>error:console break</i> cuando se pulsa CTRL+C.

El tercer argumento *tipo_cursor* establece el tipo de cursor con tres valores posibles:

Tipo	Significado
0	Muestra el cursor en cruz habitual.
1	Omite el cursor.
2	Muestra en el cursor la mira de designación de objetos.

En todos los casos `GRREAD` devuelve una lista cuyo primer elemento es un código que indica el tipo de dato que viene a continuación. El segundo elemento es el dato concreto que se trate. Los códigos son los siguientes:

Primer elemento		Segundo elemento	
Valor	Tipo de entrada	Valor	Descripción
2	Entrada por teclado	<i>Varía</i>	Código de caracteres
3	Punto designado	Punto 3D	Coordenadas de punto
4	Opción de menú de pantalla/desplegable (desde dispositivo señalador)	0-999	Nº cuadro menú pantalla
		1001-1999	Nº cuadro menú POP1
		2001-3999	Nº cuadro menú POP2
		3001-3999	Nº cuadro menú POP3
		... hasta...	
		16001-16999	Nº cuadro menú POP16
5	Dispositivo señalador	Punto 3D	Coordenada modo arrastrar
6	menú BUTTONS	0-999	Nº botón menú BUTTONS1
		1001-1999	Nº botón menú BUTTONS2
		2000-2999	Nº botón menú BUTTONS3
		3000-3999	Nº botón menú BUTTONS4
7	Opción de menú TABLET1	0-32767	Nº de cuadro digitalizado
8	Opción de menú TABLET2	0-32767	Nº de cuadro digitalizado
9	Opción de menú TABLET3	0-32767	Nº de cuadro digitalizado
10	Opción de menú TABLET4	0-32767	Nº de cuadro digitalizado
11	Opción de menú AUX	0-999	Nº de botón menú AUX1
		1001-1999	Nº de botón menú AUX2
		2000-2999	Nº de botón menú AUX3
		3000-3999	Nº. de botón menú AUX4
12	Botón de puntero (sigue a un resultado tipo 6 o tipo 11)	Punto 3D	Coordenadas de punto

NOTA: Para poder utilizar convenientemente estas últimas funciones es necesario estudiar el acceso a la Base de Datos de **AutoCAD**, cosa que se verá en su momento oportuno. Será entonces cuando se proponga un ejemplo explicativo.

ONCE.18.4. Atribuir expresión a símbolo literal

`(SET símbolo_literal expresión)`

La función `SET` no es muy utilizada, por lo que se propone aquí ya que es interesante verla.

`SET` atribuye el valor de una expresión especificada a un literal de un símbolo. Este símbolo se considera sin evaluar. La diferencia de `SET` con `SETQ` es que aquí se atribuye o asocia el valor de la expresión al literal del símbolo, haciendo ambos equivalentes. Con `SETQ` se almacena, como sabemos ya, valores en símbolos de variables no literales. Veamos un ejemplo. Si hacemos primero:

```
(SET 'x 'a)
```

y luego:

```
(SET x 25)
```

atribuimos al símbolo `x` el valor del símbolo `a`. Si extraemos el valor asociado a `x`:

```
!x
```

AutoLISP devolverá:

```
A
```

Si extraemos ahora el valor de `a`:

```
!a
```

AutoLISP devolverá:

```
25
```

16ª fase intermedia de ejercicios

- Responder a las siguientes preguntas:
 - ¿Qué función AutoLISP nos permite cambiar a pantalla gráfica?
 - ¿Y a pantalla de texto?
 - ¿Para qué se utilizan ambas?
 - ¿Qué devolvería la función `VER` en **AutoCAD** versión americana?
 - ¿Qué diferencia hay entre `SET` y `SETQ`?

ONCE.19. ACCESO A OTRAS CARACTERÍSTICAS

En su momento estudiamos el acceso a variables de **AutoCAD**, a comandos externos, a aplicaciones, etcétera. Así mismo, acabamos de ver el acceso a la pantalla gráfica. Lo que en esta sección se tratará es el acceso a otras características del programa, como pueden ser los menús desplegables, la tableta digitalizadora, los modos de referencia a objetos o los archivos de ayuda (visto esto último por encima en el **MÓDULO SEIS**, sobre creación de archivos de ayuda).

ONCE.19.1. Modos de referencia

Empezaremos por los modos de referencia, aplicación que nos permitirá manejar esta característica de **AutoCAD** en determinados momentos.

```
(OSNAP punto modo)
```

Esta función aplica el modo o modos de referencia indicados al punto especificado. OSNAP devuelve un punto como resultado. El modo será una cadena de texto, por lo que habrá de ir entre comillas. Si se indican varios modos, estos irán separados por comas. Un ejemplo:

```
(SETQ PuntoMedio (OSNAP '(10 10 0) "med"))
```

Esto equivaldría a haber seleccionado un modo de referencia *Punto medio* y haber señalado con el cursor en el punto indicado. Como sabemos, dependiendo del valor del punto de mira de los modos de referencia (variable APERTURE), si se encuentra un objeto dentro de dicha mirilla, su punto medio quedaría almacenado en la variable PuntoMedio. En caso de no encontrar ningún objeto o no existir un punto medio, devuelve nil.

NOTA: No confundir el valor APERTURE de la mirilla (cruceta) de modos de referencia a objetos, con el valor PICKBOX de la mira (cuadrado) de designación de objetos. La mira de referencia actúa de la siguiente forma: cuando tenemos un modo de referencia a objetos activado, por ejemplo el *Punto medio*, al pasar cerca de objetos, la mira captura o se “engancha” a sus puntos medios (aparece el marcador triangular si la característica *AutoSnap* está activada). Lo “cerca” que ha de pasar para que esto ocurra es precisamente el tamaño de dicha mirilla.

Se observa entonces que el hecho de que se encuentre o no el punto buscado depende en gran medida del valor actual de la variable APERTURE. Un valor demasiado pequeño dificulta la operación; un valor demasiado grande puede atrapar otro punto próximo que no interese.

La técnica mejor que impone la costumbre no consiste en ampliar o reducir el valor de APERTURE en tiempo de ejecución, sino el asegurar —siempre que se pueda— puntos reales de los objetos. Por ejemplo, si dibujamos una línea y queremos guardar en una variable su punto medio, porque luego nos interesa, lo lógico es proporcionarle a OSNAP un punto conocido de la línea:

```
(COMMAND "_line" '(0 0) '(100 100) "")
```

Así dibujamos la línea. Ahora guardaremos su punto medio en la variable PtoMed:

```
(SETQ PtoMed (OSNAP '(100 100) "_mid"))
```

De esta manera podremos dibujar ahora, por ejemplo, un círculo con centro en el punto medio de la línea:

```
(COMMAND "_circle" PtoMed 25)
```

Al indicar un punto de la línea, APERTURE siempre lo va a englobar, por muy baja que esté definida, ya que es un punto exacto coincidente pues con el cruce de los ejes del cursor.

NOTA: Se observa que los modos de referencia son cadenas que llaman a los modos incluidos en **AutoCAD**, por lo que dependiendo de la versión idiomática del programa

habremos de introducirlos en un idioma o en otro. Existe la posibilidad, como vemos, de especificarlos con el guión bajo de subrayado para cualquier versión en cualquier idioma.

Como hemos dicho, si queremos aplicar más de un modo de referencia a la función OSNAP, hemos de especificarlos entre comas:

```
(SETP PIn (OSNAP Ptol "med,int,fin"))
```

NOTA: Los modos pueden introducirse con su nombre completo o con su abreviatura por convenio. Así, valdría igual *medio* que *med*.

La variable que controla los modos de referencia actualmente activados en **AutoCAD** es OSMODE. Esta variable ya la hemos utilizado en algún programa, accediendo a ella para guardar su valor, estableciendo otro y recuperando el valor inicial al terminar el programa.

OSMODE representa los modos almacenados como fijos con los comandos REFENT (OSNAP en inglés) o DDOSNAP. Estos modos se pueden cambiar con REFENT (OSNAP), que abre el mismo cuadro de diálogo que DDOSNAP o, desde la línea de comandos, con -REFENT (-OSNAP en versiones inglesas), introduciendo el guión (-) para ello. Desde línea de comandos se introducen como con la función OSNAP de AutoLISP, con términos literales (aquí sin comillas) y separados por comas si hay más de uno.

Pero lo más lógico de un programa en AutoLISP es acceder a la variable OSMODE y cambiar su valor, para así variar los modos establecidos como fijos. Los valores posibles para OSMODE son los que siguen:

Valor OSMODE	Modo castellano	Modo inglés	Descripción
0	NIN	NON	<i>Ninguno</i>
1	FIN	END/ENDP	<i>Punto final</i>
2	MED	MID	<i>Punto medio</i>
4	CEN	CEN	<i>Centro</i>
8	PTO	NOD	<i>Punto</i>
16	CUA	QUA	<i>Cuadrante</i>
32	INT	INT	<i>Intersección</i>
64	INS	INS	<i>Inserción</i>
128	PER	PER	<i>Perpendicular</i>
256	TAN	TAN	<i>Tangente</i>
512	CER	NEA	<i>Cercano</i>
1024	RAP	QUICK	<i>Rápido</i>
1028	FIC	APPINT	<i>Intersección ficticia</i>

Hay que tener en cuenta que el modo *Rápido* no puede establecerse aisladamente sino en combinación con algún otro modo de referencia, evidentemente.

Los modos con OSMODE se establecen sumando valores; así *Punto medio*, *Inserción*, *Tangente* y *Cuadrante* activados, darían un valor para la variable de 338 (2 + 64 + 256 + 16).

Tengamos también en cuenta que en sucesivas llamadas a OSMODE (que haremos con GETVAR y SETVAR evidentemente) los modos no se acumulan. Así si una vez le hemos dado un valor de 12 para activar *Punto* y *Centro*, si posteriormente le damos 1 para activar *Punto final*, los dos anteriores se eliminarán y sólo quedará activado el último. Para anular todos ellos, utilizaremos el modo *Ninguno* (valor 0 para OSMODE).

Las siguientes líneas suelen ser típicas en muchos programas. Al iniciar:

```
...  
(SETQ Modos (GETVAR "osmode"))  
(SETVAR "osmode" 1)  
...
```

Se guarda en una variable la actual configuración de modos de referencia y se especifica la que nos interesa. Y al acabar el programa (sea naturalmente o en rutina de control de errores):

```
...  
(SETVAR "osmode" Modos)  
...
```

Para restituir la configuración primitiva del usuario y éste no aprecie nada.

Veamos ahora un método que tenemos —bastante completo— de redibujado de objetos.

ONCE.19.2. El redibujado

```
(REDRAW [nombre_entidad [modo]])
```

La función REDRAW efectúa un redibujado total —al igual que el comando de **AutoCAD**— de toda de ventana gráfica actual si se suministra sin argumento alguno:

```
(REDRAW)
```

Si se indica un nombre de entidad solamente se redibujará esa entidad. Para especificar dicho nombre habremos de extraerlo de la lista de definición de la entidad en Base de Datos, cosa que se verá más adelante.

Si se ha despejado la ventana gráfica con la función GRCLEAR (ya estudiada en la sección **ONCE.18.2.**), con REDRAW se pueden redibujar las entidades que se deseen y sólo ellas se harán visibles en pantalla. Por ejemplo, si en un dibujo de circuitería electrónica se desea averiguar cuántos bloques de un componente determinado se encuentran insertados, se extrae de la Base de Datos del dibujo todas las inserciones de bloque (se verá), se despeja la pantalla de GRCLEAR y se redibujan con REDRAW sólo las entidades capturadas.

Si además de un nombre se suministra el argumento *modo*, se puede controlar la manera en que se hace el redibujado, de acuerdo a la tabla siguiente:

Modo	Efecto
1	Redibuja la entidad en pantalla.
2	Elimina la entidad de la pantalla, es decir, la oculta. Reaparece con 1.
3	Visualiza la entidad en relieve (vídeo inverso o doble intensidad).
4	Suprime la visualización en relieve.

Si el nombre de la entidad indicado se corresponde con un objeto compuesto (bloque con atributo, polilínea no optimizada...), el efecto de REDRAW se extiende a sus componentes simples. Sin embargo, si el código de *modo* se indica con signo negativo (-1, -2, -3 ó -4), el efecto de REDRAW sólo afectará a la entidad principal.

Esta función es muy utilizada en el manejo de la Base de Datos del programa. Veremos ahora `TRANS`, que también se usa mucho en dicho menester.

ONCE.19.3. Transformación entre Sistemas de Coordenadas

`(TRANS punto sistema_origen sistema_destino [desplazamiento])`

Esta función convierte un punto o un vector de desplazamiento desde un Sistema de Coordenadas a otro. El valor del punto o del desplazamiento se indica como una lista de tres números reales. Si se indica el argumento *desplazamiento* y su valor es diferente de `nil`, entonces la lista de tres números reales se considera un vector de desplazamiento.

Los argumentos para los Sistemas de Coordenadas se pueden especificar de tres maneras diferentes. La primera es mediante un código especificativo; los códigos son:

Código	Sistema de Coordenadas
0	Sistema de Coordenadas Universal (SCU)
1	Actual Sistema de Coordenadas Personal (SCP)
2	Actual Sistema de Coordenadas de la Vista (SCV)
3	Sistema de Coordenadas del Espacio Papel (sólo en combinación con 2)

La segunda forma es mediante un nombre de entidad que indica el Sistema de Coordenadas de la Entidad (SCE) relativo —o Sistema de Coordenadas del Objeto (SCO), según convención—. Esto es equivalente a la opción *Objeto* del comando SCP de **AutoCAD**.

La tercera y última manera es con un vector de altura de objeto 3D, indicado como lista de tres números reales. Este vector expresa la orientación de la altura de objeto en el nuevo SCP con respecto al SCU. Este procedimiento no sirve cuando la entidad ha sido dibujada en el SCU (su SCE coincide con el SCU).

La función `TRANS` devuelve el punto o vector de desplazamiento como una lista de tres elementos expresada en el nuevo Sistema de Coordenadas indicado en *sistema_destino*. Por ejemplo, si el SCP actual se ha obtenido girando desde el SCU 90 grados sobre el eje Y:

```
(TRANS '(1 2 3) 0 1) devuelve (-3.0 2.0 1.0)
(TRANS '(-3 2 1) 1 0) devuelve (1.0 2.0 3.0)
```

En el primer caso, el punto `(1 2 3)` en el SCU (código 0) se expresa en el SCP actual (código 1) como `(-3 2 1)`. En el segundo caso se hace la operación inversa.

A la hora de introducir coordenadas o desplazamientos para utilizar comandos de **AutoCAD**, hay que tener muy presente que siempre se consideran respecto al SCP actual (salvo que vayan precedidas de asterisco). Por eso si se dispone de unas coordenadas calculadas en otro sistema, hay que pasarlas siempre al SCP actual mediante `TRANS`.

En la Base de Datos de **AutoCAD** los puntos característicos de cada entidad se encuentran expresados en el Sistema de coordenadas del Objeto (SCO). Es necesario siempre tener en cuenta cuál es el SCP actual y utilizar el comando `TRANS` para convertir esos puntos. Para ello, como ya se ha dicho, se indica el nombre de la entidad en vez de un código. Se verá.

El Sistema de Coordenadas de la Vista (SCV) es el sistema hacia el cual se convierten las imágenes antes de ser visualizadas en pantalla. Su origen es el centro de la pantalla y el eje Z la línea de visión (perpendicular hacia la pantalla). Es importante cuando se pretende controlar cómo van a visualizarse las entidades. Por ejemplo, si el usuario señala un punto y se desea averiguar a qué extremo de una línea existente se encuentra más próximo, se convierte el punto señalado desde el SCP actual al SCV de la forma:

```
(TRANS punto 1 2)
```

Después, para efectuar la comparación, hay que convertir cada uno de los puntos finales de la línea (extraídos de la Base de Datos tal como se explicará más adelante) al SCV también.

```
(TRANS punto_final1 nombre_de_línea 2)  
(TRANS punto_final2 nombre_de_línea 2)
```

Una vez hecho esto, ya se pueden calcular las distancias entre el punto del usuario y los dos puntos finales, medida en el SCV, para determinar cuál es la menor.

Si el punto o vector de desplazamiento indicado en TRANS es en 2D, la propia función lo convierte en 3D suministrando la coordenada Z que falta. Esta coordenada Z dependerá de cuál es el Sistema de Coordenadas desde el cual se considera el punto:

- Si es el Universal (SCU), la coordenada Z es 0.0.
- Si es el Personal actual (SCP), la coordenada Z es el valor de la elevación actual.
- Si es el Sistema de Coordenadas de Objeto (SCO), Z es 0.0.
- Si es el Sistema Coordenadas de la Vista (SCV) en Espacio Modelo o Papel, el valor de Z es la proyección del punto en el plano XY actual de acuerdo con la elevación actual.

ONCE.19.4. Ventanas y vistas

`(VPORTS)`

Devuelve una lista con la configuración de ventanas actual. La lista contiene en forma de sublistas los descriptores de todas las ventanas de la configuración actual. Cada descriptor es a su vez una lista con tres elementos: número de identificación de ventana (correspondiente a la variable de sistema de **AutoCAD** CVPOR_T), esquina inferior izquierda y esquina superior derecha de cada ventana.

Las dos esquinas aparecen en fracciones de anchura y altura de pantalla, igual que en el listado del comando VENTANAS opción ?. Así, (0 0) corresponde al vértice inferior izquierdo de la pantalla, (1 1) al superior derecho y (0.5 0.5) al centro de la pantalla.

Por ejemplo si la configuración actual en pantalla es de cuatro ventanas iguales de tamaño, VPORTS podría devolver:

```
((3 (0.5 0.5) (1.0 1.0))  
 (2 (0.5 0.0) (1.0 0.5))  
 (6 (0.0 0.5) (0.5 1.0))  
 (9 (0.0 0.0) (0.5 0.5))  
)
```

El primer número de identificación que aparece (en el ejemplo el 3) es el de la ventana activa actual.

Si TILEMODE tiene valor 0, la lista devuelta describirá las entidades de ventanas gráficas creadas con VMULT en el Espacio Papel. La ventana del Espacio Papel siempre tiene el número 1 y su tamaño se expresa en unidades del dibujo. Por ejemplo, podría devolver la lista:

```
(1 (0.0 0.0) (483.717 297.0))
```

```
(SETVIEW descriptor_vista [identificador_ventana])
```

Si existe una vista en el dibujo previamente almacenada, esta función restablece dicha vista en una ventana gráfica. Si se omite el segundo argumento, la vista se restablece en la ventana gráfica actual. Si se indica el identificador de la ventana, que es el valor almacenado en la variable de **AutoCAD** CVPOR, la vista se restablece en esa ventana.

El descriptor de la vista almacenada debe ser del tipo de la lista devuelta por TBLSEARCH, función que se estudiará al hablar del acceso a la Base de Datos de **AutoCAD**.

ONCE.19.5. Calibración del tablero digitalizador

```
(TABLET modo [fila1 fila2 fila3 dirección])
```

Se utiliza para almacenar, recuperar y crear calibraciones de tablero. Si el argumento *modo* es 0, se devuelve una lista con la calibración actual. Si es 1, se define la nueva calibración aportándola mediante tres puntos 3D que forman las tres filas de la matriz de transformación del tablero, y un vector 3D que define la dirección normal al plano formado por los tres puntos anteriores. La variable de **AutoCAD** TABMODE permite activar o desactivar el modo Tablero.

ONCE.19.6. Control de elementos de menú

```
(MENUCMD cadena)
```

Esta función controla la visualización de submenús del menú actual cargado por **AutoCAD**. Muestra, modifica o solicita un submenú, permitiendo desarrollar alguna acción en el mismo. De este modo se puede ejecutar un programa en AutoLISP asociándole un menú (ya sea de pantalla, tableta, desplegable, etc.) que podría contener por ejemplo opciones para seleccionar por el usuario desde el propio programa en AutoLISP.

MENUCMD devuelve siempre nil. La cadena de texto indicada (entre comillas) es de la forma:

```
identificador de menú = nombre de submenú  
área de menú = acción
```

En el primer caso, se especifica la inicial del identificador de sección de menú. Estas iniciales son las siguientes:

Iniciales	Menús
B1-B4	Menús de pulsadores 1 a 4
A1-A4	Menús auxiliares 1 a 4
P0	Menú de cursor
P1-P16	Menús desplegables 1 a 16
I	Menú de imágenes

S	Menú de pantalla
T1-T4	Menús de tablero 1 a 4
M	Expresiones DIESEL
<i>Gmenugroup.id</i>	Grupo de menú e identificador de menú

El nombre de submenú al que se llama tiene que existir en la sección de menú correspondiente (tiene que haber un identificador de submenú con el nombre en cuestión precedido por dos asteriscos). Por ejemplo:

```
(MENUCMD "S=REFENT" )
```

haría aparecer en el área de menú de pantalla el submenú REFENT.

```
(MENUCMD "P2=MALLAS" )
```

En este caso se llama a un submenú MALLAS en la sección de menú desplegable 2 (POP2). Este submenú debe existir en esa sección con su correspondiente identificador **MALLAS.

Existe una segunda forma de cadena de texto para MENUCMD. Consiste en especificar un área de menú, indicando el identificador de menú y el número de elemento que se ha de examinar o modificar, separados por un punto. Por ejemplo:

```
(SETQ Act (MENUCMD "P12.5=?" ))  
(IF (= Act "" )  
  (MENUCMD "P12.5=~" )  
)
```

La primera expresión examina el estado actual de la opción número 5 del desplegable P12. Si la opción contiene una señal de activación, se devolvería la misma. En caso de no contener dicha señal, devuelve cadena vacía. Si esto ocurre se utiliza de nuevo MENUCMD para asignar a la opción el valor ~ que hace el efecto de poner una señal de activación en el menú. Hay que tener en cuenta que esta señal de activación podría cambiar en diferentes plataformas (podría ser por ejemplo !), como ya debemos saber.

Veremos un par de ejemplos de esta función en la sección **ONCE.19.9.**

```
(MENUGROUP nombre_grupo)
```

Se especifica el nombre de un grupo de menús. Si el grupo está cargado, la función devuelve su nombre. Si no existe, devuelve nil.

ONCE.19.7. Letrero de selección de color

```
(ACAD_COLORDLG número_color [indicador])
```

Muestra en pantalla el cuadro de diálogo estándar para la selección de color. El primer argumento especifica el número de color que se ofrece por defecto. Hay que tener en cuenta que un valor 0 significa PorBloque y un valor 256 PorCapa. Si el segundo argumento *indicador* es nil, entonces se desactivan en el cuadro los botones PORCAPA y PORBLOQUE. Si no se especifica o es diferente de nil ambas casillas se encuentran disponibles. La función devuelve el número de color seleccionado mediante el cuadro.

ONCE.19.8. Funciones de manejo de ayuda

```
(ACAD_HELPDLG archivo_ayuda tema)
```

Esta es la función de ayuda en todas las plataformas. Se conserva únicamente por compatibilidad, pero ha sido sustituida totalmente por la siguiente.

```
(HELP [archivo_ayuda [tema [comando]]])
```

Esta función la vimos someramente al explicar la creación de archivos de ayuda en el **MÓDULO SEIS**. Ahora explicaremos todas sus características.

HELP llama a la utilidad de ayuda en todas las plataformas. El primer argumento es el nombre del archivo de ayuda con el que se va a trabajar. Si se indica uno de **AutoCAD** (extensión **.AHP**) se utiliza el lector de ayuda de **AutoCAD** para examinarlo. Si se indica un archivo de ayuda de Windows tipo *WinHelp* (extensión **.HLP**) se utiliza el programa Ayuda de Windows (**WINHLP32.EXE**) para mostrarlo, funcionando como la ayuda en entornos Windows. Si se indica una cadena vacía o se omite, se abre el archivo de ayuda por defecto de **AutoCAD**.

El segundo argumento *tema* especifica el tema cuya ayuda se muestra en primer lugar en la ventana del texto de ayuda. Si es una cadena vacía se mostrará la ventana inicial de la ayuda. El tercer argumento *comando* es una cadena de texto que especifica el estado inicial de la ventana de ayuda. Sus valores posibles son:

- **HELP_CONTENTS**: Muestra el primer tema del archivo de ayuda.
- **HELP_HELPONHELP**: Muestra la ayuda sobre la utilización de ayuda.
- **HELP_PARTIALKEY**: Muestra el diálogo de búsqueda utilizando *tema* como búsqueda inicial

Si no se producen errores, la función devuelve el nombre del archivo de ayuda. Si hay errores, devuelve **nil**.

```
(SETFUNHELP C:nombre_comando [archivo_ayuda [tema [comando]]])
```

Registra una comando para que se pueda utilizar con él la ayuda sensible al contexto. Normalmente el comando será nuevo, definido desde un programa en AutoLISP, por lo que hay que indicar los caracteres **C:**.

Cuando se crea un nuevo comando mediante **DEFUN**, si existe ya como comando registrado mediante **SETFUNHELP** se suprime del registro. Por eso **SETFUNHELP** sólo debe ser utilizada en el programa después de crear el comando nuevo con **DEFUN**.

Una vez registrado el nuevo comando, durante su utilización se podrá llamar a la ayuda transparente mediante **'?** o **'AYUDA** (**'HELP** en inglés) y mediante la tecla de función **F1**. Automáticamente se muestra el texto de ayuda disponible para el nuevo comando. Los argumentos *archivo_ayuda*, *tema* y *comando* tienen el mismo significado que para **HELP**.

Por ejemplo, se ha creado un nuevo comando llamado **MUESTRA** y se dispone de un archivo de ayuda **LISP.AHP** con texto de ayuda específica dentro del tema **MUESTRA**:

```
(DEFUN c:muestra ()  
...  
)
```

```
(SETFUNHELP "c:muestra" "lisp.ahp" "MUESTRA")
```

ONCE.19.9. Expresiones DIESEL en programas de AutoLISP

Como vimos en su momento, las expresiones en DIESEL que se referían a la personalización de la línea de estado se guardaban en una variable de sistema llamada MODEMACRO. Nada nos impide rellenar esta variable desde AutoLISP con la función SETVAR, o recoger su contenido con GETVAR.

Desde AutoLISP incluso lo tenemos más fácil y menos engorroso, porque no necesitamos escribir toda la expresión en una sola línea, ya que podemos dividirla en varias que queden concatenadas finalmente con la función STRCAT. Veamos un ejemplo:

```
(SETVAR "modemacro"
  (STRCAT
    "MI SISTEMA Capa: $(SUBSTR,$(GETVAR,clayer),1,8)"
    "$(IF,$(GETVAR,snapmode),"
    "ForzC X: $(RTOS,$(INDEX,0,$(GETVAR,snapunit)),2,0)"
    "Y: $(RTOS,$(INDEX,1,$(GETVAR,snapunit)),2,0))"
    "$(IF,$(GETVAR,osmode), REFENT)"
  )
)
```

NOTA: Como se explicó en su momento, el tamaño máximo de una cadena para MODEMACRO es de 255 caracteres, sin embargo desde AutoLISP se pueden incluir tamaños mayores, concatenando cadenas mediante STRCAT (que por cierto, únicamente admite 132 caracteres por cadena, según ya se dijo).

NOTA: Para que una línea de estado se muestre modificada permanentemente, podemos recurrir a archivos del tipo ACAD.LSP (sección **ONCE.15.1.**) donde incluir la rutina AutoLISP que acceda a MODEMACRO.

Las expresiones DIESEL en menús, por ejemplo, pueden ser combinadas con acciones desde programas AutoLISP, utilizando la función estudiada MENUCMD:

```
DEFUN c:ventanam ()
  (SETVAR "tilemode" 0)
  (COMMAND "espaciop")
  (SETQ ptb (GETPOINT "Vértice inferior izquierdo de ventana: "))
  (MENUCMD "p12=ventanam")
  (PROMPT "\nSeleccione tamaño desde el menú de pantalla.")
  (SETQ alto (GETREAL "Altura de ventana en fracción de
    pantalla:"))
  (SETQ propor (/ (CAR (GETVAR "screensize"))
    (CADR (GETVAR "screensize"))))
  )
  (SETQ ancho (* alto propor))
  (SETQ pt1 (POLAR ptb 0 ancho))
  (SETQ pt2 (POLAR pt1 (/ PI 2) alto))
  (COMMAND "vmult" ptb pt2)(PRIN1)
)
```

La función define un nuevo comando, desactiva TILEMODE y llama a la orden ESPACIOP. Solicita señalar en pantalla un punto de base ptb. A continuación llama a un menú

desplegable POP12 con las opciones de tamaño, mediante la función `MENUCMD`. Se visualiza un mensaje y a continuación se solicita la altura de la ventana en fracciones de pantalla, esperando a que el usuario señale una de las opciones. Este submenú de opciones de tamaño, dentro de la sección de menú `***POP12`, podría ser:

```
**ventanam
[Tamaños]
[ 1]$M=$(getvar,viewsize)
[ 3/4]$M=$(*,$(getvar,viewsize),0.75)
[ 5/8]$M=$(*,$(getvar,viewsize),0.625)
[ 1/2]$M=$(*,$(getvar,viewsize),0.5)
[ 3/8]$M=$(*,$(getvar,viewsize),0.375)
[ 1/4]$M=$(*,$(getvar,viewsize),0.25)
```

Al seleccionar una opción, la expresión `DIESEL` obtiene la altura actual de la ventana del Espacio Papel de la variable `VIEWSIZE`. Según la fracción seleccionada, multiplica dicha altura por la fracción. El valor resultante se admite como respuesta a `GETREAL`, que lo almacena en la variable `alto`. A continuación calcula la proporción entre anchura y altura, dividiendo los dos valores almacenados en `SCREENSIZE`. La anchura de la ventana que se desea abrir será el producto de la altura seleccionada por el usuario y el factor de proporción. De esta manera la ventana obtenida guardará la misma relación ancho/alto que la pantalla. Por último, el programa calcula el segundo vértice de la ventana `pt2` y llama al comando `VMULT` para abrir la ventana. Rizar el rizo.

También disponemos de la posibilidad de utilizar expresiones `DIESEL` como tales en el propio programa de AutoLISP. Esto lo haremos llamando a la función `MENUCMD` con la inicial de identificador `M` para lenguaje `DIESEL` (parecido a lo que hacíamos para los menús con `DIESEL`).

Por ejemplo, un nuevo comando de **AutoCAD** llamado `FECHA` para obtener la fecha y hora del sistema en un formato completo podría ser:

```
(DEFUN c:fecha ()
  (SETQ fecha
    (MENUCMD "M=$(edtime,$(getvar,date),DDDD D MON YY - H:MMam/pm)" )
  )
)
```

La expresión `DIESEL` lee la variable `DATE` y le aplica el formato de fecha especificado mediante `EDTIME`, así al introducir el nuevo comando `FECHA` en **AutoCAD** se podría devolver:

Sábado 1 Ago 98 - 4:02pm

Se puede utilizar este mecanismo para introducir expresiones `DIESEL` desde la línea de comando de **AutoCAD** y observar los resultados que devuelven. El programa en AutoLISP para conseguirlo sería:

```
(DEFUN C:Diesel ()
  (WHILE (/= Expr "M=")
    (SETQ Expr (STRCAT "M=" (GETSTRING T "\nExpresión DIESEL: ")))
    (PRINC (MENUCMD Expr))
  )(PRIN1)
)
```

Al llamar al nuevo comando `DIESEL`, se visualiza una solicitud de expresión. Cuando el usuario la introduce, `GETSTRING` la acepta como una cadena de texto, `STRCAT` le añade por delante `M=` y la variable `Expr` almacena el resultado. A continuación, `MENUCMD` llama a esa cadena con la expresión `DIESEL` y devuelve su resultado. `PRINC` lo escribe en pantalla.

NOTA: Recordemos la función de la variable `MACROTRACE` (véase **MÓDULO NUEVE**).

NOTA: Las variables de **AutoCAD** `USERS1`, `USERS2`, `USERS3`, `USERS4` y `USERS5` pueden ser utilizadas para traspasar información de una rutina AutoLISP a una expresión DIESEL. Véanse en el **MÓDULO NUEVE** y en el **APÉNDICE B**.

ONCE.19.10. Macros AutoLISP en menús y botones

Es totalmente factible la inclusión de funciones AutoLISP en las definiciones de opciones en archivos de menús, del tipo:

```
[Dibujar &Línea](command "_ .line" "0,0" "10,10" "")
```

o más complejas, y también en macros de botones de barras de herramientas. E inclusive llamadas a programas AutoLISP, escribiendo el nombre de la función definida o del nuevo comando.

Piénsese que, en última instancia, lo que se ejecuta al hacer clic en una opción de menú o en un botón, es lo mismo que podríamos escribir en la línea de comandos de **AutoCAD**.

ONCE.19.11. Macros AutoLISP en archivos de guión

Por último, decir que también es posible escribir instrucciones AutoLISP (o llamadas a programas) dentro de un archivo de guión o *script*. Estos archivos como sabemos, ejecutan por lotes las líneas incluidas en ellos como si de su escritura en la línea de comando se tratara.

ONCE.19.12. Variables de entorno

```
(GETENV nombre_variable)
```

Devuelve una cadena de texto (entre comillas) que es el valor atribuido a la variable de entorno indicada. La variable en cuestión habremos de indicarla también como cadena, por ejemplo:

```
(GETENV "acadcfg")
```

Este ejemplo podría devolver `"C:\\misdoc~1\\autocad"`, que es el directorio donde se guarda el fichero de configuración `ACAD14.CFG`. Esto puede resultar útil para saber dónde se encuentra y acceder a él para leer datos de la configuración de **AutoCAD** en el equipo donde se ejecuta la función y actuar en consecuencia. La manera de acceder a archivos directamente desde AutoLISP se estudia más adelante en este mismo **MÓDULO**, en la sección **ONCE.21**, concretamente.

17ª fase intermedia de ejercicios

- Realizar un programa que permita insertar un bloque en medio de una línea ya dibujada, partiendo ésta automáticamente para permitir la inserción. La única condición impuesta al usuario será que el bloque sea unitario en la dirección de la línea, para no complicar mucho el cálculo de los puntos entre los cuales se abrirá hueco.

ONCE.20. ACCESO A LA BASE DE DATOS DE AutoCAD

Si lo que hemos programado hasta ahora nos parecía importante, donde de verdad se aprovecha al máximo la potencial real de AutoLISP es en el acceso directo a la Base de Datos de **AutoCAD**. Hasta ahora únicamente habíamos producido objetos de dibujo —además del tratamiento de otros temas—; con los conocimientos que adquiriremos en esta sección **ONCE.20** tendremos la posibilidad de acceder a lo que ya está dibujado para editarlo, eliminarlo, copiarlo, moverlo, y un sinfín de acciones que sólo tienen final en nuestra capacidad para discurrir y en nuestra imaginación y perspicacia como programadores avanzados.

ONCE.20.1. Organización de la Base de Datos

ONCE.20.1.1. Introducción

NOTA: Si con la integración de **AutoCAD** en Windows (desde la versión 12) lo que siempre se había denominado entidad pasó a llamarse objeto, en relación a la Base de Datos interna seguimos refiriéndonos a entidades de dibujo. Cuestión de convenciones.

Todas las entidades de dibujo de **AutoCAD** se encuentran definidas en su Base de Datos interna por medio de una serie de códigos encerrados en listas. Como ya hemos comentado alguna vez, cuando **AutoCAD** guarda un dibujo, éste no se guarda como tal. Por ejemplo, un círculo no se guarda como el objeto de dibujo que es, sino como una serie de códigos que describen que es un círculo, las coordenadas de su centro y su radio, además de su capa, color, tipo de línea, etcétera. Al volver a abrir el dibujo, **AutoCAD** interpreta dichos códigos y representa en pantalla los objetos correspondientes basándose en ellos —en los códigos—.

Este tipo de almacenamiento, que podríamos denominar vectorial, es mucho más eficaz para nosotros que, por ejemplo un mapa de bits, ya que podemos acceder a la Base de Datos y modificar los parámetros correspondientes para alterar la geometría; y todo ello sin tocar el círculo en cuestión para nada, por ejemplo.

En el caso del almacenamiento como mapa de bits, además, nos percatamos de que al acercarnos cada vez más al dibujo, éste pierde su definición. Sin embargo, en **AutoCAD** no ocurre esto, ya que el programa tiene la posibilidad de recalcular la Base de Datos y volver a representar el círculo como tal. Es por ello, que si actuamos sobre una geometría con el comando **ZOOM** (en cualquiera de sus modalidades) ésta no pierde definición. A veces, es necesario provocarle a **AutoCAD** para que recalcule toda la geometría; es lo que se consigue con el comando **REGEN**.

Esta posibilidad de la que disponemos, de acceder directamente a la Base de Datos interna, proporciona una potencia extra a AutoLISP, ya que podremos diseñar programas que modifiquen directamente los objetos de un dibujo en tiempo real y sin la intervención del usuario.

ONCE.20.1.2. Estructura para entidades simples

Como decíamos, las entidades se representan en la Base de Datos mediante un conjunto de listas, una para cada entidad. Además existen otras listas correspondientes a tablas de símbolos y diccionarios, las cuales representan objetos no gráficos como capas, tipos

de línea y demás. Por último, existen también definiciones propias para los bloques. Nos centraremos por ahora en las entidades gráficas simples.

La lista de cada entidad es una lista de pares punteados (ya estudiados) o sublistas normales. Los pares punteados, como sabemos, contienen dos valores: el primero es el código que indica el tipo de dato contenido en el segundo, y éste contiene el dato concreto (coordenadas de un punto, nombre, etc.). Así por ejemplo, el siguiente par punteado contiene la definición para una entidad que dice que está en la capa 0:

```
(8 . "0")
```

8 es el código que define la propiedad de capa y "0" el nombre de la capa en sí.

Según el tipo de dato almacenado, la sublista puede no ser un par punteado y contener dos o más elementos. Por ejemplo, cuando se trata de un punto, contiene cuatro elementos: el código que indica que es un punto y las coordenadas X, Y y Z de dicho punto. En el siguiente ejemplo se muestra la lista que contiene las coordenadas del punto inicial de una línea:

```
(10 0.0 10.0 25.0)
```

10 es el código para el punto inicial en el caso de las líneas, 0.0 es la coordenada X del punto en cuestión, 10.0 es la coordenada Y y 25.0 la coordenada Z.

Así pues, y visto hasta aquí, podemos explicar un ejemplo mayor que se corresponde con la lista (casi) completa de una línea más o menos ordenada. La lista podría ser la siguiente:

```
((-1 . <Nombre de objeto: bd75a0>)  
 (0 . "LINE")  
 (8 . "PIEZA")  
 (62 . 1)  
 (6 . "TRAZOS")  
 (10 0.0 10.0 25.0)  
 (11 10.0 100.0 25.0)  
)
```

Se trata pues de una lista con siete sublistas incluidas. La explicación de cada una de ellas es la que sigue.

(-1 . <Nombre de objeto: bd75a0>) es la lista que define el nombre de la entidad; -1 es el código específico para este nombre. Esta lista siempre se representa de la misma forma: es un par punteado que como primer elemento tiene el código de nombre de entidad (-1) y, como segundo elemento, el texto fijo *Nombre de objeto:*, más un nombre en hexadecimal para la entidad, encerrado todo este último elemento entre corchetes angulares.

NOTA: En versiones inglesas de **AutoCAD**, *Nombre de objeto:* cambia por *Entity name:*.

Los nombres que se asignan a cada entidad de dibujo son, en realidad, posiciones de memoria —por eso están en hexadecimal—. Es por ello que un mismo objeto de dibujo siempre va a tener un nombre identificativo y único, en una misma sesión de dibujo, que lo diferencie de todos los demás. En el momento en que cerremos el dibujo y lo volvamos a abrir, es posible (y casi seguro) que el objeto adquiriera otro nombre, pero será también único para él en esa sesión de dibujo. Debido a esto, esta propiedad de los objetos no puede ser modificada.

(0 . "LINE") es la lista para el tipo de entidad; 0 es el código que define este tipo. El segundo elemento de la lista, que es par punteado, indica el tipo de entidad. Hay que tener en cuenta que este tipo siempre se representa en la Base de Datos en inglés, por lo que cuando se haga referencia a él desde los programas en AutoLISP habrá que hacerlo en este idioma. Además, se trata de una cadena de texto, y por eso va entre comillas (y en mayúsculas).

(8 . "PIEZA") indica la capa en la cual se encuentra actualmente la línea; 8 es el código para la capa. El segundo elemento de la lista, que es par punteado, describe, como cadena de texto y en mayúsculas, el nombre de la capa en cuestión.

(62 . 1) especifica el color de la línea; 62 es el código para el color. El segundo elemento del par punteado es el código de color de **AutoCAD**, en este caso 1 que se corresponde con el rojo.

(6 . "TRAZOS") es el tipo de línea de la entidad, línea en este caso; 6 es el código correspondiente al tipo de línea. El segundo elemento del par punteado es el nombre del tipo de línea, que aparecerá en mayúsculas y entre comillas por ser cadena.

(10 0.0 10.0 25.0) indica el punto inicial de la línea; en este caso 10 es el código del punto inicial, en otros casos (con otras entidades) significará otra cosa. Esta lista no es un par punteado, sino que contiene varios elementos. Dichos elementos son las coordenadas X (0.0), Y (10.0) y Z (25.0) del punto de inicio de la entidad de línea. Los puntos son separadores decimales.

(11 10.0 100.0 25.0) es el punto final de la línea; 11 es el código de punto final para las entidades que sean líneas; en otros casos será otra cosa. De la misma forma, la lista contiene después las coordenadas cartesianas del punto.

NOTA: En un caso real, la lista de una simple línea ofrecería más sublistas con otros códigos específicos que ya se estudiarán.

Como vemos, la manera en que están definidas las entidades en la Base de Datos de **AutoCAD** es bien sencilla. Al ser listas pueden ser fácilmente tratadas desde AutoLISP. Para entidades complejas, tipo polilíneas o bloques, el formato varía un poco, es por ello que se estudiará bajo el siguiente epígrafe. Las entidades no gráficas se verán también más adelante.

Se puede observar que para acceder a las propiedades y características de las entidades es necesario conocer los códigos normalizados. Estos códigos pueden ser comunes a todas las entidades o depender del tipo de entidad de que se trate. Así por ejemplo —como se ha indicado en su momento—, el código 8 representa la capa en la cual se encuentra la entidad actualmente; este código es común a todas las entidades de dibujo. En cambio el código 10, que en el ejemplo de la línea representaba su punto inicial, en un círculo indica su centro (al igual que en un arco), en un texto o en un bloque indica su punto de inserción, etcétera. La lista completa de todos los códigos de entidades para **AutoCAD** se proporciona más adelante (**ONCE.20.1.5.**).

En algún caso especial pueden existir varias sublistas con el mismo código como primer elemento. Esto ocurre por ejemplo con las splines, que mediante el código 10 representan todos sus puntos de control, mediante el código 11 todos sus puntos de ajuste, mediante el código 40 todos los valores de nodos, y demás.

Por otro lado, decir que hay que tener muy en cuenta un aspecto importantísimo cuando se trabaja con ciertas entidades, como polilíneas, bloques u objetos en tres dimensiones, entre otras. Y es que las coordenadas de sus puntos característicos están referidas al llamado SCE (Sistema de Coordenadas de la Entidad) —o SCO (Sistema de Coordenadas del Objeto)—. Para poder trabajar con ellas habrá que convertir dichos puntos

previamente mediante la función `TRANS` (ya estudiada). Por ejemplo, en una polilínea trazada mediante varios puntos, el primero de ellos está referido al SCP absoluto actual, sin embargo las demás coordenadas son relativas a dicho primer punto, es decir, están en el SCE. Esto hace que el proceso de regeneración de un dibujo en **AutoCAD** ahorre bastante tiempo, ya que sólo ha de recalcular el primer punto (los demás son relativos a él).

ONCE.20.1.3. Estructura para entidades compuestas

Nos dedicaremos ahora al estudio de la estructura que representa en la Base de Datos a las entidades complejas o compuestas.

Estas entidades compuestas no poseen una sola lista con sublistas, sino que se estructuran en varias listas separadas (cada una con sus propias sublistas de asociación): una lista de cabecera, varias listas de componentes y al final una lista de fin de secuencia. Las entidades compuestas que presentan estas características son:

- Polilíneas no optimizadas (versiones anteriores a la 14): tanto polilíneas 2D adaptadas a curva como polilíneas 3D y también mallas.
- Inserciones de bloque con atributos.

ONCE.20.1.3.1. Polilíneas no optimizadas

En versiones anteriores a la 14 de **AutoCAD**, las polilíneas tenían un tratamiento diferente en la Base de Datos interna de la aplicación. En **AutoCAD** aparece el concepto de polilíneas optimizadas, las cuales tienen una forma de guardarse junto al resto del dibujo más eficaz y controlada.

Estas nuevas polilíneas optimizadas están consideradas como entidades simples en la Base de Datos, y así se representan (su tipo de entidad, código 0, es `LWPOLYLINE`). No obstante, las polilíneas adaptadas a curva o curva B (spline) se convierten a polilíneas de versiones anteriores (tipo de entidad `POLYLINE`). También las polilíneas 3D y las mallas de **AutoCAD** son tipos de polilíneas no optimizadas. En este apartado pues, se verá este tipo de polilíneas, ya que son entidades compuestas.

NOTA: La variable de sistema de **AutoCAD** `PLINETYPE` especifica si el programa utiliza polilíneas 2D optimizadas. Sus valores se pueden observar en la lista de variables proporcionada en el **APÉNDICE B**.

Las polilíneas no optimizadas aparecen en la Base de Datos de **AutoCAD** de la siguiente manera:

1º La lista correspondiente a la entidad compuesta (tipo de entidad `POLYLINE`), conteniendo las características y propiedades globales de la polilínea. Es la denominada *cabecera* de la entidad en la Base de Datos. Esta lista contiene un código especial 66 que especifica que las siguientes listas pertenecen a la polilínea.

2º Un conjunto de listas de los vértices de la polilínea. A cada vértice le corresponde una lista (tipo de entidad `VERTEX`) que contiene las propiedades y características individuales de cada elemento (segmentos por ejemplo) de la polilínea.

3º Una lista de "fin de secuencia", que es un tipo de entidad especial llamado `SEQEND` que especifica el final de las listas de vértices, y por lo tanto de la polilínea. Esta lista contiene el código especial -2 que indica el nombre de la entidad principal, es decir de la cabecera.

Por ejemplo, una polilínea sencilla que contuviera únicamente dos segmentos, cada uno de ellos con espesores diferentes, podría contener en la Base de Datos las siguientes listas y sublistas (entre otras):

```
((-1 . <Nombre de objeto: 26a0a20>)
 (0 . "POLYLINE")
 (8 . "PIEZA")
 (66 . 1)
 (67 . 0)
 (10 0.0 0.0 0.0)
 (70 . 0)
 (40 . 5.0)
 (41 . 5.0)
 (210 0.0 0.0 1.0)
 )
```

Esta primera lista de cabecera contiene el tipo de entidad `POLYLINE`, que es la entidad principal o compuesta. En este caso se encuentra en la capa `PIEZA`. Su color es `PorCapa`, pues no aparece ninguna lista con el código 62. El grosor inicial y el grosor final globales de la polilínea (códigos 40 y 41) son ambos de 5.0. El resto de sublistas —alguna de ellas no se muestra aquí— hacen referencia a aspectos tales como si la polilínea es abierta o cerrada, si está adaptada curva, la orientación de su altura de objeto respecto al SCP actual, etc. El código 66 con valor 1 es el que indica que siguen otras listas de componentes de vértice.

NOTA: Apreciamos un aspecto aún no comentado, que es el hecho de que si no existe alguna de las listas significa que se toman unos valores por defecto, como por ejemplo `PorCapa` para el color (veremos más ejemplos).

La segunda de las listas de la polilínea podría ser la siguiente:

```
((-1 . <Nombre de objeto: 26a0a28>)
 (0 . "VERTEX")
 (8 . "PIEZA")
 (67 . 0)
 (10 100.0 110.0 0.0)
 (70 . 0)
 (40 . 5.0)
 (41 . 5.0)
 (42 . 0.0)
 (50 . 0.0)
 )
```

Esta segunda lista, con el tipo de entidad `VERTEX`, contiene las sublistas para el primer vértice de la polilínea. Su capa es `PIEZA`, las coordenadas (código 10) son en este caso $X = 100$, $Y = 110$ y $Z = 0$. El grosor inicial y final del segmento que empieza en ese vértice (códigos 40 y 41) son ambos 5.0. El resto de datos se refieren a si hay curva adaptada, la dirección de la tangente, etc.

La tercera lista:

```
((-1 . <Nombre de objeto: 26a09e8>)
 (0 . "VERTEX")
 (8 . "PIEZA")
 (67 . 0)
 (10 120.0 130.0 0.0)
 (70 . 0)
 (40 . 5.0)
 (41 . 0.0)
 )
```

```
(42 . 0.0)
(50 . 0.0)
)
```

Esta tercera lista corresponde al segundo vértice. En este caso sus coordenadas son X = 120, Y = 130 y Z = 0. El grosor inicial del segmento que empieza en ese vértice es 5.0, pero el grosor final aquí es 0.0.

Último vértice de la polilínea:

```
((-1 . <Nombre de objeto: 26a12a5>)
(0 . "VERTEX")
(8 . "PIEZA")
(67 . 0)
(10 160.0 140.0 0.0)
(70 . 0)
(40 . 0.0)
(41 . 0.0)
(42 . 0.0)
(50 . 0.0)
)
```

Esta cuarta lista contiene los datos del tercer vértice. Sus coordenadas son X = 160, Y = 140 y Z = 0). Al ser una polilínea abierta, el grosor inicial y final coinciden puesto que no hay ningún segmento que parta de este vértice. El grosor en este caso es 0.

Y la lista de fin de secuencia:

```
((-1 . <Nombre de objeto: 26a12e7>)
(0 . "SEQEND")
(8 . "PIEZA")
(67.0)
(-2 . <Nombre de objeto: 26a0a20>)
)
```

Esta última lista indica que la polilínea ya no contiene más vértices. El tipo de entidad es SEQEND. El código -2 indica el nombre de entidad de cabecera que repite el ya indicado en la primera lista.

ONCE.20.1.3.2. Inserciones de bloque con atributos

Las inserciones de un bloque son referencias a dicho bloque y están consideradas como entidades simples, con una única lista en la Base de Datos de **AutoCAD**. La definición de las entidades que forman dicho bloque se encuentran contenidas en las denominadas *tablas de símbolos*, que veremos en el apartado siguiente.

Sin embargo, si el bloque en cuestión contiene atributos es considerado como una entidad compuesta o compleja, de forma que la referencia del bloque es la entidad de cabecera y cada uno de los atributos viene definido en otras tantas listas siguientes. Al final, una lista de fin de secuencia cierra la entidad compuesta, como en el caso de las polilíneas no optimizadas.

Las referencias de bloques con atributos aparecen en la Base de Datos de **AutoCAD** de la siguiente manera:

1º La lista correspondiente a la entidad compuesta, en este caso una referencia de bloque (tipo de entidad INSERT). Es la cabecera.

2º Un conjunto de listas para cada atributo (tipo de entidad `ATTRIB`).

3º Una lista de "fin de secuencia" (tipo de entidad `SEQEND`).

Así pues, un sencillo bloque que representara una resistencia eléctrica, conteniendo un atributo para el valor de la resistencia, tendría en la Base de Datos una serie de listas parecidas a las que siguen:

```
((-1 . <Nombre de objeto: 26a0af8>)
(0 . "INSERT")
(8 . "ELEMENTOS")
(67 . 0)
(66 . 1)
(2 . "RES")
(10 80.0 100.0 0.0)
(41 . 1.0)
(42 . 1.0)
(50 . 0.0)
(43 . 1.0)
(210 0.0 0.0 1.0)
)
```

El tipo de entidad `INSERT` indica que se trata de una inserción de bloque. El bloque se llama `RES` (código 2). Se encuentra insertado en la capa `ELEMENTOS`. El código 66 con un valor 1 indica que el bloque contiene atributos y que siguen listas para cada atributo. El código 10 indica las coordenadas del punto de inserción del bloque. Los códigos 41, 42 y 43 indican las escalas X, Y y Z de inserción del bloque. El código 50 es el ángulo de rotación de la inserción. El último código es para la orientación de la altura de objeto.

La siguiente lista que aparece en la Base de Datos es la correspondiente al atributo incluido en el bloque:

```
((-1 . <Nombre de objeto: 26a0b00>)
(0 . "ATTRIB")
(8 . "ELEMENTOS")
(10 120.0 80.0 0.0)
(40 . 16.0)
(1 . "250K")
(2 . "RESIS")
(67 . 0)
(70 . 4)
(73 . 0)
(50 . 0.0)
(41 . 1.0)
(51 . 0.0)
(7 . "TS1")
(71 . 0)
(72 . 4)
(11 150.0 100.0 0.0)
(210 0.0 0.0 1.0)
)
```

El tipo de entidad es `ATTRIB` y se encuentra en la misma capa que la inserción del bloque. El punto de inserción del texto del atributo tiene coordenadas X = 120, Y = 80 y Z = 0. La altura de este texto (código 40) es 16. Los códigos 1 y 2 indican el valor del atributo (aquí es 250K) y el identificador (aquí `RESIS`) respectivamente. El código 70 con valor 4 indica que se trata de un atributo *verificable*. El código 50 indica el ángulo de rotación del texto. Los códigos siguientes hacen referencia al estilo de texto del atributo; en este caso tiene factor de

proporción 1, ángulo de inclinación 0, estilo TS1, generación normal, y es un texto con opción *rodear* (código 72 con valor 4). El punto indicado para *rodear* está en el código 11.

La última lista con el tipo de entidad `SEQEND`, puesto que ya no hay más atributos sería la siguiente:

```
((-1 . <Nombre de objeto: 26a0b08>)  
 (0 . "SEQEND")  
 (8 . "ELEMENTOS")  
 (67 . 0)  
 (-2 . <Nombre de objeto: 26a0af8>)  
)
```

El último código -2 contiene el nombre de la entidad principal o de cabecera, que es la referencia del bloque.

ONCE.20.1.4. Estructura para objetos no gráficos

Los objetos no gráficos de **AutoCAD** son datos que contiene el dibujo pero que no pertenecen a objeto gráficos como tal. Se subdividen en *diccionarios*, *tablas de símbolos* y bloques. Su estructura en la Base de Datos consiste también en una serie de listas de asociación similares a las de las entidades gráficas, pero que no contienen entidades propiamente dichas sino definiciones y características propias de los objetos no gráficos.

Las tablas de símbolos y los bloques comprenden 9 objetos no gráficos:

- Capas
- Estilos de texto
- Tipos de línea
- Estilos de acotación
- Vistas almacenadas
- Configuraciones de ventanas múltiples
- SCPs almacenados
- Definiciones de bloques
- Aplicaciones registradas

Los objetos de diccionario reúnen 2 objetos añadidos en la versión 13 de **AutoCAD**:

- Estilos de línea múltiple
- Grupos de selección

A partir de la revisión c4 de la versión 13, y en la versión 14, se han añadido otros tipos de objetos más específicas como `XRECORD` para almacenar datos arbitrarios o `ACAD_PROXY_OBJECT` para objetos *proxy* procedentes de aplicaciones no reconocidas por **AutoCAD**.

La organización de estos objetos en la Base de Datos, como hemos dicho, es similar a la de las entidades gráficas, a base de listas de asociaciones que poseen un código como primer elemento que indica el tipo de dato que viene a continuación. Los tipos de datos correspondientes a estos objetos para el tipo de entidad (código 0) son los que siguen:

Tipo de entidad	Objeto no gráfico
LAYER	Capa
LTYPE	Tipo de línea

Tipo de entidad	Objeto no gráfico
STYLE	Estilo de texto
DIMSTYLE	Estilo de acotación
VIEW	Vista almacenada
VPORT	Configuración de ventanas múltiples
UCS	SCP almacenado
BLOCK	Definición de bloque
ENDBLK	Final de definición de bloque
APPID	Aplicación registrada
GROUP	Grupo de selección
MLINESTYLE	Estilo de línea múltiple

En el caso de las definiciones de bloque, además de la lista con las características globales existen listas con todas las entidades que componen el bloque. A continuación veremos ejemplos de listas de algunos de los objetos no gráficos.

ONCE.20.1.4.1. Capa

La lista de definición de una capa en la tabla de símbolos podría ser la siguiente:

```
((0 . "LAYER")
 (2 . "EJES")
 (70 . 4)
 (62 . -7)
 (6 . "TRAZO_Y_PUNTO")
)
```

En este caso, el código 0 con valor asociado `LAYER` indica que se trata de una capa. Su nombre es `EJES`. El color asociado es blanco y, como el número de color aparece con signo negativo (-7), quiere decir que se encuentra en este momento desactivada. El tipo de línea asociado a la capa es `TRAZO_Y_PUNTO`.

ONCE.20.1.4.2. Estilo de texto

La lista de definición de un estilo de texto podría ser como la que sigue:

```
((0 . "STYLE")
 (2 . "TS1")
 (40 . 0.0)
 (41 . 1.0)
 (50 . 0.0)
 (71 . 0)
 (42 . 3.5)
 (3 . "romans.shx")
 (4 . "")
)
```

El nombre del estilo de texto es `TS1`; la altura en la definición del estilo (código 40) es 0; el factor de proporción (código 41) es 1; el ángulo de inclinación del estilo (código 50) es 0; la generación del estilo es *normal* (código 71 igual a 0); la altura actual de los textos por defecto (código 42) es 3.5; y el archivo de definición de tipos de letra en que está basado el estilo es `ROMANS.SHX`.

ONCE.20.1.4.3. Tipo de línea

La lista correspondiente a la definición de un tipo de línea cargado en el dibujo presenta un aspecto como el siguiente:

```
((0 . "LTYPE")
 (2 . "TRAZOS")
 (3 . "--- -- -- -- -- -- -- --")
 (72 . 65)
 (73 . 2)
 (40 . 0.75)
 (49 . 0.5)
 (49 . -0.25)
 )
```

El nombre del tipo de línea es `TRAZOS`; el aspecto que presenta en el archivo de definición `ACAD.LIN` o `ACADISO.LIN`, según preferencias, es el que aparece asociado como una cadena de texto al código 3; el código 72 indica un factor de alineamiento A , que como se sabe es obligatorio para los tipos de línea de **AutoCAD**. El código 73 indica el número de parámetros (sucesión de tramos, huecos y puntos) que definen el patrón de la línea. En este caso son dos, un trazo y un hueco. El código 40 proporciona la longitud total del patrón y el código 49 da en varias listas los valores de ese patrón: positivos para trazos, negativos para huecos y 0 para puntos.

ONCE.20.1.4.4. Definición de bloque

La lista de cabecera de la definición de un bloque tiene una característica especial y es que contiene un código `-2` con el nombre de la primera entidad de las que compone ese bloque. Por ejemplo:

```
((0 . "BLOCK")
 (2 . "RES")
 (70 . 66)
 (10 0.0 0.0 0.0)
 (-2 . <Nombre de objeto: 26a0b07>)
 )
```

El bloque definido se llama `RES`, su punto de inserción es el de coordenadas $X = 0$, $Y = 0$ y $Z = 0$) y el nombre de la primera entidad de la que está compuesta el bloque es `26a0b07`.

Como veremos seguidamente, a partir de ese nombre se podría acceder a todas las listas con las entidades que componen el bloque, exactamente del mismo modo que para las entidades sueltas. Aquí al explorar las listas de entidades en orden correlativo (`ENTNEXT`), después de la última se devuelve `nil`. No obstante, al construir las listas mediante `ENTMAKE` se requiere una lista final del tipo `ENDBLK` que indica que ya no hay más listas de componentes del bloque.

NOTA: Estas funciones AutoLISP se estudiarán tras la tabla de códigos de entidades.

ONCE.20.1.5. Códigos de acceso a Base de Datos

A continuación se proporciona la totalidad de los códigos de acceso directo a la Base de Datos de **AutoCAD**. Por comodidad y eficacia en la localización de códigos se ofrece una serie

de tablas: una de códigos ordenados por orden numérico y varias de códigos ordenados por entidades gráficas y no gráficas.

NOTA: Se han excluido de las tablas aquellos códigos específicos del formato DXF y que no se emplean desde aplicaciones AutoLISP y ARX.

TABLA 1. Códigos por orden numérico

NOTA: Los que se marcan con *—fijo—* no varían nunca en su definición. Los demás dependen de la entidad en la que se vean envueltos para trocar su significado.

Código	Descripción
-5	Cadena de reactivo permanente
-4	Operador condicional (utilizado sólo con SSGET)
-3	Centinela de datos extendidos (XDATA) <i>—fijo—</i>
-2	Referencia a nombre de entidad <i>—fijo—</i>
-1	Nombre de entidad. Cambia cada vez; al abrir un dibujo. No se guarda <i>—fijo—</i>
0	Cadena de texto que indica el tipo de entidad <i>—fijo—</i>
1	Valor de texto principal de una entidad
2	Un nombre: identificador de atributos, nombre de bloque, etc.
3-4	Otros valores de nombre o de texto
5	Identificador de entidad. Cadena de texto de hasta 16 dígitos hexadecimales. No cambia, aunque se cierre y vuelva a abrir el dibujo <i>—fijo—</i>
6	Nombre de tipo de línea <i>—fijo—</i>
7	Nombre de estilo de texto <i>—fijo—</i>
8	Nombre de capa <i>—fijo—</i>
10	Punto principal. Punto inicial de línea, inserción de texto, centro de círculo, etc.
11-18	Otros puntos
39	Altura de la entidad si no es cero <i>—fijo—</i>
40-48	Valores de coma flotante (altura de texto, factores de escala, etc.)
48	Escala del tipo de línea. Valor escalar de coma flotante
49	Valor de coma flotante repetido. Una misma entidad puede contener diversos grupos 49 para tablas de longitud variable (como las longitudes de trazo de la tabla LTYPE). El grupo 7x aparece siempre antes del primer grupo 49 y especifica la longitud de la tabla
50-58	Ángulos en radianes
60	Visibilidad de la entidad. Valor 0 (por defecto) = visibilidad; 1 = invisibilidad
62	Número de color <i>—fijo—</i>
66	Indica que siguen entidades <i>—fijo—</i>
67	Espacio (modelo o papel) <i>—fijo—</i>
68	Estado de activación de ventana gráfica
69	Número de identificación de ventana gráfica
70-78	Valores enteros, como número de repeticiones, bits indicadores o modos
90-99	Valores enteros de 32 bits
100	Marca de datos de subclase (con nombre de clase derivada como cadena). Es precisa para todos los objetos y clases de entidad que deriven de otra clase concreta
102	Cadena de control, seguida de {nombre_aplicación}. Indica que siguen datos de la aplicación, hasta encontrar una cadena de control } que finaliza el grupo de datos
105	Identificador de objetos de entrada para la tabla de símbolos DIMSTYLE
210	Dirección de la extrusión <i>—fijo—</i>
280-289	Valores enteros de 8 bits
300-309	Cadenas de texto arbitrarias
310-319	Bloques binarios arbitrarios
320-329	Identificadores de objeto arbitrarios
330-339	Identificador suave de dispositivo señalador. La referencia de dispositivo señalador indica uso, pero no posesión ni responsabilidad sobre otro objeto. Al ser suave, se permite que los objetos sean depurados
340-349	Identificador duro de dispositivo señalador. El mismo significado que antes pero, al ser duro, no se permite que los objetos sean depurados
350-359	Identificador suave de propietario. La referencia de propietario significa que el objeto propietario es responsable de los objetos para los cuales posee identificador de propiedad. Al ser suave, se permite depuración
360-369	Identificador duro de propietario. El mismo significado que antes pero, al ser duro,

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

Código	Descripción
	no se permite depuración
1000	Cadena ASCII (255 bytes como máximo) en datos extendidos XDATA
1001	Nombre de aplicación registrada (cadena ASCII de 31 bytes como máximo) para datos extendidos
1002	Cadena de control en datos extendidos ("{ "o " } ")
1003	Nombre de capa en datos extendidos
1004	Bloque de bytes (127 bytes como máximo) en datos extendidos
1005	Identificador de entidad en datos extendidos. Cadena de texto de hasta 16 dígitos hexadecimales
1010	Un punto en datos extendidos
1011	Posición del espacio universal 3D en datos extendidos
1012	Desplazamiento del espacio universal 3D en datos extendidos
1013	Dirección del espacio universal 3D en datos extendidos
1040	Valor de coma flotante en datos extendidos
1041	Valor de distancia en datos extendidos
1042	Factor de escala en datos extendidos
1070	Entero de 16 bits con signo en datos extendidos
1071	Entero grande de 32 bits con signo en datos extendidos

TABLA 2. Códigos por entidades gráficas

TABLA 2.1. Códigos comunes para todas la entidades

Código	Descripción
-1	Nombre de entidad (cambia cada vez que se abre un dibujo)
0	Tipo de entidad
5	Identificador (no cambia cada vez que se abre un dibujo)
6	Nombre del tipo de línea (presente si no es PorCapa). El nombre especial PorBloque indica un tipo de línea flotante
8	Nombre de capa
48	Escala del tipo de línea (si se omite, es 1)
60	Visibilidad del objeto. Valor 0 (o si se omite) = visible; valor 1 = invisible
62	Número de color (presente si no es PorCapa). Si su valor es 0, indica el color PorBloque. El valor 256 indica PorCapa. Un valor negativo indica que la capa está desactivada
67	Si se omite o es 0, indica que la entidad está en espacio modelo. El valor 1 indica que se encuentra en espacio papel
100	Marca de subclase (AcDbEntity)

TABLA 2.2. Códigos por tipo de entidad

Entidad	Código	Descripción
3DFACE	100	Marca de subclase (AcDbFace)
	10	Primer vértice (en SCU)
	11	Segundo vértice
	12	Tercer vértice
	13	Cuarto vértice. Si sólo hay tres, es igual al tercero
	70	Indicadores de lado invisible (por defecto = 0). Es la suma de: 1 = Primer lado invisible 2 = Segundo lado invisible 4 = Tercer lado invisible 8 = Cuarto lado invisible
3DSOLID	100	Marca de subclase (AcDbModelerGeometry)
	70	Número de versión de formato del modelador de sólidos (actualmente = 1)
	1	Datos de propiedad (varias líneas de menos de 255 caracteres cada una)

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

Entidad	3 Código	Líneas adicionales de datos de propiedad (si la cadena Descripción
		del grupo 1 anterior tiene más de 255 caracteres)
ARC	100	Marca de subclase (AcDbCircle)
	39	Altura de objeto (por defecto = 0)
	10	Punto central (en SCO)
	40	Radio
	100	Marca de subclase (AcDbArc)
	50	Ángulo inicial
	51	Ángulo final
	210	Dirección de la extrusión (por defecto = 0, 0, 1 que es el eje Z del SCU)
ATTDEF	100	Marca de subclase (AcDbText)
	39	Altura de objeto (por defecto = 0)
	10	Primer punto de alineación (en SCO)
	40	Altura del texto
	1	Valor por defecto (cadena)
	50	Rotación del texto (por defecto = 0)
	41	Factor de escala X relativa; anchura (por defecto = 1). Este valor se ajusta también cuando se sitúa el texto
	51	Ángulo oblicuo (por defecto = 0)
	7	Nombre de estilo (por defecto = STANDARD)
	71	Indicadores de generación del texto (por defecto = 0) Los mismos valores que el código 71 de TEXT
	72	Tipo de justificación de texto horizontal (por defecto = 0). Los mismos valores que el código 72 de TEXT
	11	Segundo punto de alineación (en SCO). Significativo sólo si los valores del grupo 72 o 74 son distintos de cero
	210	Dirección de la extrusión (por defecto = 0, 0, 1 que es el eje Z del SCU)
	100	Marca de subclase (AcDbAttributeDefinition)
	3	Cadena de solicitud
	2	Cadena de identificador
	70	Cadena de modo (por defecto = 0, atributo normal). Es la suma de: 1 = Atributo invisible. 2 = Atributo constante. 4 = Se requiere verificación al indicar este atributo. 8 = Atributo predefinido.
	73	Longitud de campo (por defecto = 0); no utilizado actualmente
	74	Tipo de justificación de texto vertical (por defecto = 0). Los mismos valores que el código 73 de TEXT
ATTRIB	100	Marca de subclase (AcDbText)
	39	Altura de objeto (por defecto = 0)
	10	Punto inicial del texto (en SCO)
	40	Altura del texto
	1	Valor por defecto (cadena).
	100	Marca de subclase (AcDbAttribute)
	2	Cadena de solicitud
	70	Cadena de modo. Es la suma de: 1 = Atributo invisible 2 = Atributo constante 4 = Se requiere verificación al indicar este atributo 8 = Atributo predefinido
	73	Longitud de campo (por defecto = 0) (no utilizado actualmente)
	50	Rotación del texto (por defecto = 0)
	41	Factor de escala X relativa; anchura (por defecto = 1).

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

Entidad	Código	Este valor se ajusta también cuando se sitúa el texto Descripción
	51	Ángulo oblicuo (por defecto = 0)
	7	Nombre del estilo de texto (por defecto = STANDARD)
	71	Indicadores de generación del texto (por defecto = 0). Los mismos valores que el código 71 de TEXT
	72	Tipo de justificación de texto horizontal (por defecto = 0). Los mismos valores que el código 72 de TEXT
	74	Tipo de justificación de texto vertical (por defecto = 0) Los mismos valores que el código 73 de TEXT
	11	Punto de alineación (en SCO). Presente sólo si el grupo 72 ó el 74 está presente y es distinto de cero (0)
	210	Dirección de la extrusión (por defecto = 0, 0, 1 que es el eje Z del SCU)
BODY	100	Marca de subclase (AcDbModelerGeometry)
	70	Número de versión de formato del modelador (actualmente = 1)
	1	Datos de propiedad (varias líneas de menos de 255 caracteres cada una)
	3	Líneas adicionales de datos de propiedad (si la cadena del grupo 1 anterior tiene más de 255 caracteres)
CIRCLE	100	Marca de subclase (AcDbCircle)
	39	Altura de objeto (por defecto = 0)
	10	Punto central (en SCO)
	40	Radio
	210	Dirección de la extrusión (por defecto = 0, 0, 1 que es el eje Z del SCU)
DIMENSION	100	Marca de subclase (AcDbDimension)
	2	Nombre del bloque que contiene las entidades que constituyen la imagen de cota
	11	Punto medio del texto de cota (en SCO)
	70	Tipo de cota. Los valores 0 a 6 son valores enteros que representan el tipo de cota. Los valores 32, 64 y 128 son valores de bit que se añaden a los valores enteros (el valor 32 está siempre añadido en la versión 13 y posteriores):
		0 = Girada, horizontal o vertical
		1 = Alineada
		2 = Angular
		3 = Diámetro
		4 = Radio
		5 = Angular de tres puntos
		6 = Coordenada
		32 = Se añade si sólo esta cota hace referencia a la referencia de bloque (código de grupo 2)
		64 = Tipo de coordenada. Es un valor de bit utilizado sólo con el valor entero 6. Si se añade, la coordenada es del tipo X, si no del tipo Y
		128 = Se añade si el texto de cota se sitúa en una posición definida por el usuario en lugar de la posición por defecto
	1	Texto de cota introducido explícitamente por el usuario. Si se omite o es nulo "" o vale "<>", el texto de cota ofrece la medida real; si es un espacio en blanco " " se suprime el texto. Cualquier otro valor se toma como texto de cota
	53	Ángulo de rotación del texto de cota respecto a su orientación por defecto, que es la dirección de la línea de cota. Por defecto = 0
	51	Ángulo respecto a la horizontal, de texto y línea de cota

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

Entidad	Código	Descripción
		para cotas lineales horizontales, verticales y giradas. Representa el valor
		negativo del ángulo entre el eje X del SCO y el eje X del SCP, siempre en el plano XY del SCO
	210	Dirección de la extrusión (por defecto = 0, 0, 1 que es el eje Z del SCU)
	3	Nombre del estilo de cota
	-3	Inicio de sección de la aplicación "ACAD" con los datos extendidos que indican las sustituciones de variables en el estilo de cota
<i>Cotas lineales, alineadas y giradas.</i> Además de los códigos comunes a todas las cotas, contienen:		
	100	Marca de subclase (AcDbAlignedDimension)
	10	Punto definidor de posición de la línea de cota (en SCU)
	12	Punto de inserción para los clones de una cota línea base y continua (en SCO)
	13	Punto definidor, inicio de primera línea de referencia (en SCU)
	14	Punto definidor, inicio de segunda línea de referencia (en SCU)
	50	Ángulo de cota (sólo para giradas, horizontales o verticales)
	52	Ángulo oblicuo (sólo para lineales). Si se añade al ángulo de rotación de la cota (código 50), se obtiene el ángulo de las líneas de cota
	100	Marca de subclase (AcDbRotatedDimension), sólo para giradas y lineales
<i>Cotas radiales.</i> Además de los códigos comunes a todas las cotas, contienen:		
	100	Marca de subclase (AcDbRadialDimension)
	10	Punto definidor de centro de círculo o arco (en SCU)
	15	Punto definidor de extremo de cota en el círculo o arco (en SCU)
	40	Longitud directriz de la cota
<i>Cotas de diámetro.</i> Además de los códigos comunes a todas las cotas, contienen:		
	100	Marca de subclase (AcDbDiametricDimension)
	15	Primer extremo de cota en el círculo o arco (en SCU)
	10	Punto opuesto al anterior en el círculo o arco (en SCU)
	40	Longitud directriz de la cota
<i>Cotas angulares entre dos líneas.</i> Además de los códigos comunes a todas las cotas, contienen:		
	100	Marca de subclase (AcDb2LineAngularDimension)
	10	Primer punto final de la segunda línea acotada (en SCU)
	13	Primer punto final de la primera línea acotada (en SCU)
	14	Segundo punto final de la primera línea acotada (en SCU)
	15	Segundo punto final de la segunda línea acotada (en SCU)
	16	Posición del arco de cota (en SCO)
<i>Cotas angulares en círculos, arcos o tres puntos.</i> Además de los códigos comunes, contienen:		
	100	Marca de subclase (AcDb3PointAngularDimension)
	10	Posición del arco de cota (en SCO)
	13	Primer punto final de línea de referencia (en SCU)
	14	Segundo punto final de línea de referencia (en SCU)
	15	Vértice del ángulo (en SCU)
<i>Cotas de coordenadas.</i> Además de los códigos comunes a todas las cotas, contienen:		
	100	Marca de subclase (AcDbOrdinateDimension)
	10	Origen del SCP al acotar (en SCO)
	13	Ubicación del punto a acotar (en SCU)
	14	Punto final de la directriz de cota (en SCU)
<i>ELLIPSE</i>		
	100	Marca de subclase (AcDbEllipse)
	10	Punto central (en SCU)
	11	Extremo del eje mayor respecto al centro
	210	Dirección de la extrusión. (por defecto = 0, 0, 1 que es el eje Z del SCU)
	40	Radio del eje menor como porcentaje respecto al eje mayor

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

Entidad	Código	Descripción
HATCH	41	Parámetro de inicio (este valor es 0.0 para una elipse completa)
	42	Parámetro final (este valor es $2 \times \text{PI}$ para una elipse completa)
	100	Marca de subclase (AcDbHatch)
	10	Punto de elevación (en SCO). X e Y son siempre iguales a 0; Z representa la elevación
	210	Dirección de la extrusión (por defecto = 0, 0, 1 que es el eje Z del SCU)
	2	Nombre del patrón de sombreado
	70	Indicador de relleno sólido (sólido = 1; relleno de patrón = 0)
	71	Indicador de asociatividad (asociativo = 1; no asociativo = 0)
	91	Número de caminos de contorno (bucles). Para cada camino existen los siguientes códigos, que se repiten el número de veces especificado en el código 91:
	92	Indicador de tipo de camino de contorno (expresado en bits): 0 = Por defecto 1 = Externo 2 = Polilínea 4 = Derivado 8 = Cuadro de texto 16 = Más externo Si el tipo de contorno es una polilínea, siguen los códigos: 72 Indicador de curvatura 73 Indicador de cerrada 93 Número de vértices de la polilínea 10 Emplazamiento de vértices (en SCO); varias entradas 42 Curvatura (por defecto = 0)
	93	Número de lados del camino de contorno (sólo si no es polilínea)
	72	Tipo de lado (sólo si el contorno no es una polilínea): 1 = Línea 2 = Arco circular 3 = Arco elíptico 4 = Spline Si el tipo de contorno es una línea, siguen los códigos: 10 Punto inicial (en SCO) 11 Punto final (en SCO) Si el tipo de contorno es un arco circular, siguen los códigos: 10 Punto central (en SCO) 40 Radio 50 Ángulo inicial 51 Ángulo final 73 Indicador de sentido contrario de las agujas del reloj Si el tipo de contorno es un arco elíptico, siguen los códigos: 10 Punto central (en SCO) 11 Punto final del eje principal respecto al punto central (en SCO) 40 Longitud del eje menor (porcentaje del eje principal) 50 Ángulo inicial 51 Ángulo final 73 Indicador de sentido contrario de las agujas del reloj Si el tipo de contorno es un lado de spline, siguen los códigos:

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

		94	Grado
		73	Racional
Entidad	Código	Descripción	
		74	Periódica
		95	Número de nudos
		96	Número de puntos de control
		40	Valores de nudos (varias entradas)
		10	Punto central (en SCO)
		42	Altura (por defecto = 1)
	97	Número de objetos de contorno de origen	
	330	Referencia a nombres de objetos de contorno (varias entradas)	
	75	Estilo de sombreado	
		0 = Área de "paridad impar" (estilo <i>normal</i>)	
		1 = Sólo área más externa de sombreado (estilo <i>exterior</i>)	
		2 = Toda el área de sombreado (estilo <i>ignorar</i>)	
	76	Tipo de patrón de sombreado	
		0 = Definido por el usuario	
		1 = Predefinido	
		2 = Personalizado	
	52	Ángulo de patrón de sombreado (sólo relleno de patrón)	
	41	Escala o intervalo de patrón de sombreado (sólo patrón)	
	77	Indicador de sombreado doble (doble = 1, no doble = 0)	
	78	Número de líneas de definición de patrón. Para cada línea existen los siguientes códigos, que se repiten el número de veces especificado en el código 78:	
		53	Ángulo de línea de patrón
		43	Punto base de línea de patrón, componente X
		44	Punto base de línea de patrón, componente Y
		45	Desplazamiento de línea de patrón, componente X
		46	Desplazamiento de línea de patrón, componente Y
		79	Número de elementos de longitud del trazo
		49	Longitud del trazo (varias entradas)
	47	Tamaño de pixel (opcional)	
	98	Número de puntos base	
	10	Punto base (en SCO); varias entradas	
IMAGE	100	Marca de subclase (AcDbRasterImage)	
	90	Versión de clase	
	10	Punto de inserción (en SCO)	
	11	Vector U de un pixel (puntos a lo largo de la parte inferior visual de la imagen, empezando en el punto de inserción) en el SCO	
	12	Vector V de un pixel (puntos a lo largo de la parte inferior visual de la imagen, empezando en el punto de inserción) en el SCO	
	13	Tamaño de la imagen en pixeles (valores U y V)	
	340	Referencia a objeto IMAGEDEF	
	70	Propiedades de visualización de imagen. Es la suma de:	
		1 = Mostrar imagen	
		2 = Mostrar imagen cuando no esté alineada con pantalla	
		4 = Utilizar contorno de delimitación	
		8 = Transparencia activada	
	280	Estado de delimitación: 0 = desactivado, 1 = activado	
	281	Valor de brillo (0 - 100; por defecto = 50)	
	282	Valor de contraste (0 - 100; por defecto = 50)	
	283	Valor de difuminado (0 - 100; por defecto = 0)	
	360	Referencia a objeto IMAGEDEF_REACTOR	
	71	Tipo de contorno de delimitación: 1 = rectangular, 2 = poligonal	
	91	Número de vértices del contorno de delimitación que siguen	
	14	Vértice del contorno de delimitación (en SCO); varias entradas:	

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

Entidad	Código	Rectangular: dos esquinas opuestas
		Descripción
INSERT		Poligonal: tres o más vértices, indicados secuencialmente
	100	Marca de subclase (AcDbBlockReference)
	66	Indicador de que siguen atributos variables (por defecto = 0). Si el valor es 1, la entidad de inserción terminada por un SEQEND irá seguida de una serie de entidades de atributo
	2	Nombre de bloque
	10	Punto de inserción (en SCU)
	41	Factor de escala X (por defecto = 1)
	42	Factor de escala Y (por defecto = 1)
	43	Factor de escala Z (por defecto = 1)
	50	Ángulo de rotación (por defecto = 0)
	70	Número de columnas en INSERTM (por defecto = 1)
	71	Número de filas en INSERTM (por defecto = 1)
	44	Intervalo entre columnas en INSERTM (por defecto = 0)
	45	Intervalo entre filas en INSERTM (por defecto = 0)
	210	Dirección de la extrusión. (por defecto = 0, 0, 1 que es el eje Z del SCU)
LEADER	100	Marca de subclase (AcDbLeader)
	3	Nombre de estilo de cota
	71	Indicador de extremo de cota: 0 = desactivado; 1 = activado
	72	Tipo de camino: 0 = segmentos de línea recta; 1 = spline
	73	Indicador de creación de directriz (por defecto = 3): 0 = Directriz creada con anotación de texto 1 = Creada con anotación de tolerancia 2 = Creada con anotación de referencia 3 = Creada sin anotación
	74	Indicador de dirección de línea de conexión (si la hay): 0 = La línea de conexión (o final de tangente para directriz spline) tiene la dirección opuesta al vector horizontal 1 = La línea de conexión (o final de tangente de directriz spline) tiene la dirección del vector horizontal
	75	Indicador de línea de conexión: 0 = sin línea; 1 = con línea
	40	Altura de la anotación de texto
	41	Anchura de la anotación de texto
	76	Número de vértices en la directriz
	10	Coordenadas de vértice (una entrada para cada vértice).
	77	Color que utilizar si DIMCLRE = PorBloque para la directriz
	340	Referencia a anotación asociada (texto múltiple, tolerancia o inserción)
	210	Vector normal
	211	Dirección horizontal para una directriz
	212	Desplazamiento del punto de inserción de referencia a bloque desde el vértice de la directriz
	213	Desplazamiento del punto de emplazamiento de anotación desde el último vértice de la directriz
	-3	Inicio de sección de la aplicación "ACAD" con los datos extendidos que indican las sustituciones de variables en el estilo de cota
LINE	100	Marca de subclase (AcDbLine)
	39	Altura de objeto (por defecto = 0)
	10	Punto inicial (en SCU)

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

Entidad	11 Código	Punto final (en SCU) Descripción
LWPOLYLINE	210	Dirección de la extrusión (por defecto = 0, 0, 1 que es el eje Z del SCU)
	100	Marca de subclase (AcDbPolyline)
	90	Número de vértices
	70	Indicador de polilínea (por defecto = 0) 1 = Cerrada 128 = <i>Plinegen</i>
	43	Anchura constante (por defecto = 0); no se utiliza si se establece una anchura variable (códigos 40 y/o 41).
	38	Elevación (por defecto = 0)
	39	Altura de objeto (por defecto = 0)
	10	Coordenadas de vértice (en SCU); una entrada por vértice
	40	Grosor inicial; una entrada por vértice (por defecto = 0). No se utiliza si se establece un grosor constante (código 43)
	41	Grosor final; una entrada por vértice (por defecto = 0). No se utiliza si se establece un grosor constante (código 43)
	42	Curvatura; una entrada por vértice (por defecto = 0)
	210	Dirección de la extrusión (por defecto = 0, 0, 1 que es el eje Z del SCU)
MLINE	100	Marca de subclase (AcDbMline)
	2	Cadena de hasta 32 caracteres. Nombre del estilo utilizado. Debe haber una entrada de este estilo en el diccionario MLINESTYLE
	340	Identificador del dispositivo señalador del diccionario MLINESTYLE
	40	Factor de escala
	70	Justificación: 0 = superior, 1 = ninguna, 2 = inferior
	71	Indicador de abierta/cerrada: 1 = abierta, 3 = cerrada
	72	Número de vértices
	73	Número de elementos de la definición MLINESTYLE
	10	Punto inicial (en SCU)
	210	Dirección de la extrusión (por defecto = 0, 0, 1 que es el eje Z del SCU)
	11	Coordenadas de vértice; una entrada para cada vértice
	12	Vector de dirección del segmento que empieza en el vértice una entrada para cada vértice
	13	Vector de dirección del inglete en el vértice; una entrada para cada vértice
	74	Número de parámetros para este elemento (se repite para todos los elementos de un segmento)
	41	Parámetros del elemento (se repite según lo especificado en el código 74 anterior)
	75	Número de parámetros del área de relleno para este elemento (se repite para todos los elementos del segmento)
	42	Parámetros del área de relleno (se repite según lo especificado en el código 75 anterior)
MTEXT	100	Marca de subclase (AcDbMText)
	10	Punto de inserción
	40	Altura del texto por defecto
	41	Anchura del rectángulo de referencia
	71	Punto de unión: 1 = Superior izquierdo 2 = Superior centro 3 = Superior derecho 4 = Central izquierdo

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

Entidad	Código	Descripción
		5 = Medio centro 6 = Central derecho 7 = Inferior izquierdo 8 = Inferior centro 9 = Inferior derecho
	72	Dirección del dibujo: 1 = De izquierda a derecha 2 = De derecha a izquierda 3 = De arriba abajo 4 = De abajo arriba
	1	Cadena de texto. Si la cadena de texto tiene menos de 250 caracteres, aparece en el grupo 1. Si tiene más, la cadena se divide en bloques de 250 caracteres, que aparecen en uno o más códigos de grupo 3. En este caso, el último grupo siempre es un grupo 1 y tiene menos de 250 caracteres
	3	Texto adicional (siempre en bloques de 250 caracteres)
	7	Nombre del estilo de texto (por defecto STANDARD)
	210	Dirección de la extrusión (por defecto = 0, 0, 1 que es el eje Z del SCU)
	11	Vector de dirección del eje X (en SCU)
	42	Anchura horizontal de los caracteres
	43	Altura vertical de los caracteres
	50	Ángulo de rotación en radianes
OLEFRAME	100	Marca de subclase (AcDbOleFrame)
	70	Número de versión de OLE
	90	Longitud de datos binarios
	310	Datos binarios (varias líneas)
	1	Fin de datos OLE (la cadena "OLE")
OLE2FRAME	100	Marca de subclase (AcDbOle2Frame)
	70	Número de versión de OLE
	3	Longitud de datos binarios
	10	Esquina superior izquierda (en SCU)
	11	Esquina inferior derecha (en SCU)
	71	Tipo de objeto OLE: 1 = vinculado, 2 = incrustado, 3 = estático
	72	Descriptor de modo mosaico: 0 = El objeto está en una ventana gráfica en mosaico 1 = El objeto no está en una ventana gráfica en mosaico
	90	Longitud de datos binarios
	310	Datos binarios (varias líneas)
	1	Fin de datos OLE (la cadena "OLE")
POINT	100	Marca de subclase (AcDbPoint)
	10	Ubicación del punto (en SCU)
	39	Altura de objeto (por defecto = 0)
	210	Dirección de la extrusión (por defecto = 0, 0, 1 que es el eje Z del SCU)
	50	Ángulo del eje X para el SCP en vigor cuando se dibuja el punto (por defecto = 0); se utiliza cuando PDMODE es distinto de cero
POLYLINE	100	Marca de subclase (AcDb2dPolyline o AcDb3dPolyline)
	10	Punto <i>ficticio</i> ; los valores de X e Y son siempre 0, y el valor de Z es la elevación de la polilínea (en SCO en 2D y SCU en 3D)
	39	Altura de objeto (por defecto = 0)

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

Entidad	Código	Descripción
	0	Indicador de polilínea (por defecto es 0). Es la suma de:
		1 = Polilínea cerrada (o malla poligonal cerrada en M)
		2 = Se han añadido vértices convertidos en curva
		4 = Se han añadido vértices convertidos en spline
		8 = Polilínea 3D
		16 = Malla poligonal 3D
		32 = La malla poligonal está cerrada en la dirección N
		64 = La polilínea es una malla poligonal
		128 = El tipo de línea se genera de forma continua
	40	Grosor inicial por defecto (por defecto = 0)
	41	Grosor final por defecto (por defecto = 0)
	71	Número de vértices M de malla poligonal (por defecto = 0)
	72	Número de vértices N de malla poligonal (por defecto = 0)
	73	Densidad M de superficie amoldada (por defecto = 0)
	74	Densidad N de superficie amoldada (por defecto = 0)
	75	Curvas y tipo de superficie amoldada (por defecto = 0):
		0 = No adaptada a superficie amoldada
		5 = Superficie de curva B (spline) cuadrática
		6 = Superficie de curva B (spline) cúbica
		8 = Superficie Bézier
	210	Dirección de la extrusión (por defecto = 0, 0, 1 que es el eje Z del SCU)
RAY	100	Marca de subclase (AcDbRay)
	10	Punto inicial (en SCU)
	11	Vector unitario de dirección (en SCU)
REGION	100	Marca de subclase (AcDbModelerGeometry)
	70	Número de versión de formato del modelador (actualmente = 1)
	1	Datos de propiedad (varias líneas de menos de 255 caracteres cada una)
	3	Líneas adicionales de datos de propiedad (si la cadena del grupo 1 anterior tiene más de 255 caracteres)
SEQEND	-2	Nombre de la entidad que inicia la secuencia. Este tipo de entidad marca el final de vértices para una polilínea o el final de atributos para una inserción de bloque con atributos
SHAPE	100	Marca de subclase (AcDbShape)
	39	Altura de objeto (por defecto = 0)
	10	Punto de inserción (en SCU)
	40	Tamaño
	2	Nombre de la forma
	50	Ángulo de rotación (por defecto = 0)
	41	Factor de escala X relativa (por defecto = 1)
	51	Ángulo de rotación (por defecto = 0)
	210	Dirección de la extrusión (por defecto = 0, 0, 1 que es el eje Z del SCU)
SOLID	100	Marca de subclase (AcDbTrace)
	10	Primera esquina
	11	Segunda esquina

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

Entidad	Código	Descripción
SPLINE	12	Tercera esquina
	13	Cuarta esquina. Si sólo hay tres, coincide con la tercera
	39	Altura de objeto (por defecto = 0)
	210	Dirección de la extrusión (por defecto = 0, 0, 1 que es el eje Z del SCU)
	100	Marca de subclase (AcDbSpline)
	210	Vector normal (se omite si la spline no es plana)
	70	Indicador de la spline. Es la suma de: 1 = Spline cerrada 2 = Spline periódica 4 = Spline racional 8 = Plana 16 = Lineal (también se especifican bits para plana)
	71	Grado de la curva spline
	72	Número de nudos
	73	Número de puntos de control
	74	Número de puntos de ajuste (si los hay)
	42	Tolerancia de nudo (por defecto = 0.0000001)
	43	Tolerancia de los puntos de control (por defecto = 0.0000001)
	44	Tolerancia de ajuste (por defecto = 0.0000000001)
	12	Tangente inicial; puede omitirse (en SCU)
	13	Tangente final; puede omitirse (en SCU)
	40	Valor de nudo (una entrada por nudo)
	41	Grosor (si no es 1); con varios pares de grupos, están presentes si todos son distintos de 1
	10	Puntos de apoyo (en SCU); una entrada por punto de apoyo
	11	Puntos de ajuste (en SCU); una entrada por punto de ajuste
TEXT	100	Marca de subclase (AcDbText)
	39	Altura de objeto (por defecto = 0)
	10	Primer punto de alineación (en SCO)
	40	Altura del texto
	1	Valor del texto (la cadena de texto en sí)
	50	Angulo de rotación del texto (por defecto = 0)
	41	Factor de escala X o anchura (por defecto = 1). Este valor se ajusta también cuando se sitúa en el texto
	51	Ángulo de oblicuidad (por defecto = 0)
	7	Nombre del estilo de texto (por defecto = STANDARD)
	71	Indicadores de generación del texto (por defecto = 0). Suma de: 2 = El texto mira hacia atrás (simetría en X) 4 = El texto mira hacia arriba (simetría en Y)
	72	Tipo de justificación del texto en horizontal (por defecto = 0): 0 = Izquierdo 1 = Centrado 2 = Derecho 3 = Alineado (si el alineamiento vertical = 0) 4 = Medio (si el alineamiento vertical = 0) 5 = Ajustado (si el alineamiento vertical = 0)
	11	Segundo punto de alineación (en SCO). Sólo tiene sentido si el valor del grupo 72 ó 73 es distinto de 0
	210	Dirección de la extrusión (por defecto = 0, 0, 1 que es el eje Z del SCU)
	73	Tipo de justificación del texto en vertical (por defecto = 0): 0 = Línea base 1 = Inferior

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

2 = Medio

Entidad	Código	Descripción
3 = Superior		
TOLERANCE	100	Marca de subclase (AcDbFcf)
	3	Nombre de estilo de cota
	10	Punto de inserción (en SCU)
	210	Dirección de la extrusión (por defecto = 0, 0, 1 que es el eje Z del SCU)
	11	Vector de dirección del eje X (en SCU)
TRACE	100	Marca de subclase (AcDbTrace)
	10	Primera esquina (en SCO)
	11	Segunda esquina (en SCO)
	12	Tercera esquina (en SCO)
	13	Cuarta esquina (en SCO)
	39	Altura de objeto (por defecto = 0)
	210	Dirección de la extrusión (por defecto = 0, 0, 1 que es el eje Z del SCU)
VERTEX	100	Marca de subclase (AcDbVertex)
	100	Marca de subclase (AcDb2dVertex o AcDb3dPolylineVertex)
	10	Punto de ubicación (en SCO si es 2D y en SCU si es 3D)
	40	Grosor inicial (por defecto = 0)
	41	Grosor final (por defecto = 0)
	42	Curvatura (por defecto = 0). La curvatura es la tangente de la cuarta parte del ángulo incluido para un segmento de arco, expresada en números negativos si el arco se dibuja en sentido de las agujas del reloj desde el punto inicial al punto final. Una curvatura 0 indica un segmento recto y una curvatura 1 indica un semicírculo
	70	Indicadores de vértice. Es la suma de: 1 = Vértice extra creado por una adaptación a curva 2 = Tangente adaptada a curva definida por este vértice 4 = Sin utilizar 8 = Vértice spline creado por un ajuste a spline 16 = Punto de control de armadura spline 32 = Vértice de polilínea 3D 64 = Vértice de malla poligonal 3D 128 = Vértice de malla policara
	50	Dirección de tangente adaptada a curva
	71	Índice de vértice de malla policara
	72	Índice de vértice de malla policara
	73	Índice de vértice de malla policara
	74	Índice de vértice de malla policara. En los cuatro casos, si se omite, su valor es 0. Si es negativo, indica arista invisible
VIEWPORT	100	Marca de subclase (AcDbViewport)
	10	Punto central (en SCU)
	40	Anchura en unidades del espacio papel
	41	Altura en unidades del espacio papel
	68	Campo de estado de ventana gráfica: 0 = Desactivada 1 = Activada, pero no se ve en pantalla, o se ha excedido el número especificado por MAXACTVP número = Activada y en funcionamiento. El

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

valor indica el orden de apilamiento de las ventanas gráficas: 1 es la ventana gráfica activa, 2 la siguiente, etc.

Entidad	Código	Descripción
	69	Número de identificación. Cambia cada vez que se abre un dibujo, excepto la ventana gráfica del espacio papel que tiene siempre valor 1
	-3	Inicio de sección de la aplicación "ACAD" con los datos extendidos que describen la ventana.
	1001	Identificador de la aplicación ("ACAD")
	1000	Datos de inicio de la ventana gráfica. Es siempre una cadena VMULT
	1002	Datos descriptores del inicio de la ventana gráfica. Es siempre una cadena "{ "
	1070	Número de versión de los datos extendidos. Es siempre el 16
	1010	Punto de mira de la vista (en SCU)
	1010	Vector de la línea de mira (en SCU)
	1040	Ángulo de ladeo de la vista
	1040	Altura de la vista
	1040	Valor X del punto central de la vista (en SCV)
	1040	Valor Y del punto central de la vista (en SCV)
	1040	Longitud de la lente de perspectiva
	1040	Valor Z del plano delimitador frontal
	1040	Valor Z del plano delimitador posterior
	1070	Modo de visualización
	1070	Valor de RESVISTA
	1070	Parámetro de zoom rápido
	1070	Parámetro de SIMBSPC
	1070	Forzcursor ACT/DES
	1070	Rejilla ACT/DES
	1070	Estilo de Forzcursor
	1070	Isoplano de Forzcursor
	1040	Ángulo de Forzcursor
	1040	Valor de coordenada X en el SCP del punto base de Forzcursor
	1040	Valor de coordenada Y en el SCP del punto base de Forzcursor
	1040	Intervalo X de Forzcursor
	1040	Intervalo Y de Forzcursor
	1040	Intervalo X de Rejilla
	1040	Intervalo Y de Rejilla
	1070	Indicador de oculto en trazo
	1002	Comienza la lista de capas inutilizadas (probablemente esté vacía). Este campo es siempre la cadena "{ "
	1003	Nombres de las capas inutilizadas en la ventana gráfica. Esta lista puede incluir capas dependientes de una referencia. Aquí puede aparecer cualquier número de grupos 1003
	1002	Final de la lista de capas inutilizadas. Es siempre la cadena " } "
	1002	Final de los datos de la ventana. Es siempre la cadena " } "
XLINE	100	Marca de subclase (AcDbXline)
	10	Primer punto (en SCU)
	11	Vector unitario de dirección (en SCU)

TABLA 3. Códigos para tablas de símbolos

TABLA 3.1. Códigos del grupo de tablas de símbolos

Código	Descripción
--------	-------------

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

-1	Nombre de entidad (cambia cada vez que se abre un dibujo)
0	Tipo de objeto (TABLE)
2	Nombre de tabla
5	Identificador (se mantiene cada vez que se abre el dibujo)
Código	Descripción
100	Marca de subclase (AcDbSymbolTable)
70	Número máximo de entradas en una tabla

TABLA 3.2. Códigos comunes a todos los tipos de tablas de símbolos

Código	Descripción
-1	Nombre de entidad (cambia cada vez que se abre un dibujo)
0	Tipo de objeto (nombre de la tabla)
2	Nombre de tabla
5	Identificador (excepto DIMSTYLE); se mantiene cada vez que se abre el dibujo
105	Identificador, sólo para DIMSTYLE
102	Inicio del grupo definido por la aplicación "{nombre_aplicación}". Por ejemplo, "{ACAD_REACTORS}" indica el inicio del grupo de reactivos permanentes de AutoCAD , y "{ACAD_XDICTIONARY}" indica el inicio de un grupo de extensión de diccionario
<i>códigos def. por aplicación</i>	Los códigos y valores contenidos entre los grupos 102 están definidos por la aplicación
102	Fin del grupo, "}" indica que terminan los valores definidos por la aplicación
100	Marca de subclase (AcDbSymbolTableRecord)

TABLA 3.3. Códigos para cada tipo de tabla de símbolos

Tipo de tabla	Código	Descripción
APPID	100	Marca de subclase (AcDbRegAppTableRecord)
	2	Nombre de aplicación proporcionado por el usuario (para datos extendidos). Estas entradas de tabla mantienen un conjunto de nombres para todas las aplicaciones registradas
	70	Valores de indicadores estándar. Es la suma de: 1 = Los XDATA asociados no se escriben cuando se ejecuta GUARDCOM12 16 = La aplicación es dependiente de una RefX. 32 = Se ha resuelto satisfactoriamente la RefX. 64 = Al menos una entidad del dibujo hizo referencia a la aplicación la última vez que se editó el dibujo
BLOCK_RECORD	100	Marca de subclase (AcDbBlockTableRecord)
	2	Nombre de bloque
DIMSTYLE	100	Marca de subclase (AcDbDimStyleTableRecord).
	2	Nombre de estilo de cota
	70	Valores de indicador estándar. Es la suma de: 16 = El estilo de cota es dependiente de una RefX. 32 = Se ha resuelto satisfactoriamente la RefX. 64 = Al menos una entidad del dibujo hizo referencia al estilo de cota la última vez que se editó el dibujo
	3	DIMPOST (ACOPOST)
	4	DIMAPOST (ACOPOSTA)

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

Tipo de tabla	5	DIMBLK (ACOBLOQ)
	6	DIMBLK1 (ACOBLOQ1)
	7	DIMBLK2 (ACOBLOQ2)
	40	DIMSCALE (ACOECSAL)
Tipo de tabla	Código	Descripción
	41	DIMASZ (ACOTF)
	42	DIMEXO (ACODLRO)
	43	DIMDLI (ACOILA)
	44	DIMEXE (ACOLREC)
	45	DIMRND (ACORED)
	46	DIMDLE (ACOPLC)
	47	DIMTP (ACOTOLMA)
	48	DIMTM (ACOTOLME)
	140	DIMTXT (ACOALTXT)
	141	DIMCEN (ACOCEN)
	142	DIMTSZ (ACOTAMTR)
	143	DIMALT (ACOCALT)
	144	DIMLFAC (ACOFACL)
	145	DIMTVP (ACOPVT)
	146	DIMTFAC (ACOFACOT)
	147	DIMGAP (ACODIST)
	71	DIMTOL (ACOTOL)
	72	DIMLIM (ACOLIM)
	73	DIMTIH (ACOTIH)
	74	DIMTOH (ACOTEH)
	75	DIMSE1 (ACOSLR1)
	76	DIMSE2 (ACOSLR2)
	77	DIMZIN (ACOCP)
	170	DIMALT (ACOALT)
	171	DIMALTD (ACOPALT)
	172	DIMTOFL (ACOTELI)
	173	DIMSAH (ACOFD)
	174	DIMTIX (ACOTIL)
	175	DIMSOXD (ACOSLCE)
	176	DIMCLDR (ACOCOLAC)
	177	DIMCLRE (ACOCOLRE)
	178	DIMCLRT (ACOCOTEX)
	270	DIMUNIT (ACOUN)
	271	DIMDEC (ACODEC)
	272	DIMTDEC (ACOTOLDEC)
	273	DIMALTU (ACOUNALT)
	274	DIMALTTD (ACOTDALT)
	340	Identificador de objeto ESTILO referenciado (utilizado en lugar de almacenar el valor DIMSTYLE (ACOESTEXT))
	275	DIMAUNIT (ACOUNANG)
	280	DIMJUST (ACOSTJUST)
	281	DIMSD1 (ACOSLC1)
	282	DIMSD2 (ACOSLC2)
	283	DIMTOLJ (ACOSTJUSTOL)
	284	DIMTZIN (ACOCPT)
	285	DIMALT (ACOCALT)
	286	DIMALTTZ (ACOTCALT)
	287	DIMFIT (ACOAJUS)
	288	DIMUPT (ACOTSITU)
LAYER	100	Marca de subclase (AcDbLayerTableRecord)
	2	Nombre de capa
	70	Indicadores estándar. Es la suma de:
		1 = La capa está inutilizada 2 = La capa está inutilizada por defecto en las nuevas ventanas gráficas 4 = La capa está bloqueada 16 = La capa es dependiente de una refx. 32 = Se ha resuelto satisfactoriamente la RefX.

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

		64 = Al menos una entidad del dibujo hizo referencia a la capa la última vez que se editó el dibujo
Tipo de tabla	Código	Descripción
LTYPE	62	Número de color (si es negativo, la capa está desactivada)
	6	Nombre de tipo de línea
	100	Marca de subclase (AcDbLinetypeTableRecord)
	2	Nombre de tipo de línea
	70	Indicadores estándar. Es la suma de: 16 = El tipo de línea es dependiente de una RefX. 32 = Se ha resuelto satisfactoriamente la RefX. 64 = Al menos una entidad del dibujo hizo referencia al tipo de línea la última vez que se editó el dibujo
	3	Texto descriptivo para el tipo de línea
	72	Código de alineación; es siempre 65 (código ASCII de la letra A)
	73	Número de elementos de tipo de línea
	40	Longitud total de patrón
	49	Longitud de trazo, punto o espacio (una entrada por elemento)
	74	Tipo de elemento de tipo de línea complejo (uno por elemento): 0 = No complejo 2 = Cadena de texto incrustado 4 = Forma incrustada
	75	Código de forma compleja: uno por elemento si el código 74 > 0; sólo uno si el código 74 = 2
	340	Dispositivo señalador para objeto ESTILO (uno por elemento si el código 74 > 0)
	46	S = valor de escala (opcional). Puede haber varias entradas
	50	R = valor de rotación (opcional). Puede haber varias entradas
	44	X = desplazamiento en X (opcional). Puede haber varias entradas
	45	Y = desplazamiento en Y (opcional). Puede haber varias entradas
	9	Cadena de texto (uno por elemento si el código 74 = 2)
STYLE	100	Marca de subclase (AcDbTextStyleTableRecord)
	2	Nombre de estilo
	70	Valores de indicador estándar. Es la suma de: 1 = Si se establece, esta entrada describe una forma 4 = Texto vertical 16 = El estilo de texto es dependiente de una RefX. 32 = Se ha resuelto satisfactoriamente la RefX. 64 = Al menos una entidad del dibujo hizo referencia al estilo de texto la última vez que se editó el dibujo
	40	Altura de texto fija; 0 si no es fija
	41	Factor de anchura
	50	Ángulo de oblicuidad
	71	Indicadores de generación de texto. 2 = El texto mira hacia atrás (simetría en X) 4 = El texto mira hacia arriba (simetría en Y)
	42	Última altura utilizada
	3	Nombre de archivo de tipo de letra principal
	4	Nombre de archivo de tipos de letra grandes; vacío si no hay

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

UCS

100 Marca de subclase (AcDbUCSTableRecord)
2 Nombre de SCP

Tipo de tabla	Código	Descripción
	70	Valores de indicador estándar. Es la suma de: 16 = El SCP es dependiente de una RefX. 32 = Se ha resuelto satisfactoriamente la RefX. 64 = Al menos una entidad del dibujo hizo referencia al SCP la última vez que se editó el dibujo
	10	Origen (en SCU)
	11	Dirección del eje X (en SCU)
	12	Dirección del eje Y (en SCU)

VIEW

100 Marca de subclase (AcDbViewTableRecord)
2 Nombre de la vista
70 Valores de indicador estándar. Es la suma de:
1 = si se especifica, es una vista en espacio papel
16 = La vista es dependiente de una RefX.
32 = Se ha resuelto satisfactoriamente la RefX.
64 = Al menos una entidad del dibujo hizo referencia a la vista la última vez que se editó el dibujo

40 Altura de la vista (en SCV)
10 Punto central de la vista (en SCV)
41 Anchura de la vista (en SCV)
11 Línea de mira desde el punto de mira (en SCU)
12 Punto de mira (en SCU)
42 Longitud de lentes
43 Plano delimitador frontal (desplazamiento desde punto de mira)
44 Plano delimitador posterior (desplazamiento desde punto de mira)
50 Ángulo de ladeo
71 Modo de visualización (mismos valores que la variable de sistema VIEWMODE)

VPORT

100 Marca de subclase (AcDbViewportTableRecord)
2 Nombre de ventana gráfica
70 Valores de indicador estándar. Es la suma de:
16 = La ventana es dependiente de una RefX.
32 = Se ha resuelto satisfactoriamente la RefX.
64 = Al menos una entidad del dibujo hizo referencia a la ventana la última vez que se editó el dibujo

10 Esquina inferior izquierda de la ventana gráfica
11 Esquina superior derecha de la ventana gráfica
12 Punto central de la vista (en SCV)
13 Punto base de Forzcursor
14 Intervalo de Forzcursor X e Y
15 Intervalo de Rejilla X e Y
16 Línea de mira desde el punto de mira (en SCU)
17 Punto de mira de la vista (en SCU)
40 Altura de la vista
41 Proporción de aspecto de la ventana gráfica
42 Longitud de la lente
43 Plano delimitador frontal (desplazamiento desde punto de mira)
44 Plano delimitador posterior (desplazamiento desde el punto de mira)
0 Ángulo de rotación de Forzcursor
51 Ángulo de ladeo de la vista
68 Campo de estado

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

69	Número de ID (identificación)
71	Modo de visualización (mismos valores que la variable de sistema VIEWMODE)
72	Porcentaje de precisión de círculos
73	Valor de zoom rápido
74	Valor de SIMBSCP
75	Forzcursor activado/desactivado
76	Rejilla activada/desactivada
77	Estilo de Forzcursor
78	Isoplano de Forzcursor

TABLA 4. Códigos para definiciones de bloque

Tipo	Código	Descripción
BLOCK	0	Tipo de entidad (BLOCK)
	5	Identificador
	102	Inicio del grupo definido por la aplicación "{nombre_aplicación}". Por ejemplo, "{ACAD_REACTORS}" indica el inicio del grupo de reactivos permanentes de AutoCAD , y "{ACAD_XDICTIONARY}" indica el inicio de un grupo de extensión de diccionario
	<i>códigos def. por aplicación</i>	Los códigos y valores contenidos entre los grupos 102 están definidos por la aplicación
	102	Fin del grupo, "}" indica que terminan los valores definidos por la aplicación
	100	Marca de subclase (AcDbEntity).
	8	Nombre de capa
	100	Marca de subclase (AcDbBlockBegin).
	2	Nombre de bloque
	70	Indicadores de tipo de bloque. Es la suma de: 1 = Bloque sin nombre (sombreado, acotación asociativa, otras operaciones internas o una aplicación) 2 = Este bloque tiene definiciones de atributos 4 = Este bloque es una referencia externa (RefX.) 8 = Este bloque es una RefX. superpuesta 16 = Este bloque es externamente dependiente. 32 = El bloque es una referencia externa resuelta o depende de una referencia externa 64 = Esta definición es una referencia externa referenciada
	10	Punto base
	3	Nombre de bloque
	1	Nombre del camino de la referencia externa (sólo aparece si el bloque es una referencia externa)
ENDBLK	0	Tipo de entidad (ENDBLK)
	5	Identificador
	102	Inicio del grupo definido por la aplicación "{nombre_aplicación}". Por ejemplo, "{ACAD_REACTORS}" indica el inicio del grupo de reactivos permanentes de AutoCAD , y "{ACAD_XDICTIONARY}" indica el inicio de un grupo de extensión de diccionario
	<i>códigos def. por aplicación</i>	Los códigos y valores contenidos entre los grupos 102 están definidos por la aplicación.
	102	Fin del grupo, "}" indica que terminan los valores definidos por la aplicación
	100	Marca de subclase (AcDbBlockEnd)

TABLA 5. Códigos para objetos no gráficos

TABLA 5.1. Códigos comunes a todos los objetos no gráficos

Código	Descripción
0	Tipo de objeto (DICTIONARY o XRECORD)
5	Identificador
102	Inicio del grupo definido por la aplicación "{nombre_aplicación}". Por ejemplo, "{ACAD_REACTORS}" indica el inicio del grupo de reactivos permanentes de AutoCAD , y "{ACAD_XDICTIONARY}" indica el inicio un grupo de extensión de diccionario
<i>códigos def. por aplicación</i>	Los códigos y valores contenidos entre los grupos 102 están definidos por la aplicación
102	Fin del grupo, "}" indica que terminan los valores definidos por la aplicación

TABLA 5.2. Códigos comunes a todos los objetos de tipo diccionario

Código	Descripción
100	Marcador de subclase (AcDbDictionary)
3	Nombre de la entrada (uno para cada entrada)
350	Identificador del objeto de la entrada (uno por cada entrada)

TABLA 5.3. Códigos para cada tipo de objeto de diccionario

Tipo diccionario	Código	Descripción
GROUP	0	Nombre del objeto (GROUP)
	5	Identificador
	102	Inicio del grupo de reactivos permanentes de AutoCAD "{ACAD_REACTORS}". Aparece en todos los diccionarios salvo el principal
	330	Identificador suave del dispositivo señalador para un diccionario de propietarios
	102	Fin del grupo de reactivos permanentes, "}"
	100	Marcador de subclase (AcDbGroup)
	300	Descripción del grupo
	70	Indicador de "sin nombre": 1 = sin nombre; 0 = con nombre
	71	Indicador de seleccionabilidad: 1 = seleccionable; 0 = no seleccionable
	340	Identificador de la entidad en el grupo (una entrada por objeto)
MLINESTYLE	0	Nombre del objeto (MLINESTYLE)
	5	Identificador
	102	Inicio del grupo de reactivos permanentes de AutoCAD "{ACAD_REACTORS}". Aparece en todos los diccionarios salvo el principal
	330	Identificador suave del dispositivo señalador para un diccionario de propietarios
	102	Fin del grupo de reactivos permanentes, "}"
	100	Marcador de subclase (AcDbMlineStyle)
	2	Nombre de estilo de línea múltiple
	70	Indicadores. Suma de:
		1 = relleno activado 2 = visualizar ingletes 16 = Extremo inicial cuadrado (línea) 32 = Extremo inicial de arcos interiores

		64 = Extremo inicial redondeado (arcos exteriores)
		256 = Extremo final cuadrado (línea)
		512 = Extremo final de arcos internos
Tipo diccionario	Código	Descripción
		1024 = Extremo final redondeado (arcos exteriores)
	3	Descripción del estilo (cadena; máximo 255 caracteres)
	62	Color de relleno (por defecto = 256). Puede haber varias entradas; una entrada por cada elemento
	51	Ángulo inicial, por defecto es 90 (grados)
	52	Ángulo final, por defecto es 90 (grados)
	71	Número de elementos
	49	Desfase del elemento (número real). Puede haber varias entradas, una entrada por cada elemento
	62	Color del elemento (por defecto = 0). Puede haber varias entradas; una entrada por cada elemento
	6	Tipo de línea del elemento (por defecto = PorCapa)
		Puede haber varias entradas, una entrada por cada elemento

TABLA 5.4. Códigos comunes a todos los objetos de tipo XRECORD

Código	Descripción
100	Marcador de subclase (AcDbXrecord)
1 a 369	Estos valores están a disposición de las aplicaciones en cualquier forma (salvo 5 y 105)

ONCE.20.2. Funciones de gestión de la Base de Datos

Una vez vista toda la teoría acerca de la Base de Datos interna de **AutoCAD**, pasaremos a la práctica estudiando las funciones que nos permiten gestionarla. En este apartado vamos a tratar cuatro grupos de funciones distintos: funciones para trabajar con conjuntos designados, funciones que gestionan nombres de entidades, funciones para acceder a los datos de las entidades y funciones específicas para gestionar objetos no gráficos. Todas ellas, convenientemente combinadas (y con algunas funciones de manejo de listas concretas), nos darán acceso libre y directo a la Base de Datos de **AutoCAD**.

De todas formas no vamos a ver dichas funciones de un modo ordenado. Para sacarle más partido a la explicación las mezclaremos, de este modo podremos ir siguiendo paso a paso el modo de acceder a la Base de Datos.

ONCE.20.2.1. Crear un conjunto de selección

Si bien las funciones del tipo *GET...* nos permitían obtener del usuario puntos, distancias, ángulos y demás, evidente es que necesitaremos otro tipo de funciones con las que obtener entidades completas o grupos de entidades para su proceso. Este conjunto de funciones *SS...* son las que vamos a comenzar a ver con *SSGET*, aunque como ya hemos dicho no veremos las demás en orden temático.

```
(SSGET [modo] [punto1 [punto2]] [lista_puntos] [lista_filtros])
```

La función *SSGET* acepta el conjunto designado de entidades. Se puede utilizar sin parámetros así:

(SSGET)

SSGET presenta el mensaje Designar objetos: en línea de comandos (Select objects: en versiones inglesas de **AutoCAD**). La designación puede hacerse mediante cualquiera de los modos de designación permitidos: por un punto, una ventana, una captura, un polígono, etcétera. En el momento de pulsar INTRO, la función acepta del conjunto de designación, presentando un mensaje del tipo <Conjunto de selección: *n*> (<Selection set: *n*> en ingles), donde *n* es un número de orden entero que representa el conjunto de selección actual.

La forma lógica de utilización de SSGET pasa por el almacenamiento en una variable del conjunto de selección en cuestión, el cual luego podrá ser llamado por los diversos comandos de **AutoCAD** o por otra función de AutoLISP para su proceso. Por ejemplo:

```
(SETQ Conjunto (SSGET))  
(COMMAND "_.move" Conjunto "" "10,0" "")
```

NOTA: Recordemos la utilidad del punto (.) antes del comando por si estuviera redefinido (**MÓDULO SIETE**).

Esta rutina solicita al usuario una designación de objetos, los cuales serán recogidos en un conjunto de selección y guardados en la variable *Conjunto*. Posteriormente, se desplazan los objetos una distancia de 10 unidades de dibujo en el eje X.

NOTA: Si queremos hacer pruebas en línea de comandos para comprobar la funcionalidad de SSGET, una vez ejecutada la función, podemos hacer una llamada a cualquier comando que solicite designar objetos e introducir el nombre de la variable como conjunto de selección, pero con el signo de cerrar admiración (!) por delante. Desde rutinas o programas AutoLISP no es necesario incluir dicho símbolo, evidentemente.

NOTA: La diferencia entre SSGET y el comando DESIGNA es evidente. SSGET permite designar cualquier conjunto y guardarlo en una variable; de este modo podemos disponer de varios conjuntos de selección para ser utilizados. DESIGNA permite designar una serie de objetos que se incluirán en un conjunto de selección, el cual puede ser referenciado posteriormente mediante el modo *Previo* de selección; en el momento en que volvamos a hacer DESIGNA el conjunto anterior desaparece. Además no tiene sentido guardar en una variable el resultado de un DESIGNA ya que luego no funciona a la hora llamarlo para algún comando. DESIGNA viene muy bien para su utilización en macros, en los programas de AutoLISP usaremos como norma general SSGET.

El argumento *modo* especifica el método de designación. *modo* puede tener cualquiera de los siguiente valores:

- "P". Crea un conjunto de selección con el último conjunto de objetos previamente seleccionado. Equivale al modo de designación *Previo*.
- "U". Crea un conjunto de selección con la última entidad añadida a la Base de Datos del dibujo de las visibles en pantalla, es decir la última entidad dibujada y no borrada de las visibles en pantalla. Equivale al modo de designación *Último*.
- "I". Crea un conjunto de selección con el conjunto implícito designado (variable PICKFIRST de **AutoCAD** activada).
- *p1*. Crea un conjunto de selección con la entidad que pasa por el punto *p1*. Equivale a señalar ese punto en la pantalla. El resultado dependerá del modo o modos de referencia a objetos actuales, es decir del valor de la variable OSMODE.

- "V" *p1 p2*. Crea un conjunto de selección a partir de la *Ventana* cuyos vértices son los puntos *p1* y *p2*. Ambos puntos no se pueden omitir.
- "C" *p1 p2*. Crea un conjunto de selección a partir de la *Captura* cuyos puntos son los puntos *p1* y *p2*. Ambos puntos no se pueden omitir.
- "PV" *lista_puntos*. Crea un conjunto de selección a partir del *Polígono-Ventana* cuyos vértices son los puntos indicados en la lista. La lista no se puede omitir.
- "PC" *lista_puntos*. Crea un conjunto de selección a partir del *Polígono-Captura* cuyos vértices son los puntos indicados en la lista. La lista no se puede omitir.
- "B" *lista_puntos*. Crea un conjunto de selección a partir del *Borde* cuyos vértices son los puntos indicados en la lista. La lista no se puede omitir.
- "X". Crea un conjunto de selección todas las entidades de la Base de Datos, visibles o no visibles en pantalla. Equivale al modo de designación *Todo*.

NOTA: Los valores entre comillas son cadenas que deben indicarse como tales.

NOTA: Los valores entre comillas representan a los modos de designación de **AutoCAD** y se introducen como cadenas por ser una llamada a dichos modos. Es por ello, que en versiones idiomáticas diferentes a la castellana han de indicarse de forma conveniente. Por ejemplo, en lugar de "Ú" indicar "L", para *Último*; es factible la sintaxis "_L" , por ejemplo, para cualquier versión idiomática del programa.

Veamos un ejemplo sencillo. La siguiente rutina dibuja una línea en pantalla y luego la borra:

```
(COMMAND "_line" "0,0" "100,100" "")  
(COMMAND "_erase" (SSGET "_l") "")
```

Otro ejemplo; éste dibuja un rectángulo y luego lo borra también:

```
(COMMAND "_rectang" "100,100" "300,300")  
(SETQ Conjunto (SSGET "_c" '(100 100) '(300 300)))  
(COMMAND "_erase" Conjunto "")
```

Además de todo esto, disponemos de la posibilidad de introducir filtros de selección. Estos filtros han de ser listas de asociaciones que filtran o realizan una criba de los objetos según determinadas condiciones, quedándose con aquellas entidades de la Base de Datos que cumplen dichas condiciones.

Se puede añadir una lista de filtros a cualquiera de las modalidades de selección expuesta arriba. Los filtros de selección se añaden detrás de los parámetros propios de selección (como "P", "V" *p1 p2* o "X").

Las listas de filtros hacen referencia a las propiedades de la entidad, como el color, su capa, tipo de línea, etc. O también a puntos u otras características. Para construir una lista de filtro deberemos construir la propia lista con cada una de sus sublistas, las cuales serán las características o propiedades de las entidades que queremos filtrar. Las sublistas pueden ser pares puntuados contruidos con CONS o no.

Así, un ejemplo muy típico es aquel que permite seleccionar todos los objetos de un dibujo que tengan unas determinadas características. Por ejemplo, para designar todos los

círculos del dibujo actual, que además estén en la capa `PIEZA` y tengan asignado el color rojo, haríamos:

```
(SSGET "x" (LIST (CONS 0 "CIRCLE")
                  (CONS 8 "Pieza")
                  (CONS 62 1)
                )
)
```

Construimos pues una lista con `LIST` que recoge las condiciones del filtro, que no son otra cosa sino propiedades de la entidad. AutoLISP explorará toda ("x") la Base de Datos del dibujo actual y seleccionará ("`SSGET`") las entidades que posean dichas propiedades.

Otro ejemplo puede ser la designación o selección de todas las líneas que comiencen en un punto:

```
(SSGET "x" (LIST (CONS 0 "LINE")
                  '(10 10.0 10.0 0.0)
                )
)
```

Como se ve aquí, la segunda condición no es par punteado, ya que dice relación a los puntos iniciales de las líneas (en este caso de coordenadas $X = 10$, $Y = 10$ y $Z = 0$), por lo que se construye como una lista normal (con el apóstrofo ' de literal).

Las listas de filtros pueden ser enteramente construidas como literales también. En el primero de los ejemplos sería así:

```
(SSGET "x" '((0 . "CIRCLE")
              (8 . "Pieza")
              (62 . 1)
            )
)
```

En estos casos habremos de describir correctamente la notación de los pares punteados, es decir: el primer valor, un espacio, el punto, otro espacio y el segundo valor, todo ello entre paréntesis. Aunque resulta más elegante y sencillo, quizás, hacer mediante `CONS`.

En principio, cada elemento de una lista de filtros de selección se añade como una condición más que se debe cumplir. Sin embargo, existe la posibilidad de añadir operadores relacionales y booleanos a estos filtros. Esto se realiza con el código especial `-4`, por ejemplo:

```
(SSGET "x" (LIST (CONS 0 "TEXT")
                  (CONS -4 "<=")
                  (CONS 40 5)
                )
)
```

El operador relacional, que ha de ser una cadena, se aplica a la condición que le siga en la lista. En este último ejemplo, `SSGET` selecciona entidades de texto cuya altura (código 40) sea menor o igual ("`<=`") que 5 unidades de dibujo.

La tabla siguiente muestra cuáles son los operadores relacionales que se pueden incluir en los filtros, con su correspondiente descripción:

Operador relacional	Descripción
" * "	Cualquier valor (siempre verdadero)

" = "	Igual que
" ! = "	Distinto de
Operador relacional	Descripción
" + / = "	Distinto de
" < > "	Distinto de
" < "	Menor que
" < = "	Menor o igual que
" > "	Mayor que
" > = "	Mayor o igual que
" & "	AND binario (sólo grupos de números enteros)
" & = "	Igual a enmascarado binario (sólo grupos de números enteros)

No se puede especificar un nombre de capa menor o igual que otro (ni por orden alfabético), por ejemplo, por lo que estos operadores se aplican sólo a datos numéricos. Para establecer filtros con los datos textuales o alfanuméricos se utilizan los caracteres comodín explicados con la función WCMATCH, en la sección **ONCE.12..**

Para condiciones que afectan a puntos es factible agrupar operadores en grupos de tres, separados por comas. Por ejemplo:

```
(SSGET "x" (LIST (CONS 0 "LINE")
                  (CONS -4 "<,>,*")
                  '(11 10 100)
                )
)
```

En este ejemplo se buscan todas las líneas cuyas coordenadas de punto final sean: en X menores que 10, en Y mayores que 100 y sus coordenadas Z cualesquiera.

Como hemos dicho, además de operadores relacionales, los cuales afectan sólo a la siguiente condición, se pueden agrupar condiciones mediante operadores booleanos, empleando un operador de comienzo y otro de final. Estos operadores también se especifican con el código -4 y son:

Operador booleano inicial	Encierra...	Operador booleano final
" < AND "	uno o más operandos	" AND > "
" < OR "	uno o más operandos	" OR > "
" < XOR "	dos operandos	" XOR > "
" < NOT "	un operando	" NOT > "

En el ejemplo siguiente se designarán todos los textos de la capa NOTAS y todos los arcos de radio 10:

```
(SSGET "X" '((-4 . "<OR")
              (-4 . "<AND")(0 . "TEXT")(8 . "NOTAS")(-4 . "<AND>")
              (-4 . "<AND")(0 . "ARC") (40 . 10) (-4 . "<AND>")
              (-4 . "<OR>")
            )
)
```

Los conjuntos de selección ocupan archivos temporales de **AutoCAD**. Por esa razón existe una limitación en cuanto al número máximo de conjuntos almacenados en variables,

que es de 128 a la vez. Una vez alcanzado este límite, `SSGET` rechaza la posibilidad de crear un nuevo conjunto y devuelve `nil`. Para acceder a más conjuntos de selección es preciso eliminar alguno de los almacenados poniendo la variable a `nil`.

ONCE.20.2.2. Obtener el nombre de una entidad

Una vez seleccionadas las entidades y agrupadas en un conjunto de selección, deberemos extraer el nombre de la que o las que nos interesen para trabajar con ellas.

Como sabemos, en la Base de Datos la entidades se encuentran referenciadas mediante un nombre que es en realidad un número hexadecimal que indica la posición en memoria. Este nombre es único para cada entidad en cada sesión de dibujo, de forma que la identifica inequívocamente y la distingue de todas las demás del dibujo.

Por todo ello, para capturar una entidad, y solo esa, de un conjunto de selección lo más lógico es referirse a su nombre, y para extraer su nombre disponemos de la función `SSNAME`.

`(SSNAME conjunto índice)`

`SSNAME` devuelve el nombre de entidad (código `-1`), del conjunto de selección especificado, situada en el lugar indicado por el índice. Las entidades dentro de un conjunto comienzan a numerarse por el `0`. Así, en un conjunto con cuatro entidades, éstas estarían numeradas de `0` a `3`. Veamos un ejemplo:

```
(SETQ Entidad (SSNAME (SSGET) 0))
```

De esta forma designaríamos una o varias entidades en pantalla cuando nos lo pidiera `SSGET` y el nombre de la primera de ellas (`SSNAME` con índice `0`) se guardaría en la variable `Entidad`.

Una vez obtenido el nombre de una entidad, puede ser procesado por otras funciones que veremos a continuación.

NOTA: Los nombres que devuelve `SSNAME` son sólo de entidades principales; esta función no puede acceder a las entidades que forman parte de una polilínea o un bloque.

ONCE.20.2.3. Extraer la lista de una entidad

Ahora que ya hemos designado una entidad y que podemos conocer su nombre, sólo queda extraer su lista de definición para poder modificarla o editarla. Para ello disponemos de la función `ENTGET`.

`(ENTGET nombre_entidad [lista_aplicaciones])`

Esta función `ENTGET` busca en la Base de Datos el nombre indicado de una entidad y devuelve la lista completa correspondiente a esa entidad. Se observa que se requiere el nombre de una entidad (código `-1` en Base de Datos) y por lo tanto se hace necesario obtenerlo previamente con `SSNAME` —u otras que veremos—.

El ejemplo típico de descubrimiento de esta función (combinada con las anteriores) es:

```
(SETQ ListaEntidad (ENTGET (SSNAME (SSGET) 0)))
```

Con este ejemplo obtendremos la lista de definición en Base de Datos de cualquier entidad que designemos en pantalla y lo guardaremos en la variable `ListaEntidad` para su posterior proceso; si designamos varias sólo aceptará la primera de ellas (debido al índice 0 de `SSNAME`, como sabemos). Una lista de un círculo designado, por ejemplo, podría ser:

```
((-1 . <Nombre de objeto: 2770500>)
 (0 . "CIRCLE")
 (5 . "20")
 (100 . "AcDbEntity")
 (67 . 0)
 (8 . "0")
 (100 . "AcDbCircle")
 (10 144.409 168.958 0.0)
 (40 . 17.2339)
 (210 0.0 0.0 1.0)
 )
```

NOTA: Ahora podremos comprobar de manera factible toda la teoría explicada anteriormente sobre la Base de Datos de **AutoCAD**.

La manera más sencilla y utilizada de acceder a los datos de una entidad es mediante `ASSOC` (ya estudiada), para obtener la lista de asociaciones deseada, y luego con `CAR`, `CDR`, `CADR`, `CADDR`, etc., para capturar sus componentes (recordemos que funciones del tipo `NTH` no funcionan directamente con pares punteados).

De este modo, en el ejemplo anterior, y si hubiésemos designado el círculo cuya lista se propone, podríamos extraer la capa en la que se encuentra directamente:

```
(SETQ Capa (CDR (ASSOC 8 ListaEntidad)))
```

Lo que hacemos aquí es guardar en `Capa` el segundo elemento (`CDR`) de la sublista de asociación cuyo primer elemento (`ASSOC`) sea 8 (código para la capa).

NOTA: Recuérdese la necesidad de utilizar `CDR` y no `CADR` para capturar el segundo elemento de un par punteado.

Si quisiéramos extraer ahora por ejemplo la coordenada Y del centro del círculo haríamos:

```
(SET CentroY (CADDR (ASSOC 10 ListaEntidad)))
```

Ya que esta lista no es par punteado, el primer elemento saldría con `CAR` (el código 10 de coordenadas del centro para un círculo), el segundo con `CADR` (la coordenada X) y el tercero (coordenada Y) con `CADDR`. Para la coordenada Z utilizaríamos `CADDR`.

NOTA: Como se explicó en su momento, las coordenadas de los puntos de las entidades se expresan en el Sistema de Coordenadas de Entidad o de Objeto (SCE o SCO según la denominación adoptada). En la mayoría de las entidades dibujadas en el Sistema de Coordenadas Universal SCU, las coordenadas en la base de datos coinciden con las universales. Pero si se han dibujado en cualquier Sistema de Coordenadas Personal (SCP), se hace necesario recurrir a la función `TRANS` para efectuar las conversiones.

Por último decir que el argumento opcional `lista_aplicaciones` de `ENTGET` permite incluir en la lista devuelta los datos extendidos de entidades (los que siguen al código específico -3). De este tema hablaremos en la sección **ONCE.20.2.14..**

ONCE.20.2.4. Actualizar lista y Base de Datos

Una vez accedido al elemento que nos interesa, deberemos realizar los cambios necesarios en la lista para modificar la entidad. Para ello utilizaremos básicamente funciones del tipo SUBST o APPEND, ya explicadas.

NOTA: Recordar que nos estamos moviendo entre listas y que cualquier función de manejo de listas nos puede ser útil: CONS, LIST, etcétera.

Así pues, retomemos el ejemplo anterior del círculo. Tras escribir:

```
(SETQ ListaEntidad (ENTGET (SSNAME (SSGET) 0)))
```

y designar un círculo, AutoLISP nos devuelve:

```
((-1 . <Nombre de objeto: 2770500>)
 (0 . "CIRCLE")
 (5 . "20")
 (100 . "AcDbEntity")
 (67 . 0)
 (8 . "0")
 (100 . "AcDbCircle")
 (10 144.409 168.958 0.0)
 (40 . 17.2339)
 (210 0.0 0.0 1.0)
 )
```

Si ahora quisiéramos, por ejemplo, cambiar la capa del círculo, habríamos de hacer:

```
(SETQ ListaEntidad (SUBST (CONS 8 "Ejes") (CONS 8 "0") ListaEntidad))
```

NOTA: Si la capa no existe se crea.

Como hemos de saber ya, CONS nos devolverá la nueva lista renovada. Si ocurre algún error, CONS devolverá la lista sin renovar.

Pero si quisiéramos en este caso cambiar el color del círculo, no podríamos utilizar CONS, ya que la lista del color (código 62) no aparece porque es PorCapa, por lo que no podemos decir que sustituya una nueva lista por otra que no existe.

En estos casos se utiliza APPEND, que nos permite añadir nuevos elementos a la lista por su manera de actuar. Así pues, para cambiar el círculo de color haríamos:

```
(SETQ ListaEntidad (APPEND ListaEntidad (LIST (CONS 62 1))))
```

NOTA: Todo ello en línea de comandos.

NOTA: Procúrese con APPEND añadir a la lista de una entidad una nueva lista, y no una nueva lista a la lista de una entidad. Es decir, la lista general irá antes, como argumento de APPEND, que la lista que debe agregarse. Si se realiza esto al revés, la nueva lista se añadirá por delante a la lista de la entidad y esto hará que no funcione correctamente. Como norma general los dos primeros elementos de una lista de entidad habrán de ser el nombre (con código -1) y el tipo (código 0), respectivamente; con las restantes sublistas (capa, color, punto inicial, centro, tipo de línea...) no importa el orden generalmente.

La razón de tener que formar una lista con la propia lista de asociación del color es que, si no lo hacemos, APPEND añade los valores de la lista sin asociar y provoca un error bad list.

Si recordamos, `APPEND` toma los componentes de las listas y los junta todos en una. Si hacemos que la lista de par punteado se encuentre dentro de otra lista, `APPEND` tomará el par punteado como un solo elemento y se lo añadirá al resto (que también son sublistas) de la lista de la entidad.

NOTA: Repásense estas funciones en la sección **ONCE.17.**

Hemos de comprender que actualizar así la variable que contiene la lista de la entidad no actualiza el dibujo. Esta variable contiene una copia de la definición de la entidad en la Base de Datos, pero no la definición propiamente dicha.

Para actualizar pues, y como paso último, la Base de Datos de **AutoCAD** y que los objetos se vean realmente modificados, debemos recurrir a una función que se encarga de ello:

`(ENTMOD lista_entidad)`

`ENTMOD` pues actualiza la lista de una entidad en la Base de Datos de **AutoCAD**. Su funcionamiento es tan sencillo como pasarle como argumento único la lista de la entidad que hay que modificar, y ella se encarga del resto.

Así pues, en el ejemplo del círculo que venimos arrastrando sólo quedaría escribir:

```
(ENTMOD ListaEntidad)
```

para que ese círculo cambiara su capa y su color.

El funcionamiento principal de modificación de las entidades de la Base de Datos se basa en los pasos que hemos venido siguiendo, esto es, la designación de la entidad o entidades que queremos tratar, la obtención de sus nombres y con ello sus listas, la modificación de las mismas y, por último, la actualización de la Base de Datos mediante `ENTMOD`. El resto de las funciones que veremos aquí se refieren a otros tipos de extraer nombres de entidades o sus listas, o de trabajar con los conjuntos.

La función `ENTMOD` presenta algunas restricciones en cuanto al tipo de dato que puede actualizar para una entidad. No puede modificar ni el nombre de entidad (código -1) ni el tipo (código 0), evidentemente. Si se modifica el nombre de estilo de texto, tipo de línea o nombre de bloque, estos deben estar previamente definidos o cargados en el. Si se modifica el nombre de capa en cambio, se crea una nueva capa si no existiera previamente —como hemos podido comprobar—. Si se modifica la lista de una entidad principal, se actualiza su imagen en pantalla. Si se modifica la lista de una subentidad (vértices de polilínea o atributos) la imagen en pantalla no cambia hasta que se utiliza `ENTUPD` (la veremos a continuación).

No se pueden modificar con `ENTMOD` entidades de ventanas gráficas (tipo `VIEWPORT`). Tampoco las entidades incluidas en la definición de un bloque.

`(ENTUPD nombre_entidad)`

Como hemos comentado pues, mediante `ENTMOD` se actualiza en la Base de Datos una entidad a la que se le han cambiado sus características. Si se trata de una entidad principal, `ENTMOD` regenera directamente la entidad y ésta se visualiza en pantalla ya actualizada. Pero si se modifica un componente de una entidad compuesta —como vértices de una polilínea o un atributo de un bloque—, aunque se actualice la Base de Datos el aspecto de la entidad no cambiará en pantalla hasta que se produzca una regeneración general de todo el dibujo.

Mediante `ENTUPD`, indicando simplemente el nombre de la entidad (por ejemplo, el vértice de una polilínea modificada), se busca cuál es la cabecera de esa entidad y se regenera, con lo que se actualiza su aspecto en pantalla. En general, `ENTUPD` regenera la entidad cuyo nombre se especifique, incluidas todas las subentidades.

Pues llegados a este punto, ya podemos ver algún ejemplo un poco más trabajado. El siguiente es un ejemplo típico de toda la vida. Un programa AutoLISP que permite cambiar la capa actual de trabajo simplemente designando un objeto que se encuentre en la capa a la que queremos cambiar. Además, y para introducir una pequeña variación, las demás capas serán desactivadas. El listado sencillo es el siguiente:

```
(DEFUN C:DesCapa ()
  (SETQ ListaEnt (ENTGET (SSNAME (SSGET) 0)))
  (SETQ ListaCapa (ASSOC 8 ListaEnt))
  (SETQ Capa (CDR ListaCapa))
  (SETVAR "CLAYER" Capa)
  (COMMAND "_.layer" "_off" "*" "_n" "")
)
```

Como podemos ver, aquí se crea un nuevo comando de **AutoCAD** llamado `DESCAPA` para realizar lo propuesto. Lo primero es solicitar al usuario un objeto (`SSGET`), si se seleccionan varios únicamente se elige el primero después (índice 0 de `SSNAME`), y se guarda su lista de especificación (`ENTGET`) en la variable `ListaEnt`. A continuación se extrae la lista de la capa (código 8 con `ASSOC`) de la lista completa `ListaEnt` y se guarda en `ListaCapa`, y luego se almacena en la variable `Capa` la capa en cuestión, que es el segundo elemento (`CDR`) del par punteado que guarda la capa (`ListaCapa`). Por último se establece dicha capa como actual con la variable `CLAYER` y se desactivan todas las demás (exceptuando la actual, como decimos) con el comando `CAPA` (`LAYER` en inglés) de **AutoCAD**.

NOTA: Al recibir el comando `CAPA` desde un archivo `.LSP`, **AutoCAD** lo activa en su modo de línea de comandos automáticamente, por lo que no es necesario indicar `-CAPA` (o incluso `_.-LAYER`). Esa notación es más que nada para macros, aunque puede ser interesante incluirla aquí para no perder la costumbre.

En el ejemplo que acabamos de ver, es factible seleccionar más de un objeto y, aunque sólo se elija el primero después, no parece lógico utilizar este método en este caso. A continuación conoceremos una función que nos abrirá los ojos un poco más en este sentido.

ONCE.20.2.5. Nombre de entidad por un punto

Existen diversos comandos de **AutoCAD** que procesan entidades teniendo en cuenta el punto de designación con el que se ha actuado sobre las mismas, es el caso por ejemplo del comando `PARTE`. Veamos la sintaxis de la siguiente función:

```
(ENTSEL [mensaje_solicitud])
```

La función `ENTSEL` espera a que el usuario designe una única entidad mediante un punto y devuelve una lista cuyo primer elemento es el nombre de la entidad (código -1) designada, y su segundo elemento las coordenadas X, Y y Z del punto de designación. De esta forma se tienen asociados ambos valores para procesarlos posteriormente.

Esta lista devuelta por `ENSEL` se puede indicar en las llamadas a los comandos de **AutoCAD que requieren señalar una entidad por un punto. Así por ejemplo, si en línea de comandos hacemos:**

```
(SET PuntoParte (ENTSEL))
```

y marcamos un punto de una línea, por ejemplo. Y después ejecutamos el comando PARTE, y como primer punto para seleccionar objeto le decimos:

```
!PuntoParte
```

dicho punto será aceptado por el comando de manera normal.

Se puede indicar una cadena de texto como argumento opcional de ENTSEL; esta cadena es un mensaje de solicitud. Por ejemplo:

```
(SETQ NomEnt (ENSET "Designar entidad por un punto: "))
```

Si no se especifica un mensaje, ENTSEL presenta el mismo que SSGET o DESIGNA, pero en singular, es decir, si para esos dos comentados era Designar objetos:, para ENTSEL es Designar objeto: (Select object: en inglés).

Una vez visto ENTSEL nos parecerá más lógico utilizarlo para el ejemplo anterior DESCAPA. El ejercicio tratado con ENTSEL, y un poco más trabajado, se presenta a continuación:

```
(DEFUN C:Iracapa (/ Refent0 error0 Ent EntName Capa)
  (SETVAR "cmdecho" 0)
  (SETQ error0 *error* *error* Errores)
  (SETQ Refent0 (GETVAR "osmode"))(SETVAR "osmode" 0)
  (WHILE (NOT (SETQ Ent (ENTSEL "\nDesignar objeto: "))))
  (SETQ EntName (CAR Ent))
  (SETQ Capa (CDR (ASSOC 8 (ENTGET EntName))))
  (COMMAND "_.-layer" "_s" Capa "")
  (SETVAR "osmode" Refent0)
  (SETVAR "cmdecho" 1)(SETQ *error* error0)(PRIN1)
)

(DEFUN C:IC ()
  (C:Iracapa)
)

(DEFUN Errores (Mensaje)
  (SETQ *error* error0)
  (PRINC (STRCAT "\nError: " Mensaje))
  (SETVAR "osmode" Refent0)
  (SETVAR "cmdecho" 1)
  (PRIN1)
)

(PROMPT "Nuevo comando IRACAPA definido. Alias IC.")(PRIN1)
```

El ejemplo es tan sencillo que se ha definido todo él en una sola orden de usuario nueva.

Tras designar el objeto por un punto (ENTSEL) y guardar su nombre y punto de designación (que es lo que devuelve ENTSEL) en Ent, se guarda en EntName únicamente su nombre, extrayendo el primer elemento de la lista con CAR.

A continuación se guarda en Capa el segundo elemento (CDR) del par punteado que comience con 8 (ASSOC) dentro de la lista de la entidad (ENTGET) cuyo nombre es EntName.

Por último se define la capa como actual, esta vez con el propio comando `CAPA` y se acaba el programa. El resto se corresponde con el procedimiento de rigor de control de errores y control de variables de **AutoCAD**.

ONCE.20.2.6. Añadir, eliminar y localizar entidades

Veremos ahora tres funciones que nos permitirán añadir, eliminar y localizar una entidad dentro de un conjunto de selección. Pero antes, expliquemos un poco otra que nos permite averiguar el número de entidades de un conjunto. Su sintaxis es:

`(SLENGTH conjunto)`

Como decimos, `SLENGTH` determina el número de entidades que existen en el conjunto de selección indicado. El número es siempre entero positivo, salvo si es mayor de 32.767, en cuyo caso es un número real. Veamos un ejemplo simple:

```
(SETQ Conjunto (SSGET "_1")) (SLENGTH Conjunto)
```

`SLENGTH` devolverá siempre 1 en este caso, ya que `SSGET` almacena en `Conjunto` el último ("`_1`") objeto dibujado y visible. Sencillo.

Por otro lado, para añadir una entidad a un conjunto de selección ya existente se utiliza la función `SSADD`. Su sintaxis es la siguiente:

`(SSADD [nombre_entidad [conjunto]])`

Si se emplea sin ningún argumento construye un conjunto de selección vacío, si elementos. Si se indica sólo un nombre de entidad, construye un conjunto de selección que contiene sólo esa entidad. Si se especifica un nombre de entidad y también un conjunto de selección existente, añade la entidad al conjunto, con lo que este pasa a tener un elemento más.

La función `SSADD` siempre devuelve un valor de conjunto. Si se indica sin argumentos o sólo con el nombre de una entidad, dado que crea un conjunto nuevo, devuelve su valor. Si se especifica un conjunto ya existente, devuelve ese mismo valor especificado puesto que el efecto es añadirle una entidad, pero el identificador del conjunto sigue siendo el mismo.

El siguiente ejemplo muestra el funcionamiento de `SSADD`:

```
(DEFUN C:BorraEnt ()
  (SETQ Entidades (SSGET))
  (SETQ NuevaEntidad (CAR (ENTSEL "Designar objeto que se añadirá: ")))
  (SSADD NuevaEntidad Entidades)
  (COMMAND "_erase" Entidades "")
)
```

Este programa permite designar un conjunto de objetos en pantalla con `SSGET` para luego añadir uno al conjunto, seleccionándolo mediante `ENTSEL`. `SSADD` añade el nuevo objeto al conjunto de selección `Entidades` ya existente, lo cual se puede comprobar cuando en la última línea se borran los objetos del conjunto.

`(SSDEL nombre_entidad conjunto)`

`SSDEL`, por su lado, elimina la entidad, cuyo nombre se especifique, del conjunto de selección indicado. Digamos que es el proceso contrario a `SSADD`.

El nombre del conjunto sigue siendo el mismo y por eso `SSDEL` devuelve ese nombre. Si la entidad no existe en el conjunto, `SSDEL` devuelve `nil`.

El siguiente ejemplo de `SSDEL` es el contrario del anterior:

```
(DEFUN C:BorraEnt2 ()
  (SETQ Entidades (SSGET))
  (SETQ ViejaEntidad (CAR (ENTSEL "Designar objeto que se eliminará: ")))
  (SSDEL ViejaEntidad Entidades)
  (COMMAND "_erase" Entidades "")
)
```

Este programa permite designar un conjunto de objetos en pantalla con `SSGET` para luego eliminar uno del conjunto, seleccionándolo mediante `ENTSEL`. `SSDEL` elimina el nuevo objeto del conjunto de selección `Entidades` ya existente, lo cual se puede comprobar cuando en la última línea se borran los objetos del conjunto.

`(SSMEMB nombre_entidad conjunto)`

Esta última función de esta sección examina el conjunto de selección especificado para ver si la entidad cuyo nombre se indica está en él. Si la encuentra devuelve su nombre, si no devuelve `nil`.

ONCE.20.2.7. Aplicar y determinar pinzamientos

`(SSSETFIRST conjunto_pinzamientos conjunto_seleccionados)`

Esta función aplica pinzamientos a los conjuntos especificados. Los pinzamientos se aplican al primer conjunto, pero sin que queden sus entidades seleccionadas. Si se indica un segundo conjunto, además de aplicar pinzamientos, sus entidades quedan seleccionadas. Si hay entidades comunes en ambos conjuntos, sólo se pinza y selecciona del segundo, ignorándose el primero. La función devuelve una lista con los dos conjuntos de selección.

`(SSGETFIRST)`

Esta función devuelve una lista con dos conjuntos de selección: el primero con todas las entidades del dibujo que tienen aplicados pinzamientos pero sin estar seleccionadas, y el segundo con todas las entidades que además de tener aplicados pinzamientos están seleccionadas.

ONCE.20.2.8. Obtener nombre con modo de selección

`(SSNAMEX conjunto [índice])`

Esta función, al igual que `SSNAME`, devuelve el nombre de la entidad del conjunto especificado que se encuentre en el lugar determinado por *índice*. La diferencia con la otra función de nombre casi igual, es que `SSNAMEX`, junto con dicho nombre, devuelve también los datos que describen el tipo de selección realizada sobre dicha entidad por el usuario.

Los datos son devueltos en forma de una lista con sublistas, una para cada entidad, de la forma:

```
(id_selección nombre_entidad (datos))
```


Cada una de estas sublistas tiene tres elementos, como vemos. Dichos elementos se comentan detalladamente a continuación.

— *id_selección* es un número identificador del método de selección empleado con la entidad. Los números posibles se indican en la siguiente tabla:

Identificador	Descripción
0	Selección no específica (<i>Último, Previo, Todo...</i>)
1	Designación mediante cursor
2	<i>Ventana</i> o <i>Polígono-Ventana</i>
3	<i>Captura</i> o <i>Polígono-Captura</i>
4	<i>Borde</i>

— *nombre_entidad* es el número hexacimal que es nombre de la entidad; el del código -1.

— *datos* es el conjunto de datos de información sobre el tipo de selección para la entidad. Si la selección ha sido no específica (identificador 0) no hay datos añadidos. Si ha sido señalado directamente con el cursor (identificador 1), se ofrece una lista con el punto de designación. Dependiendo del punto de vista 3D, el punto de designación se puede representar como una lista infinita, un rayo (semi-infinita) o un segmento de línea (finita). El punto en cuestión se ofrece en una lista con sus tres coordenadas, precedida esta lista por un código que puede ser uno de los siguientes:

Código	Descripción
0	Línea infinita
1	Rayo semi-infinito
2	Segmento de línea finito

Además, a las tres coordenadas sigue un vector opcional que describe la dirección de la línea infinita o el desplazamiento al otro extremo del segmento de línea. Si se omite en el listado, significa que el punto de vista es en planta. Por ejemplo, la siguiente lista es devuelta para un objeto designado en el punto de X = 50 e Y = 50 en una vista en planta del SCU:

```
(1 <Nombre de objeto: 26a0b07> (0 (50.0 50.0)))
```

Si la selección es del tipo *Ventana* o *Captura*, sus datos se ofrecen en una lista que empieza por un identificador de polígono con número negativo (-1, -2, -3, etc.) y después sublistas con todos los vértices del rectángulo o polígono, indicándose en cada una el tipo de punto (0, 1 ó 2) y las tres coordenadas. Por ejemplo, la siguiente lista es devuelta por una designación de *Captura* en una vista en planta:

```
((3 <Nombre de objeto: 26a0c12> -1 ) ((-1 (0 (20.0 10.0 0.0)))  
                                     (0 (50.0 10.0 0.0))  
                                     (0 (50.0 40.0 0.0))  
                                     (0 (20.0 40.0 0.0))  
                                    )  
 )
```

Si la selección es del tipo *Borde*, los datos son una lista de puntos de intersección entre la línea de *Borde* y la entidad. Por ejemplo, la siguiente lista es devuelta por una designación de *Borde*, que ha intersectado con la entidad en el punto 32,56.

```
(4 <Nombre de objeto: 26a5c09> (0 (32.0 56.0 0.0)))
```

ONCE.20.2.9. Otras formas de obtener nombres

Como hemos comprobado para acceder a una entidad en la Base de Datos necesitamos su nombre. Funciones que lo extraigan hemos visto hasta ahora `SSNAME`, `SSNAMEX` y `ENTSEL`, pero existen también otras que cumplen esa misión dependiendo de la situación en la que nos encontremos. Veámoslas.

`(ENTNEXT [nombre_entidad])`

Esta función devuelve el nombre de la primera entidad de la Base de Datos que sigue a la entidad cuyo nombre se indica en `nombre_entidad`. Como vemos, este argumento o parámetro de la sintaxis de `ENTNEXT` es opcional, y es que si no lo especificamos, la función devuelve el nombre de la primera entidad no eliminada de la Base de Datos.

Un ejemplo típico de uso de `ENTNEXT` es la de rastrear de modo repetitivo la Base de Datos del dibujo para obtener todos los nombres de todas las entidades de dibujo (de un dibujo no muy extenso, claro está), o los de las tres primeras, por ejemplo:

```
(SETQ Entidad1 (ENTNEXT))  
(SETQ Entidad2 (ENTNEXT Entidad1))  
(SETQ Entidad3 (ENTNEXT Entidad2))
```

`ENTNEXT` no sólo accede a las entidades principales, sino también a las contenidas en ellas. Así pues, otro ejemplo más lógico y utilizado se refiere a la extracción de todos los vértices de una polilínea no optimizada, o los atributos de un bloque. Se puede poner como condición que recorra todas las entidades hasta que encuentre `"SEQUEND"`.

Si lo que queremos es saber a qué polilínea de antigua definición pertenece determinado vértice, habrá que ir explorando todas las polilíneas del dibujo hasta dar con alguna característica de dicho vértice. En dicho momento, y tras la entidad `"SEQUEND"`, se extrae el valor del código -2 que da el nombre de la entidad principal.

`(ENTLAST)`

Esta función devuelve el nombre de la última entidad principal no eliminada de la Base de Datos. Sólo entidades principales.

Habitualmente se utiliza para capturar el nombre de una entidad recién dibujada mediante `COMMAND` desde un programa AutoLISP. Presenta la gran ventaja de que obtiene el nombre de la entidad aunque no sea visible en pantalla o se encuentre en una capa inutilizada.

En el siguiente ejemplo, la variable `EntidadLínea` almacena el nombre de la entidad tipo línea recién dibujada, sin necesidad de tener que designarla. Con este nombre ya es posible acceder a la Base de Datos y realizar las operaciones convenientes:

```
(COMMAND "_line" "0,0" "100,100" "")  
(SETQ EntidadLínea (ENTLAST))
```

`(NENTSEL [mensaje_solicitud])`

Esta función permite acceder en la Base de Datos a una entidad que se encuentre formando parte de una entidad compuesta (polilínea no optimizada o bloque). La cadena de texto opcional es el mensaje para la solicitud de designación de entidad.

Cuando la entidad que se designa no forma parte de otra compuesta, `NENTSEL` devuelve la misma información que `ENTSEL`, es decir, una lista cuyo primer elemento es el nombre de la entidad y su segundo elemento el punto de designación.

Cuando con `NENTSEL` se designa un componente de una polilínea no optimizada, devuelve una lista cuyo primer elemento es el nombre de la subentidad, es decir el vértice (tipo de entidad "VERTEX") inicial del segmento de polilínea designado. El segundo elemento de la lista sigue siendo el punto de designación. Por ejemplo:

```
(NENTSEL "Designar segmento de polilínea: ")
```

podría devolver:

```
(<Nombre de objeto: 26b004e> (5.65 6.32 0.0))
```

Cuando con `NENTSEL` se designa un componente de un bloque devuelve una lista con cuatro elementos:

- El primer elemento es el nombre de la entidad componente del bloque, extraída de la tabla de símbolos con la definición de ese bloque.
- El segundo elemento es una lista con las coordenadas del punto de designación.
- El tercer elemento es una lista que contiene a su vez cuatro listas: es la matriz de transformación del Sistema de Coordenadas Modelo (SCM) al Universal (SCU). El Sistema de Coordenadas Modelo (SCM) es aquél al que están referidas todas las coordenadas en la definición del bloque. Su origen es el punto de inserción del bloque. La matriz de transformación permite trasladar las coordenadas de la definición del bloque al Sistema de Coordenadas Universal (SCU), para a partir de ahí referirlas al Sistema de Coordenadas más conveniente para el usuario.
- El cuarto elemento es una lista con el nombre de entidad que contiene a la designada. Si existen varios bloques anidados, la lista contiene todos los nombres desde el bloque más interior hasta el más exterior de los que engloban a la entidad designada.

Por ejemplo, la designación de una entidad que forma parte de un bloque, que a su vez se encuentra incluido en otro bloque podría hacer que `NENTSEL` devolviera:

```
(<Nombre de objeto: 26c009d>
 (6.65 5.67 0.0)
 ( (1.0 0.0 0.0)
   (0.0 1.0 0.0)
   (0.0 0.0 1.0)
   (5.021 4.021 0.0)
 )
 (<Nombre de objeto: 26c010e> <Nombre de objeto: 26c01ba>)
)
```

La excepción a lo dicho son los atributos contenidos en un bloque. Si se designa un atributo, `NENTSEL` devuelve una lista con sólo dos elementos: el nombre de la entidad de atributo y el punto de designación.

```
(NENTSELP [mensaje_solicitud][punto])
```

De manera similar a la función anterior, `NENTSELP` permite acceder a todos los datos de definición de entidades contenidas en un bloque. Se puede especificar un mensaje de solicitud y un punto de designación.

`NENTSELP` obtiene una matriz de transformación de 4×4 elementos definida así:

M ₀₀	M ₀₁	M ₀₂	M ₀₃
M ₁₀	M ₁₁	M ₁₂	M ₁₃
M ₂₀	M ₂₁	M ₂₂	M ₂₃
M ₃₀	M ₃₁	M ₃₂	M ₃₃

Las tres primeras columnas de la matriz expresan la escala y rotación, y a cuarta columna es un vector de traslación. La última fila de la matriz no se toma en cuenta en las funciones que operan con este tipo de matrices. Esta matriz sirve para aplicar transformaciones a puntos.

(HANDENT *identificador*)

Devuelve el nombre de la entidad asociada al rótulo o identificador indicado. Hasta ahora se ha visto que los nombres de entidades eran únicos y las identificaban inequívocamente. Pero al terminar la sesión y salir de **AutoCAD**, esos nombres se pierden. En cambio, los identificadores se asocian a cada entidad y no cambian en las diferentes sesiones de dibujo. Estos identificadores se obtienen en la Base de Datos mediante el código 5.

Para comprobarlo podemos ejecutar el comando DDMODIFY con cualquier objeto. En el cuadro de diálogo que despliega este comando, arriba a la derecha, aparece el identificador en cuestión de cada objeto. Si deseamos, podemos extraer luego la lista de dicho objeto —con un simple (ENTGET (CAR (ENTSEL)))— para ver que asociado al código 5 se encuentra dicho identificador.

ONCE.20.2.10. Borrar/recuperar entidades

Veremos ahora una función muy sencilla que elimina o recupera una entidad. Esta función es:

(ENTDEL *nombre_entidad*)

ENTDEL elimina de la Base de Datos la entidad, cuyo nombre se indica, si existe en ella en el dibujo actual; ENTDEL recupera la entidad cuyo nombre se indica si había sido previamente borrada de la Base de Datos.

Esto quiere decir que las entidades borradas con ENTDEL pueden ser posteriormente recuperadas con el mismo ENTDEL, antes de salir de la actual sesión de dibujo evidentemente. Al salir del dibujo actual, las entidades borradas con ENTDEL se pierden definitivamente, sin haber posibilidad de recuperación.

ENTDEL solo accede a entidades principales; no es posible eliminar vértices de polilíneas sin optimizar ni atributos de bloque.

Ejemplo:

```
(DEFUN C:Eli ()
  (SETQ Nombre (CAR (ENTSEL "Designar un objeto para ser borrado: ")))
  (ENTDEL Nombre)
  (INITGET 1 "Sí No")
  (SETQ Recup (GETKEYWORD "¿Recuperarlo ahora (S/N)? "))
  (IF (= Recup "Sí")
    (ENTDEL Nombre)
  )
)
```

Este ejemplo permite eliminar cualquier objeto del dibujo actual. Tras ello, nos ofrece la

posibilidad de recuperarlo o no.

ONCE.20.2.11. Obtener rectángulo de texto

La función TEXTBOX devuelve las coordenadas de la caja de abarque o rectángulo contenedor de una entidad texto cuya lista se especifique. Su sintaxis es la que sigue:

```
(TEXTBOX lista_entidad_texto)
```

lista_entidad_texto puede ser la lista completa de la entidad de texto en cuestión o una lista parcial que contenga únicamente el valor de cadena del texto (par punteado con código 1 en la lista de entidad de texto). Por ejemplo:

```
(TEXTBOX (ENTGET (CAR (ENTSEL "Seleccione texto: "))))
```

Este ejemplo trabaja con la lista completa de un texto; el siguiente únicamente con la sublista de asociación de la cadena de texto:

```
(TEXTBOX '((1 . "Hola")))
```

Si se indica sólo la lista parcial con el texto, se utilizan los valores actuales por defecto de los parámetros de definición del texto. Si se indica la lista completa, se utilizan los valores contenidos en ella.

Las coordenadas devueltas por TEXTBOX son el vértice inferior izquierdo y el superior derecho del rectángulo de abarque, tomándose siempre como punto de inserción el 0,0,0. El primer punto únicamente es diferente de 0 cuando se trata de un texto de generación vertical o contiene letras con astas verticales por debajo de la línea de base.

ONCE.20.2.12. Construcción de una entidad

```
(ENTMAKE [lista_entidad])
```

Esta función permite añadir una entidad nueva al dibujo, construyendo directamente su lista completa en la Base de Datos. Si la lista introducida es correcta, devuelve esa lista. En caso contrario devuelve nil. Las listas se indican generalmente como literales, con QUOTE (').

La lista debe contener todas las sublistas de asociaciones necesarias para definir completamente cada tipo de entidad. Si se omite alguna se produce un error. Es posible omitir algún dato optativo y entonces se asume la opción por defecto. Así por ejemplo si no se indica la capa, la entidad construida asume la capa actual.

Una forma cómoda de añadir una nueva entidad a la Base de Datos es partir de una entidad ya existente, obtener su lista con ENTGET modificar y añadir lo que sea preciso y crear la nueva entidad con ENTMAKE. Esto evita tener que construir la lista completa desde el programa en AutoLISP.

El tipo de entidad debe ir siempre en inglés, como sabemos (por ejemplo "CIRCLE", "LINE", "ARC", etc.) y debe ser el primer o segundo elemento de la lista. Lógicamente, todos los códigos de las sublistas de asociaciones deberán ser correctos.

Para construir una entidad compleja como definiciones de bloques, referencias de bloque con atributos, o polilíneas no optimizadas, es preciso construir todas las listas

necesarias empleando varias veces `ENTMAKE`: la lista de cabecera o de la entidad principal, las listas con cada subentidad componente y la lista final del tipo `"SEQEND"` o `"ENDBLK"` para las definiciones de bloque.

Aunque para explorar todas las entidades contenidas en las definiciones de bloque con `ENTNEXT` no es necesario buscar un tipo de entidad final (como `"SEQEND"` para las polilíneas y atributos) pues al llegar a la última `ENTNEXT` devuelve `nil`, a la hora de construir las listas completas de la definición de un bloque, es preciso añadir como última lista un tipo de entidad llamado `"ENDBLK"`.

Por ejemplo, para construir un cuadrado como polilínea no optimizada, evidentemente con cuatro vértices, en la capa actual y con color rojo (número 1) se podría hacer:

```
(ENTMAKE '((0 . "POLYLINE")
          (62 . 1)
          (66 . 1)
          (70 . 1)
        )
)
(ENTMAKE '((0 . "VERTEX")
          (10 0.0 0.0 0.0)
        )
)
(ENTMAKE '((0 . "VERTEX")
          (10 0.0 10.0 0.0)
        )
)
(ENTMAKE '((0 . "VERTEX")
          (10 10.0 10.0 0.0)
        )
)
(ENTMAKE '((0 . "VERTEX")
          (10 10.0 0.0 0.0)
        )
)
(ENTMAKE '((0 . "SEQEND")
)
)
```

En la cabecera de la polilínea, el código 66 debe ir seguido obligatoriamente del valor 1 que indica que siguen vértices. Para que la polilínea sea cerrada, hay que incluir una lista con código 70 y valor 1.

NOTA: También se pueden construir directamente listas de objetos no gráficos mediante `ENTMAKE`.

<code>(ENTMAKEX [lista_entidad])</code>

Esta función es similar a `ENTMAKE`, pero la entidad se crea sin propietario. Se suministra una lista correcta de definición y se crea un objeto, gráfico o no gráfico. Pero al no tener propietario, este objeto no se escribe en los archivos `.DWG` o `.DXF`.

ONCE.20.2.13. Manejo de tablas de símbolos

Existe una serie de funciones específicas para gestionar los objetos no gráficos de **AutoCAD**. Es posible modificar algunos de los datos de estos objetos, aunque la mayoría de

ellos no se pueden crear expresamente mediante `ENTMAKE`. El grupo de objetos de diccionario se denomina así por contener el tipo de símbolo `DICTIONARY` y aparecer en los formatos de intercambio DXF en ese grupo. Pero en realidad contiene dos tipos de objetos no gráficos incorporados en la Versión 13 de **AutoCAD**: estilos de línea múltiple y grupos de selección.

Veremos a continuación dichas funciones.

`(TBLNEXT nombre_tabla [retroceso])`

Esta función devuelve una lista con el contenido de la tabla de símbolos cuyo nombre se indique. El nombre tiene que ser `"LAYER"`, `"LTYPE"`, `"VIEW"`, `"STYLE"`, `"BLOCK"`, `"UCS"`, `"VPORT"`, `"DIMSTYLE"` o `"APPID"`, que son los únicos admitidos. La función devuelve la primera tabla de símbolos existente de ese tipo, la primera vez que se utiliza. Después va devolviendo las siguientes conforme se utiliza repetidamente.

Por ejemplo, en un dibujo con tres capas: 0, `PIEZA` y `OCULTAS`, `TBLNEXT` se utilizaría tres veces para obtener las características de las tres capas. Al escribir:

```
(TBLNEXT "layer")
```

se devuelve:

```
((0 . "LAYER")
 (2 . "0")
 (6 . "CONTINUOUS")
 (70 . 0)
 (62 . 7)
 )
```

La capa 0 tiene asociados un tipo de línea `CONTINUOUS` y un color 7 (blanco). Empleando de nuevo `TBLNEXT` se devuelve la siguiente definición de capa. Al escribir:

```
(TBLNEXT "layer")
```

se devuelve:

```
((0 . "LAYER")
 (2 . "PIEZA")
 (6 . "CONTINUOUS")
 (70 . 0)
 (62 . 1)
 )
```

La capa `PIEZA` tiene asociados un tipo de línea `CONTINUOUS` y un color 1 (rojo). Por último, al escribir:

```
(TBLNEXT "layer")
```

se devuelve:

```
((0 . "LAYER")
 (2 . "OCULTAS")
 (6 . "TRAZOS")
 (70 . 3)
 )
```

La capa `OCULTAS` tiene asociados un tipo de línea `TRAZOS` y un color 3 (verde).

Si se empleara `TBLNEXT` para la tabla `LAYER` por cuarta vez, se devolvería `nil` puesto que ya no existen más definiciones de capas.

Para examinar los componentes de la definición de un bloque, se accede a su tabla de símbolos mediante `(TBLNEXT "block")` o `TBLSEARCH` (que ahora veremos). El código -2 de la lista devuelta, contiene el nombre de la primera entidad de la definición del bloque. Se obtiene y se suministra a `ENTNEXT`, de manera que sucesivos `ENTNEXT` van devolviendo todas las listas de los componentes del bloque, hasta que al llegar a la última `ENTNEXT` devuelva `nil`.

Si el argumento *retroceso* no se omite y tiene un valor diferente de `nil` la función `TBLNEXT` empieza a buscar desde la primera tabla de símbolos.

`(TBLSEARCH nombre_tabla símbolo [siguiente])`

Esta función busca en el tipo de tabla que se indique, el nombre de símbolo especificado a continuación y devuelve la lista correspondiente. De esta forma se puede buscar por ejemplo directamente la lista correspondiente a la capa llamada `PIEZA`, haciendo:

```
(TBLSEARCH "layer" "pieza")
```

Normalmente se utiliza para determinar si existe una capa, un estilo de texto, etcétera. En el ejemplo siguiente se controla la existencia de carga de un tipo de línea:

```
(TBLSEARCH "LTYPE" "Vías")
```

Si el tipo de línea existe se devuelve su lista de definición, si no existe se devuelve `nil`. Esto puede ser muy útil, ya que, como sabemos por ejemplo, el tipo de línea no se representa en la definición de una entidad si es `PorCapa`, por lo que no surtiría efecto alguno el que un usuario intente asignar, mediante un programa, dicho tipo de línea a una entidad si no está cargado. Nos preocuparemos de comprobar su existencia para emitir un mensaje de error si no estuviera cargado.

Si se pretendiera acceder a la definición de un estilo de texto definido en el dibujo y llamado `TS1`, haciendo `(TBLSEARCH "STYLE" "TS1")`, podría ser devuelta la siguiente lista:

```
((0 . "STYLE")
 (2 . "TS1")
 (3 . "ROMANS")
 (4 . "")
 (70 . 0)
 (40 . 0.0)
 (41 . 1.0)
 (50 . 0.0)
 (71 . 0)
 )
```

El contenido de la tabla informa que el estilo está basado en la fuente o tipo de letra `ROMANS`, con altura 0, factor de proporción 1, ángulo de inclinación 0 y generación *normal*.

Si el argumento *siguiente* no se omite y tiene un valor diferente de `nil` el contador de `TBLNEXT` se ajusta de manera que la próxima llamada de `TBLNEXT` buscará la siguiente tabla a la obtenida por `TBLSEARCH`.

`(TBJOBJNAME nombre_tabla símbolo)`

Busca en la tabla indicada el nombre de símbolo especificado, devolviendo el nombre de entidad de dicha tabla (recordemos que la función anterior hacía lo mismo pero devolvía la

lista completa). A pesar de no ser objetos gráficos, las tablas de símbolos pueden gestionarse mediante `ENTGET` y `ENTMOD` como si fueran entidades gráficas. Para ello se necesita suministrar su nombre de entidad (código -1) y éste es el que obtiene `TBLOBJECTNAME`. Este mecanismo permite modificar directamente en la Base de Datos el nombre de un estilo de texto, el color asociado a una capa, etc. Muy interesante; además puede utilizarse como la anterior para controlar la existencia de este tipo de objetos.

```
(SNVALID nombre_tabla [indicador])
```

Esta función comprueba la validez de los caracteres del nombre de tabla de símbolos. Si es un nombre válido devuelve `T` y en caso contrario `nil`. Los nombres deben contener sólo caracteres alfanuméricos y caracteres especiales como el de dólar \$, guión de subrayado _ y guión normal -. También muy utilizado a la hora de comprobar si los nombres son válidos.

```
(NAMEDOBJDICT)
```

Esta función es básica para acceder a todos los objetos no gráficos del grupo de diccionarios. Devuelve el nombre de entidad del diccionario de objetos no gráficos del dibujo actual. Se utiliza en las funciones de exploración de esos objetos `DICTNEXT` y `DICTSEARCH`.

```
(DICTNEXT nombre_diccionario [retroceso])
```

Devuelve la lista con el contenido de objetos no gráficos del grupo de diccionarios. El nombre de diccionario suministrado debe haberse obtenido previamente mediante `NAMEDOBJDICT`. La función devuelve el primer objeto de diccionario cuando se utiliza por primera vez. Después devuelve sucesivamente los demás objetos no gráficos hasta el último, tras el cual devuelve `nil`. Su funcionamiento es similar a `TBLNEXT`.

Actualmente, los dos únicos objetos no gráficos accesibles en el grupo de diccionarios son los estilos de línea múltiple `ACAD_MLINESSTYLE` y los grupos de selección `ACAD_GROUP`, por lo que `DICTNEXT` devolverá dos listas: la primera con los nombres de todos los estilos de línea múltiple creados y la segunda con los nombres de todos los grupos de selección creados.

Si el argumento *retroceso* no se omite y tiene un valor diferente de `nil` la función `DICTNEXT` empieza a buscar desde el primer objeto no gráfico de diccionario.

```
(DICTSEARCH nombre_diccionario símbolo [retroceso])
```

Esta función busca en el grupo de diccionarios el tipo de objeto no gráfico indicado en *símbolo* y devuelve la lista correspondiente. El tipo de objeto sólo puede ser `"ACAD_MLINESSTYLE"`, para los estilos de línea múltiple, y `"ACAD_GROUP"`, para los grupos de selección. La lista devuelta es la misma que en `DICTNEXT`.

Si el argumento *siguiente* no se omite y tiene un valor diferente de `nil` el contador de `DICTNEXT` se ajusta de manera que la próxima llamada de `DICTNEXT` buscará la siguiente tabla a la obtenida por `DICTSEARCH`.

```
(DICTADD nombre_diccionario símbolo nuevo_objeto)
```

Añade el nuevo objeto no gráfico al diccionario especificado. Los objetos no gráficos son estilos de línea múltiple y grupos de selección. El argumento *símbolo* es el nombre clave del objeto que se va a añadir.

```
(DICTREMOVE nombre_diccionario símbolo)
```

Elimina el objeto no gráfico indicado en *símbolo* del diccionario especificado.

```
(DICTRENAME nombre_diccionario símbolo_antiguo símbolo_nuevo)
```

Cambia el nombre de la entrada representada por *símbolo_antiguo*, por el nuevo nombre indicado a continuación en el diccionario especificado en primer lugar.

ONCE.20.2.14. Funciones relativas a datos extendidos

En **AutoCAD**, al emplear aplicaciones ADS o ARX, en la Base de Datos del dibujo se añade un nuevo tipo de tabla de símbolos llamado "APPID". Cada tabla de este tipo contiene el nombre de una aplicación ADS o ARX utilizada para datos extendidos de entidades. De esta forma, cada tipo de aplicación puede añadir los datos necesarios para su funcionamiento a las entidades de **AutoCAD**. Como en los demás casos, estos datos son accesibles a través de listas de asociaciones. Para distinguirlos del resto comienzan con el código -3.

```
(REGAPP nombre_aplicación)
```

Esta función registra un nombre de aplicación externa en el actual dibujo de **AutoCAD**. Registrando las aplicaciones con un nombre, es posible después acceder a los datos extendidos de las entidades. Si el registro de la aplicación es correcto, *REGAPP* devuelve el nombre registrado. En caso contrario (por ejemplo al especificar un nombre de aplicación que ya existe), devuelve *nil*. El nombre puede contener hasta 31 caracteres alfanuméricos y algunos especiales como dólar \$, subrayado _ y guión -. Una vez registrado, el nombre de la aplicación se añade en la Base de Datos como una tabla de símbolos del tipo "APPID".

Si en la función *ENTGET* se especifica una lista de nombres de aplicaciones registrados con *REGAPP*, la lista devuelta incluye también el código -3 que es el denominado *centinela* o indicador de que la entidad contiene datos extendidos, y todas las listas de datos que siguen a dicho código. Estos datos se encuentran asociados a códigos de 1000 a 1071. Los datos extendidos propios de **AutoCAD** se obtienen indicando como nombre de aplicación el de "ACAD". Por ejemplo:

```
(ENTGET (ENTLAST) '("acad"))
```

Sólo algunos tipos de entidades contienen en **AutoCAD** datos extendidos, como por ejemplo las ventanas gráficas, los bloques de sombreados, las cotas, directrices y tolerancias geométricas.

```
(XDROOM nombre_entidad)
```

XDROOM devuelve el espacio de memoria disponible para los datos extendidos de la entidad cuyo nombre se indica. El espacio máximo disponible para cada entidad es de 16383 octetos. El valor devuelto por *XDROOM* es entonces la diferencia entre este máximo y lo que ocupan los datos extendidos ya existentes para la entidad. Por ejemplo:

```
(XDROOM (ENTLAST))
```

podría devolver:

16264

`(XDSIZE lista_datos_extendidos)`

Devuelve la longitud, en octetos o bytes, que la lista indicada ocupa cuando es añadida como datos extendidos de una entidad. Es complementario del anterior `XDROOM` y se utiliza para controlar cuánto van ocupando en memoria los datos extendidos de una entidad.

La lista debe contener un nombre de aplicación previamente registrado con `REGAPP`. Si existen varios nombres de aplicaciones, se forma una lista que englobe a las demás (por ejemplo con `LIST`).

Hasta aquí todas las funciones relacionadas directamente con el acceso a la Base de Datos de **AutoCAD**. A continuación, estudiaremos cuatro ejemplos de programas completos que nos ayudarán a la comprensión práctica de este tema.

El primer ejemplo que veremos se corresponde con un programa que permite distribuir un texto indicado por el usuario a lo largo de cualquier entidad de condición curva, sea arco, círculo, elipse, polilínea o spline, y/o de líneas. Veamos el listado del programa en primera instancia:

```
(DEFUN datos_txcurva (/ mens lconj grad prop tent)
  (IF altx0 () (SETQ altx0 1))
  (SETQ mens (STRCAT "Altura del texto <" (RTOS altx0 2 2) ">: "))
  (INITGET 6)
  (IF (SETQ altx (GETREAL mens)) () (SETQ altx altx0))(TERPRI)
  (SETQ altx0 altx)
  (IF sep0 () (SETQ sep0 0))
  (SETQ mens (STRCAT "Separación entre texto y entidad (+ = encima;- = debajo)"
    " <" (RTOS sep0 2 2) ">: "))
  (IF (SETQ sep (GETREAL mens)) () (SETQ sep sep0))(TERPRI)
  (SETQ sep0 sep)
  (IF (NOT (TBLSEARCH "block" "$txcurva"))
    (PROGN (COMMAND "_point" "0,0")
      (COMMAND "_block" "$txcurva" "0,0" (entlast) ""))
  )
  (SETQ prop (CDR (ASSOC 41 (TBLSEARCH "style" (GETVAR "textstyle")))))
  (SETQ esp (* altx prop 1.2))
  (SETQ mens "Designar entidad para alinear texto: ")
  (WHILE (NOT (SETQ grad (ENTSEL mens)))(SETQ grad (ENTSEL mens)))
  (SETQ tent (CDR (ASSOC 0 (ENTGET (CAR grad)))))
  (WHILE (NOT (OR (= tent "ARC")(= tent "CIRCLE")(= tent "ELLIPSE")
    (= tent "LINE")(= tent "POLYLINE")(= tent "LWPOLYLINE")
    (= tent "SPLINE"))))
    (PROMPT "\nEntidad no válida para situar texto\n")
    (WHILE (NOT (SETQ grad (ENTSEL mens)))(SETQ grad (ENTSEL mens)))
    (SETQ tent (CDR (ASSOC 0 (ENTGET (CAR grad)))))
  )
  (COMMAND "_measure" grad "_b" "$txcurva" "_y" esp)(TERPRI)
  (SETQ conj (SSGET "_p"))
  (SETQ lconj (SSELENGTH conj) lontx (+ lconj 1))
  (WHILE (> lontx lconj)
    (PROMPT (STRCAT "Número máximo de caracteres: " (ITOA lconj) "\n"))
    (SETQ cadtx (GETSTRING T "Introducir Texto: "))(TERPRI)
    (SETQ lontx (STRLEN cadtx))
  )
)

(DEFUN txcurva (/ ntx desig tx nent lent pin rot )
  (SETQ ntx 0 desig (SSADD))
  (REPEAT lontx
```

```
(SETQ tx (SUBSTR cadtx (+ ntx 1) 1))
(SETQ nent (SSNAME conj ntx))
(SETQ lent (ENTGET nent))
(SETQ pin (CDR (ASSOC 10 lent)))
(SETQ rot (CDR (ASSOC 50 lent)))
(IF (OR (= tx "l") (= tx "i") (= tx "I") (= tx "j") (= tx "1")))
  (SETQ pin (POLAR pin rot (/ altx 6))))
(IF (= tx "m")
  (SETQ pin (POLAR pin rot (- (/ altx 6)))))
(SETQ pin (POLAR pin (+ rot (/ PI 2)) sep))
(SETQ rot (/ (* rot 180) PI))
(COMMAND "_text" pin altx rot tx)
(SETQ desig (SSADD (SSNAME (SSGET "_l") 0) desig))
(SETQ ntx (+ ntx 1))
)
(COMMAND "_erase" conj "")
(COMMAND "_select" desig "")
(COMMAND "_redraw")
)

(DEFUN c:txcurva (/ altx sep esp conj lontx cadtx refnt0 pdmod0 error0)
  (SETVAR "cmdecho" 0)
  (SETQ error0 *error* *error* err_txcurva)
  (SETQ pdmod0 (GETVAR "pdmode")) (SETVAR "pdmode" 0)
  (SETQ refnt0 (GETVAR "osmode")) (SETVAR "osmode" 0)
  (COMMAND "_undo" "_begin")
  (datos_txcurva)
  (txcurva)
  (COMMAND "_purge" "_block" "$txcurva" "_n")
  (COMMAND "_undo" "_end")
  (SETVAR "osmode" refnt0) (SETVAR "pdmode" pdmod0)
  (SETVAR "cmdecho" 1) (PRIN1)
)

(DEFUN err_txcurva (mens)
  (SETQ *error* error0)
  (IF (= mens "quitar / salir abandonar")
    (PRIN1)
    (PRINC (STRCAT "\nError: " mens " ")))
  )
  (IF conj (COMMAND "_erase" conj ""))
  (COMMAND "_purge" "_block" "$txcurva" "_n")
  (COMMAND "_undo" "_end")
  (SETVAR "cmdecho" 1)
  (SETVAR "osmode" refnt0) (SETVAR "pdmode" pdmod0) (PRIN1)
)

(PROMPT "Nuevo comando TXCURVA definido.")(PRIN1)
```

El programa tiene el siguiente funcionamiento. El comando nuevo de **AutoCAD** TXCURVA, llama primero a la función `datos_txcurva` y después a `txcurva`.

La función de `datos` solicita en primer lugar la altura del texto y la separación entre el texto y la curva en la cual se va a alinear. Un valor positivo deja al texto a un lado o “por encima” y un valor negativo por el otro lado o “por debajo”, siempre según el sentido de creación de la curva. Para ambos datos, ofrece como valores por defecto los últimos utilizados (cosa que ya hemos estudiado con otros ejemplos).

El programa crea un bloque que le resulta necesario para insertarlo con el comando GRADUA (MEASURE). Este bloque contiene simplemente un punto y se le da el nombre de

\$txcurva. El carácter \$ al principio es para distinguirlo de otros bloques que pueda contener el dibujo. Este bloque será limpiado al final de la rutina. Primero se comprueba mediante TBLSEARCH si ya existe.

Para el espaciado entre caracteres el programa tiene en cuenta el estilo actual, extraído de la variable de **AutoCAD** TEXTSTYLE. El código 41 contiene la anchura del estilo. El producto de ese factor por la altura y una cantidad de 1.2 es el espaciado entre caracteres. Se trata de un cálculo estimativo. El valor adecuado dependerá del estilo de texto. Más adelante se tendrán en cuenta determinados caracteres como "i", "l", "j", etcétera para modificar ese espaciado.

Se solicita designar la entidad para alinear el texto. Se establece un control por si se falla en la designación. La función ENTSEL sólo permite designar una entidad. A continuación se extrae el tipo de entidad (código 0) y se almacena en tent. Si el tipo de entidad no es una de las válidas para graduar, se visualiza un mensaje de advertencia y se vuelve a solicitar.

Se llama al comando GRADUA, con el espaciado entre caracteres calculado más arriba, y se inserta el bloque \$txcurva alineándolo con la curva. Con SSGET "_p" se almacena el conjunto de puntos de graduación, y se averigua su número en lconj. Se utiliza lontx para controlar el número máximo de caracteres que caben a lo largo de la curva. Se ofrece esa información al usuario y se solicita el texto que se alineará (cadtx). Se almacena su longitud en lontx y si es superior a la máxima, se vuelve a solicitar.

Por su lado, en la función txcurva se inicializa la variable ntx que controla el número de orden de cada carácter en la cadena de texto para alinear, y desig que almacena un conjunto de selección vacío con SSADD.

A continuación se establece un ciclo que se repetirá tantas veces como caracteres haya en la cadena de texto. Se extraen de la cadena los caracteres uno a uno. Se averigua en el conjunto de bloques de punto el punto de inserción (código 10) y el ángulo de inserción (código 50), mediante el mecanismo con SSNAME, ENTGET y ASSOC que ya conocemos.

El punto de inserción de cada carácter se calcula a partir del punto de inserción del bloque, llevando en perpendicular una distancia igual a la separación indicada por el usuario. Si el carácter es estrecho ("l", "i", "I", "j" o "1"), se desplaza previamente el punto de inserción hacia delante una sexta parte de la altura del texto. Si es "m", se desplaza hacia atrás. Con esto se persigue que esos caracteres no se solapen o se queden más pegados al precedente que al siguiente. Se traslada el ángulo de rotación de radianes a grados, puesto que así hay que indicarlo en el comando TEXTO. Se van insertando los caracteres uno a uno. Se añade cada uno al conjunto de selección desig para poder almacenar mediante el comando DESIGNA dicho conjunto. Esto permitirá referirse posteriormente a todo el texto a la vez con el modo *Previo*.

Por último se borran todas las inserciones del bloque y se redibuja. El resto de mecanismos de control de errores y demás ya está lo suficientemente explicado.

El segundo programa traza una línea de tubería en tres dimensiones. Tuberías y codos se generan mediante mallas. El trazado deberá haber sido previamente mediante líneas (sólo líneas), de forma que los extremos de los tramos se toquen. Veamos el programa:

```
(DEFUN inic_tubos ( / n nent lent)
  (PROMPT "Designar líneas de conducción una a una y en orden: \n")
  (WHILE (NOT (SETQ lin (SSGET))))
  (SETQ n 0)(SETQ num (SSLENGTH lin))
  (REPEAT num
    (SETQ nent (SSNAME lin n))
```

```
(SETQ lent (ENTGET nent))
(IF (/= (CDR (ASSOC 0 lent)) "LINE")
  (PROGN (SSDEL nent lin)(SETQ n (- n 1)))
)
(SETQ n (+ n 1))
)
(SETQ num (SSLENGTH lin))
(IF (= num 0)(PROMPT "Ninguna de las entidades es una línea\n"))
(IF (= num 1)(PROMPT "Sólo una entidad de línea designada\n"))
)

(DEFUN datos_tubos ( / mens capac)
  (INITGET 7)
  (SETQ rtub (/ (GETREAL "Diámetro exterior de tubería: ") 2))
  (TERPRI)
  (SETQ rcod rtub)
  (SETQ mens (STRCAT "Diámetro exterior de los codos <" (RTOS (* rtub 2) 2 2)
    ">: "))
  (INITGET 6)
  (IF (SETQ rcod (GETREAL mens))
    (SETQ rcod (/ rcod 2))
    (SETQ rcod rtub)
  )
  (WHILE (< rcod rtub)
    (PROMPT "Diámetro de codos debe ser mayor o igual que el de tubería\n")
    (INITGET 6)
    (IF (SETQ rcod (GETREAL mens))
      (SETQ rcod (/ rcod 2)) (SETQ rcod rtub))
    )
  (INITGET 7)
  (SETQ rpeq (GETREAL "Radio interior de curvatura del codo: "))
  (TERPRI)
  (INITGET 6)
  (IF (SETQ tab (GETINT "Precisión de mallas <16>: "))((SETQ tab 16))(TERPRI)
  (SETVAR "surftab1" tab)(SETVAR "surftab2" tab)
  (SETQ capac (GETVAR "clayer"))
  (SETQ capl (STRCAT "$c-" capac))
  (COMMAND "_layer" "_new" capl "_off" capl "")
  (SETQ pte1 (CDR (ASSOC 10 (ENTGET (SSNAME lin 0)))))
  (SETQ npte1 (CDR (ASSOC -1 (ENTGET (SSNAME lin 0)))))
  (SETQ pt1 (TRANS pte1 npte1 1))
  (SETQ pte2 (CDR (ASSOC 10 (ENTGET (SSNAME lin 1)))))
  (SETQ npte2 (CDR (ASSOC -1 (ENTGET (SSNAME lin 1)))))
  (SETQ pt2 (TRANS pte2 npte2 1))
  (SETQ pte3 (CDR (ASSOC 11 (ENTGET (SSNAME lin 1)))))
  (SETQ npte3 (CDR (ASSOC -1 (ENTGET (SSNAME lin 1)))))
  (SETQ pt3 (TRANS pte3 npte3 1))
  (SETQ n 2)(SETQ prim T)
)

(DEFUN tramo_tubos (/ ang alfa tabcod pri prf cir1 cir2 cir3 cir4 cen pfeje
  eje)
  (COMMAND "_ucs" "_3" pt1 pt2 pt3)
  (SETQ pt2 (TRANS pte2 npte2 1))(SETQ pt3 (TRANS pte3 npte3 1))
  (SETQ ang (ANGLE pt2 pt3))
  (SETQ alfa (- pi ang))
  (SETQ dis (/ (+ rcod rpeq)((sin (/ alfa 2))(cos (/ alfa 2)))))
  (SETQ ang (/ (* 180 ang) pi))
  (SETQ tabcod (FIX (* tab (/ ang 90))))
  (IF (< tabcod 6)(SETQ tabcod 6))
  (COMMAND "_ucs" "_y" "90")
)
```

```
(SETQ pt1 (TRANS pte1 npte1 1))(SETQ pt2 (TRANS pte2 npte2 1))
(IF prim (SETQ pri pt1)
  (SETQ pri (MAPCAR '+ pt1 (LIST 0 0 disant)))
)
(SETQ prf (MAPCAR '- pt2 (LIST 0 0 dis)))
(COMMAND "_circle" pri rtub)
(SETQ cir1 (LIST (ENTLAST) (POLAR pri 0 rtub)))
(IF (OR prim (= rcod rtub)) ()
  (PROGN (COMMAND "_circle" pri rcod)
    (SETQ cir2 (LIST (ENTLAST) (POLAR pri 0 rcod)))
  )
)
(COMMAND "_circle" prf rtub)
(SETQ cir3 (LIST (ENTLAST) (POLAR prf 0 rtub)))
(IF (= rcod rtub) ()
  (PROGN (COMMAND "_circle" prf rcod)
    (SETQ cir4 (LIST (ENTLAST) (POLAR prf 0 rcod)))
  )
)
(SETQ cen (POLAR prf (/ pi 2) (+ rcod rpeq)))
(SETQ pfeje (MAPCAR '- cen (LIST rtub 0 0)))
(COMMAND "_line" cen pfeje "")
(SETQ eje (LIST (ENTLAST) cen))
(IF (OR prim (= rcod rtub)) ()
  (COMMAND "_rulesurf" cir1 cir2)
)
(COMMAND "_rulesurf" cir1 cir3)
(IF (= rcod rtub)
  (PROGN (SETVAR "surftab1" tabcod)
    (COMMAND "_revsurf" cir3 eje "" ang))
  (PROGN (COMMAND "_rulesurf" cir3 cir4)
    (SETVAR "surftab1" tabcod)
    (COMMAND "_revsurf" cir4 eje "" ang))
)
(SETVAR "surftab1" tab)
(IF (= rcod rtub)
  (COMMAND "_chprop" cir1 cir3 eje "" "_p" "_la" capl "")
  (IF prim (COMMAND "_chprop" cir1 cir3 cir4 eje "" "_p" "_la" capl "")
    (COMMAND "_chprop" cir1 cir2 cir3 cir4 eje "" "_p" "_la" capl")
  )
)
(SETQ prim nil)
)

(DEFUN act_tubos ()
  (SETQ pte1 pte2 npte1 npte2 disant dis)
  (SETQ pt1 (TRANS pte1 npte1 1))
  (SETQ pte2 pte3 npte2 npte3)
  (SETQ pt2 (TRANS pte2 npte2 1))
  (SETQ pte3 (CDR (ASSOC 11 (ENTGET (SSNAME lin n)))))
  (SETQ npte3 (CDR (ASSOC -1 (ENTGET (SSNAME lin n)))))
  (SETQ pt3 (TRANS pte3 npte3 1))
  (SETQ n (+ n 1))
)

(DEFUN tramof_tubos ( / pri prf cir1 cir3 cir4)
  (SETQ pt1 (TRANS pte1 npte1 1))
  (SETQ pt2 (TRANS pte2 npte2 1))
  (SETQ pt3 (TRANS pte3 npte3 1))
  (COMMAND "_ucs" "3" pt3 pt2 pt1)
  (COMMAND "_ucs" "y" "90")
)
```

```
(SETQ pt3 (TRANS pte3 npte3 1))(SETQ pt2 (TRANS pte2 npte2 1))
(SETQ pri pt3)
(SETQ prf (MAPCAR '- pt2 (LIST 0 0 dis)))
(COMMAND "_circle" pri rtub)
(SETQ cir1 (LIST (ENTLAST) (POLAR pri 0 rtub)))
(COMMAND "_circle" prf rtub)
(SETQ cir3 (LIST (ENTLAST) (POLAR prf 0 rtub)))
(IF (= rcod rtub) ()
  (PROGN (COMMAND "_circle" prf rcod)
    (SETQ cir4 (LIST (ENTLAST) (POLAR prf 0 rcod)))
  )
)
(COMMAND "_rulesurf" cir1 cir3)
(IF (= rcod rtub)
  (COMMAND "_chprop" cir1 cir3 "" "_p" "_la" cap1 "")
  (PROGN (COMMAND "_rulesurf" cir3 cir4)
    (COMMAND "_chprop" cir1 cir3 cir4 "" "_p" "_la" cap1 "")
  )
)
(COMMAND "_ucs" "_w")
)

(DEFUN c:tubos ( / lin num rtub rcod rpeq tab pte1 pte2 pte3
  nptel npte2 npte3 pt1 pt2 pt3 n prim dis disant)
  (SETQ error0 *error* *error* err_tubos)
  (SETVAR "cmdecho" 0)
  (COMMAND "_undo" "_begin")
  (inic_tubos)
  (IF (> num 1)
    (PROGN
      (datos_tubos)
      (SETVAR "gridmode" 0)(SETQ refnt0 (GETVAR "osmode"))(SETVAR "osmode" 0)
      (REPEAT (- num 1)
        (tramo_tubos)
        (IF (< n num)(act_tubos)
          (tramof_tubos)
        )
      )
      (COMMAND "_erase" (SSGET "X" (LIST (CONS 8 cap1))) "")
      (COMMAND "_purge" "_layer" cap1 "_n")
    )
    (PROMPT "**No vale**")
  )
  (SETQ *error* error0)(COMMAND "_undo" "_end")
  (SETVAR "osmode" refnt0)(SETVAR "blipmode" 1)(SETVAR "cmdecho" 1)(PRIN1)
)

(DEFUN err_tubos (mens)
  (SETQ *error* error0)
  (IF (= mens "quitar / salir abandonar")
    (PRIN1)
    (PRINC (STRCAT "\nError: " mens " "))
  )
  (COMMAND "_erase" (SSGET "X" (LIST (CONS 8 cap1))) "")
  (COMMAND "_undo" "_end")
  (SETVAR "cmdecho" 1) (SETVAR "osmode" refnt0)(PRIN1)
)
```

La función `inic_tubos` solicita la designación de líneas que son los ejes del trazado de tubos, una a una y en orden. Establece un control con `WHILE` para garantizar que se ha designado alguna entidad. Examina el conjunto designado para rechazar las entidades que no

sean líneas. Si no existen líneas o sólo hay una, visualiza mensajes de advertencia. Al final, el conjunto `lin` contiene únicamente las entidades de línea señaladas por el usuario.

La función `datos_tubos` va solicitando los datos necesarios. En primer lugar el diámetro exterior de los tubos, después el de los codos (ofreciendo como opción por defecto el mismo que para los tubos y evitando que sea menor que el de ellos). Las variables almacenan en cada caso el valor de los radios. A continuación se pide el radio interior de curvatura del codo. Por último la precisión del mallado para las superficies de cada tubo.

Las curvas de apoyo para el trazado se situarán en una capa cuyo nombre se forma con el de la capa actual y un prefijo `$c-`.

Se toman los tres primeros puntos de la conducción (primer punto de la primera línea y primer y último punto de la segunda línea) y se almacenan esos puntos en el Sistema de Coordenadas de Entidad que es como se encuentran en la Base de Datos. Se trasladan esos puntos al SCP actual. Se almacenan también los nombres de las entidades de línea a que pertenecen. Por último se inicializa el contador de tramos `n`, y se pone a cierto `T` la variable `prim` para controlar que se trata del primer tramo que se dibuja.

`tramo_tubos` se encarga de dibujar cada tramo con su tubo y su codo. Establece el SCP adecuado de acuerdo con el plano formado por las dos líneas que definen el tramo. Calcula el ángulo formado por esas líneas en el nuevo SCP y la distancia `dis` del vértice a la que termina el tubo y empieza el codo. Calcula la precisión de mallado `tabcod` del codo de acuerdo con el ángulo abarcado por el mismo.

Cambia a un nuevo SCP perpendicular al anterior para trazar los círculos que van a definir las mallas. Los centros de estos círculos serán los puntos en que empieza y termina el tubo `pri` y `prf`. Se trazan los círculos y se forman listas que contengan el nombre de cada uno con un punto de designación (del tipo de las devueltas por `ENTSEL`). Hay que tener la precaución de trasladar los puntos al SCP actual.

El codo se va a trazar como una superficie de revolución y esto obliga a dibujar el eje y almacenarlo también como una lista con nombre y punto de designación. Si se trata del primer tramo, o bien si el diámetro del codo es igual al de la tubería, entonces se evita dibujar las mallas de transición entre ambos diámetros, que visualizan el espesor del codo.

Una vez trazadas todas las curvas, se generan las mallas con `SUPREGLA` y `SUPREV`. Se cambian las curvas de apoyo a la capa `cap1`. Una vez recorrido el primer tramo se cambia la variable de control `prim` a `nil` para el resto de tramos.

`act_tubos` se encarga de actualizar los valores de las variables para empezar el nuevo tramo en la repetitiva. Almacena la distancia `dis` de final del último tramo para utilizarla en `disant` como distancia de principio del nuevo tramo. Calcula los puntos necesarios para obtener el SCP del plano formado por las dos líneas del nuevo tramo y suma uno al contador `n`.

La subrutina `tramof_tubos` dibuja el tramo final, alterando el orden de los puntos utilizados para formar el SCP adecuado. Dibuja los últimos círculos y obtiene las mallas correspondientes. Para obtener los puntos necesarios, siempre se parte del SCE de la Base de Datos y de su traslado al SCP actual.

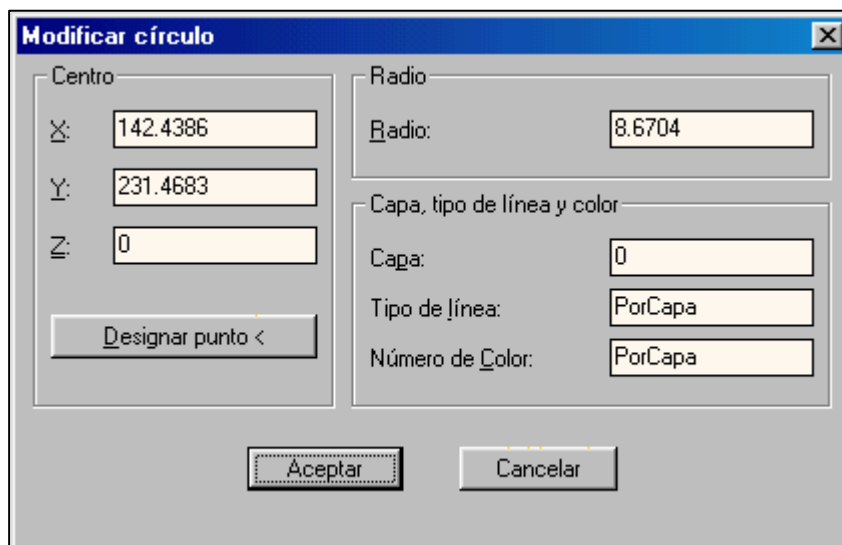
`c:tubos` es la función que compone el comando de **AutoCAD**. Llama a la función `inic_tubos` y establece una condición. Si el usuario ha designado más de una entidad de línea, entonces pueden trazarse los tubos. En caso contrario se visualiza un mensaje `*No vale*`.

Por lo demás, funciones de control de errores y demás.

Veamos un tercer ejemplo que maneja un cuadro de diálogo. El siguiente programa actúa como DDMODIFY a pequeña escala, es decir, permite modificar las propiedades de una entidad y/o visualizarlas, pero en este caso sólo de círculos. Concretamente se puede actuar sobre las coordenadas del centro del círculo, su radio, capa, tipo de línea y color. Veamos primeramente el diseño en DCL y el aspecto del cuadro de diálogo en cuestión:

```
modicir:dialog {label="Modificar círculo";
:row {
:boxed_column {label="Centro";
:edit_box {label "&X:";edit_width=15;key="txcentro";}
:edit_box {label "&Y:";edit_width=15;key="tycentro";}
:edit_box {label "&Z:";edit_width=15;key="tzcentro";}
spacer_1;
:button {label="&Designar punto <";key="tpoints";}
spacer_1;
}
:column {
:boxed_column {label="Radio";
:edit_box {label "&Radio:";edit_width=15;key="tradio";}
spacer_1;
}
:boxed_column {label="Capa, tipo de línea y color";
:edit_box {label "Ca&pa:";edit_width=15;key="tcapa";}
:edit_box {label "Tipo de &línea:";edit_width=15;key="tlin";}
:edit_box {label "Número de &Color:";edit_width=15;key="tcolor";}
spacer_1;
}
}
}
spacer_1;
ok_cancel;
errtile;
}
```

El cuadro es este:



Ahora mostramos el programa AutoLISP:

```
(DEFUN Datos ()
  (SETQ PuntoCentro nil)
  (WHILE (NOT (SETQ Entidad (ENTSEL "\nDesignar círculo: ")))
    (SETQ ListaCir (ENTGET (CAR Entidad)))
    (IF (/= (CDR (ASSOC 0 ListaCir)) "CIRCLE")
      (Datos)
      (Cuadro)
    )
  )
)

(DEFUN Cuadro (/ SD)
  (SETQ SD nil)
  (IF (NOT PuntoCentro)
    (SETQ Ind (LOAD_DIALOG "c:/misdoc~1/autocad/autolisp/modicir.dcl"))
  )
  (NEW_DIALOG "modicir" Ind)
  (IF PuntoCentro
    (PROGN
      (SET_TILE "txcentro" (RTOS (CAR PuntoCentro)))
      (SET_TILE "tycentro" (RTOS (CADR PuntoCentro)))
      (SET_TILE "tzcentro" (RTOS (CADDR PuntoCentro)))
    )
    (PROGN
      (SET_TILE "txcentro" (RTOS (CADR (ASSOC 10 ListaCir))))
      (SET_TILE "tycentro" (RTOS (CADDR (ASSOC 10 ListaCir))))
      (SET_TILE "tzcentro" (RTOS (CADDR (ASSOC 10 ListaCir))))
    )
  )
  (SET_TILE "tradio" (RTOS (CDR (ASSOC 40 ListaCir))))
  (SET_TILE "tcapa" (CDR (ASSOC 8 ListaCir)))
  (IF (NOT (ASSOC 6 ListaCir))
    (SET_TILE "tlin" "PorCapa")
    (SET_TILE "tlin" (CDR (ASSOC 6 ListaCir)))
  )
  (IF (NOT (ASSOC 62 ListaCir))
    (SET_TILE "tcolor" "PorCapa")
    (SET_TILE "tcolor" (ITOA (CDR (ASSOC 62 ListaCir))))
  )

  (ACTION_TILE "tpoints" "(DONE_DIALOG 2)")
  (ACTION_TILE "accept" "(Tiles) (IF ErrorDCL () (PROGN (DONE_DIALOG 1)
                                                         (UNLOAD_DIALOG Ind)))")
  (ACTION_TILE "cancel" "(DONE_DIALOG 0) (UNLOAD_DIALOG Ind)")

  (SETQ SD (START_DIALOG))
  (COND
    ((= 1 SD) (Aceptar))
    ((= 2 SD) (Designar))
    ((= 0 SD) ())
  )
)

(DEFUN Tiles ()
  (ControlDCL)
  (SETQ NoLin nil NoColor nil)

  (SETQ X (ATOF (GET_TILE "txcentro")))
  Y (ATOF (GET_TILE "tycentro"))
  Z (ATOF (GET_TILE "tzcentro"))
)
```

```
(IF (OR (< (ATOF (GET_TILE "tradio")) 0) (= (ATOF (GET_TILE "tradio")) 0))
  (PROGN
    (SETQ ErrorDCL T)
    (SET_TILE "error" "El radio no puede ser cero ni menor que cero.")
    (MODE_TILE "tradio" 2)
  )
  (SETQ Radio (ATOF (GET_TILE "tradio"))))
)

(SETQ Capa (GET_TILE "tcapa"))

(IF (= (GET_TILE "tlin") "PorCapa")
  (SETQ NoLin T)
  (PROGN
    (IF (TBLSEARCH "LTYPE" (GET_TILE "tlin"))
      (SETQ Lin (GET_TILE "tlin"))
      (PROGN
        (SETQ ErrorDCL T)
        (SET_TILE "error" "El tipo de línea indicado no está cargado.")
        (MODE_TILE "tlin" 2)
      )
    )
  )
)

(IF (OR (< (ATOI (GET_TILE "tcolor")) 0) (> (ATOI (GET_TILE "tcolor")) 256))
  (PROGN
    (SETQ ErrorDCL T)
    (SET_TILE "error" "El número de color no puede ser menor que cero ni mayor
                      que 256.")
    (MODE_TILE "tcolor" 2)
  )
  (PROGN
    (IF (= (GET_TILE "tcolor") "PorCapa")
      (SETQ NoColor T)
      (SETQ Color (ATOI (GET_TILE "tcolor"))))
    )
  )
)

(DEFUN Designar ()
  (SETQ PuntoCentro (GETPOINT "\nNuevo centro del círculo: "))
  (Cuadro)
)

(DEFUN Aceptar (/ ViejoCentro NuevoCentro ViejoRadio NuevoRadio NuevaCapa
                  ViejaCapa NuevoColor ViejoColor)
  (SETQ ViejoCentro (ASSOC 10 ListaCir))
  (SETQ NuevoCentro (LIST 10 X Y Z))
  (SETQ ListaCir (SUBST NuevoCentro ViejoCentro ListaCir))

  (SETQ ViejoRadio (ASSOC 40 ListaCir))
  (SETQ NuevoRadio (Cons 40 Radio))
  (SETQ ListaCir (SUBST NuevoRadio ViejoRadio ListaCir))

  (SETQ ViejaCapa (ASSOC 8 ListaCir))
  (SETQ NuevaCapa (Cons 8 Capa))
  (SETQ ListaCir (SUBST NuevaCapa ViejaCapa ListaCir))
)
```

```
(IF (NOT NoLin)
  (PROGN
    (IF (NOT (ASSOC 6 ListaCir))
      (SETQ ListaCir (APPEND ListaCir (LIST (CONS 6 Lin)))))
    (PROGN
      (SETQ ViejaLínea (ASSOC 6 ListaCir))
      (SETQ NuevaLínea (CONS 6 Lin))
      (SETQ ListaCir (SUBST NuevaLínea ViejaLínea ListaCir))
    )
  )
)

(IF (NOT NoColor)
  (PROGN
    (IF (NOT (ASSOC 62 ListaCir))
      (SETQ ListaCir (APPEND ListaCir (LIST (CONS 62 Color)))))
    (PROGN
      (SETQ ViejoColor (ASSOC 62 ListaCir))
      (SETQ NuevoColor (CONS 62 Color))
      (SETQ ListaCir (SUBST NuevoColor ViejoColor ListaCir))
    )
  )
)

(ENTMOD ListaCir)
)
(DEFUN ControlDCL ()
  (SETQ ErrorDCL nil)
  (IF (= (GET_TILE "txcentro") "")
    (PROGN
      (SETQ ErrorDCL T)
      (SET_TILE "error" "Falta la coordenada X del centro del círculo.")
      (MODE_TILE "txcentro" 2)
    )
  )
  (IF (= (GET_TILE "tycentro") "")
    (PROGN
      (SETQ ErrorDCL T)
      (SET_TILE "error" "Falta la coordenada Y del centro del círculo.")
      (MODE_TILE "tycentro" 2)
    )
  )
  (IF (= (GET_TILE "tzcentro") "")
    (PROGN
      (SETQ ErrorDCL T)
      (SET_TILE "error" "Falta la coordenada Z del centro del círculo.")
      (MODE_TILE "tzcentro" 2)
    )
  )
  (IF (= (GET_TILE "tradio") "")
    (PROGN
      (SETQ ErrorDCL T)
      (SET_TILE "error" "Falta el radio del círculo.")
      (MODE_TILE "tradio" 2)
    )
  )
  (IF (= (GET_TILE "tcapa") "")
    (PROGN
      (SETQ ErrorDCL T)

```

```
(SET_TILE "error" "Falta la capa del círculo.")
(MODE_TILE "tcapa" 2)
)
)
(IF (= (GET_TILE "tlin") "")
  (PROGN
    (SETQ ErrorDCL T)
    (SET_TILE "error" "Falta el tipo de línea del círculo.")
    (MODE_TILE "tlin" 2)
  )
)
(IF (= (GET_TILE "tcolor") "")
  (PROGN
    (SETQ ErrorDCL T)
    (SET_TILE "error" "Falta el color del círculo.")
    (MODE_TILE "tcolor" 2)
  )
)
)

(DEFUN ControlLSP (Mensaje)
  (SETQ *error* ErrorLSP)
  (PRINC Mensaje)(TERPRI)
  (COMMAND "_.undo" "_e")
  (SETVAR "cmdecho" 1)
  (PRIN1)
)

(DEFUN C:ModiCir (/ ListaCir X Y Z Capa Color Lin NoLin NoColor PuntoCentro
                  Entidad ErrorDCL ErrorLSP)
  (SETVAR "cmdecho" 0)
  (GRAPHSCR)
  (COMMAND "_.undo" "_be")
  (SETQ ErrorLSP *error* *error* ControlLSP)
  (Datos)
  (COMMAND "_.undo" "_e")
  (SETQ *error* ErrorLSP)
  (SETVAR "cmdecho" 1)
  (PRIN1)
)

(DEFUN C:MC ()
  (C:Modicir)
)
```

Se define un nuevo comando MODICIR y una abreviatura MC a él. Tras la declaraciones de rigor, el programa llama a la función Datos, en la cual se nos pide designar un círculo. Se comprueba si realmente es un círculo extrayendo el par punteado de su nombre; si no lo es se vuelve a realizar la pregunta y, si sí es un círculo, se llama a la función Cuadro.

Cuadro carga el letrero de diálogo y rellena sus casillas accediendo a la lista de la entidad. En el caso del punto del centro, se distingue entre tomar las coordenadas de la lista de la entidad designada o del punto designado mediante el botón *Designar punto* <, dependiendo pues si se entra al cuadro por primera vez o tras haber designado un nuevo centro. Una vez hecho esto se establecen las correspondencias de los ACTION_TILE.

La función Designar permite designar un nuevo centro y vuelve al cuadro. Recuérdese la técnica ya comentada de asignar una variable a lo devuelto por START_DIALOG y de hacerla nil al entrar al cuadro para que no dé problemas. También hemos de tener presente la

posibilidad de cargar o no cargar el cuadro, dependiendo de si ya lo estuviera porque se viene de Designar, o si no lo estuviera porque se acaba de ejecutar el comando.

Tras pulsar el botón *Aceptar* se nos manda a la subrutina *Tiles*, la cual lo primero que hace es llevarnos a *ControlDCL*. *ControlDCL* controla si alguna casilla del letrero está vacía. Si así fuera, se muestra un mensaje de error, se resalta la casilla y da valor *T* a la variable *ErrorDCL*, que hace que el cuadro no se cierre al establecer esa condición en el *ACTION_TILE* para *accept*.

Después de *ControlDCL* seguimos en *Tiles*. En esta subrutina se asignan los valores correspondientes a cada una de las variables implicadas, extrayéndolo de los propios *tiles*. En el caso del radio se comprueba que no sea cero o negativo; si así fuera se establece *ErrorDCL* a *T* para lo mismo que lo explicado anteriormente. En el caso del tipo de línea y del color, se comprueba que el tipo esté cargado y que el color no sea menor de 0 ni mayor de 256, respectivamente. Y se hace lo propio con *ErrorDCL*.

Tras acabar aquí, nos vamos a *Aceptar*, que se encarga de modificar la lista y sustituirla en la Base de Datos. En los casos del centro, el radio y la capa no hay problema alguno, simplemente hay que sustituir una lista por otra. Pero en el caso del tipo de línea y el color, el problema se presenta.

Si el tipo de línea es *PorCapa*, en la lista de definición de la Base de Datos el par punteado correspondiente no aparecerá, por lo que habrá que añadirlo (*APPEND*). Pero si es distinto de *PorCapa*, el par punteado sí aparece, así que habrá que sustituirlo (*SUBST*). En el caso del color ocurre exactamente lo mismo.

Por último, se introducen los cambios en la Base de Datos y se cierra el cuadro.

El resto corresponde a lo que ya conocemos: control de errores AutoLISP y otras características de relleno.

El último ejemplo dice relación a un programa que permite juntar dos curvas splines en una sola. La única condición es que deben tocarse. Se pueden juntar sucesivas splines dos a dos. Veamos el listado:

```
(DEFUN Inic_UneSpline ()
  (WHILE (NOT (SETQ splb (ENTSEL "\nDesignar spline: "))))
  (SETQ splb (CAR splb))
  (WHILE (OR (/= (CDR (ASSOC 0 (ENTGET splb))) "SPLINE")
    (= 1 (REM (CDR (ASSOC 70 (ENTGET splb))) 2)))
    (PROMPT "\nEntidad designada no es una spline abierta.")(TERPRI)
    (WHILE (NOT (SETQ splb (ENTSEL "\nDesignar spline: "))))
    (SETQ splb (CAR splb))
  )
  (WHILE (NOT (SETQ splj (ENTSEL "\nDesignar spline para juntar: "))))
  (SETQ splj (CAR splj))
  (WHILE (OR (EQUAL splb splj)
    (/= (CDR (ASSOC 0 (ENTGET splj))) "SPLINE")
    (= 1 (REM (CDR (ASSOC 70 (ENTGET splj))) 2)))
    (IF (EQUAL splb splj)
      (PROMPT "\nSe ha designado la misma spline.")
      (PROMPT "\nEntidad designada no es una spline abierta.")(
    (WHILE (NOT (SETQ splj (ENTSEL "\nDesignar spline para juntar: "))))
    (SETQ splj (CAR splj))
  )
)
)

(DEFUN Juntar_UneSpline (/ lisb lisj lisv numv pb1 pb2 pj1 pj2 expr n)
```

```
(SETQ lisb (ENTGET splb))
(SETQ lisv (MEMBER (ASSOC 11 lisb) lisb))
(SETQ numv (CDR (ASSOC 74 lisb)))
(SETQ pb1 (CDR (NTH 0 lisv)) pb2 (CDR (NTH (1- numv) lisv)))
(SETQ lisj (ENTGET splj))
(SETQ lisv (MEMBER (ASSOC 11 lisj) lisj))
(SETQ numv (CDR (ASSOC 74 lisj)))
(SETQ pj1 (CDR (NTH 0 lisv)) pj2 (CDR (NTH (1- numv) lisv)))
(COND ((EQUAL pb2 pj1 0.0000001)
      ((EQUAL pb2 pj2 0.0000001) (SETQ lisv (REVERSE lisv)))
      ((EQUAL pb1 pj2 0.0000001) (SETQ lisv (REVERSE lisv) pb2 pb1))
      ((EQUAL pb1 pj1 0.0000001) (SETQ pb2 pb1))
      (T (PROMPT "Entidades no se tocan por un extremo")(QUIT)))
)
(SETQ expr "(command \"_splinedit\" splb \"_f\" \"_a\" pb2 ")
(SETQ n 1)
(REPEAT (1- numv)
  (SETQ expr (STRCAT expr "(cdr (nth " (ITOA n) " lisv)))")
  (SETQ n (1+ n))
)
(SETQ expr (STRCAT expr " \"_x\" \"_x\" \"_x\" \"_x\" )"))
(EVAL (READ expr))
(ENTDEL splj)
(REDRAW splb)
)

(DEFUN C:UneSpline (/ splb splj)
  (SETQ error0 *error* *error* Err_UneSpline)
  (SETVAR "cmdecho" 0)
  (Inic_UneSpline)
  (COMMAND "_undo" "_begin")
  (Juntar_UneSpline)
  (COMMAND "_undo" "_end")
  (SETVAR "cmdecho" 1)(PRIN1)
)

(DEFUN Err_UneSpline (Mensaje)
  (SETQ *error* error0)
  (IF (= Mensaje "quitar / salir abandonar")
    (PRIN1)
    (PRINC (STRCAT "\nError: " Mensaje " ")))
  )
  (COMMAND "_undo" "_end")
  (SETVAR "cmdecho" 1)(PRIN1)
)
```

El programa utiliza el comando EDITSPLINE con la primera spline y, mediante la opción Ajustar, subopción añadir, va añadiendo todos los puntos de ajuste de la segunda spline. Las tangencias en el punto de contacto de ambas splines se pierden. Según cuál de las dos se señale primero, la curva final resultante diferirá ligeramente. Esto se hace apreciable sobre todo en el primer tramo de la segunda spline que se junta.

La función inicial solicita señalar las splines que unir. Se establece mediante WHILE un primer control para obligar al usuario a que señale al menos una entidad. La función ENTSEL se utiliza para señalar una única entidad por un punto. Una vez señalada una entidad, es preciso comprobar que se trata de una spline. Esto se hace accediendo a la Base de Datos y extrayendo el tipo de entidad, asociado al código 0. Además, podría tratarse de una spline cerrada, en cuyo caso no sería posible juntarle a otra. Las splines de este tipo tienen un valor impar asociado al código 70. Por eso se extrae dicho valor, se divide entre 2 y se obtiene el resto mediante REM. Si es 1 significa que el valor es impar y por lo tanto una spline cerrada.

En resumen, si la entidad señalada por el usuario no es spline, o es una spline cerrada, entonces se vuelve a solicitar su designación. Cuando se obtiene una entidad correcta, su nombre de identificación (extraído con `CAR`) se almacena en `splb`. A continuación se solicita señalar la spline para juntar. Se establecen los mismos controles, pero se añade otra posibilidad. Si la entidad señalada es un spline no cerrada, se comprueba si se trata de la misma curva señalada en primer lugar. Esto se hace comparando los dos nombres de identificación `splb` y `splj`. Sólo cuando no sean iguales, se aceptará la segunda spline.

La función para juntar extrae las listas en la Base de Datos de ambas splines mediante `ENTGET` y las almacena en `lisb` y `lisj`. Los vértices de ajuste se encuentran en sucesivas sublistas con código 11. Mediante `MEMBER` se extrae el resto de lista a partir de la primera aparición de un vértice. Así, la lista `lisv` contiene las coordenadas de todos los vértices en sublistas asociadas al código 11. El número de vértices de ajuste se obtiene de la sublista asociada al código 74. Los vértices extremos, que son el primero y último de la spline se obtienen mediante `NTH`. Se almacenan en `pb1` y `pb2`.

La misma operación se realiza con la lista de la segunda spline que se juntará a la primera. Sus datos se almacenan en las mismas variables, pues los de la primera spline ya no interesan, una vez obtenidos sus vértices extremos. Así, `lisv` contendrá la lista con sublistas de vértices, `numv` el número de vértices y `pj1` y `pj2` los vértices extremos de la segunda spline.

Se utiliza `COND` para examinar las cuatro posibilidades y ver por qué extremos se tocan las splines. Si no se tocan, se visualiza un mensaje y se aborta el programa mediante `QUIT`. En función de qué extremos se toquen, se invierte la lista de vértices o se modifica `pb2` para que almacene siempre el punto de contacto. Se utiliza `EQUAL` para comparar la igualdad de puntos, estableciendo una cifra de aproximación.

El mecanismo de unión de las splines va a ser el siguiente: mediante el comando `EDITSPLINE` se designa la primera curva. Se utiliza la opción `Ajustar` y dentro de ella la subopción `añadir`. Se señala el vértice de contacto como punto a partir del cual añadir los demás, y se van proporcionando en orden todos los vértices de la segunda spline que quedan así incorporados a la primera. Pero como el número de vértices es variable, la expresión debe formarse concatenando una cadena de texto que después será convertida en expresión de AutoLISP y evaluada mediante el mecanismo (`EVAL (READ expr)`) ya estudiado, que comentamos en su momento de pasada y aquí lo vemos en acción.

Los vértices se suministran extrayéndolos de la lista `lisv` con `NTH` y `CDR`. Una vez añadidos todos los vértices de la segunda spline a la primera, se borra aquella mediante `ENTDEL`. Se redibuja la spline global resultante mediante `REDRAW`. Si las splines tienen muchos vértices, la operación de juntar puede llevar unos cuantos segundos de tiempo.

18ª fase intermedia de ejercicios

- Realizar un programa que facilite la modificación global de las propiedades de varios textos a la vez.
- Realizar un programa que permita juntar dos polilíneas 3D en una sola.
- Diseñar un programa que haga resaltar las inserciones de todos los bloques de un dibujo utilizando la generación de vectores virtuales (explicado esto en la sección **ONCE.18.2**). Las inserciones serán resaltadas poniéndolas en relieve mediante vídeo inverso

y, al mismo tiempo, visualizando una flecha virtual que señale el punto de inserción de cada bloque.

ONCE.21. ACCESO A ARCHIVOS

Bajo esta sección vamos a estudiar una importante aplicación de los programas en AutoLISP, esto es, la posibilidad de gestionar directamente archivos de texto ASCII. Todo lenguaje de programación que se precie goza de esta característica, ya que le permite acceder a un archivo de texto para leerlo, crearlo o modificarlo.

Si nos damos cuenta, la totalidad de los archivos que hemos venido estudiando en este curso como archivos de personalización, .MNU, .LIN, .PAT, .AHP, .SCR, etcétera, son archivos de texto ASCII. Es por ello, que desde un programa en AutoLISP podremos acceder a sus líneas en cualquier momento, así como crear archivos propios de una manera automática.

ONCE.21.1. Fundamento teórico somero sobre el acceso a archivos

A grandes rasgos, los archivos o ficheros de texto o datos a los que podemos acceder desde la mayoría de los lenguajes de programación son dos: archivos de acceso secuencial y archivos de acceso aleatorio o directo. La diferencia principal entre ambos estriba en la forma en la que almacenan los datos. Por un lado, los archivos de acceso secuencial guardan la información secuencialmente, es decir, dato tras dato organizados todo ellos en una serie de líneas. Tras cada línea se reconoce un retorno de carro con salto de línea (INTRO) y al final del archivo un marca de fin de archivo. Son archivos ASCII que si se abren con cualquier editor se pueden examinar fácilmente.

Por otro lado, los archivos de acceso directo tienen una forma más eficaz de almacenar la información. En la teoría del acceso a estos archivos entra en juego el concepto de *campo clave*. Los datos son guardados según un campo clave o principal que los ordena basándose en su condición. De esta forma, una serie de registros almacenados según un campo clave numérico, serán introducidos en el archivo en su posición correcta, única e inequívoca.

Como norma general pues, los programas que manejen archivos de acceso aleatorio o directo llegarán mucho antes a la información buscada por el usuario, mientras que los programas que manejen archivos de acceso secuencial verán incrementado su tiempo de acceso, ya que han de recorrer todo el fichero hasta dar con la información buscada. Sin embargo, este último tipo de archivos ocupa bastante menos espacio en disco, por la no organización de sus registros (unos detrás de otros), mientras que los archivos de acceso directo ocupan más espacio, ya que los datos se escriben ordenados según dicho campo clave, dejando espacio para los registros vacíos aún no rellenados (registros con "basura").

AutoLISP sólo es capaz de acceder a ficheros de acceso secuencial. Estos ficheros, pueden ser archivos de texto como un archivo de menú o de definición de tipos de línea, o ficheros de datos —también de texto ASCII— que contengan, por ejemplo, coordenadas de puntos de un levantamiento topográfico (típico fichero de salida de una estación total, por ejemplo).

Los ficheros o archivos de texto que no guarden datos divididos en campos y registros, estarán escritos como su autor lo quiso hacer, sin ningún orden lógico aparente. Sin embargo, los ficheros de datos de salida de muchos utensilios (instrumentos topográficos, máquinas de control numérico, etc.) o programas (bases de datos, hojas de cálculo o el propio **AutoCAD**) sí guardan una estructura tipificada y normalizada. La mayoría de ellos dividen sus registros

(líneas) por retornos de carro con salto de línea (INTRO) y separan sus campos (columnas) por algún carácter especial: una coma (formato CDF), un espacio blanco (formato SDF), u otros. Además, los campos suelen estar encerrados entre comillas dobles o simples si son cadenas o sin ningún carácter delimitador si son campos numéricos. Otros ficheros separarán registros y campos, o encerrarán datos, con cualquier otro carácter o técnica.

Sea cual fuere la opción o el tipo de fichero, el programador en **AutoLISP** habrá de examinar antes bien su contenido para estudiar la forma de llegar a extraer los datos necesarios del mismo.

Por último decir que existen tres formas básicas de acceder a un fichero (enseguida veremos cómo se hace en AutoLISP) desde los lenguajes de programación: la de lectura, en la que un imaginario puntero se coloca al principio del archivo exclusivamente para leer datos; la de escritura, en la que el puntero se coloca también al principio para escribir datos, eliminando los que ya hay; y la de añadir datos, en la que el puntero se coloca al final del archivo para agregar datos al mismo, sin afectar a lo que ya está guardado. Es muy importante tener esta última característica en cuenta para evitar errores fatales para el usuario.

ONCE.21.2. Funciones para el manejo de archivos

A continuación iremos describiendo cada una de las funciones de las que disponemos en AutoLISP para acceder a y/o manejar ficheros. Comenzaremos evidentemente por la manera de abrir uno de estos archivos. Para ello debemos recurrir a la función inherente **OPEN**, cuya sintaxis es:

```
(OPEN nombre_archivo modo)
```

Esta función abre un archivo para permitir el acceso a las funciones de entrada y salida de AutoLISP. **OPEN** devuelve un valor de descriptor de archivo que posteriormente utilizaremos para acceder a él, por lo que la práctica lógica consistirá en guardar dicho descriptor para ser posteriormente utilizado. Viene a ser algo parecido que lo que hacíamos con la función **LOAD_DIALOG** para guardar el índice devuelto y luego usarlo en el acceso al cuadro de diálogo.

El nombre de archivo que proporcionamos a **OPEN** será una cadena entrecomillada en la que se especificará el nombre, y en su caso (si no se encuentra en el directorio actual) la ruta de acceso completa, del archivo o fichero en cuestión. Por su lado, el argumento *modo*, indica la manera en que se abrirá el archivo, según la tabla siguiente (el modo también se indica entre comillas dobles):

Argumento <i>modo</i>	Descripción
-----------------------	-------------

"r"	Abre en modo lectura. Sólo se pueden leer o extraer datos del archivo. Si el archivo indicado no existe se devuelve nil.
"w"	Abre en modo escritura. Se escriben datos en el archivo y, si ya existían otros datos, se sobrescriben. Si el archivo indicado no existe se crea.
"a"	Abre en modo aditivo. Se escriben datos en el archivo al final del mismo, tras los datos existentes si hay. Si el archivo indicado no existe se crea.

Veamos varios ejemplos:

```
(SETQ Archivo (OPEN "ejemplo.txt" "w"))  
(SETQ Arch (OPEN "c:/clientes/datos/text/bilbao.dat" "a"))  
(SETQ Des (OPEN "a:\\move.scr" "r"))
```

```
(SETQ Archivo (OPEN "acadiso.lin" "r"))
```

```
(CLOSE descriptor_archivo)
```

CLOSE cierra el archivo válido abierto identificado por su descriptor de archivo (el obtenido con OPEN). Una vez cerrado el archivo, se devuelve `nil` y ya no se puede hacer ninguna operación con él. El descriptor de archivo deja de ser válido y al volver a abrirse mediante OPEN cambiará.

Es necesario cerrar los archivos cuando ya no se van a utilizar. Ejemplo:

```
(CLOSE Archivo)
```

```
(READ-LINE [descriptor_archivo])
```

Una vez abierto un archivo para lectura, utilizaremos la función `READ-LINE` para leer una línea completa de dicho archivo, es decir, hasta el salto de línea. Para ello deberemos indicar el descriptor de archivo que devolvió la función `OPEN` al abrirlo. `READ-LINE` devuelve cada vez una de las líneas como cadena, es decir, entre comillas dobles, hasta que al llegar al final del archivo devuelve `nil`.

Por ejemplo, imaginemos un archivo de texto llamado `DATOS.ML1` que se encuentra en el directorio raíz de un disco duro (por ejemplo `c:\`). Dicho archivo contiene las siguientes líneas:

```
Esto es una prueba  
de lectura de archivos.
```

Primero, pues, deberemos abrirlo para lectura, guardando el descriptor de archivo en una variable que llamaremos `Arch`:

```
(SETQ Arch (OPEN "c:\\datos.ml1" "r"))
```

A continuación, si hacemos:

```
(READ-LINE Arch)
```

AutoLISP devuelve:

```
"Esto es una prueba"
```

Una siguiente instrucción igual `READ-LINE` devolverá:

```
"de lectura de archivos."
```

Una siguiente instrucción igual devolverá `nil`, ya que se ha llegado al final del archivo. Si se desea retornar el puntero de lectura al principio del archivo, es necesario cerrarlo y volverlo a abrir, si no, como observamos, el puntero se sitúa en la siguiente línea a la última leída.

El siguiente pequeño programa comprueba si el tipo de línea `TRAZO_Y_PUNTOX2` se encuentra definido en el archivo `ACADISO.LIN` de **AutoCAD**:

```
(DEFUN C:BuscaLin ()  
  (SETQ Existe nil)  
  (SETQ Arch (OPEN "c:\\archiv~1\\autoca~1\\support\\acadiso.lin" "r"))
```

```
(WHILE (SETQ Lin (READ-LINE Arch))
  (SETQ Lin (STRCASE Lin T))
  (IF (WCMATCH Lin "*trazo_y_puntox2*")
    (SETQ Existe T)
    (IF Existe () (SETQ Existe nil)))
  )
)
(IF Existe
  (PROMPT "\nExiste.")
  (PROMPT "\nNo existe.")
)
(CLOSE Arch)
(PRIN1)
)
```

En un principio se establece la variable `Existe` como `nil`, para futuras ejecuciones del programa, y se abre el archivo en cuestión para lectura. Se establece una condicional que dice que mientras exista la variable `Lin`, que almacena una línea leída del archivo, se realiza todo lo demás. Es decir, mientras `Lin` tenga valor, es igual aquí a mientras queden líneas por leer en el archivo. En el momento en que `Lin` valga `nil` (fin de archivo) ya no se ejecutará ninguna función de la repetitiva condicional.

En esa repetitiva se convierte el contenido de `Lin` (cada línea del archivo) a minúsculas con `STRCASE` y la opción `T`. Esto se realiza porque al ser una cadena se distingue entre mayúsculas y minúsculas, de tal forma que al intentar buscar posteriormente la subcadena `trazo_y_puntox2` en `Lin` (con `WCMATCH`) no se produzca ningún error de interpretación.

Si la subcadena buscada existe se establece `Existe` como `T`, y si no existe se mira si `Existe` ha tomado alguna vez el valor `T` (ha existido). Si así fuera no se haría nada, si no se establecería a `nil`.

Una vez terminado de leer el fichero, se comprueba si la variable `Existe` tiene valor `T` o `nil` (existe o no) y se actúa en consecuencia, emitiendo el mensaje adecuado. Se termina cerrando el archivo y con un `PRIN1` para evitar el `nil` de `CLOSE`.

Si a la función `READ-LINE` no se le especifica un descriptor de archivo, lee una cadena de texto completa desde el teclado. Así por ejemplo, si escribimos en línea de comandos:

```
(READ-LINE)
```

y ahora escribimos en la propia línea de comandos `Prueba de lectura desde teclado.`, AutoLISP devuelve:

```
"Prueba de lectura desde teclado."
```

```
(WRITE-LINE cadena [descriptor_archivo])
```

`WRITE-LINE` funciona de forma inversa a `READ-LINE`, esto es, escribe la cadena indicada como una línea completa en el archivo especificado por su descriptor válido. Evidentemente el archivo deberá estar abierto para escribir o añadir datos, dependiendo de la modalidad que nos interese.

El siguiente programa permite crear automáticamente archivos de guión:

```
(DEFUN C:CreaSCR ()
  (SETQ NombreArchivo (GETSTRING "Introduzca nombre de archivo: "))
  (SETQ DesArchivo (OPEN NombreArchivo "a"))
)
```

```
(WHILE (/= (SETQ Línea (GETSTRING "Introduzca comando (INTRO fin): " T)) "")  
  (WRITE-LINE Línea DesArchivo)  
)  
(CLOSE DesArchivo)  
)
```

Tras introducir un nombre para el archivo y abrirlo, se piden continuamente los comandos necesarios, los cuales se van escribiendo al archivo en cuestión. Al introducir un INTRO se termina el programa —ya que Línea es igual a una cadena nula ("")— y se cierra el archivo.

En este tipo de programas es lógico introducir bastantes controles, por ejemplo para que el usuario introduzca la ruta de acceso con contrabarras y el propio programa las cambie a formato de AutoLISP; es cuestión de jugar con las variables de manejo de cadenas estudiadas.

WRITE-LINE devuelve la cadena en cuestión entre comillas. Si no se indica un descriptor de archivo, la función escribe la cadena, capturada desde el teclado, en la línea de comandos.

```
(READ-CHAR [descriptor_archivo])
```

La siguiente función que veremos, READ-CHAR, lee un carácter del archivo (abierto para lectura) especificado por su descriptor cada vez y devuelve su código ASCII. Así por ejemplo, si un archivo de texto llamado EJEMPLO.DAT (en C:\) contuviera las dos siguientes líneas:

```
Hola  
Hola yo
```

y si se abriera así:

```
(SETQ Hola (OPEN "c:/ejemplo.dat" "r"))
```

sucesivas llamadas con READ-CHAR así:

```
(READ-CHAR Hola)
```

devolverían lo siguiente:

```
72  
111  
108  
97  
10  
72  
111  
108  
97  
32  
121  
111  
nil
```

Cada vez se lee un carácter más y se devuelve su código ASCII. El código 10 corresponde al INTRO entre las dos líneas (retorno de carro con salto de línea). Por compatibilidad con UNIX, en el que el final de línea es siempre el código ASCII 10, READ-CHAR devuelve este código cuando se utiliza en plataformas DOS/Windows y se detecta un INTRO, a

pesar de que en realidad el código ASCII de este carácter es el 13. Véase la lista de códigos ASCII en el **APÉNDICE F**. Cuando no hay más caracteres, READ-CHAR devuelve nil.

Como sabemos, para volver a leer el archivo desde el principio (que el puntero de lectura se coloque en el inicio) deberemos cerrarlo y volverlo a abrir.

NOTA: Para poder tratar con estos códigos ASCII, recuérdense las funciones ASCII y CHR de conversión carácter/código y código/carácter respectivamente, estudiadas en la sección **ONCE.12.**

Si a la función READ-CHAR no le acompaña un descriptor de archivo, lee un carácter de la memoria temporal de entrada del teclado (*buffer* del teclado) y devuelve su código ASCII. Si el *buffer* contiene varios caracteres en el momento de ejecución de READ-CHAR, devuelve el primero de ellos. Al igual que con los ficheros de texto, sucesivas llamadas devolverán los siguientes caracteres. Si no hay ningún carácter en el *buffer* de teclado en dicho momento, READ-CHAR espera a que el usuario entre una serie de caracteres, finalizados con INTRO. Entonces devuelve el primero de los caracteres introducidos; los siguientes READ-CHAR devolverán el resto.

El siguiente ejemplo de READ-CHAR (combinado con READ-LINE) controla si el primer carácter de cada línea de un archivo .LSP es un punto y coma (;). Si así fuera agrega una unidad a un contador. Al final, muestra el número de líneas íntegras de comentarios que existen en el archivo:

```
(DEFUN C:Coment ()
  (SETQ Contador 0)
  (SETQ Archivo (GETSTRING "\nCamino y nombre del archivo .LSP: "))
  (SETQ Desc (OPEN Archivo "r"))
  (WHILE (SETQ Lin (READ-CHAR Desc))
    (IF (= (CHR Lin) ";")
      (PROGN
        (SETQ Contador (1+ Contador))
        (READ-LINE Desc)
      )
    )
  )
  (CLOSE Desc)
  (PROMPT (STRCAT "\nEl total de líneas íntegras de comentarios es: " (ITOA
    Contador) "."))
  (PRIN1)
)
```

Obsérvese la manera conjunta de trabajar READ-CHAR y READ-LINE. READ-CHAR lee el primer carácter de la línea, si es un punto y coma (convertido el código con CHR) añade una unidad al contador y con READ-LINE se lee el resto de la línea. De esta manera hacemos que el puntero de lectura se coloque al principio de la siguiente línea, con lo que podemos volver a empezar. En el caso en el que el primer carácter no sea un punto y coma, se realiza la misma función con READ-LINE pero sin incrementar el contador.

(WRITE-CHAR código_ASCII [descriptor_archivo])

WRITE-CHAR realiza la función inversa a READ-CHAR, es decir, escribe en un archivo cuyo descriptor se especifica, o en pantalla si no se especifica ningún descriptor, el carácter cuyo código ASCII se indica. Además, devuelve este código ASCII.

Así pues, si escribimos:

```
(WRITE-CHAR 72 Desc)
```

siendo Desc un descriptor válido de archivo, en dicho archivo se escribirá H, esto es, el carácter correspondiente al código ASCII 72. Si escribiéramos en línea de comandos:

```
(WRITE-CHAR 72)
```

AutoLISP devolvería:

```
H72
```

ya que, como hemos dicho, escribe el carácter y devuelve el código.

Por las mismas razones de compatibilidad con UNIX que en el caso de READ-CHAR, tanto el código ASCII 10 como el 13 pueden indicarse para escribir retornos de carro con salto de línea (INTRO) o fines de línea.

Veamos ahora otras tres funciones muy útiles para escribir datos en un archivo. La primera es PRIN1, y las otras dos derivadas (PRINT y PRINC) son similares a ella con alguna pequeña diferencia que explicaremos. La sintaxis de PRIN1 es:

```
(PRIN1 [expresión [descriptor_archivo]])
```

Esta función escribe expresiones en un archivo, si se indica un descriptor válido, o en la línea de comandos, si no se indica descriptor alguno. Devuelve la propia expresión.

A diferencia de WRITE-LINE y WRITE-CHAR, PRIN1 permite escribir cualquier expresión en un fichero, sin necesidad de que sea una cadena de texto. Así por ejemplo, la siguiente secuencia:

```
(SETQ Archivo (OPEN "c:/datos.dat" "w"))  
(SETQ Mensualidad (* 2 2350))  
(PRIN1 "Mensualidad" Archivo)  
(PRIN1 Mensualidad Archivo)  
(CLOSE Archivo)
```

abre un archivo para escritura. Asigna a la variable Mensualidad el valor de un producto y, a continuación, escribe en dicho archivo un texto o cadena fija y el valor de la variable expuesta. Por último, el fichero es cerrado. La apariencia ahora del archivo DATOS.DAT sería la que sigue:

```
"Mensualidad"4700
```

Es decir, las expresiones se añaden sin separación de líneas o interlineado y, además las cadenas literales son incluidas con sus comillas dobles correspondientes, al contrario que con WRITE-LINE. Por el contrario, si las expresiones son literales con apóstrofo:

```
(PRIN1 'Mensualidad Archivo)  
(PRIN1 '(SetQ x 5.5) Archivo)
```

se añadirían como tales, pero en mayúsculas, evidentemente:

```
MENSUALIDAD(SETQ X 5.5)
```

Si la expresión es una cadena con caracteres de control, PRIN1 escribe esos caracteres con contrabarra y el número de su código octal. Por ejemplo:

(PRIN1 (CHR 2) Archivo)	escribe y devuelve "\002"
(PRIN1 (CHR 10) Archivo)	escribe y devuelve "\n"
(PRIN1 (CHR 13) Archivo)	escribe y devuelve "\r"

Se puede utilizar PRIN1 sin ningún argumento, que es lo que venimos haciendo en los programas hasta ahora. De esta forma, la función devuelve una cadena nula, es decir, simplemente salta una línea en la línea de comandos de **AutoCAD**, sin ningún otro mensaje. Al incluir pues PRIN1 sin argumentos como última expresión de un programa, haremos que su final sea "limpio". Esta particularidad no se puede utilizar con archivos evidentemente.

NOTA: La utilidad real de PRIN1 es la de escribir expresiones compatibles por ejemplo con LOAD o con COMMAND, que llama, este último, a comandos de **AutoCAD** cuyos mensajes no admiten todos los códigos ASCII.

`(PRINT [expresión [descriptor_archivo]])`

Totalmente idéntica a PRIN1, salvo que salta a nueva línea antes de visualizar o escribir la expresión y añade un espacio blanco al final. La siguiente secuencia:

```
(SETQ Archivo (OPEN "c:/datos.dat" "w"))
(SETQ Mensualidad (* 2 2350))
(PRINT "Mensualidad" Archivo)
(PRINT Mensualidad Archivo)
(CLOSE Archivo)
```

escribiría:

```
"Mensualidad"
4700
```

dejando la primera línea en blanco, ya que antes de escribir la cadena salta una línea también. Al final de ambas líneas hay un espacio blanco.

`(PRINC [expresión [descriptor_archivo]])`

Totalmente idéntica a PRIN1, salvo que los caracteres de control se escriben como tales, no representados por su código octal. Alguno de estos caracteres puede ser representado en el archivo por un símbolo, cosa que apreciaremos al abrirlo con un editor ASCII.

A diferencia de PRIN1, la función PRINC escribe cualquier carácter admitido en un archivo de texto y las expresiones pueden ser leídas directamente con funciones como READ-LINE.

Por ejemplo las siguientes inclusiones del código ASCII del salto de línea con retorno de carro, no aparecerán en octal en el archivo, sino que serán caracteres INTRO verdaderos, es decir, se producirán los saltos de línea con sus retornos de carro:

```
(SETQ k (OPEN "c:/ejem1.doc" "a"))
(PRIN1 "w" k)
(PRINC (CHR 10) k)
(PRIN1 "x" k)
(PRIN1 "y" k)
(PRINC (CHR 10) k)
(PRIN1 "z" k)
```

El resultado será:

```
"w"  
"x"y"  
"z"
```

Y veamos por último otras dos funciones muy utilizadas.

```
(FINDFILE nombre_archivo)
```

La función `FINDFILE` explora directorios en busca del archivo especificado. El archivo se indica entre comillas por ser cadena. Si se especifica sin ruta o camino de acceso, `FINDFILE` buscará en los caminos de archivos de soporte incluidos en el cuadro *Preferencias*, en la pestaña *Archivos* (carpeta *Camino de búsqueda de archivo de soporte*). Si se escribe una ruta de acceso en el argumento *nombre_archivo*, `FINDFILE` buscará el archivo en la ruta especificada.

Si la función `FINDFILE` encuentra el archivo buscado devuelve la cadena que indica su camino y nombre en formato válido de AutoLISP, si no es así, devuelve `nil`.

`FINDFILE` se utiliza eminentemente para comprobar la existencia de archivos o ficheros indicados por el usuario para ser abiertos, ya que de no existir se produciría un error en tiempo de corrida de AutoLISP. Vamos a ver un ejemplo ya explicado en el que se ha añadido esta nueva característica; se corresponde con el programa contador de líneas íntegras de comentarios en los ficheros `.LSP`:

```
(DEFUN C:Coment ()  
  (SETQ Contador 0)  
  (SETQ Archivo (GETSTRING "\nCamino y nombre del archivo .LSP: "))  
  (IF (NOT (FINDFILE Archivo))  
    (PROGN  
      (PROMPT "\nError: el archivo especificado no existe.\n")  
      (EXIT)  
    )  
  )  
  (SETQ Desc (OPEN Archivo "r"))  
  (WHILE (SETQ Lin (READ-CHAR Desc))  
    (IF (= (CHR Lin) ";")  
      (PROGN  
        (SETQ Contador (1+ Contador))  
        (READ-LINE Desc)  
      )  
      (READ-LINE Desc)  
    )  
  )  
  (CLOSE Desc)  
  (PROMPT (STRCAT "\nEl total de líneas íntegras de comentarios es: " (ITOA  
    Contador) "."))  
  (PRIN1)  
)
```

```
(GETFILED título_letrero archivo_defecto patrón_extensión modo)
```

La función `GETFILED` muestra el letrero estándar de gestión de archivos de **AutoCAD**. Puede ser muy útil cuando se programe con cuadros de diálogo en DCL, ya que es más vistoso escoger un archivo de un letrero que escribirlo en línea de comandos.

Esta función devuelve una cadena que contiene la ruta y el nombre del archivo seleccionado en el cuadro. Este nombre y ruta están en formato AutoLISP, por lo que pueden

ser perfectamente guardados en una variable y posteriormente utilizados por la función `OPEN`, por ejemplo. Expliquemos los argumentos.

- `título_letrero` es una cadena que formará el nombre en la parte superior el cuadro, en la barra de título (azul comúnmente). Si se indica una cadena nula (""), aparecerá el título por defecto *Abrir dibujo*.

- `archivo_defecto` es el archivo que aparecerá en la casilla *Nombre de archivo:* por defecto. Su extensión será la indicada con el siguiente argumento, si se indica. Si se introduce cadena nula (""), no aparecerá ningún archivo por defecto.

- `patrón_extensión` especifica la extensión o extensiones que admitirá el cuadro. Se pueden indicar varias separadas por punto y coma (`mnu;mns;mn1;*`), siendo la primera de ellas la que aparecerá primera en la lista desplegable *Tipos de archivos:* y la que aparecerá por defecto en la casilla *Nombre de archivo:*, siempre que haya un archivo indicado por defecto. Si se indica una cadena nula (""), se especifica el patrón *Todos los archivos* (`*.*`). Este patrón también se consigue con una cadena que encierre un único asterisco `"*"`.

- `modo` es un valor entero (codificado en bits) que condiciona el letrero según los valores siguientes:

Valor de <i>modo</i>	Descripción
0	No es en realidad un valor de bit. Se define cuando no se indica ninguno de los siguientes. El archivo seleccionado habrá de existir y en el botón de la derecha de la casilla <i>Nombre de archivo:</i> aparecerá la leyenda <i>Abrir</i> , ya que es un cuadro para leer de un archivo.
1	El archivo seleccionado habrá de ser nuevo, es decir, será un archivo que se creará. Por lo tanto, lo lógico será escribir un nombre de archivo no existente en la casilla <i>Nombre de archivo:</i> , tras escoger el directorio donde se creará. A la derecha de esta casilla la leyenda del botón adjunto será <i>Guardar</i> , ya que es un cuadro para escribir en un archivo. Si se elige un archivo existente aparecerá un mensaje de error advirtiendo de su existencia, ofreciéndonos la posibilidad de sobrescribirlo. El valor 1 es el bit 0.
2	Desactiva el botón <i>Teclearlo</i> . Este bit se define si se llama a la función <code>GETFILED</code> mientras otro cuadro de diálogo está activo (en caso contrario, obliga a cerrar el otro cuadro de diálogo). Si no se define este bit, se activa el botón <i>Teclearlo</i> . Cuando el usuario selecciona el botón, el cuadro de diálogo desaparece y <code>GETFILED</code> devuelve 1. El valor 2 es el bit 1.
4	Permite al usuario escribir una extensión de nombre de archivo arbitraria o bien no escribir ninguna. Si no se define este bit, <code>GETFILED</code> sólo acepta la extensión o extensiones especificadas en el argumento <code>patrón_extensión</code> , y si el usuario la escribe en la casilla de texto <i>Nombre de archivo:</i> , la añade al nombre del archivo. El valor 4 es el bit 2.
8	Si se define este bit y no se define el bit 0 (valor 1), <code>GETFILED</code> inicia en la biblioteca una búsqueda del nombre de archivo escrito. Si encuentra el archivo y el directorio en el orden de búsqueda en la estructura, descompone el camino y sólo devuelve el nombre del archivo. Esto no ocurre si los archivos que se buscan tienen el mismo nombre pero se encuentran en distintos directorios. Si no se define este bit, <code>GETFILED</code> devuelve el nombre completo del

Valor de <i>modo</i>	Descripción
	archivo, incluido el nombre del camino. Definiremos este bit si utilizamos el cuadro de diálogo para abrir un archivo existente
	cuyo nombre deseamos guardar en el dibujo (u otra base de datos). El valor 8 es el bit 3.

Los valores mencionados pueden sumarse para combinar las distintas acciones, pudiendo pues indicar valores desde 0 hasta 15 para el argumento *modo*.

Si se abre un cuadro de gestión de archivos para abrir uno (con valor de *modo* 0, por ejemplo) y la extensión patrón única o principal (la primera) es *dwg*, al cuadro se le añade un anexo por la derecha para mostrar una presentación preliminar del dibujo de **AutoCAD**, si la tuviere.

Como ya se ha dicho, la devolución de `GETFILED` está en formato AutoLISP, concretamente con doble contrabarra (contrabarra carácter de control y carácter contrabarra) para separar directorios, por lo que una secuencia como la que sigue, por ejemplo, sería totalmente válida:

```
(SETQ Archivo (GETFILED "Abrir archivo ASCII" "" "txt;dat;*" 0))  
(SETQ Des (OPEN Archivo "r"))
```

19ª fase intermedia de ejercicios

- Crear un programa para realizar conversiones de unidades con respecto a las existentes en el archivo `ACAD.UNT`. Presentará un menú textual en el que se podrá escoger la categoría de las unidades y, posteriormente, el tipo de unidad de origen y el de destino.
- Realizar un programa que permita crear de un modo automático tipos de línea complejos con texto. Se solicitarán al usuario todos los datos necesarios y después de presentará una muestra del tipo de línea creado. Tras la aceptación por parte del usuario, se incluirá la definición en el archivo indicado por él también.

ONCE.22. FUNCIONES DE CHEQUEO

Se estudian ahora las funciones de AutoLISP que se utilizan para examinar los símbolos de los programas, sean variables de usuario, *subrs*, valores concretos, funciones, etcétera, y detectar una característica en concreto. Se puede conocer de este modo si un dato es cero o no, si una variable existe o es nil, y demás.

Todas estas funciones las hemos venido sustituyendo por otros mecanismos totalmente lícitos y que, en el fondo, se utilizan más. Sin embargo, hemos de conocerlas e incluso veremos alguna que nos descubrirá nuevas posibilidades.

La primera que veremos se corresponde con la sintaxis:

`(ATOM elemento)`

`ATOM` examina el elemento indicado como argumento único y devuelve `T` si se trata de un átomo o `nil` si es una lista. Todo lo que no sea una lista se considera un átomo. Veamos algunos ejemplos:

(ATOM '(3 5))	devuelve nil
(ATOM 6)	devuelve T
(ATOM k)	devuelve T
(ATOM (SETQ x 6.7))	devuelve T
(ATOM '(SETQ x 6.7))	devuelve nil

`(BOUNDP elemento)`

Esta función devuelve T si el símbolo indicado está asociado a un valor distinto de nil. El símbolo puede ser una variable, un nombre de función de usuario o incluso el nombre de una subrutina de AutoLISP. Únicamente si el símbolo es nil, BOUNDP devuelve nil.

Puesto que BOUNDP examina símbolos, hay que indicar estos sin evaluar, es decir precedidos del carácter de QUOTE: el apóstrofo ('). Veamos unos ejemplos; supongamos las siguientes declaraciones:

```
(SETQ x 24)
(DEFUN FuncMi () (SETQ n (1+ n)))
(SETQ y x)
```

Veamos los resultados de la aplicación de BOUNDP:

(BOUNDP 'x)	devuelve T
(BOUNDP 'FuncMi)	devuelve T
(BOUNDP 'y)	devuelve T
(BOUNDP 'h)	devuelve nil
(BOUNDP 'FuncOt)	devuelve nil
(BOUNDP 'getpoint)	devuelve T

Esta función puede resultarnos útil en ciertos momentos. Quizá para saber si una variable es nil o T no la utilizaremos nunca, ya que estas expresiones son iguales:

```
(IF Radio...
(IF (BOUNDP 'Radio)...
```

Pero estudiemos un ejemplo: quizás un programa necesite saber en tiempo real si una variable a la que se va a dar un valor existe como función inherente de AutoLISP. Sabemos que no debemos utilizar variables que sean nombres de funciones AutoLISP, ya que perderán, las funciones, su utilidad y valor. Sabemos también que existe la función ATOMS-FAMILY que nos permite saber si cierta *subr* existe o no, o incluso si un nombre de variable está ya utilizado o no. Puede que ATOMS-FAMILY sea más engorroso de utilizar que BOUNDP dentro de un programa.

En estos casos podremos utilizar BOUNDP con este objetivo: conocer si una variable que va a ser creada en tiempo real existe ya, sea de usuario o como nombre de inherente de AutoLISP, o no.

`(LISTP elemento)`

LISTP, por su lado, explora el elemento indicado y devuelve T en el caso de que se trate de una lista y nil si no lo es. Supongamos:

```
(SETQ JogtGt '(10 25 0))
(SETQ x 45)
```

Veamos los resultados:

(LISTP JogtGt)	devuelve T
(LISTP 'JogtGt)	devuelve nil
(LISTP x)	devuelve nil
(LISTP (LIST y z))	devuelve T

`(NUMBERP elemento)`

NUMBERP examina el resultado de la evaluación del elemento indicado como argumento y devuelve T si dicho resultado es un número entero o real. De lo contrario devuelve nil. Supongamos:

```
(SETQ x 25)
(SETQ a "Hola, ¿qué tal?")
```

Veamos los resultados:

(NUMBERP x)	devuelve T
(NUMBERP a)	devuelve nil
(NUMBERP (SETQ y (+ 4.5 67)))	devuelve T
(NUMBERP 'x)	devuelve nil

Una función que puede resultar interesante a la hora de operar con ciertas variables de las cuales no conocemos su tipo de contenido exacto.

`(MINUSP elemento)`

MINUSP devuelve T si el valor del elemento indicado es un número real o entero negativo. En caso contrario devuelve nil. Ejemplos:

(MINUSP -5)	devuelve T
(MINUSP -1.8936)	devuelve T
(MINUSP (* 2 5))	devuelve nil
(MINUSP (+ -5 6))	devuelve nil
(MINUSP (+ -5 4))	devuelve T
(MINUSP (SETQ z -78))	devuelve T

`(ZEROP elemento)`

ZEROP devuelve T si el valor del elemento indicado es cero; en caso contrario devuelve nil. El resultado ha de ser numérico, si no se produce un error `bad argument type`. Supongamos:

```
(SETQ x 25)
(SETQ y 0)
```

Vemos ahora los resultados:

(ZEROP x)	devuelve nil
(ZEROP y)	devuelve T
(ZEROP (- x x))	devuelve T
(ZEROP (* y -1))	devuelve T

`(NULL elemento)`

Esta función examina el valor asociado al elemento y devuelve `T` si dicho valor es `nil`. En caso contrario devuelve `nil`.

Existe una diferencia importante entre esta función y `BOUNDP`. En este caso se examinan resultados de evaluación de símbolos y no los propios símbolos. Por eso no interesa indicar literales con `NULL`, puesto que el resultado de la evaluación de un literal de un símbolo es el propio nombre del símbolo. Como ese valor no es `nil`, `NULL` aplicado a literales devolverá siempre `nil`.

Supongamos las siguientes igualdades y la función definida:

```
(SETQ x 35)
(DEFUN GoiBeh () (SETQ n (1+ n)))
(SETQ y x)
```

Veamos los distintos resultados:

<code>(NULL x)</code>	devuelve <code>nil</code>
<code>(NULL GoiBeh)</code>	devuelve <code>nil</code>
<code>(NULL 'GoiBeh)</code>	devuelve <code>nil</code>
<code>(NULL ghtF)</code>	devuelve <code>T</code>
<code>(NULL w)</code>	devuelve <code>T</code>

Veremos ahora una función importante a la que seguro podremos sacar mucho partido.

`(TYPE elemento)`

La función `TYPE` devuelve el tipo de elemento indicado. Este elemento puede ser un símbolo, un valor concreto, una expresión AutoLISP, un nombre de función...

`TYPE` devuelve el nombre del tipo atendiendo a la siguiente tabla:

Devolución	Significado
REAL	Valor numérico real.
INT	Valor numérico entero.
STR	Valor textual (cadena alfanumérica).
FILE	Descriptor de archivo.
PICKSET	Conjunto de selección de AutoCAD .
ENAME	Nombre de entidad de AutoCAD .
SYM	Símbolo (variable).
LIST	Lista o función de usuario.
SUBR	Función inherente de AutoLISP o subrutina.
PAGETB	Tabla de paginación de funciones.
EXSUBR	Subrutina externa.

Supongamos las siguientes declaraciones como ejemplo:

```
(SETQ x 53 y 27.5 Txt "HOLA" Listal '(a b c))
(SETQ IndArch (OPEN "prueba.lsp" "a"))
(SETQ Conj (SSGET) NomEnt (SSNAME Conj 0))
```

Veamos ahora como responderían los diversos elementos a una función TYPE:

(TYPE x)	devuelve INT
(TYPE 'x)	devuelve SYM
(TYPE y)	devuelve REAL
(TYPE IndArch)	devuelve FILE
(TYPE Conj)	devuelve PICKSET
(TYPE NomEnt)	devuelve ENAME
(TYPE Txt)	devuelve STR
(TYPE Listal)	devuelve LIST
(TYPEsetq)	devuelve SUBR
(TYPE (SETQ z (+ 1 3)))	devuelve INT

NOTA: Las tablas de paginación (tipo PAGETB) son elementos que se añaden como primer término de las listas que definen funciones de usuario, cuando la paginación virtual de funciones se encuentra activada. De todo esto se hablará al explicar la función VMON en la sección **ONCE.24.**

El tipo devuelto por TYPE siempre está en mayúsculas y es un nombre no una cadena (no está entrecomillado).

ONCE.22.1. Rastreo

Para acabar esta sección veremos dos últimas funciones denominadas de rastreo. Se llaman así porque se dedican a la parte de depuración de un programa en AutoLISP. Estas dos funciones se llaman TRACE y UNTRACE.

`(TRACE función1 [función2...])`

TRACE marca las funciones indicadas con un atributo de rastreo y devuelve el nombre de la última función. La forma de utilizar TRACE es, generalmente, escribiéndola en la línea de comando de **AutoCAD**. Una vez definidas así las funciones que queremos rastrear, ejecutamos el programa en cuestión.

Cada vez que la función indicada en TRACE es evaluada se visualiza en pantalla la llamada a esa función y, cuando termina, el resultado de la evaluación. Las distintas llamadas aparecen con sangrados de línea proporcionales al nivel de anidación de las funciones. El texto que indica la entrada en la función es *Entrada de Función:*, siendo en las versiones anglosajonas del programa: *Entering Función:* (*Función* se refiere al nombre de la función). El resultado se precede del texto fijo *Resultado:* (*Result:* en inglés).

NOTA: El nombre de la función utilizada en TRACE ha de ser uno de una función definida por el usuario. TRACE se puede utilizar antes de cargar el programa en memoria o después.

`(UNTRACE función1 [función2...])`

Esta función desactiva los atributos de rastreo activados por TRACE. Devuelve el nombre de la última función indicada.

Para aclarar el significado de estas funciones veamos un pequeño ejemplo. Supongamos el siguiente programa:

```
(DEFUN PidePto ()  
  (SETQ pt2 (GETPOINT pt1 "Nuevo punto: "))(TERPRI)
```



```
(COMMAND "linea" pt1 pt2 "")
(SETQ pt1 pt2)
)
(DEFUN C:DibuLin ()
(SETQ pt1 (GETPOINT "Introducir punto: "))(TERPRI)
(REPEAT 4
  (PidePto)
)
)
```

Si hiciéramos:

```
(TRACE PidePto)
```

y ejecutáramos el nuevo comando de **AutoCAD** DIBULIN, el resultado podría ser:

```
Entrada de PIDEPTO:
Comando: Resultado: (200.0 50.0 0.0)
Entrada de PIDEPTO:
Comando: Resultado: (205.0 65.0 0.0)
Entrada de PIDEPTO:
Comando: Resultado: (230.0 70.0 0.0)
Entrada de PIDEPTO:
Comando: Resultado: (250.0 100.0 0.0)
```

Cada vez que se llama a la función dentro de la repetitiva (REPEAT 4), esto es un total de cuatro veces, aparece el mensaje Entrada de PIDEPTO:, que aparece con dos sangrados sucesivos, porque esta llamada está incluida dentro del DEFUN C:Dibulin y a su vez dentro de REPEAT 4, por lo tanto en un segundo nivel de anidación. Una vez evaluada la función se visualiza el resultado.

ONCE.23. OPERACIONES BINARIAS LÓGICAS

Bajo esta sección se agrupa una serie de funciones de AutoLISP que realizan operaciones a nivel binario. Esto quiere decir que, aunque admiten que se indiquen los números en decimal o en hexadecimal, los consideran como binarios (conjunto de unos y ceros). Su utilidad es por tanto muy específica.

NOTA: La explicación exhaustiva del sistema binario (funcionamiento, operaciones, conversiones...) o del álgebra de Boole (tablas de verdad, verdad y falsedad...) escapa a los objetivos de esta sección. Aquí nos limitaremos única y exclusivamente a mostrar las funciones AutoLISP que manejan cifras binarias a nivel lógico.

```
(~ valor_numérico)
```

Esta función devuelve la negación lógica (NOT) de una cifra binaria, es decir el complemento a 1. El número indicado ha de ser entero. Veamos unos ejemplos:

(~ 5)	devuelve -6
(~ -6)	devuelve 5
(~ 0)	devuelve -1
(~ 54)	devuelve -55

Recordemos que el carácter ~ (tilde) corresponde al código ASCII 126, por lo que se escribe con la combinación ALT+126.

`(BOOLE operación [valor_entero1 valor_entero2...])`

BOOLE realiza una operación booleana general a nivel binario. El argumento operación es un número entre 0 y 15 que representa una de las 16 operaciones booleanas posibles. Los valores enteros indicados se combinarán bit a bit de acuerdo con la función booleana especificada.

Esto significa que el primer bit del primer entero se combina con el primer bit del segundo entero y así sucesivamente. El resultado final será 0 ó 1 según la tabla de verdad de la función booleana indicada. Lo mismo con el resto de bits de los valores enteros especificados. La combinación final de todos los bits resultantes dará el número entero final que devuelve la función.

Algunos de los valores de *operación* se corresponden con las operaciones booleanas estándar. Estos valores, sus correspondencias y el resultado cierto (1) dependiendo de los números enteros se muestran en la tabla siguiente:

Valor de <i>operación</i>	Booleana estándar	Resultado es 1 si...
1	AND (Y lógico)	todos los bits de entrada son 1 (A y B...).
6	XOR (O lógico exclusivo)	sólo uno de los bits de entrada es 1 (o A o B...).
7	OR (O lógico)	al menos 1 de los bits de entrada es 1 (A o B...).
8	NOT (NO lógico)	ninguno de los bits de entrada es 1 (no A no B...).

Veamos un ejemplo. Supongamos que escribimos:

```
(BOOLE 6 8 12 7)
```

Esto equivale a un XOR de:

8	que en binario es	1000
12	que en binario es	1100
7	que en binario es	111

Por lo tanto, la operación XOR se realizará cuatro veces:

XOR	de	1	1	0	devuelve 0
XOR	de	0	1	1	devuelve 0
XOR	de	0	0	1	devuelve 1
XOR	de	0	0	1	devuelve 1

El resultado final es el número binario 0011 que equivale al 3 decimal. Por lo tanto:

```
(BOOLE 6 8 12 7)           devuelve 3
```

`(LOGAND [valor_entero1 valor_entero2...])`

Esta función realiza un Y lógico (AND) a nivel binario de los valores indicados. Estos valores han de ser enteros; el resultado es también un entero.

En un caso particular de operación booleana con `BOOLE`, `LOGAND` equivale a hacer `BOOLE 1`. Ejemplo:

`(LOGAND 5 7 12 14)` devuelve 4

`(LOGIOR [valor_entero1 valor_entero2...])`

Realiza un *O* lógico (`OR`) a nivel binario. Los valores serán enteros y el resultado también (equivale a `BOOLE 7`). Ejemplo:

`(LOGIOR 1 4 9)` devuelve 13

`(LSH valor_entero número_bits)`

Esta función devuelve el desplazamiento a nivel binario de un registro del número entero indicado en un valor del número de bits especificado como segundo argumento. Si éste es positivo, el entero se desplaza hacia la izquierda. Si el número de bits es negativo, el entero se desplaza a la derecha. Los dígitos que faltan entran como ceros, y se pierden los dígitos salientes.

En la siguiente tabla se muestran varios ejemplos donde se indica en primer lugar la función AutoLISP, después el número entero original indicado en la función; a continuación el binario equivalente a dicho entero; después el valor del desplazamiento especificado; a continuación el binario resultante una vez aplicado ese desplazamiento; por último el entero equivalente a ese binario, que es el resultado final que devuelve `LSH`:

Función	E	Binario	Desplazam.	Binario res.	Entero res.
<code>(LSH 2 1)</code>	2	10	1	100	4
<code>(LSH 2 -1)</code>	2	10	-1	1	1
<code>(LSH 40 2)</code>	40	101000	2	10100000	160
<code>(LSH 11 -1)</code>	11	1011	-1	101	6

ONCE.24. GESTIÓN DE LA MEMORIA

Esta última sección hábil contiene funciones de AutoLISP que gestionan la memoria disponible para almacenar las variables, definiciones de funciones y valores de los programas cargados. Resultan útiles cuando los requerimientos de memoria son grandes o la memoria disponible es reducida, lo que puede provocar problemas.

AutoCAD utiliza automáticamente la cantidad de memoria que necesita en cada caso. Pero en sistemas con una capacidad limitada puede hacerse necesario un ajuste de la misma. Hay que tener siempre presente que cuanto mayor es el número de variables y de funciones definidas, mayores son las necesidades de memoria. Al mismo tiempo, si sus nombres ocupan más de seis caracteres se consume más memoria. Todas estas circunstancias pueden forzar al usuario a liberar memoria o a activar la llamada paginación virtual de funciones, concepto que se explica a continuación.

Enfilemos pues la recta final de este **MÓDULO**.

`(VMON)`

Esta función activa la *paginación virtual de funciones*. Ésta no resulta necesaria hasta que se agotan todos los demás tipos de memoria virtual, lo que ocurre muy raramente en la

mayoría de las plataformas. En algún caso, cuando se tiene cargado en memoria un programa extenso de AutoLISP o varios programas, puede ser insuficiente el espacio de memoria nodal disponible. La memoria nodal almacena todos los símbolos de funciones de usuario y de variables cargados en memoria. Estos símbolos son los también llamados *nodos*.

Si esto ocurre, se puede emplear `VMON` para activar la paginación virtual de funciones. Desde el momento en que se llame a `VMON`, esa paginación virtual afectará únicamente a las funciones de usuario cargadas a partir de ese momento. Cada vez que se carguen nuevas funciones de usuario y la memoria nodal sea insuficiente, AutoLISP irá evacuando las funciones poco utilizadas a un archivo temporal controlado por la paginación de archivos de **AutoCAD**. Si esas funciones evacuadas se volvieran a necesitar, AutoLISP volvería a cargarlas en memoria nodal. Estos intercambios son automáticos y transparentes de cara al usuario.

Si se dispone de suficiente memoria RAM ampliada o paginada, la utilización de esa memoria será lógicamente mucho más rápida que la paginación en disco.

Las funciones de usuario cargadas antes de la llamada a `VMON` no pueden ser evacuadas de la memoria nodal, por lo tanto seguirán ocupando espacio. De la misma manera, todos los símbolos de variables continúan en memoria nodal y no son afectados por `VMON`. Esto quiere decir que aunque se hayan ampliado mucho las posibilidades de cargar programas largos, sigue existiendo una limitación de memoria nodal disponible.

Con `VMON` activada, todas las nuevas `DEFUN` colocan un nuevo nodo llamado *tabla de paginación* como primer elemento añadido a la lista de AutoLISP que define la función. Este primer elemento no es visible a la hora de escribir (en pantalla o en un archivo) esa función. Pero se puede obtener con `CAR`. La función `TYPE` devuelve el tipo de elemento `PAGETB` para esas tablas de paginación (como se explicó en su momento).

`(GC)`

Cada vez que se pretende cargar en memoria un nuevo símbolo de función o de variable, AutoLISP busca en la memoria nodal nodos libres donde almacenarlo. La simple operación de crear una variable y atribuirle un valor, por ejemplo:

```
(SETQ Var 27.53)
```

requiere dos nodos: uno para almacenar en memoria el nombre del símbolo (en este caso `Var`) y otro para almacenar su valor (aquí `27.53`).

Si el nombre del símbolo tiene seis caracteres o menos, se almacena directamente en el nodo. Si tiene más, se toma espacio adicional de la memoria nodal o montón. De ahí la importancia de utilizar nombres de símbolos (funciones de usuario o variables) con menos de seis caracteres.

Si no existen nodos libres para almacenar ese símbolo, AutoLISP recupera automáticamente el espacio inutilizado (nodos que se han liberado porque ya no tienen ningún símbolo asociado). Si esto resulta insuficiente, solicita memoria adicional del montón para crear nuevos nodos. En el momento en que se agote esta memoria, tendría que recurrir a la paginación virtual de funciones si se encuentra activada (si se ha empleado `VMON`).

El espacio de memoria tomada del montón para crear nuevos nodos ya no puede ser devuelto al montón hasta salir de **AutoCAD**. Sin embargo existe la posibilidad de forzar la recuperación de esa memoria mediante `GC`. Esta recuperación de memoria lleva bastante tiempo y su utilidad es limitada. Es preferible dejar que AutoLISP vaya recuperando automáticamente los nodos liberados en memoria sin necesidad de llamar a esta función.

`(ALLOC valor_entero)`

Los nodos en AutoLISP tienen un tamaño de 12 octetos. Para evitar un fraccionamiento excesivo de la memoria nodal, los nodos se agrupan en segmentos. Cada segmento tiene un tamaño implícito de 514 nodos, es decir, 6168 octetos.

Si se desea especificar un número de nodos diferente a 514 como tamaño del segmento, se puede hacer con `ALLOC`:

```
(ALLOC 1024)
```

Este ejemplo asignará a cada segmento un tamaño de 1024 nodos, lo que representará 10240 octetos.

Asignando un tamaño adecuado a los segmentos se pueden reducir las operaciones de recuperación de memoria inutilizada, ganando en velocidad de ejecución del programa. De todas formas, es preferible dejar la tarea de gestión de memoria al mecanismo automático de atribución de nodos de AutoLISP.

`(EXPAND valor_entero)`

El espacio de memoria para almacenar valores de cadenas se toma de la misma memoria montón que los segmentos de nodos. Estos valores de cadenas son de todo tipo: nombres de símbolos de más de seis caracteres, valores concretos de variables de texto, textos de mensajes, textos de opciones de menú, etcétera.

Por lo tanto en la memoria de montón, además de la utilizada para los segmentos de nodos, se requiere una parte para cadenas de texto. La función `EXPAND` reserva explícitamente para nodos un número determinado de segmentos de la memoria montón. Ejemplo:

```
(EXPAND 10)
```

Este ejemplo reservará 10 segmentos que, si son de 514 nodos cada uno, representarán un espacio de 51400 octetos. No es posible reservar toda la memoria montón disponible para segmentos de nodos. AutoLISP necesita que exista una parte libre para la memoria de cadenas.

`(MEM)`

Esta función visualiza el estado actual de la memoria en AutoLISP, y devuelve `nil`. Por ejemplo la función podría visualizar:

```
Nodos:          20224
Nodos libres:    40
Segmentos:       76
Asignado:        256
Colecciones:     5
nil
```

En versiones idiomáticas inglesas de **AutoCAD** la lista sería así:

```
Nodes:          20224
Free nodes:     40
Segments:       76
Allocate:       256
Collections:    5
```

nil

Cada término indica lo siguiente:

- Nodos: el número total de nodos atribuidos hasta ese momento.
- Nodos libres: es el número de nodos que se encuentran libres como consecuencia de la recuperación de memoria no utilizada.
- Segmentos: es el número de segmentos atribuidos.
- Asignado: indica el tamaño en nodos del segmento actual.
- Colecciones: es el total de recuperaciones de memoria no utilizada, que han sido efectuadas (ya sea automática o manualmente).

ONCE.25. CÓDIGOS Y MENSAJES DE ERROR

ONCE.25.1. Códigos de error

En la tabla siguiente se muestran los valores de códigos de error generados por AutoLISP. La variable de sistema `ERRNO` toma uno de estos valores cuando una llamada de función de AutoLISP provoca un error que **AutoCAD** detecta después.

Código	Significado
0	Sin errores
1	Nombre no válido de tabla de símbolos
2	Nombre no válido de conjunto de selección o entidad
3	Se ha superado el número máximo de conjuntos de selección
4	Conjunto de selección no válido
5	Uso no adecuado de definición de bloque
6	Uso no adecuado de referencia externa
7	Selección de objeto: fallo en la designación
8	Final de archivo de entidad
9	Final de archivo de definición de bloque
10	No se ha encontrado la última entidad
11	Intento no válido de suprimir un objeto de ventana gráfica
12	Operación no permitida durante el comando <code>POL</code>
13	Identificador no válido
14	Identificadores no activados
15	Argumentos no válidos en la petición de transformación de coordenadas
16	Espacio no válido en la petición de transformación de coordenadas
17	Uso no válido de entidad suprimida
18	Nombre de tabla no válido
19	Argumento de función de tabla no válido
20	Intento de definir una variable de sólo lectura
21	No puede utilizarse el valor cero
22	Valor incorrecto
23	Regeneración compleja en curso
24	Intento de cambiar el tipo de entidad
25	Nombre de capa erróneo
26	Nombre de tipo de línea erróneo
27	Nombre de color erróneo
28	Nombre de estilo de texto erróneo
29	Nombre de forma erróneo
30	Campo para el tipo de entidad erróneo
31	Intento de modificar una entidad suprimida
32	Intento de modificar una subentidad <code>SEQEND</code>
33	Intento de cambiar identificador

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en AutoLISP

Código	Significado
34	Intento de modificar la visibilidad en pantalla gráfica
35	Entidad en capa bloqueada
36	Tipo de entidad errónea
37	Entidad polilínea errónea
38	Entidad compleja incompleta en el bloque
39	Campo de nombre de bloque no válido
40	Campos de identificador de bloque repetidos
41	Campos de nombre de bloque repetidos
42	Vector normal erróneo
43	Falta el nombre de bloque
44	Faltan los indicadores de bloques
45	Bloque anónimo no válido
46	Definición de bloque no válida
47	Falta un campo obligatorio
48	Tipo de datos extendidos (XDATA) no reconocido
49	Anidación errónea de lista en XDATA
50	Ubicación errónea del campo APPID
51	Se ha superado el tamaño máximo de XDATA
52	Selección de entidad: respuesta nula
53	APPID duplicada
54	Intento de crear o modificar objeto de pantalla gráfica
55	Intento de crear o modificar RefX, XDef o XDep
56	Filtro SSGET: fin de lista no esperado
57	Filtro SSGET: falta operando de prueba
58	Filtro SSGET: cadena <i>opcode</i> (-4) no válida
59	Filtro SSGET: anidación errónea o cláusula condicional vacía
60	Filtro SSGET: incoherencia en principio y final de cláusula condicional
61	Filtro SSGET: número erróneo de argumentos en cláusula condicional (con NOT o XOR)
62	Filtro SSGET: superado el límite máximo de anidación
63	Filtro SSGET: código de grupo no válido
64	Filtro SSGET: prueba de cadena no válida
65	Filtro SSGET: prueba de vector no válida
66	Filtro SSGET: prueba de número real no válida
67	Filtro SSGET: prueba de número entero no válida
68	El digitalizador no es un tablero
69	El tablero no está calibrado
70	Argumentos de tablero no válidos
71	Error ADS: no es posible asignar buffer nuevo de resultados
72	Error ADS: se ha detectado un puntero nulo
73	No es posible abrir el archivo ejecutable
74	La aplicación ya está cargada
75	Número máximo de aplicaciones ya cargadas
76	No es posible ejecutar la aplicación
77	Número de versión no compatible
78	No es posible descargar la aplicación anidada
79	La aplicación no se descarga
80	La aplicación no está cargada
81	Memoria insuficiente para cargar la aplicación
82	Error ADS: matriz de transformación no válida
83	Error ADS: nombre de símbolo no válido
84	Error ADS: valor de símbolo no válido
85	Operación AutoLISP/ADS no permitida mientras se presenta un cuadro de diálogo

ONCE.25.2. Mensajes de error

Como ya sabemos, cuando AutoLISP detecta una condición de error, cancela la función en proceso y llama a la función **error** del usuario con un mensaje que indica el tipo de error. Si no se define ninguna función **error** (o si **error** se limita a cero), la acción estándar es presentar el mensaje en la forma siguiente:

```
error: mensaje
```

El mensaje va seguido de un indicador de función. Si hay una función **error** definida por el usuario, el indicador no actúa, sino que se hace referencia a dicha función de usuario con *mensaje* como argumento único.

A continuación se muestra una lista completa de los mensajes de error que puede proporcionar AutoLISP en programas de usuario, además de una explicación de cada uno. La lista recoge los mensajes únicamente en castellano, excepto los que se encuentran en inglés en todas las plataformas idiomáticas.

Lista 1

Los argumentos de defun no pueden tener el mismo nombre

Una función definida con varios argumentos que tengan el mismo nombre fallará y generará este mensaje.

AutoCAD rechazó la función

Los argumentos suministrados a una función de **AutoCAD** no son válidos (como en SETVAR con una variable de sistema de sólo lectura o TBLNEXT con un nombre de tabla no válido) o la propia función no es válida en el contexto actual. Por ejemplo, no se puede utilizar una función GET... de introducción de datos del usuario dentro de la función COMMAND.

Desbordamiento de pila de AutoLISP

Se ha superado el espacio de almacenamiento de pila de AutoLISP. El motivo puede ser una repetición excesiva de funciones o listas de argumentos muy largas.

tipo de argumento erróneo

Se ha pasado un tipo de argumento incorrecto a una función. Por ejemplo, no se puede obtener la STRLEN de un número entero.

bad association list

La lista suministrada a la función ASSOC no está formada por listas de valor clave.

código de conversión erróneo

El identificador SPACE suministrado a la función TRANS no es válido.

bad ENTMOD list

El argumento suministrado a ENTMOD no es una lista de datos de entidades adecuada (tal y como lo devuelve ENTGET).

bad ENTMOD list value

Una de las sublistas de la lista de asociación suministrada a ENTMOD contiene un valor erróneo.

lista de argumento formal inadecuada

Al evaluar esta función, AutoLISP ha detectado una lista de argumento formal no válida. Quizás no se trate de una función, sino más bien de una lista de datos.

función incorrecta

El primer elemento de la lista no es un nombre de función válido; quizás se trate de un nombre de variable o de un número. Este mensaje también puede indicar que la función especificada no está bien definida (no olvidar la lista de argumentos formal obligatoria).

código de función erróneo

El identificador de función suministrado al comando TABLET no es correcto.

bad grvecs list value

El valor pasado en una lista GRVECS no es un punto 2D o un punto 3D.

bad grvecs matrix value

La matriz que se ha suministrado a GRVECS contiene errores de formación o de tipo de datos (por ejemplo: STR, SYM, etc.).

lista incorrecta

Se ha suministrado una lista con errores de forma a una función. Esto puede suceder si un número real empieza con un separador decimal. Hemos de incluir un cero inicial en este caso.

lista de puntos incorrecta

La solicitud F, CP o WP lleva adjunta una lista nula o con elementos que no son puntos. Se utilizan con SSGET y GRVECS.

bad node

La función TYPE ha encontrado un tipo de elemento no válido.

tipo de nodo erróneo en la lista

La función FOREACH ha encontrado un tipo de elemento no válido.

argumento de puntos incorrecto valor de punto incorrecto

Se ha pasado un punto mal definido (una lista de dos números reales) a una función que esperaba un punto. Un número real no puede empezar por un separador decimal; en dicho caso, hay que incluir el cero inicial.

detectado número real incorrecto

Se ha intentado transmitir un número real no válido de AutoLISP a **AutoCAD**.

bad ssget list

El argumento suministrado a SSGET "X" no es una lista de datos de entidad adecuada (tal y como devuelve ENTGET).

bad ssget list value

Una de las sublistas de la lista de asociaciones suministrada a SSGET "X" contiene un valor erróneo.

cadena modo ssget incorrecta

Este error se produce cuando SSGET recibe una cadena no válida en el argumento modo.

lista de xdata incorrecta

Este error se genera cuando XDSIZE, SSGET, ENTMOD, ENTMAKE o TEXTBOX recibe una lista de datos de entidad extendida con errores de forma.

se requiere punto de base

Se ha llamado a la función GETCORNER sin el argumento de punto base obligatorio.

Boole arg1 0 or 15

El primer argumento de la función booleana ha de ser un número entero entre 0 y 15.

imposible evaluar la expresión

Separador decimal mal colocado o alguna otra expresión incorrecta.

no es posible abrir (archivo) para entrada; fallo de LOAD

No se ha encontrado el archivo designado en la función LOAD, o el usuario no tiene permisos de lectura sobre ese archivo.

imposible volver a entrar en AutoLISP

Una función activa está utilizando el buffer de comunicación **AutoCAD**/AutoLISP; no se podrá llamar a otra nueva función hasta que la actual haya finalizado.

interrupción desde el teclado

El usuario ha tecleado CTRL+C durante el proceso de una función.

divide by zero

No se puede dividir por cero.

desbordamiento en división

La división por un número muy pequeño ha dado como resultado un cociente no válido.

exceeded maximum string length

Se ha pasado a una función una cadena con más de 132 caracteres.

extra right paren

Se ha encontrado uno o más paréntesis cerrados de los necesarios.

file not open

El descriptor de archivo para la operación de E/S no es el de un archivo abierto.

lectura de archivo, memoria de cadenas insuficiente

Memoria de cadenas agotada mientras AutoLISP leía un archivo.

file size limit exceeded

Un archivo ha superado el límite de tamaño permitido por el sistema operativo.

floating-point exception

Sólo sistemas UNIX. El sistema operativo ha detectado un error aritmético de coma flotante.

función cancelada

El usuario ha tecleado CTRL+C o ESC (cancelar) en respuesta a una solicitud de datos.

function undefined for argument

El argumento pasado a LOG o SQRT sobrepasa los límites permitidos.

function undefined for real

Se ha suministrado un número real como argumento de una función que exige un número entero, por ejemplo (LSH 2.2 1).

falta punto final grvecs

Falta un punto final en la lista de vectores que se ha suministrado a GRVECS.

illegal type in left

El archivo .LSP no es ASCII puro, sino que se ha guardado con un programa de tratamiento de textos e incluye códigos de formato.

improper argument

El argumento para GCD es negativo o cero.

`inappropriate object in function`

El paginador de funciones VMON ha detectado una función mal expresada.

`número incorrecto de argumentos`

La función QUOTE sólo espera un argumento concreto, pero se le han proporcionado otros.

`número incorrecto de argumentos para una función`

El número de argumentos para la función creada por el usuario no coincide con el número de argumentos formales especificado en DEFUN.

`solicitud inadecuada de datos sobre lista de comandos`

Se ha encontrado una función de comando pero no se puede ejecutar porque hay otra función activa o porque el intérprete de comandos no está completamente inicializado. Este error puede producirse desde una llamada a la función COMMAND en ACAD.LSP, ACADR14.LSP o en un archivo .MNL.

`entrada interrumpida`

Se ha detectado un error o condición de fin de archivo prematuro, lo que ha provocado la finalización de la introducción de datos en el archivo.

`insufficient node space`

No hay espacio suficiente en la pila de almacenamiento para la acción solicitada.

`insufficient string space`

No hay espacio en la pila de almacenamiento para la cadena de texto especificada.

`invalid argument`

Tipo de argumento erróneo o argumento sobrepasa los límites permitidos.

`lista de argumentos no válida`

Se ha pasado a una función una lista de argumentos que contiene errores.

`invalid character`

Una expresión contiene un carácter erróneo.

`par punteado no válido`

No se ha dejado el espacio pertinente antes y después del punto en la construcción manual del par punteado. Este mensaje de error también puede deberse a un número real que empieza por el separador decimal, en cuyo caso ha de incluir el cero inicial.

`valor de número entero no válido`

Se ha encontrado un número menor que el entero más pequeño o mayor que el número entero más grande.

`desbordamiento LISPSTACK`

Se ha superado el espacio de almacenamiento de pila de AutoLISP. El motivo puede ser una repetición excesiva de funciones o listas de argumentos muy largas.

`malformed list`

Se ha terminado prematuramente una lista que se estaba leyendo en un archivo. La causa más común es una incoherencia en el emparejamiento de las aperturas y cierres de paréntesis o comillas.

`malformed string`

Se ha terminado prematuramente una cadena que se estaba leyendo en un archivo.

`misplaced dot`

Un número real comienza con el separador decimal. Se ha de incluir un cero inicial en este caso.

`función nula`

Se ha intentado evaluar una función que tiene una definición vacía.

`quitar/salir abandonar`

Se ha llamado a la función `QUIT` o `EXIT`.

`cadena demasiado larga`

La cadena que se ha pasado a `SETVAR` es demasiado larga.

`too few arguments`

Se han pasado pocos argumentos a una función integrada.

`argumentos insuficientes para grvecs`

No se han pasado suficientes argumentos a `GRVECS`.

`demasiados argumentos`

Se han pasado demasiados argumentos a una función integrada.

La siguiente lista muestra los errores internos de AutoLISP que no tienen que ver con el programa de usuario. Estos errores, si se producen, hay que notificarlos a Autodesk de forma inmediata ya que se pueden deber a *bugs* o fallos en la programación de **AutoCAD** o en la implementación de AutoLISP en él. Para más información véase la ayuda del programa.

Lista 2

`bad argument to system call`

Sólo sistemas UNIX. El sistema operativo ha detectado una llamada errónea generada por AutoLISP.

`bus error`

Sólo sistemas UNIX. El sistema operativo ha detectado un error de bus.

`hangup`

Sólo sistemas UNIX. El sistema operativo ha detectado una señal *hangup* (colgado).

`illegal instruction`

Sólo sistemas UNIX. El sistema operativo ha detectado una instrucción de máquina no válida.

`segmentation violation`

Sólo sistemas UNIX. El sistema operativo ha detectado un intento de direccionamiento externo al área de memoria especificada para este proceso.

`unexpected signal nnn`

Sólo sistemas UNIX. El sistema operativo ha emitido una señal inesperada.

ONCE.FIN. EJERCICIOS PROPUESTOS

- I. Diseñar un programa en AutoLISP que permita dibujar un perfil doble T parametrizado. El programa solicitará en línea de comandos la altura total y la altura del alma, la anchura del ala y del alma y el punto de inserción. El perfil se dibujará con ángulo de inserción 0.

- II. Realizar un programa que extraiga todos los bloques existentes en el dibujo actual y los guarde en disco en la unidad y directorio que el usuario indique. El programa contendrá una rutina de control de errores y funcionará desde línea de comandos. El resto al gusto del programador.
- III. Realizar un programa que permita dibujar un agujero en alzado seccionado. Tras solicitar el punto de inserción se irán pidiendo los demás datos. El usuario dispondrá de la posibilidad de dibujar el agujero con o sin cajera y/o roscado o no. La cajera (si existiera) y el agujero se dibujarán en la capa actual; la rosca (si existiera) y el eje de simetría se dibujarán en sendas capas cuyos nombres serán solicitados al usuario. El programa funcionará bajo línea de comandos únicamente.
- IV. Crear un programa en AutoLISP que dibuje una curva de nivel cartográfica a partir de los puntos contenidos en un fichero topográfico. El programa funcionará mediante un cuadro diseñado en DCL que tendrá el siguiente aspecto:

The dialog box titled "Generar curva de nivel" contains the following elements:

- Fichero:**
 - Entrada: C:\carto.txt (with "Examinar..." button)
 - Salida: C:\salid.dat (with "Examinar..." button)
 - ☒ Ver fichero de salida al terminar
 - ☐ No generar fichero de salida
- Generar curva:**
 - ☐ Polilínea
 - ☒ Polilínea adaptada
 - ☐ Spline
 - ☐ Líneas
- Propiedades de la curva:**
 - Capa: 0 (dropdown)
 - Color: PorCapa (dropdown)
 - Tipo de línea: CONTINUOUS (dropdown)
- Generar marcas:**
 - Estilo: STANDARD
 - Altura: 10
 - Rotación: 45
 - Comenzar en: 1
 - ☐ No generar marcas
- Separador:**
 - ☒ Coma
 - ☐ Espacio blanco
 - ☐ Otro
 - Carácter: (empty text box)
- Buttons:** Aceptar, Cancelar

Como se puede apreciar en este letrero de diálogo, el usuario dispondrá de la posibilidad de escoger un fichero en entrada —donde están los datos— y un fichero de salida. En este último se guardarán las coordenadas de los puntos precedidas de un número de orden o marca que será el que se dibuje en pantalla al lado de cada punto. Además, también se podrá escoger la posibilidad de ver el fichero de salida al terminar de dibujar la curva, en cuyo caso se abrirá un editor ASCII (tipo Bloc de Notas) con el fichero en cuestión, y la posibilidad de no generar este fichero, en cuyo caso no se generará (al elegir esta opción habrán de desactivarse la innecesarias).

Como segundo paso se elegirá el tipo de curva con el que se dibujará la curva de nivel (sección *Generar curva*). Después elegiremos las propiedades de la curva: capa, color y tipo de línea. En cada lista desplegable se reflejarán los elementos actuales en el dibujo.

Como se ve en el cuadro, habremos de especificar también un estilo de texto, su rotación y su altura, así como dispondremos de la posibilidad de no escribir las marcas en pantalla. También se ofrece en esta sección del letrero una casilla de edición para indicar cuál será la primera de las marcas.

Por último escogeremos el separador de coordenadas en el archivo de entrada, es decir, cuál es el carácter que divide unas componentes de otras: una coma (formato CDF), un espacio blanco (formato SDF) o cualquier otro que, en su caso, se especificará.

El programa dispondrá de una rutina de control de errores, tanto del cuadro de diálogo (para no introducir valores incorrectos) como de errores AutoLISP.

- V. Crear un programa mediante AutoLISP y DCL que dibuje escaleras de caracol en 3D. El cuadro de diálogo puede ser el que sigue:

Escalera de caracol

Escalera

Altura total: 500

Diámetro Exterior: 250

Diámetro Interior: 18

Número de vueltas: 2

☐ Hueco interior

Generación: Horaria

Punto de inserción

X: 325.1857

Y: 173.1912

Z: 0

Designar <

Peldaños

Hueija máxima peldaño: 50

Altura peldaño: 4

Distancia entre peldaños: 16

Aceptar Cancelar

Como observamos habremos de introducir todos los datos necesarios para el dibujo de la escalera. La opción Hueco interior determina si por dentro existirá hueco o una columna cilíndrica. Dispondremos de un botón para designar un punto de inserción en pantalla.

- Créense rutinas de control de errores y demás.
- VI. Diseñar un programa AutoLISP/DCL cuyo cometido será sumar o acumular todas distancias de tramos rectos y curvos de las líneas y/o polilíneas y/o arcos existentes en una capa del dibujo actual. Al final, el programa escribirá un texto personalizable con las cantidades acumuladas. Este programa puede serle muy útil a todo aquel que necesite saber en un momento el total de tramos trazados en un dibujo (en una capa), sean tuberías, circuitos eléctricos, carreteras, etcétera.

El cuadro de diálogo principal del programa será el que se puede observar más abajo.

EL área *Actuar en capa* define la capa en la cual se realizarán las mediciones. Como vemos puede ser una elegida de la lista desplegable —que contendrá todas las capas del dibujo—, la capa actual o la correspondiente al objeto que se designará (si se elige la última opción). En estos dos últimos casos la primera opción quedará inhabilitada, evidentemente; en el último caso, se pedirá designar objeto al pulsar el botón *Aceptar*, antes de proceder a la medición.

Acumular distancias

Actuar en capa

Capa: TUBERIAS

☒ Capa de la lista

☐ Capa actual

☐ Designar por objeto

Acumular

☒ Líneas

☐ Polilíneas

☒ Arcos

☐ Todo

Mostrar texto

Estilo: TITEST

Altura: 15

Rotación: 0

Texto inicial: Distancia total:

Texto final: milímetros.

Preferencias...

Inserción del texto

Capa: Actuación

X: 269.3993

Y: 124.5143

Z: 0

Designar <

Aceptar Cancelar

En el área *Acumular* se especifica el tipo de objeto sobre el que actuará: línea, polilínea, arcos o todos (con las correspondientes combinaciones).

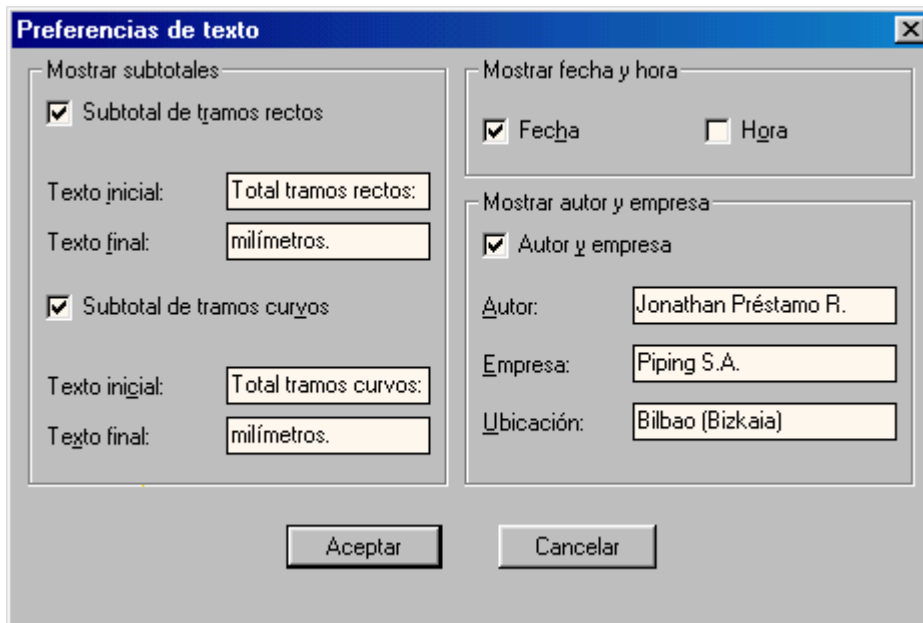
En el área *Inserción del texto* habremos de pulsar el botón designar para escoger un punto de inserción en pantalla. Al volver al cuadro, se reflejarán en las casillas correspondientes las coordenadas X, Y y Z de dicho punto. Además, escogeremos aquí la capa donde se insertará el texto (*Actuación* dice relación a la capa elegida en el área primera).

Por último, en el área *Mostrar texto* describiremos las propiedades del texto (estilo, altura y rotación), así como el texto que aparecerá antes del total acumulado y después del mismo.

El botón *Preferencias...* nos lleva a otro cuadro de diálogo, cuyo diseño lo vemos en la página siguiente. En este nuevo cuadro elegiremos diversas preferencias (si así lo deseamos) para el texto final escrito en pantalla. Así pues, dispondremos de la posibilidad de mostrar el subtotal de tramos rectos y de

tramos curvos —con sus correspondientes textos inicial y final—, la fecha y la hora de la medición y el autor de la misma, así como su empresa y la ubicación de ésta.

Este programa representa ya un alto nivel en la programación en AutoLISP/DCL. Se puede hacer de variadas formas, sobre todo el cuerpo principal donde se realiza la medición. Intentemos escoger la que nos resulte más sencilla y versátil a la hora de tener en cuenta otros aspectos del programa.



EJERCICIOS RESUELTOS DEL MÓDULO DIEZ

EJERCICIO I

Letrero a)

```
agujeros:dialog {label="Inserción de agujeros";
:row {
: image {width=15;aspect_ratio=0.8;color=0;key="imagen";}
:boxed_radio_column {label="Tipo de agujero";key="tipo";
:radio_button {label("&Sin cajera";key="sin";}
:radio_button {label("&Con cajera recta";key="recta";}
:radio_button {label("Con cajera a&vellanada";key="avella";}
}
}
:row {
:boxed_column {label="Agujero";
:edit_box {label("&Diámetro: ";edit_width=6;edit_limit=16;
key="diamagu";}
:edit_box {label("&Profundidad: ";edit_width=6;edit_limit=16;
key="profagu";}
spacer_1;
}
:boxed_column {label="Cajera";key="cajera";
```



```

:edit_box {label="Diámetro: ";edit_width=6;edit_limit=16;
           key="diamcaj";}
:edit_box {label="Profundidad: ";edit_width=6;edit_limit=16;
           key="profcaj";}
spacer_1;
}
}
spacer_1;
ok_cancel;
errtile;
}

```

Letreros b-1 y b-2)

```

insertdir :dialog {label="Inserción de bloques en un directorio";
:row {
:column {
spacer_0;
:list_box {label="Nombre de bloque:";key="lista";
           fixed_height=true;height=16;}
spacer_0;
:button {label="Mostrar...";key="mostrar";
         fixed_width=true;alignment=centered;}
}
:column {
:image_button {key="img0";width=11;aspect_ratio=0.8;color=0;}
:image_button {key="img4";width=11;aspect_ratio=0.8;color=0;}
:image_button {key="img8";width=11;aspect_ratio=0.8;color=0;}
:image_button {key="img12";width=11;aspect_ratio=0.8;color=0;}
}
:column {
:image_button {key="img1";width=11;aspect_ratio=0.8;color=0;}
:image_button {key="img5";width=11;aspect_ratio=0.8;color=0;}
:image_button {key="img9";width=11;aspect_ratio=0.8;color=0;}
:image_button {key="img13";width=11;aspect_ratio=0.8;color=0;}
}
:column {
:image_button {key="img2";width=11;aspect_ratio=0.8;color=0;}
:image_button {key="img6";width=11;aspect_ratio=0.8;color=0;}
:image_button {key="img10";width=11;aspect_ratio=0.8;color=0;}
:image_button {key="img14";width=11;aspect_ratio=0.8;color=0;}
}
:column {
:image_button {key="img3";width=11;aspect_ratio=0.8;color=0;}
:image_button {key="img7";width=11;aspect_ratio=0.8;color=0;}
:image_button {key="img11";width=11;aspect_ratio=0.8;color=0;}
:image_button {key="img15";width=11;aspect_ratio=0.8;color=0;}
}
}
spacer_1;
:row {children_fixed_width=true;
:button {label="Precedente";key="pre";}
:button {label="Siguiente";key="sig";}
:spacer {width=9;}
ok_cancel;
}
errtile;
}

```

```

muestra:dialog {label="Muestra ampliada del bloque";

```

```
:column {
  :image {width=48;aspect_ratio=0.8;key="muestra";color=0;}
  ok_only;
}
}
```

Letrero c)

```
modelar:dialog {label="Modelización de imágenes";
:row {
:column {
:boxed_radio_column {label="Modalidad de sombra";
  key="radio_edge";
:radio_button {label="256 colores &sin aristas";key="edge0";}
:radio_button {label="256 colores &con aristas";key="edge1";
  value=1;}
:radio_button {label="Ocultar líneas";key="edge2";}
:radio_button {label="Colores &planos con aristas";
  key="edge3";}
}
:spacer {}
:button {label="&Valores por defecto";key="defecto";
  fixed_width=true;alignment=centered;}
:spacer {}
:boxed_column {label="Luz difusa";
:boxed_radio_row {label="Valores predeterminados";
  fixed_width=true;alignment=centered;
:radio_button {label="&30";key="dif30";}
:radio_button {label="&50";key="dif50";}
:radio_button {label="&70";key="dif70";value=1;}
:radio_button {label="&90";key="dif90";}
}
:boxed_row {label="Valor de usuario";
:column {
:spacer {}
:text {label="Mín          Máx";alignment=centered;}
:spacer {}
:slider {min_value=0;max_value=100;small_increment=1;
  key="desliza";value=70;}
}
:column {
  spacer_1;
:edit_box {label="  ";alignment=centered;edit_width=3;
  edit_limit=3;key="valor";value=70;}
}
}
  spacer_1;
}
}
:boxed_column {label="Ventana de muestra";
:radio_row {key="muestra";
:radio_button {label="&Esfera";key="esfera";value=1;}
:radio_button {label="Cu&bo";key="cubo";}
}
:image {key="imagen";width=18;fixed_width=true;
  aspect_ratio=0.85;color=graphics_background;}
:text {label="Luz difusa = 70";}
:button {label="&Mostrar";key="mostrar";fixed_wdth=true;
  alignment=centered;}
spacer_0;
```

```
    }  
  }  
  spacer_1;  
  ok_cancel;  
  errtile;  
}
```

MÓDULO DOCE

Programación en Visual Basic orientada a AutoCAD (VBA)

DOCE.1. INTRODUCCIÓN

En este **MÓDULO** vamos a estudiar una nueva característica muy importante a la hora de realizar programas para **AutoCAD**, la cual es la programación en Visual Basic. **AutoCAD** proporciona una nueva interfaz de desarrollo en Visual Basic, uno de los lenguajes de programación más potentes que existen y que se está convirtiendo en un estándar a la hora de realizar programas en entorno Windows.

Si el desarrollador dispone de una versión de Visual Basic instalada en su ordenador podrá crear programas en este lenguaje que, tras compilarlos, podrán correr bajo **AutoCAD**, siempre que sigan una normas básicas para manejar los objetos del programa. Sin embargo esto no es imprescindible, ya que el propio **AutoCAD** implementa un módulo VBA (*Visual Basic for Applications*), el cual dispone de la sintaxis completa del lenguaje (en su versión 5.0), un depurador y un entorno de desarrollo integrado. De esta forma es totalmente factible crear un programa e irlo probando mientras se va depurando en una sesión de **AutoCAD**.

El objetivo de este **MÓDULO** no es explicar a fondo el lenguaje de programación Visual Basic, ya que para ello se necesitaría como mínimo un curso completo, y existen muchos y muy buenos; por ello, eso escapa a las pretensiones de este curso. Al lector se le suponen ya unos conocimientos medios o avanzados de programación en dicho lenguaje y simplemente habrá de actualizarse al manejo de los objetos incluidos en **AutoCAD** para Visual Basic.

DOCE.2. Visual Basic Y ActiveX Automation

Visual Basic es la desembocadura de lo que en un principio se llamó lenguaje BASIC (*Beginner's All-purpose Symbolic Instruction Code*, algo así como código de instrucción simbólica para todo propósito del principiante), que fue diseñado para iniciados en EE.UU. en 1964. Fue una derivación del lenguaje FORTRAN que se caracterizaba por su sencillez de manejo y curva de aprendizaje.

Microsoft a lo largo de su historia ha venido proporcionando una serie de editores BASIC cuyos más altos exponentes sin duda fueron GWBASIC y, sobre todo, QuickBASIC. Cuando apareció Windows como entorno las cosas comenzaron a cambiar: aparecen los lenguajes visuales. La programación visual, conocida como orientada a objetos y conducida por eventos, permite desarrollar aplicaciones para Windows de una manera sencilla e intuitiva. Visual Basic, a lo largo de su historia, ha ido creciendo en importancia y potencia, llegando en muchos momentos a no tener nada que envidiar a su más temible competidor, probablemente el lenguaje visual más potente que existe: Visual C++.

Hoy en día, esta programación orientada a objetos se sirve de una de las últimas tecnologías de programación, esto es *ActiveX Automation*. La tecnología ActiveX, presente en las últimas versiones de Windows, supera con creces al sistema anterior OLE, proporcionando una interfaz de programación que permite manejar los objetos de una aplicación desde fuera de la misma. La intercomunicación ahora entre aplicaciones que manejen objetos ActiveX es total: nosotros seremos capaces desde un programa en Visual Basic de abrir una sesión de

AutoCAD, añadir una serie de líneas a modo de tabla y, a continuación abrir una sesión en Excel, por ejemplo, para leer determinados datos, operar con ellos y volver a **AutoCAD** para añadir los datos a la tabla. Y todo esto mediante letreros de diálogo, botones y demás componentes típicos y sencillos de manejar de Microsoft Windows.

Los objetos propios de una aplicación son así expuestos a las demás aplicaciones como objetos Automation. En **AutoCAD** son objetos Automation todos los objetos de dibujo, las denominadas tablas de símbolos como bloques, capas, estilos de cota, etc. y también las *Preferencias*. Las aplicaciones que proporcionan un entorno de programación en el que los desarrolladores o usuarios pueden escribir macros y rutinas para acceder y controlar objetos ActiveX se denominan *controladores de Automation*. Estos controladores pueden ser aplicaciones de Windows como Word y Excel, o entornos de programación como Visual Basic o Visual C++.

Los objetos ActiveX exponen para su control dos conceptos específicos: métodos y propiedades. Los métodos son funciones que ejercen una acción sobre los objetos. Las propiedades son funciones que definen o devuelven información sobre el estado de un objeto. Las propiedades y métodos dependen de cada tipo de objeto y se describen a través de una biblioteca de tipos. Los desarrolladores o usuarios tienen a su disposición un examinador de biblioteca de tipos, para saber en todo momento los métodos y propiedades existentes. **AutoCAD** dispone de sus propios objetos ActiveX para ser manejados desde un programa.

La biblioteca de tipos donde se encuentran definidos los métodos y propiedades a través de los cuales expone **AutoCAD** sus objetos *Automation* se encuentra en el archivo ACAD.TLB. Las referencias se establecen desde el editor de VBA, desde el menú *Herramientas>Referencias...*, activando la casilla *AutoCAD Object Library*. Los objetos de una aplicación pueden usarse sin hacer referencia a la biblioteca de objetos, pero es preferible añadir dicha referencia por motivos de facilidad, fiabilidad y comprobación.

Es posible acceder a objetos Automation desde muchas aplicaciones de Windows. Estas aplicaciones pueden ser ejecutables independientes, archivos de bibliotecas de enlace dinámico .DLL y macros dentro de aplicaciones como Word y Excel. Mediante Automation resulta muy sencillo desarrollar y mantener una interfaz gráfica de usuario. La interfaz puede ir desde un solo cuadro de diálogo o una barra de herramientas que mejore una aplicación existente, hasta una aplicación propia completa.

DOCE.2.1. La línea de productos de Visual Basic

El lenguaje de programación Visual Basic como tal está dividido en diferentes productos que el usuario adquiere según sus propias necesidades. En líneas generales podemos dividir estos productos en tres grandes grupos:

- Visual Basic. Es la herramienta de más alto nivel dentro de la gama. Posee todas las características del lenguaje y está dividida en cuatro ediciones: Creación de Controles (*Control Creation*), que permite únicamente —que no es poco— el desarrollo de controles ActiveX, por lo que no permite la creación de aplicaciones autónomas; Estándar (*Learning*), con acceso de alta velocidad a bases de datos y que ya sirve para el desarrollo de aplicaciones; Profesional (*Professional*), con herramientas avanzadas para el diseño de aplicaciones profesionales; y Empresarial (*Enterprise*), con todas las herramientas disponibles para desarrolladores e investigadores.
- VBA (*Visual Basic for Applications*). Es una herramienta menos avanzada que la anterior y diseñada para ser implementada en las distintas aplicaciones Windows y poder desarrollar programas para ellas. Incluye importantes implementaciones para macros y guiones, la casi completa sintaxis de programación Visual Basic, un depurador y un entorno de desarrollo integrado. Es la incluida en **AutoCAD**.

- VBScript (*Visual Basic Scripting Edition*). Es la herramienta de inferior nivel. Contiene un subconjunto del lenguaje Visual Basic, pero sin acceso a sistema operativo ni a operaciones de entrada y salida de archivos. Se ha concebido fundamentalmente como herramienta de diseño *script* para páginas Web de Internet en HTML (al estilo de JavaScript).

DOCE.3. EL MÓDULO VBA DE AutoCAD

Como ya se ha comentado, **AutoCAD** proporciona un módulo VBA que es una versión *preview*, sin embargo podríamos decir que funciona completamente bien (en la versión 14.01 de **AutoCAD** ya se da una versión final del editor VBA). Este módulo no se instala en la instalación general del programa, sino que hay que hacerlo después de ella. Para este menester deberemos acudir al directorio \VBAINST\ del CD-ROM de instalación de **AutoCAD**, y ejecutar el archivo SETUP.EXE, que instalará el módulo VBA.

Una vez instalado, y tras quizá reiniciar el equipo mejor, aparecerá un nuevo menú desplegable en **AutoCAD** denominado **VBA** (entre **Herr.** y **Dibujo**). Este menú dispone de las siguientes opciones:

- **Run Macro** (ejecutar macro) ejecuta una macro o rutina. Debe haberse cargado previamente algún proyecto con la opción siguiente. El comando en línea de comandos es VBARUN.
- **Load Project** (cargar proyecto) permite cargar un nuevo proyecto (.DVB). Al cargar un proyecto existente, **AutoCAD** muestra el editor VBA con el contenido del proyecto (formularios, módulos de código...). El comando en línea de comandos es VBALOAD.
- **Unload Project** (descargar proyecto) descarga un proyecto previamente cargado. El comando en línea de comandos es VBAUNLOAD.
- **Show VBA IDE** (mostrar editor VBA) permite mostrar el editor de VBA. El comando en línea de comandos es VBAIDE.

NOTA: En la versión 14.01 de **AutoCAD** se añade un menú de persiana al menú **Herr.** llamado **Macro**.

El módulo VBA de **AutoCAD** es ligeramente distinto al entorno Visual Basic habitual, si es que estamos familiarizados con él. En principio, por defecto el *Explorador de proyectos* y la *Ventana de Propiedades* aparecen anclados a la izquierda, y no a la derecha. El *Cuadro de herramientas* no aparece por ningún lado, ya que sólo se pone de manifiesto cuando estamos trabajando con un formulario.

En general la estructura de menús y barra de herramientas es muy parecida, si bien existen ausencias —sobre todo— en VBA con respecto a Visual Basic, como por ejemplo la capacidad de incluir formularios MDI, de compilar proyectos o de abrir proyectos, entre otras. Esto último se realiza desde el menú desplegable de **AutoCAD**, y sólo desde ahí. Decir que el módulo VBA únicamente puede ser abierto desde una sesión de **AutoCAD**, y no independientemente.

Todas las nuevas características se irán aprendiendo a lo largo de este **MÓDULO**, por lo que no debemos preocuparnos si dominamos Visual Basic y creemos no sentirnos a gusto en VBA.

Dado que, como hemos dicho, se suponen conocimientos previos por parte del lector, nos meteremos de lleno ya mismo con la programación exclusivamente dirigida a **AutoCAD**.

DOCE.4. COMENZANDO CON VBA

Lo primero que quizá nos haya llamado la atención es la entrada por defecto que aparece en el Explorador de proyectos: `ThisDrawing`, dentro de una carpeta llamada **AutoCAD** Objetos. El objeto `ThisDrawing` se refiere al documento actual activo de **AutoCAD**, y no puede ser eliminado. A él nos referiremos a la hora de gestionar objetos de **AutoCAD**, ya que siempre que hagamos algo será evidentemente referenciado y ejecutado en el documento actual activo.

Antes de comenzar a programar tenemos que asegurarnos —para evitarnos problemas posteriores— que la referencia a la librería de objetos de **AutoCAD** (`ACAD.TLB`) está activada. Para ello acudiremos al menú Herramientas>Referencias....

La forma de programar en Visual Basic es muy personal, ya que hay gente que utiliza bastante los módulos de código y otra gente que incluye todo el código en el propio formulario. En **AutoCAD** normalmente trabajaremos con un único formulario (cuadro de diálogo) al estilo DCL + AutoLISP que realizará determinada función. Es por ello que se utilizarán muy poco los módulos de código (excepto para macros), y muchísimo menos los módulos de clase. A los programadores avanzados les puede ser muy interesante el disponer de estas características en VBA.

DOCE.4.1. La plantilla de objetos

Los objetos ActiveX que proporciona **AutoCAD** para su manejo desde programas VBA están divididos según una jerarquía que deberemos seguir a la hora de llamarlos o referirnos a ellos. La plantilla que se muestra a continuación nos será muy útil a la hora de programar, ya que establece dicha jerarquía.

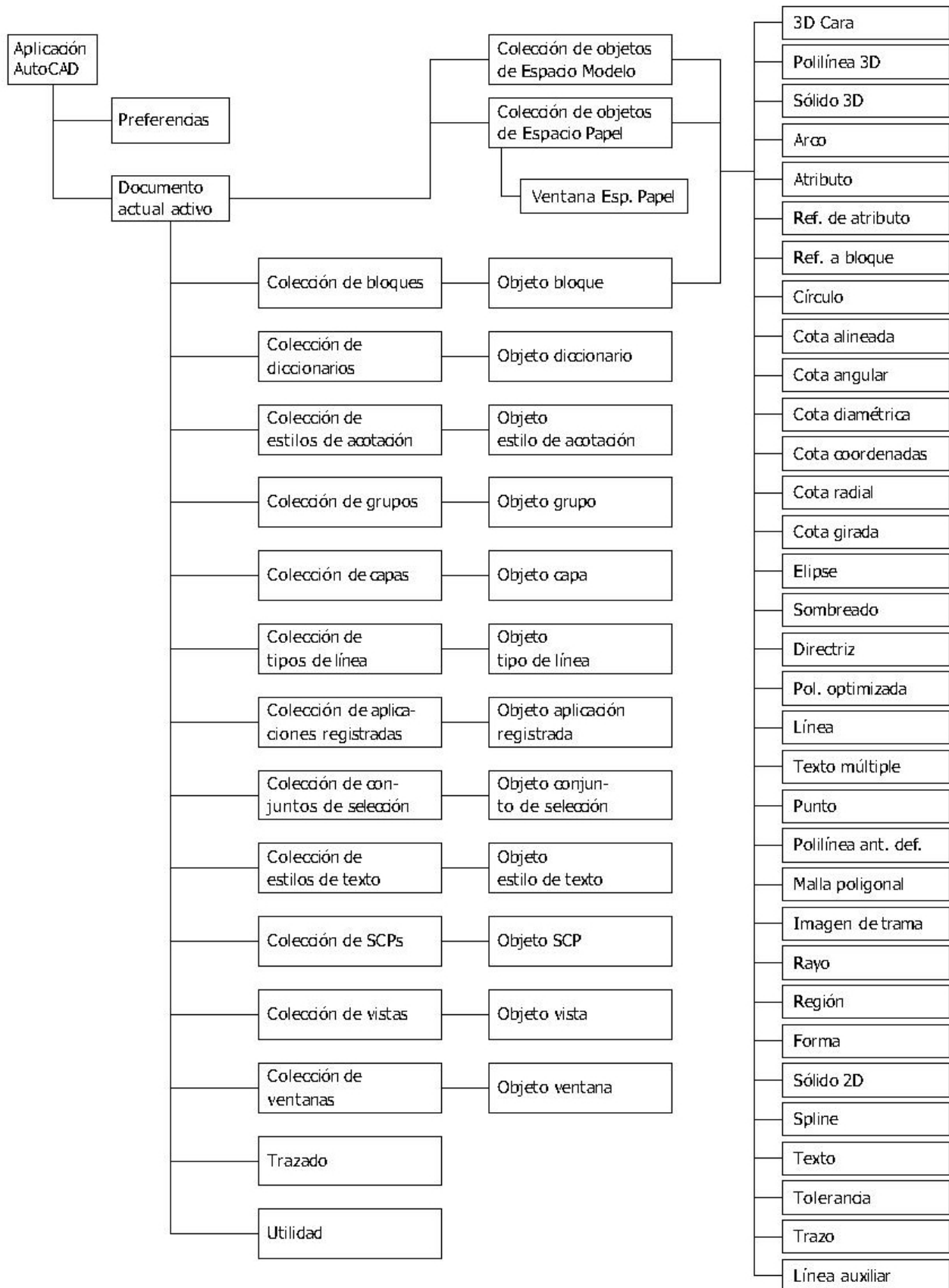
En Visual Basic es factible añadir al entorno nuevos objetos creados por nosotros para luego ser utilizados. Lo que se ha hecho en VBA es precisamente eso. Estos objetos tienen sus propiedades y métodos, al igual que los demás. Existen objetos de entidades individuales de dibujo (líneas, círculos, arcos...) con sus propiedades (color, capa, tipo de línea...) y métodos (copiar, mover, escalar).

También se han definido otros objetos no gráficos como son el Espacio Modelo, el Espacio Papel y los bloques. Estos se consideran una colección de objetos de entidades individuales de dibujo y tienen también sus propiedades para, por ejemplo, saber cuántas entidades simples contienen, y sus métodos para, por ejemplo, añadir nuevas entidades a la colección.

El propio documento actual de **AutoCAD** está definido como un objeto y tiene sus propiedades (camino de acceso, límites...) y métodos (guardar, regenerar...). Dentro de él se encuentran los mencionados anteriormente, además de otras colecciones como el conjunto de capas, de estilos de texto, etcétera, cada una con propiedades y métodos.

Y todo ello está incluido en el objeto más exterior, que es la aplicación de **AutoCAD**.

La plantilla que muestra toda la jerarquía se proporciona en la página siguiente.



DOCE.4.2. Comenzar un programa

El primer paso que vamos a dar al realizar un programa en VBA será casi siempre el mismo, y se refiere a la declaración de las variables de objeto que serán necesarias para acceder a los distintos aspectos de **AutoCAD**.

Según la plantilla de la página anterior, la propia aplicación **AutoCAD**, el documento activo, el Espacio Modelo o el Espacio Papel, entre otros, son objetos que hemos de definir en un principio para luego poder referirnos a ellos a lo largo del programa fácilmente. Como objetos que son los declararemos como tales, así por ejemplo, la aplicación en sí la podríamos declarar así (en `General_Declaraciones`):

```
AcadApp as Object
```

el documento activo:

```
AcadDoc as Object
```

y el Espacio Modelo y Papel:

```
AcadModel as Object
```

y

```
AcadPapel as Object
```

Esto en sí no tiene mucho sentido hasta que no le demos unos valores coherentes. Para ello utilizaremos el procedimiento `UserForm_Initialize` (si nuestro programa dispone de formulario; no es una macro), ya que habrán de tomar valores al iniciar el programa, así:

```
Set AcadApp = GetObject(, "AutoCAD.Application")
Set AcadDoc = AcadApp.ActiveDocument
Set AcadModel = AcadDoc.ModelSpace
Set AcadPapel = AcadDoc.PaperSpace
```

NOTA: Como sabemos, para añadir valores a variables del tipo `Object`, es necesario utilizar la instrucción `Set`.

Como vemos, la primera línea es la que realmente hace referencia a la librería o biblioteca de objetos de **AutoCAD**. No se incluye el camino por encontrarse el archivo `ACAD.TLB` en el directorio principal de instalación de **AutoCAD**, esto es, por estar en directorio de archivos de soporte. Sin embargo, la coma necesaria de la función `GetObject` que separa los dos parámetros es imprescindible incluirla.

Las líneas restantes hacen referencia a los demás objetos que, como se ve, cuelgan todos del primero. El documento activo (`ActiveDocument`) cuelga directamente de la aplicación **AutoCAD**, y tanto el Espacio Modelo (`ModelSpace`) como el Espacio Papel (`PaperSpace`) del documento actual. Esta jerarquía la podemos apreciar perfectamente en la plantilla anteriormente expuesta.

Esta asignación de objetos no es necesaria realizarla, ya que podemos referirnos a ellos con toda la secuencia de nombres. Pero parece lógico utilizar, por ejemplo, un nombre de variable como `AcadModel` cada vez que nos tengamos que referirnos al Espacio Papel (que serán muchas veces), que utilizar la secuencia entera hasta `ModelSpace`.

Además si trabajamos directamente en VBA podemos sustituir las llamadas al documento actual por `ThisDrawing` así:

```
Set AcadDoc = ThisDrawing
```

o incluso utilizar `ThisDrawing` posteriormente en las llamadas durante el programa. El problema de esto reside en que sólo el VBA de **AutoCAD** sabe lo que es `ThisDrawing`. En el momento en que queramos abrir nuestro programa en un entorno Visual Basic externo para, por ejemplo compilarlo (que ya lo veremos), necesitaremos realizar las llamadas pertinentes a la librería de objetos, si no nunca funcionará. Es por ello que es mejor acostumbrarse a esta técnica mucho más versátil y elegante.

Todo esto es necesario porque debemos decirle a VBA que vamos a trabajar con una aplicación denominada **AutoCAD**, que utilizaremos el documento actual activo y, por ejemplo, su Espacio Modelo. A la hora de añadir un círculo, por ejemplo, deberemos indicar que ha de ser al Espacio Modelo del documento actual de **AutoCAD**, sobre todo si trabajamos con un Visual Basic externo o el archivo está ya compilado.

Se emplea en estas declaraciones la función `GetObject`, cuya finalidad consiste en devolver una referencia al objeto que se le solicita, en nuestro caso a `AutoCAD.Application`, que engloba a todo el modelo de objetos. Para que esta función responda adecuadamente es necesario que **AutoCAD** esté cargado y con un dibujo abierto (en la versión 14 no es posible cerrar un dibujo), que será referenciado como `ActiveDocument`.

DOCE.5. DIBUJO Y REPRESENTACIÓN DE ENTIDADES

A continuación, bajo esta sección, estudiaremos los diferentes métodos que tenemos de añadir entidades individuales de dibujo mediante programas en VBA.

La manera de dibujar entidades dice relación a métodos que pertenecen a las colecciones de Espacio Modelo, Espacio Papel y bloques (como se ve en la plantilla de objetos). Estas colecciones también tienen propiedades, pero eso lo veremos más adelante.

DOCE.5.1. Líneas

NOTA IMPORTANTE DE SINTAXIS: La sintaxis que se utiliza en este **MÓDULO** es similar a la de **MÓDULOS** anteriores: instrucciones, funciones, métodos, propiedades y demás términos reservados como aparecen en el editor VBA una vez aceptados; los textos en cursiva son mnemotécnicos que han de ser sustituidos por su valor; los textos no en cursiva deben escribirse coma tales; las sintaxis recuadradas: métodos en blanco y propiedades con sombra; en las listas las propiedades y métodos de objetos nuevos (sin explicar) en negrita, los ya explicados no; los listados de programas se muestran como aparecen en el editor; una barra vertical indica una dualidad de valores.

La sintaxis del método `AddLine` para dibujar líneas es la que sigue:

<pre>Set ObjLínea = ObjColección.AddLine(DblPtoInicial, DblPtoFinal)</pre>

Propiedades de los objetos de línea:

Application	Color	EndPoint	EntityName	EntityType
Handle	Layer	Linetype	LinetypeScale	Normal
ObjectID	StartPoint	Thickness	Visible	

Métodos de los objetos de línea:

ArrayPolar	ArrayRectangular	Copy	Erase
GetBoundingBox	GetXData	Highlight	IntersectWith
Mirror	Mirror3D	Move	Offset
Rotate	Rotate3D	ScaleEntity	SetXData
TransformBy	Update		

Los objetos gráficos de dibujo hemos de declararlos previamente como tales. Para ello deberemos definir un nombre de variable que almacenará el objeto; es a lo que se refiere *ObjLínea*. Esta variable puede ser declarada como `Object` simplemente o como un objeto especial de VBA para **AutoCAD** que representa el tipo de objeto que almacenará. Este objeto especial tiene diferentes nombres según el objeto que almacene; su sintaxis podría definirse así: *IAcadObjeto*, es decir, primero la cadena fija *IAcad* y luego el nombre del objeto.

De esta manera, una variable que tuviera que guardar una línea podría declararse como `Object` o como *IAcadLine*. La diferencia es que si la declaramos como `Object` podrá almacenar cualquier objeto a lo largo del programa: una línea, un círculo, un rayo, un texto... Si la declaramos como *IAcadLine* esa variable única y exclusivamente podrá almacenar entidades de línea de **AutoCAD**, una o varias a lo largo de la ejecución, pero sólo líneas. Es recomendable hacerlo con la primera opción (`Object`), sino a veces hay problemas.

Las entidades de **AutoCAD**, para ser dibujadas desde VBA, han de ser guardadas en una variable de objeto, por lo tanto con `Set`. Veremos casi al final de este **MÓDULO** que esto no es rigurosamente cierto. Por ahora lo haremos así.

ObjColección se refiere a la colección donde se almacenará la línea, es decir, si se dibujará en Espacio Modelo o Espacio Papel, o si formará parte de un bloque. Los puntos inicial y final de la línea (*DblPtoInicial* y *DblPtoFinal*) deben ser agrupaciones de tres coordenadas (X, Y, y Z), por lo tanto han de definirse como matrices (*arrays* o tablas) de tres elementos, cada uno de ellos de tipo `Double`, generalmente.

Para realizar nuestra primera prueba de dibujo de una línea crearemos una macro que así lo haga. Una macro en VBA se crea y se añade automáticamente al módulo estándar de código existente si existe, si no existe se crea una nuevo. La macro se ejecutará sin necesidad de letrero de diálogo.

Para crear un macro VBA lo haremos desde el menú *Herramientas>Macros....* Esto abre un cuadro que permite introducir un nombre para la nueva macro (también se puede eliminar, editar o ejecutar una existente). Tras pulsar el botón *Crear* se añadirá un nuevo procedimiento sub al módulo de código existente (o se creará uno). Dicho procedimiento tendrá el nombre de la macro. Para ejecutar una macro podemos hacerlo desde el mismo cuadro de creación (eligiendo una) o desde el menú de **AutoCAD VBA>Run Macro...**, esto último arranca un cuadro de diálogo que permite elegir el módulo estándar en cuestión donde se encuentra la macro y la macro por su nombre (en el caso de que hubiera varias dentro de un mismo módulo).

Creemos pues una macro llamada *DibujoLínea*. A continuación añadiremos las siguientes líneas de código:

NOTA: Cuidado con lo que va tras *General_Declaraciones* (siempre hasta la primera línea de división) y el resto (en estos casos se muestra el nombre de procedimiento sub en el encabezado).

```
Option Explicit
Dim AcadDoc As Object
Dim AcadModel As Object
```

Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

```
Dim ObjLínea As IAcadLine
Dim PuntoInicial (1 To 3) As Double
Dim PuntoFinal (1 To 3) As Double

Sub DibujoLínea()
    Set AcadDoc = GetObject( , "Autocad.Application").ActiveDocument
    Set AcadModel = AcadDoc.ModelSpace
    PuntoInicial(1) = 100: PuntoInicial(2) = 100: PuntoInicial(3) = 0
    PuntoFinal(1) = 200: PuntoFinal(2) = 200: PuntoFinal(3) = 0
    Set ObjLínea = AcadModel.AddLine(PuntoInicial, PuntoFinal)
End Sub
```

NOTA: Como en la mayoría de los programas en Visual Basic se recomienda el uso de `Option Explicit` para depurar errores en tiempo de corrida.

Lo primero que hacemos en `General_Declaraciones`, tras `Option Explicit` que nos obliga a declarar todas las variables utilizadas, es definir o declarar las variables que vamos a usar. La que representa al documento actual y la del Espacio Modelo, como ya se había explicado; la que contendrá la línea, como un objeto de línea de **AutoCAD**; y las del punto inicial y final de la línea, como matrices del tipo `Double`.

En el cuerpo de la macro propiamente dicha se asigna su valor a cada variable, tanto a las de la aplicación y el dibujo actual, como a los puntos inicial y final. Por fin se utiliza el método explicado para dibujar la línea en el Espacio Modelo. Solo habremos de correr la macro por alguno de los métodos explicados para comprobar el resultado.

NOTA: Existen los correspondientes objetos `IAcadModelSpace` e `IAcadDocument`, pero se recomienda la sintaxis utilizada en el ejemplo para estos objetos.

La ventaja que lleva implícita el almacenamiento en una variable de objeto de la entidad dibujada, dice relación a su posterior utilización para la aplicación de propiedades. Ahora no es necesario (como hacíamos en AutoLISP) acceder a la Base de Datos interna de **AutoCAD** para modificar las propiedades de un objeto, ya que dichas propiedades se relacionan directamente con sus objetos. Así por ejemplo, si tras trazar la línea del ejemplo anterior quisiéramos cambiarla al color rojo, únicamente deberíamos añadir la siguiente línea al programa (tras `Set ObjLínea...`):

```
ObjLínea.Color = 1
```

`Color` es una propiedad de la línea (véase en la lista tras la sintaxis), por lo que lo único que hacemos es cambiarle el valor como a cualquier otra propiedad en Visual Basic: indicando el objeto, seguido de un punto de separación, la propiedad, un signo de igual y el nuevo valor. Así de simple. Algunas propiedades se tratan de otra forma.

A continuación se explican a fondo cada una de las propiedades de los objetos de línea. Se explican de manera ambigua, es decir, sin referirse a las líneas en sí, ya que estas propiedades son comunes a muchos otros objetos VBA.

- **Application.** Obtiene el objeto `Application` de la entidad. Este objeto representa la aplicación de **AutoCAD**. La sintaxis es:

```
Set ObjAplicación = ObjGráfico.Application
```

siendo `ObjAplicación` una variable definida como `Object` y `ObjGráfico` un objeto gráfico. En el caso de nuestro ejemplo anterior de dibujo de una línea, `ObjGráfico` sería la variable `ObjLínea`.

Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

- **Color.** Obtiene y/o asigna el color de/a una entidad. El color se representa como un número entero de 0 a 256 (colores de **AutoCAD**). También se pueden emplear algunas constantes predefinidas como:

AcByBlock	AcByLayer	AcCyan	AcRed	AcBlue	AcYellow
AcMagenta	AcGreen	AcWhite			

La sintaxis para esta propiedad es

```
ObjGráfico.Color = IntNumColor
```

Siendo *IntNumColor* el número de color o alguna de las constantes. Esta es la forma en que hemos cambiado el color a nuestra línea en el ejemplo. Existe otra sintaxis para obtener el color de un objeto dibujado y guardado en una variable:

```
IntNumColor = ObjGráfico.Color
```

donde *IntNumColor* es ahora la variable (tipo *Integer*) que guardará el número de color y *ObjGráfico* la variable (tipo *Object*) que almacena la línea. En nuestro caso anterior la línea estaba almacenada en *ObjLínea* (tipo *IAcadLine*). Si quisiéramos obtener su color y guardarlo en una variable denominada *NúmeroColor* habríamos de hacer, por ejemplo:

```
NúmeroColor = ObjLine.Color
```

Antes de esto, y si existe una instrucción *Option Explicit*, habremos de declarar la variable *NúmeroColor* como *Integer*.

NOTA: Recordemos que el color 0 es *PorBloque* y el color 256 *PorCapa*.

- **EndPoint.** Obtiene el punto final de un objeto arco, elipse o línea y, en el caso de las líneas, también puede asignarse. Así pues, la sintaxis para obtener el punto final de los objetos mencionados es:

```
VarPtoFinal = ObjGráfico.EndPoint
```

siendo *VarPtoFinal* una variable que almacena un punto. Estas variables de punto siempre han de definirse como *Variant*. En el caso de nuestra línea podríamos haber obtenido su punto final así, aunque esto no tiene mucho sentido (como con el color) porque ya sabemos su punto final; se lo hemos dado nosotros.

Si lo que deseamos es asignar un punto final (sólo para líneas) utilizaremos la siguiente sintaxis:

```
ObjLínea.EndPoint = DblPtoFinal
```

donde *ObjLínea* es una variable tipo objeto que contiene una línea y *DblPtoFinal* es una variable que contiene un punto, es decir una matriz de tres valores tipo *Double*.

- **EntityName.** Obtiene el nombre de un objeto. Este nombre es el que la clase a la que pertenece el objeto. En concreto se refiere al código 100 en la definición de la entidad en Base de Datos cuando accedíamos con *AutoLISP*, y no al código 0 que es el que utilizábamos nosotros para obtener el nombre de la entidad. La sintaxis de obtención de un nombre es:

```
StrNombre = ObjGráfico.EntityName
```

Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

donde *ObjNombre* es una variable de tipo *String* que guardará el nombre del objeto, y *ObjGráfico* es la variable que guarda el objeto. Si en el ejemplo de nuestra línea, y tras definir una variable llamada por ejemplo *NombreLínea* como *String*, escribiríamos al final:

```
NombreLínea = ObjLínea.EntityName
```

NombreLínea guardaría la cadena *AcDbLine*, nombre de clase del objeto de entidad línea.

Cuando se pregunta por objetos de **AutoCAD** se puede emplear la siguiente propiedad *EntityType*.

- *EntityType*. Obtiene el tipo de entidad. Éste ha de ser guardado en una variable de tipo *Integer* así:

```
IntNumTipo = ObjGráfico.EntityType
```

El número de tipo devuelto se corresponde con los expuestos en la siguiente tabla:

Tipo de entidad	Descripción
1	AcDB3DFace
2	AcDB3DPolyLine
3	AcDB3DSolid
4	AcDBArc
5	AcDBAttribute
6	AcDBAttributeReference
7	AcDBBlockReference
8	AcDBCircle
9	AcDBDimAligned
10	AcDBDimAngular
11	AcDBDimDiametric
12	AcDBDimOrdinate
13	AcDBDimRadial
14	AcDBDimRotated
15	AcDBEllipse
16	AcDBGroup
17	AcDBHatch
18	AcDBLeader
19	AcDBLine
20	AcDBMText
21	AcDBPoint
22	AcDBPolyline
23	AcDBPolylineLight
24	AcDBPolyMesh
25	AcDBPViewPort
26	AcDBRaster
27	AcDBRay
28	AcDBRegion
29	AcDBShape
30	AcDBSolid
31	AcDBSpline
32	AcDBText
33	AcDBTolerance
34	AcDBTrace
35	AcDBXLine

En nuestro ejemplo, una sentencia de este tipo nos devolvería en código 19.

- **Handle.** Obtiene el rótulo o código del objeto. Se trata de una cadena de texto que identifica el objeto durante toda la vida de ese objeto en el dibujo. Existe otra manera de identificar una entidad, mediante su `ObjectID` (visto más adelante). Algunas funciones de ARX requieren un `ObjectID` en lugar de `Handle`. Para los demás casos es preferible usar `Handle`. La sintaxis pues de esta propiedad es:

```
StrRótulo = ObjGráfico.Handle
```

StrRotulo es una cadena (String).

- **Layer.** Permite obtener y/o asignar la capa de/a un objeto. La sintaxis para asignar una capa a un objeto ya dibujado es:

```
ObjGráfico.Layer = StrNombreCapa
```

donde *ObjGráfico* es la variable que guarda el objeto y *StrNomCapa* una cadena de texto con el nombre de una capa existente. Si la capa no existe VBA proporciona un error de ejecución. Para evitar eso habría que crearla previamente con un método que veremos bastante más adelante. En nuestro ejemplo de la línea, si existiera una capa llamada CUERPO en el dibujo actual y hubiésemos querido colocar o introducir dicha línea en esa capa, habríamos hecho:

```
ObjLínea.Layer = "Cuerpo"
```

La sintaxis para obtener la capa de un objeto es:

```
StrNomCapa = ObjGráfico.Layer
```

StrNomCapa habrá de ser una variable del tipo String, es decir una cadena alfanumérica.

- **Linetype.** Permite obtener y/o asignar un tipo de línea de/a una entidad de dibujo. Para asignar un tipo de línea:

```
ObjGráfico.Linetype = StrNombreTipoLínea
```

siendo *ObjGráfico* el objeto en cuestión (como nuestra línea en *ObjLínea*) y *StrNomTipoLínea* una cadena que especifica el nombre del tipo de línea. Si éste no está cargado se producirá un error (cómo cargar tipos de línea se verá más adelante). Como tipos de línea especiales se pueden indicar:

ByLayer ByBlock Continuous

Para obtener el tipo de línea de un objeto:

```
StrNomTipoLínea = ObjGráfico.Linetype
```

StrNomTipoLínea será una cadena (String).

- **LinetypeScale.** Obtiene y asigna la escala de tipo de línea de/a un objeto. Para asignar:

```
ObjGráfico.LinetypeScale = RealEscalaTipoLínea
```

RealEscalaTipoLínea es el valor de la nueva escala.

Para obtener:

```
RealEscalaTipoLínea = ObjGráfico.LinetypeScale
```

RealEscalaTipoLínea es una variable positiva y real del tipo Double.

- **Normal.** Obtiene y asigna la normal de/a una entidad. La normal es un vector unitario que indica la dirección Z en el SCE (Sistema de Coordenadas de la Entidad). El vector se almacena en una variable del tipo Variant, que recogerá una matriz de tres elementos Double. Para asignar una normal:

```
ObjGráfico.Normal = DblVector
```

Para obtener una normal:

```
VarVector = ObjGráfico.Normal
```

- **ObjectID.** Extrae el identificador interno de una entidad. Este número, devuelto en notación decimal, se corresponde con el valor hexadecimal que aparecía tras *Entity name* en la definición de una entidad en Base de datos cuando accedíamos con AutoLISP; el número del código -1. Para obtenerlo:

```
LngNúmeroID = ObjGráfico.ObjectID
```

Donde *LngNúmeroID* es una variable que habrá sido declarada como Long.

- **StartPoint.** Funciona igual que *EndPoint* pero para el punto inicial de líneas, arcos y elipses. En estos dos último casos sólo se puede obtener su valor; con las líneas se puede modificar. Para asignar un nuevo punto inicial a una línea la sintaxis es:

```
ObjLínea.StartPoint = DblPtoInic
```

Donde *ObjLínea* sólo puede ser un objeto de línea exclusivamente y *DblPtoInic* una variable que almacene un punto o una matriz de tres valores tipo Double.

Para obtener el punto inicial de los objetos mencionados:

```
VarPtoInic = ObjLínea.StartPoint
```

VarPtoInic habrá sido declarada como Variant.

- **Thickness.** Obtiene y/o asigna un valor que representa la altura en Z de un objeto 2D. Para asignar, la sintaxis es la siguiente:

```
ObjGráfico.Thickness = DblAlturaObjeto
```

y para obtener la elevación de un objeto:

```
DblAlturaObjeto = ObjGráfico.Thickness
```

DblAlturaObjeto habrá sido declarada como Double.

- **Visible.** Obtiene y/o asigna la visibilidad de/a una entidad. Esta propiedad es del tipo Boolean, por lo que sus resultados sólo podrán ser True o False. Si queremos asignar un estado de visibilidad a un objeto utilizaremos la siguiente sintaxis:


```
ObjGráfico.Visible = BooEstadoVis
```

Por ejemplo, en el caso de la macro que dibujaba una línea, si al final del cuerpo de la macro (antes de End Sub) escribiéramos:

```
ObjLínea.Visible = False
```

y la ejecutáramos, la línea se habría dibujado, pero permanecería invisible. A veces puede ser interesante mantener objetos invisibles para mostrarlos en un momento dado.

Si queremos extraer el estado de visibilidad de un objeto utilizaremos la sintaxis que sigue:

```
BooEstadoVis = ObjGráfico.Visible
```

donde *BooEstadoVis* habrá sido declarado, como norma general, como Boolean.

A parte de las propiedades, los nuevos objetos VBA para **AutoCAD** también poseen métodos. Veremos a continuación todos los métodos para las líneas uno por uno.

NOTA: Al igual que para las propiedades, los métodos de los objetos son muchos de ellos comunes, por lo que nos referiremos aquí a ellos como globales, no como particulares de la creación de líneas.

- **ArrayPolar.** Este método crea una matriz polar del objeto indicado, especificando un centro, los grados abarcados y el número de objetos. La sintaxis de este método es la siguiente:

```
VarMatriz = ObjGráfico.ArrayPolar(IntNúmero, DblÁngulo, DblCentro)
```

Donde *VarMatriz* ha de ser una variable declarada como Variant que guardará todos los objetos de la matriz o *array*. *ObjGráfico* es el objeto ya creado del que se realizará la matriz polar, *IntNúmero* es el número de objetos de la matriz (Integer), *DblÁngulo* los ángulos cubiertos en radianes (Double) y *DblCentro* el centro de la matriz (un punto de tres elementos Double).

Así, y retomando el ejemplo que hemos visto de dibujo de una línea, veámoslo ampliado a continuación:

```
Option Explicit
Dim AcadDoc As Object
Dim AcadModel As Object

Dim ObjLínea As Object
Dim PuntoInicial (1 To 3) As Double
Dim PuntoFinal (1 To 3) As Double
Dim MatrizLin As Variant
Dim PuntoBase (1 To 3) As Double

Sub DibujoLínea()
    Set AcadDoc = GetObject(, "Autocad.Application").ActiveDocument
    Set AcadModel = AcadDoc.ModelSpace
    PuntoInicial(1) = 100: PuntoInicial(2) = 100: PuntoInicial(3) = 0
    PuntoFinal(1) = 200: PuntoFinal(2) = 200: PuntoFinal(3) = 0
    Set ObjLínea = AcadModel.AddLine(PuntoInicial, PuntoFinal)

    PuntoBase(1) = 10: PuntoBase(2) = 10: PuntoBase(3) = 0
```

```
MatrizLin = ObjLínea.ArrayPolar(10, 3.14159, PuntoBase)
End Sub
```

En este nuevo ejemplo, a parte de lo anterior se declara la variable `MatrizLin` como `Variant` y `PuntoBase` como una tabla de tres valores `Double`. Después se realiza la matriz (tras dar valores a `PuntoBase`). El número de elementos y los ángulos cubiertos pueden ser también variables, evidentemente declaradas como `Integer` y `Double` respectivamente.

Podemos acceder a la tabla que guarda los objetos mediante índices para obtener los propios objetos de la matriz. En el ejemplo anterior, únicamente deberemos añadir las siguientes dos líneas al listado (antes de `End Sub`):

```
Dim Línea6 As Object
Set Línea6 = MatrizLin(6)
```

para guardar en la variable `Línea6` la sexta de las líneas de la matriz en este caso. Esta variable tiene que ser declarada como `Object` porque va a guardar un objeto. Además el índice habrá de ser coherente con el número de objetos claro está, sino se produce un error de ejecución en el módulo VBA.

- **ArrayRectangular.** Este método crea una matriz polar de dos o tres dimensiones repitiendo el objeto al que se le aplica. Hay que indicar el número de filas, columnas y niveles, así como las distancias correspondientes, de la siguiente manera:

```
VarMatriz = ObjGráfico.ArrayRectangular(IntNúmFil, IntNumCol, IntNumNiv,
DblDistFil, DblDistCol, DblDistNiv)
```

Los números de filas, columnas y niveles serán enteros (`Integer`) y las distancias lo más lógico es que sean de doble precisión (`Double`).

Las demás consideraciones, así como la manera de acceder a los objetos simples de la matriz, son las mismas que para `ArrayPolar`.

- **Copy.** Método que copia la entidad a la que se aplica en la misma posición que el objeto original. Su sintaxis es simplemente:

```
Set ObjCopia = ObjGráfico.Copy
```

`ObjCopia` será un objeto, por lo que estará definido como tal. Para copiar la línea sexta de la matriz del último ejemplo, la cual habíamos guardado en `Línea6`, haríamos por ejemplo:

```
Dim Copia6 As Object
Set Copia6 = Línea6.Copy
```

Para copiar un objeto y moverlo a su vez, utilizaremos primero este método y luego el método `Move`, que veremos después.

- **Erase.** Elimina la entidad a la que se aplica el método. Como en este caso no hemos de guardar objeto ni valor alguno en ninguna variable, podemos llamar al método con `Call`, pero al no necesitar argumentos no hace falta. Su sintaxis es pues:

```
ObjGráfico.Erase
```

Para borrar por ejemplo la línea sexta, del ejemplo que venimos manejando, guardada en `Línea6` haremos:

Línea6.Erase

• **GetBoundingBox.** Este método devuelve las coordenadas inferior izquierda y superior derecha de la caja de abarque del objeto al que se le aplique. Esta caja de abarque es el rectángulo que se ajusta al objeto abarcándolo por completo. La sintaxis de **GetBoundingBox** es:

Call *ObjGráfico*.GetBoundingBox(*VarInfIzq*, *VarSupDcha*)

La forma de trabajar este método es un poco extraña. *VarInfIzq* y *VarSupDcha* han de ser forzosamente dos variables declaradas, antes de llamar al método, como **Variant**. Al final de la ejecución guardarán las coordenadas en cuestión. Si deseamos ahora utilizar esos puntos para por ejemplo dibujar una línea, no podremos utilizarlos como tales, sino que habremos de realizar un trasvase de variables e introducir los diferentes valores en una matriz de tres elementos. Veamos una ampliación más de nuestro ejemplo de la línea:

```
Option Explicit
Dim AcadDoc As Object
Dim AcadModel As Object

Dim ObjLínea As Object
Dim PuntoInicial (1 To 3) As Double
Dim PuntoFinal (1 To 3) As Double
Dim MatrizLin As Variant
Dim PuntoBase (1 To 3) As Double

Sub DibujoLínea()
    Set AcadDoc = GetObject( , "Autocad.Application").ActiveDocument
    Set AcadModel = AcadDoc.ModelSpace
    PuntoInicial(1) = 100: PuntoInicial(2) = 100: PuntoInicial(3) = 0
    PuntoFinal(1) = 200: PuntoFinal(2) = 200: PuntoFinal(3) = 0
    Set ObjLínea = AcadModel.AddLine(PuntoInicial, PuntoFinal)

    PuntoBase(1) = 10: PuntoBase(2) = 10: PuntoBase(3) = 0
    MatrizLin = ObjLínea.ArrayPolar(10, 3.14159, PuntoBase)
    Dim Línea6 As Object
    Set Línea6 = MatrizLin(6)

    Dim EsqInfIzq As Variant
    Dim EsqSupDch As Variant
    Call Línea6.GetBoundingBox(EsqInfIzq, EsqSupDch)
    Dim Pto1 (1 To 3) As Double
    Dim Pto2 (1 To 3) As Double
    Pto1(1) = EsqInfIzq(0): Pto1(2) = EsqInfIzq(1): Pto1(3) = EsqInfIzq(2)
    Pto2(1) = EsqSupDch(0): Pto2(2) = EsqSupDch (1): Pto2(3) = EsqSupDch(2)
    Set ObjLínea = AcadModel.AddLine(Pto1, Pto2)
End Sub
```

Véase cómo se realiza el trasvase de variables. En el caso de las variables **Pto1** y **Pto2**, las hemos definido con un rango de 1 a 3, sin embargo en **EsqInfIzq** y **EsqSupDch** se utiliza el rango 0 a 2, por lo que los índices están desfasados una unidad.

NOTA: Este mecanismo lo deberemos utilizar bastantes veces, como veremos. Inexplicablemente los puntos guardados en variables **Variant**, que da la casualidad de que son las variables de salida de métodos como este visto u otros más importantes (como el equivalente al **GETPOINT** de **AutoLISP**, que ya estudiaremos), no pueden ser utilizados directamente para, por ejemplo, dibujar entidades. Aspecto que debe solucionarse.

- **GetXData.** Permite recuperar los datos extendidos de una entidad. Su sintaxis es la que sigue:

```
Call ObjGráfico.GetXData(StrNomAplicación, VarTipoXData, VarValorXData)
```

Los datos extendidos no los crea **AutoCAD** sino una aplicación AutoLISP o ADS. Si embargo, estos datos se almacenan en la base de datos con los restantes del dibujo.

StrNomAplicación es el nombre de la aplicación que ha creado los datos extendido; es una cadena (String). Si se especifica una cadena vacía se devuelven todos los datos extendidos de todas las aplicaciones.

VarTipoXData y *VarValorXData* son obligatoriamente dos variables de salida definidas como Variant antes de llamar al método. Estas variables almacenarán un *array* de códigos de tipo de datos y un *array* con el tipo de dato para cada código respectivamente.

- **Highlight.** Asigna lo que se denomina puesta en relieve de una entidad, es decir, su resalte o designación (línea de trazos en lugar de continua). La sintaxis es la siguiente:

```
ObjGráfico.Highlight(BooRelieve)
```

BooRelieve es Boolean. El valor True hace que la entidad se resalte y False que no se resalte. A veces puede ser necesario realizar una regeneración o utilizar el método Update para que los resultados sean visibles. La mayoría de las veces no hace falta.

- **IntersectWith.** Calcula todos los puntos de intersección de dos entidades y los devuelve en forma de matriz de puntos. La sintaxis es:

```
VarMatrizPtos = ObjGráfico1.IntersectWith(ObjGráfico2, OpciónExt)
```

Se indican los objetos gráficos con los que se desea calcular las intersecciones y una opción de extensión cuyos valores son:

Opción	Descripción
AcExtendNone	No se prolonga ninguna entidad
AcExtendThisEntity	Se prolonga <i>ObjGráfico1</i>
AcExtendOtherEntity	Se prolonga <i>ObjGráfico2</i>
AcExtendBoth	Se prolongan ambas entidades

El resultado es una matriz de valores de puntos tipo Variant. Si no hay puntos de intersección, su valor es nulo.

- **Mirror.** Produce una simetría de la entidad en cuestión con respecto a un eje de simetría que viene definido por dos puntos, ambos *arrays* de tres elementos tipo Double. El valor devuelto es un nuevo objeto simétrico al inicial (variable tipo Object). La sintaxis de este método es:

```
Set ObjSimétrico = ObjGráfico.Mirror(DblPto1, DblPto2)
```

Los objetos no originales no son eliminados y, si deseáramos hacerlo, deberíamos utilizar el método Erase tras la ejecución de la simetría.

NOTA: Recordemos la función de la variable `MIRRTEXT` a la hora de hacer simetrías de textos. Veremos más adelante cómo dar valores a variables de sistema de **AutoCAD**.

Veamos un ejemplo de este método:

```
Option Explicit
Dim AcadDoc As Object
Dim AcadModel As Object

Dim ObjLínea As Object
Dim PuntoInicial(1 To 3) As Double
Dim PuntoFinal(1 To 3) As Double
Dim PuntoSim(1 To 3) As Double
Dim SimLin As Object

Sub DibujoLínea()
    Set AcadDoc = GetObject(, "Autocad.Application").ActiveDocument
    Set AcadModel = AcadDoc.ModelSpace
    PuntoInicial(1) = 100: PuntoInicial(2) = 100: PuntoInicial(3) = 0
    PuntoFinal(1) = 200: PuntoFinal(2) = 200: PuntoFinal(3) = 0
    Set ObjLínea = AcadModel.AddLine(PuntoInicial, PuntoFinal)
    PuntoSim(1) = 200: PuntoSim(2) = 100: PuntoSim(3) = 0
    Set SimLin = ObjLínea.Mirror(PuntoFinal, PuntoSim)
End Sub
```

En este ejemplo, tras dibujar una línea se realiza una simetría de la misma, apoyándonos en un punto propio de la línea origen y en otro recién creado.

- **Mirror3D.** De manera similar al método anterior, **Mirror3D** produce una simetría tridimensional de un objeto con respecto a un plano definido por tres puntos. Los tipos de datos son iguales que en **Mirror**. Veamos la sintaxis de **Mirror3D**:

```
Set ObjSimétrico3D = ObjGráfico.Mirror3D(DblPto1, DblPto2, DblPto3)
```

- **Move.** Se utiliza para desplazar un objeto apoyándose en un punto de base que se moverá a otro punto indicado. Ambos puntos son matrices de tres elementos del tipo **Double**. Como no obtenemos ningún objeto, matriz de puntos o valor como devolución, sino simplemente un desplazamiento, deberemos llamar al método con **Call**.

La sintaxis de este método es:

```
Call ObjGráfico.Move(DblPto1, DblPto2)
```

Si en el último ejemplo hubiésemos querido desplazar el objeto simétrico, llevando el segundo punto del eje de simetría al último punto de la primera línea, habríamos añadido el siguiente renglón tras la simetría:

```
Call SimLin.Move(PuntoSim, PuntoFinal)
```

- **Offset.** Este método crea una copia paralela o equidistante a un objeto y según una distancia dada. La copia será una matriz de objetos **Variant**; lo más lógico es que la distancia sea una variable **Double** (si es variable y no se indica directamente un valor, claro está):

```
VarEquidist = ObjGráfico.Offset(DblDistancia)
```

Para acceder posteriormente a cada objeto podemos utilizar la técnica de las matrices.

Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

• **Rotate.** Gira el objeto en cuestión con respecto a un punto base (tabla o *array* de tres elementos *Double*) y según un ángulo indicado en radianes (*Double*). Su sintaxis es:

```
Call ObjGráfico.Rotate(DblPtoBase, DblÁngulo)
```

Para girar, por poner un ejemplo, la simetría que hemos movido en el ejemplo anterior, añadiremos al código del programa:

```
Call SimLin.Rotate(PuntoFinal, (3.14159 / 2))
```

• **Rotate3D.** Al igual que **Rotate** gira un entidad, esta vez en el espacio 3D y tomando como eje de giro el que determinan dos puntos. Los tipos de datos son los mismos. Su sintaxis es:

```
Call ObjGráfico.Rotate3D(DblPto1, DblPto2, DblÁngulo)
```

• **ScaleEntity.** Aplica un factor de escala al objeto con el cual esté asociado y a partir de un punto base indicado. El tipo de dato de la escala dependerá de nuestra conveniencia, pero es lógico que sea *Double*. Su sintaxis es la que sigue:

```
Call ObjGráfico.ScaleEntity(DblPtoBase, DblFactorEscala)
```

• **SetXData.** Permite asignar datos extendidos a una entidad. Se suministra un *array Integer* con el código para cada tipo de dato y un *array Variant* con los datos, así:

```
Call ObjGráfico.SetXData(IntTipoXDato, VarValorXDato)
```

• **TransformBy.** Mueve, escala y gira el objeto mediante una matriz de transformación de 4x4. Los elementos de la matriz son de tipo *Double* y deben tener una disposición concreta:

R00	R01	R02	T0
R10	R11	R12	T1
R20	R21	R22	T2
0	0	0	1

La rotación se indica en la submatriz 3x3 que lleva la letra *R*, y en la columna marcada con *T* se indica la traslación. También en la submatriz de 3x3 es donde se indica el factor de escala. La sintaxis de **TransformBy** es:

```
Call ObjGráfico.TransformBy(DblMatriz)
```

• **Update.** Actualiza el objeto al que se le aplica, que puede ser una entidad o incluso todo el dibujo. Su sintaxis es la que sigue:

```
ObjGráfico.Update
```

Realizar una actualización del un objeto puede ser necesario en determinadas ocasiones en que una operación en cuestión no produce el efecto deseado hasta que se actualiza.

Si quisiéramos por ejemplo actualizar la primera línea dibujada del ejemplo que venimos manejando hasta ahora haríamos:

```
ObjLínea.Update
```

Si lo que queremos es actualizar la aplicación, habríamos de definir un objeto así por ejemplo:

```
Dim AcadApp As Object
```

para darle un valor así:

```
Set AcadApp = GetObject(, "AutoCAD.Application")
```

y luego:

```
AcadApp.Update
```

En este caso declararíamos el objeto de documento activo simplemente así:

```
Set AcadDoc = AcadApp.ActiveDocument
```

Después de estudiar todas las propiedades y todos los métodos del objeto de línea, pasaremos ahora a ver los demás métodos de las estas colecciones existentes para la creación de objetos o entidades gráficas de dibujo desde VBA. Muchas propiedades y/o métodos son comunes a todos los objetos gráficos, por lo que, de aquí en adelante, únicamente se expondrán aquellos que no se hayan visto ya. Es decir, iremos aprendiendo las características nuevas de cada objeto gráfico sino se han visto con anterioridad en otro. Aún así, de cada objeto se proporcionan listadas todas sus propiedades y listados sus métodos, resaltando en negrita los que se explican en la sección en curso.

DOCE.5.2. Círculos

El método `AddCircle` permite dibujar círculos en la colección de objetos de Espacio Modelo, Espacio Papel o formando parte de un bloque. Hay que indicar las coordenadas del punto del centro, que será un matriz de tres elementos (coordenadas X, Y y Z) de tipo de dato `Double`, y el radio del círculo, que también será `Double`. La sintaxis de `AddCircle` es:

```
Set ObjCírculo = ObjColección.AddCircle(DblCentro, DblRadio)
```

Propiedades de los objetos de círculo:

<code>Application</code>	<code>Area</code>	<code>Center</code>	<code>Color</code>	<code>EntityName</code>
<code>EntityType</code>	<code>Handle</code>	<code>Layer</code>	<code>Linetype</code>	<code>LinetypeScale</code>
<code>Normal</code>	<code>ObjectID</code>	<code>Radius</code>	<code>Thickness</code>	<code>Visible</code>

Métodos de los objetos de círculo:

<code>ArrayPolar</code>	<code>ArrayRectangular</code>	<code>Copy</code>	<code>Erase</code>
<code>GetBoundingBox</code>	<code>GetXData</code>	<code>Highlight</code>	<code>IntersectWith</code>
<code>Mirror</code>	<code>Mirror3D</code>	<code>Move</code>	<code>Offset</code>
<code>Rotate</code>	<code>Rotate3D</code>	<code>ScaleEntity</code>	<code>SetXData</code>
<code>TransformBy</code>	<code>Update</code>		

Como podemos observar, los métodos son los mismos que para las líneas. En cuestión de propiedades desaparecen dos evidentemente (`StartPoint` y `EndPoint`) y aparecen `Area`, `Radius` y `Center`, que son las que vamos a comentar a continuación.

- **Area.** Obtiene el área de una entidad cerrada (círculo, elipse, arco, polilínea, región o spline). La variable que guarde el dato de salida habrá sido declarado como `Double`. La sintaxis de esta propiedad es:

Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

```
DblÁrea = ObjGráfico.Area
```

- **Center.** Obtiene y asigna el centro de la entidad. La sintaxis para asignar un nuevo centro es:

```
ObjGráfico.Center = DblPtoCentro
```

siendo *DblPtoCentro* un *array* de tres elementos *Double* que guarda las coordenadas X, Y y Z del nuevo centro. Para obtener el centro utilizaremos la sintaxis siguiente:

```
VarPtoCentro = ObjGráfico.Center
```

donde *VarPtoCentro* será una variable declarada como *Variant* que guardará las coordenadas del centro. Recordemos que para utilizar luego estás coordenadas, a la hora de suministrarlas como puntos de dibujo para objetos gráficos, habrá que realizar un trasvase de variables y guardarlas en una matriz o *array* de tres elementos tipo *Double*.

- **Radius.** Obtiene y asigna el radio de un círculo (o de un arco). El valor en cuestión, tanto uno nuevo para asignar como el que recogerá una variable al obtener, será del tipo *Double*. Por lo que si se utilizan variables (al obtener seguro, pero el asignar no es obligatorio) deberán declararse como de doble precisión.

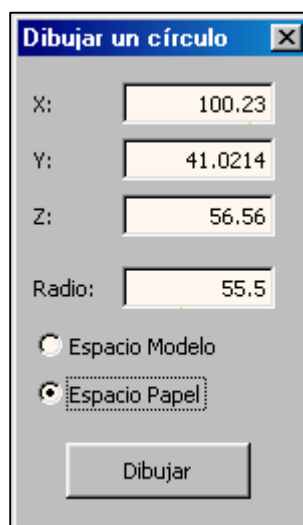
La sintaxis para asignar un radio es:

```
ObjGráfico.Radius = DblRadio
```

y la sintaxis para obtener el radio de una entidad ya dibujada es:

```
DblRadio = ObjGráfico.Radius
```

En el siguiente pequeño programa se crea un sencillo cuadro de diálogo (formulario en Visual Basic) que permite, mediante una serie de casillas de edición y un botón de acción, dibujar un círculo en el Espacio Modelo o Espacio Papel, según el botón excluyente que esté activado. El cuadro de diálogo es el siguiente:



El listado del código es el que sigue:

Option Explicit


```
Dim AcadDoc As Object
Dim AcadModel As Object
Dim AcadPapel As Object
Dim DibujoEn As Object
Dim CírculoObj As Object
Dim PtoCentro(1 To 3) As Double
Dim Radio As Double

Private Sub buttonDibujar_Click()
    PtoCentro(1) = Val(boxX.Text)
    PtoCentro(2) = Val(boxY.Text)
    PtoCentro(3) = Val(boxZ.Text)
    Radio = Val(boxRadio.Text)
    formPrincipal.Hide
    If optionModelo.Value = True Then
        Set DibujoEn = AcadModel
    Else
        Set DibujoEn = AcadPapel
    End If
    Set CírculoObj = DibujoEn.AddCircle(PtoCentro, Radio)
End Sub
```

```
Private Sub UserForm_Initialize()
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument
    Set AcadModel = AcadDoc.ModelSpace
    Set AcadPapel = AcadDoc.PaperSpace
End Sub
```

Como vemos, tras inicializar todas las variables se muestra el cuadro. La acción al pulsar el botón *Dibujar* (buttonDibujar es su nombre) pasa por extraer los valores de las casillas de coordenadas (boxX, boxY y boxZ), pasarlos a valores numéricos e introducirlos en la matriz que será el punto centro del círculo. También se realiza lo mismo con el radio (casilla boxRadio). Por último se dibuja el círculo en el Espacio Modelo o en el Espacio Papel según el valor de la variable de objeto DibujoEn, que lo que hace es almacenar un objeto u otro (Modelo o Papel) según qué botón excluyente o de opción esté marcado (optionModelo u optionPapel).

NOTA: Se podrían haber introducido también diversos controles de errores.

DOCE.5.3. Elipses

El método AddEllipse permite dibujar elipses o arcos de elipse en la colección de objetos de Espacio Modelo, Espacio Papel o en la colección de bloques. La sintaxis de AddEllipse es:

<pre>Set ObjElipse = ObjColección.AddEllipse(DblCentro, DblPtoEjeMayor, DblRelación)</pre>

El centro ha de indicarse como una matriz de elementos Double (*DblCentro*); *DblPtoEjeMayor* es un punto (matriz de tres valores Double) en uno de los extremos del eje mayor, considerando las coordenadas en el SCO y relativas al centro; *DblRelación* es un valor Double que es la relación entre la medida del eje menor y del eje mayor (valor máximo igual a 1 se corresponde con un círculo).

Propiedades de los objetos de elipse:

Application	Area	Center	Color	EndAngle
EndParameter	EndPoint	EntityName	EntityType	Handle
Layer	LineType	LinetypeScale	MajorAxis	MinorAxis
Normal	ObjectID	RadiusRatio	StartAngle	StartParameter
StartPoint	Visible			

Métodos de los objetos de elipse:

ArrayPolar	ArrayRectangular	Copy	Erase
GetBoundingBox	GetXData	Highlight	IntersectWith
Mirror	Mirror3D	Move	Offset
Rotate	Rotate3D	ScaleEntity	SetXData
TransformBy	Update		

Los métodos son los mismos que lo ya estudiados, pero hay propiedades nuevas.

- **EndAngle**. Obtiene y/o asigna el ángulo final en radianes de un objeto elipse (o arco). El ángulo se considera desde el cero trigonométrico. Recorriendo el arco en el sentido antihorario, el primer extremo será el punto inicial y el segundo extremo el final. Un círculo cerrado serían 6.28 radianes.

La sintaxis para asignar un ángulo final a una elipse (o arco) ya dibujada será pues:

```
ObjGráfico.EndAngle = DblÁngulo
```

y la sintaxis para obtener el ángulo de una entidad ya dibujada será:

```
DblÁngulo = ObjGráfico.EndAngle
```

DblÁngulo es Double.

- **EndParameter**. Sólo aplicable a elipses, obtiene y asigna el parámetro final de una elipse (variable de tipo Double). Los parámetros inicial y final de una elipse se calculan según la ecuación $P(q) = A \cdot \cos(q) + B \cdot \sin(q)$, donde *A* y *B* son la mitad del eje mayor y del menor respectivamente. El ángulo *q* será el inicial o final. Se pueden emplear los ángulos o los parámetros para crear una elipse. El parámetro final tiene un rango de 0 a $2 \times \pi$.

Para asignar un parámetro final:

```
ObjGráfico.EndParameter = DblParamFinal
```

Para obtener un parámetro final:

```
DblParamFinal = ObjGráfico.EndParameter
```

- **MajorAxis**. Sólo aplicable a elipses, obtiene y asigna el punto extremo del eje mayor (es una matriz de tres elementos tipo Double). Sus coordenadas toman como origen el centro de la elipse y se miden en el SCO (Sistema de Coordenadas del Objeto).

La sintaxis para asignar un punto extremo de eje mayor a una elipse es:

```
ObjGráfico.MajorAxis = DblPtoEjeMayor
```

Para obtener un punto extremo de eje mayor de una elipse, la sintaxis es:

```
DblPtoEjemayor = ObjGráfico.MayorAxis
```

- MinorAxis. Enteramente similar a la propiedad anterior, pero obtiene el punto extremo del eje menor de una elipse. No se puede asignar un punto extremo de eje menor, únicamente obtener:

```
DblPtoEjeMenor = ObjGráfico.MinorAxis
```

- RadiusRatio. Sólo aplicable a elipses, define la relación entre el tamaño del eje menor de la elipse y el tamaño del eje mayor. La sintaxis para asignar una relación:

```
ObjGráfico.RadiusRatio = DblRelación
```

Para obtener la relación de una elipse:

```
DblRelación = ObjGráfico.RadiusRatio
```

El valor máximo admitido para la variable Double *DblRelación* es 1, que significa que ambos ejes son iguales y por lo tanto se trata de un círculo.

- StartAngle. Igual que EndAngle pero para el ángulo inicial. Para asignar uno:

```
ObjGráfico.StartAngle = DblÁngulo
```

Para obtener uno:

```
DblÁngulo = ObjGráfico.StartAngle
```

DblÁngulo es Double.

- StartParameter. Igual que EndParameter pero para el parámetro inicial. Para asignar uno:

```
ObjGráfico.StartParameter = DblParamInicial
```

Para obtener uno:

```
DblParamInicial = ObjGráfico.StartParameter
```

DOCE.5.4. Arcos

El método AddArc permite crear un arco de circunferencia que se dibuja desde el punto inicial al final en sentido contrario a las agujas del reloj. Estos puntos se calculan a partir de las propiedades StartAngle, EndAngle y Radius.

Para crear un arco hay que indicar el centro (array de tres valores Double), el radio (Double), el ángulo inicial (Double) en radianes y el ángulo final (Double) también en radianes. La sintaxis de AddArc es:

```
Set ObjArco = ObjColección.AddArc(DblCentro, DblRadio, DblÁngInic, DblÁngFin)
```

Propiedades de los objetos de arco:

Application	Area	Center	Color	EndAngle
-------------	------	--------	-------	----------

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en Visual Basic orientada a AutoCAD (VBA)

EndPoint	EntityName	EntityType	Handle	Layer
LineType	LinetypeScale	Normal	ObjectID	Radius
StartAngle	StartPoint	Thickness	Visible	

Métodos de los objetos de arco:

ArrayPolar	ArrayRectangular	Copy	Erase
GetBoundingBox	GetXData	Highlight	IntersectWith
Mirror	Mirror3D	Move	Offset
Rotate	Rotate3D	ScaleEntity	SetXData
TransformBy	Update		

Todos los métodos y propiedades de los arcos han sido ya explicados en las secciones anteriores.

DOCE.5.5. Puntos

Para añadir puntos como entidades gráficas se utiliza el método `AddPoint`, ya sea en cualquiera de las colecciones que lo poseen (Espacio Modelo, Papel o bloques). La manera es sencilla, ya que únicamente hay que suministrar al método un punto que es, como siempre, una matriz o *array* de tres elementos (coordenadas X, Y y Z) de tipo de dato `Double`.

La sintaxis para este método es:

```
Set ObjPunto = ObjColección.AddPoint(DblPunto)
```

Propiedades de los objetos de punto:

Application	Color	Coordinates	EntityName	EntityType
Handle	Layer	LineType	LinetypeScale	Normal
ObjectID	Thickness	Visible		

Métodos de los objetos de punto:

ArrayPolar	ArrayRectangular	Copy	Erase
GetBoundingBox	GetXData	Highlight	IntersectWith
Mirror	Mirror3D	Move	Rotate
Rotate3D	ScaleEntity	SetXData	TransformBy
Update			

Los métodos han sido ya estudiados y las propiedades se han visto todas excepto una nueva.

- **Coordinates.** Obtiene y/o asigna una matriz con las coordenadas de cada uno de los vértices del objeto dado; en el caso del punto hay un único vértice.

La variable que se asigna a estos vértices (si la hubiera) se declara con una sentencia tipo `Dim Vértices (1 To n * 3) As Double`, donde *n* es el número de vértices o puntos.

Para asignar una matriz de coordenadas a un objeto punto:

```
ObjGráfico.Coordinates = DblMatrizVértices
```

Y para obtenerla:

```
VarVértices = ObjGráfico.Coordinates
```

Donde *VarVértices* es una variable declarada como *Variant*. Recordemos que si queremos utilizar después las coordenadas habrá que hacer un trasvase a una matriz de tres elementos *Double*.

DOCE.5.6. Texto en una línea

El método *AddText* permite añadir texto en una línea al dibujo. Hemos de indicar la cadena de texto en cuestión (variable *String*), el punto de inserción en el SCU (matriz *Double* de tres valores) y la altura (*Double*). La sintaxis es pues que sigue:

```
Set ObjTexto = ObjColección.AddText(StrTexto, DblPtoIns, DblAltura)
```

Propiedades de los objetos de texto:

<i>Application</i>	<i>Color</i>	<i>EntityName</i>	<i>EntityType</i>
<i>Handle</i>	Heigh	HorizontalAlignment	InsertionPoint
<i>Layer</i>	<i>LineType</i>	<i>LinetypeScale</i>	<i>Normal</i>
<i>ObjectID</i>	ObliqueAngle	Rotation	ScaleFactor
StyleName	TextAlignmentPoint	TextGenerationFlag	TextString
<i>Thickness</i>	VerticalAlignment	<i>Visible</i>	

Métodos de los objetos de texto:

<i>ArrayPolar</i>	<i>ArrayRectangular</i>	<i>Copy</i>	<i>Erase</i>
<i>GetBoundingBox</i>	<i>GetXData</i>	<i>Highlight</i>	<i>IntersectWith</i>
<i>Mirror</i>	<i>Mirror3D</i>	<i>Move</i>	<i>Rotate</i>
<i>Rotate3D</i>	<i>ScaleEntity</i>	<i>SetXData</i>	<i>TransformBy</i>
<i>Update</i>			

Los métodos se han visto todos ya; las nuevas propiedades se estudian a continuación.

- **Heigh**. Obtiene y/o asigna la altura de un objeto. En el caso del texto se refiere a la altura de las letras mayúsculas. La sintaxis para asignar una altura a un objeto es:

```
ObjGráfico.Heigh = DblAltura
```

Y para obtenerla:

```
DblAltura = ObjGráfico.Heigh
```

Siendo *DblAltura* una variable declarada como *Double*.

- **HorizontalAlignment**. Obtiene o asigna la alineación horizontal de y a los textos exclusivamente. Las sintaxis son:

para asignar:

```
ObjTexto.HorizontalAlignment = IntAlineaciónH
```

para obtener:

```
IntAlineaciónH = ObjTexto.HorizontalAlignment
```

siendo los valores para *IntAlineaciónH* del tipo *Integer*, aunque también se admiten las siguientes constantes:

Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

acHorizontalAlignmentLeft
acHorizontalAlignmentRight
acHorizontalAlignmentMiddle

acHorizontalAlignmentCenter
acHorizontalAlignmentAligned
acHorizontalAlignmentFit

- **InsertionPoint.** Esta propiedad obtiene y asigna el punto de inserción de una entidad. El punto de inserción es una matriz de tres elementos *Double*, como cualquier otro punto:

```
ObjGráfico.InsertionPoint = DblPtoIns
```

Con esta sintaxis asignamos un nuevo punto de inserción a un objeto. Para extraer u obtener el punto de inserción de una entidad (ahora *Variant*) utilizaremos:

```
VarPtoIns = ObjGráfico.InsertionPoint
```

- **ObliqueAngle.** Asigna y obtiene el ángulo de oblicuidad de un texto medido en radianes. Para valores positivos se produce una inclinación a la derecha; los valores negativos se transforman en su equivalente positivo para almacenarlos, para ello se les suma 2π .

Para asignar un ángulo de oblicuidad:

```
ObjTexto.ObliqueAngle = DblÁngulo
```

Para obtener el ángulo de oblicuidad de un texto:

```
DblÁngulo = ObjTexto.ObliqueAngle
```

DblÁngulo es un variable de doble precisión *Double*.

- **Rotation.** Obtiene y/o asigna un ángulo de rotación en radianes (*Double*) para un texto, medido a partir del eje X y en sentido antihorario.

Para asignar:

```
ObjTexto.Rotation = DblÁngulo
```

Para obtener:

```
DblÁngulo = ObjTexto.Rotation
```

- **ScaleFactor.** Obtiene y/o asigna el factor de escala de anchura (*Double*) para una entidad de texto.

Para asignar:

```
ObjTexto.ScaleFactor = DblEscala
```

Para obtener:

```
DblEscala = ObjTexto.ScaleFactor
```

- **StyleName.** Obtiene y/o asigna el nombre (*String*) de estilo de texto (o acotación) empleado para el objeto. El estilo deberá estar definido en el dibujo, si no es así se producirá un error. A definir un estilo desde VBA ya aprenderemos más adelante. Si no se asigna estilo se asume por defecto el actual.

Para asignar:

```
ObjGráfico.StyleName = StrEstilo
```

Para obtener:

```
StrEstilo = ObjGráfico.StyleName
```

- **TextAlignmentPoint**. Obtiene y/o asigna la posición del punto de alineación del texto. Se trata pues de un *array* de tres elementos de tipo **Double** a la hora de asignar, y de una variable **Variant** a la hora de recoger los valores de un objeto texto.

Para asignar pues:

```
ObjTexto.TextAlignmentPoint = DblPtoAlineación
```

Para obtener:

```
VarPtoAlineación = ObjTexto.TextAlignmentPoint
```

- **TextGenerationFlag**. Obtiene y/o asigna el tipo de generación del texto. Es un valor **Integer** pero que posee también estas dos constantes:

acTextFlagBackward

acTextFlagUpsideDown

para aplicar una generación de "atrás hacia adelante" y de "arriba a abajo" respectivamente. Si se quieren aplicar los dos efectos se deben ejecutar dos instrucciones, una con cada valor.

Para asignar pues:

```
ObjTexto.TextGenerationFlag = IntGeneración
```

Para obtener:

```
IntGeneración = ObjTexto.TextGenerationFlag
```

- **TextString**. Obtiene y/o asigna la cadena de texto de una entidad texto. Se pueden incluir en la cadena (valor tipo **String**) los caracteres especiales de **AutoCAD** (los precedidos por %%).

Para asignar una cadena la sintaxis es:

```
ObjTexto.TextString = StrTexto
```

Para obtener una cadena la sintaxis es:

```
StrTexto = ObjTexto.TextString
```

- **VerticalAlignment**. Obtiene o asigna la alineación vertical de y a los textos exclusivamente. Las sintaxis son:

para asignar:

```
ObjTexto.VerticalAlignment = IntAlineaciónV
```

para obtener:

<code>IntAlineaciónV = ObjTexto.VerticalAlignment</code>

siendo los valores para `IntAlineaciónV` del tipo Integer, aunque también se admiten las siguientes constantes:

<code>acVerticalAlignmentBaseline</code>	<code>acVerticalAlignmentBottom</code>
<code>acVerticalAlignmentMiddle</code>	<code>acVerticalAlignmentTop</code>

Veamos a continuación un ejemplo de un programa que maneja textos.

Primero mostraremos es código del programa:

Option Explicit

```
Dim AcadDoc As Object
Dim AcadModel As Object
Dim ObjTexto As Object
Dim PuntoIns(1 To 3) As Double
Dim TextoV As String
Dim ColorV As Integer: Dim CapaV As String: Dim TipoLinV As String
Dim EstiloV As String: Dim AlturaV As Double: Dim OblicuidadV As Double
Dim AlV As String
Dim AlH As String
Dim PI As Double
```

```
Private Sub Aceptar_Click()
    On Error Resume Next
    PuntoIns(1) = Val(X.Text)
    PuntoIns(2) = Val(Y.Text)
    PuntoIns(3) = Val(Z.Text)
    TextoV = Texto.Text
    Select Case LCase(Color.Text)
        Case "porcapa"
            ColorV = 256
        Case "porbloque"
            ColorV = 0
        Case Else
            ColorV = Val(Color)
            If ColorV < 0 Or ColorV > 256 Then
                MsgBox ("Color no válido")
                Exit Sub
            End If
        End Select
    End Select
    CapaV = Capa.Text
    TipoLinV = UCase(TipoLin.Text)
    EstiloV = UCase(Estilo.Text)
    AlturaV = Val(Altura.Text)
    OblicuidadV = (Val(Oblicuidad.Text) * 2 * PI) / 360
    If AlDerecha.Value = True Then AlH = "acHorizontalAlignmentRight"
    If AlIzquierda.Value = True Then AlH = "acHorizontalAlignmentLeft"
    If AlCentro.Value = True Then AlH = "acHorizontalAlignmentCenter"
    If AlSuperior.Value = True Then AlV = "acVerticalAlignmentTop"
    If AlMitad.Value = True Then AlV = "acVerticalAlignmentMiddle"
    If AlLíneaBase.Value = True Then AlV = "acVerticalAlignmentBaseLine"
    If AlInferior.Value = True Then AlV = "acVerticalAlignmentBottom"

    Set ObjTexto = AcadModel.AddText(TextoV, PuntoIns, AlturaV)
    ObjTexto.Color = ColorV
    ObjTexto.Layer = CapaV
```



```
ObjTexto.Linetype = TipoLinV
ObjTexto.StyleName = EstiloV
ObjTexto.ObliqueAngle = OblicuidadV
ObjTexto.HorizontalAlignment = AlH
ObjTexto.VerticalAlignment = AlV
End
End Sub
```

```
Private Sub Cancelar_Click()
    End
End Sub
```

```
Private Sub UserForm_Initialize()
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument
    Set AcadModel = AcadDoc.ModelSpace

    PI = 3.14159
End Sub
```

Como vemos es bastante sencillo de entender, según lo estudiado ya acerca de los textos. Únicamente decir que estaría un poco incompleto, ya que sólo se controla —como ejemplo— que el usuario introduzca bien el número de color. Los demás controles (casillas vacías, estilos de texto existentes...) deberían introducirse para que el programa estuviera completo.

NOTA: La manera de controlar si un estilo de texto existe, entre otros muchos controles, se estudiará más adelante.

Veamos a continuación el cuadro diálogo perteneciente a este código:

Manejo de textos en AutoCAD 14

Punto de inserción

X: 25.5

Y: 56

Z: 0

Texto

Prueba de manejo de textos desde el módulo VBA de AutoCAD 14.

Propiedades generales

Color: PorCapa

Capa: Cajetín

Tipo de línea: CONTINUOUS

Propiedades de texto

Estilo: TEXTO_ES_1

Altura: 5

Oblicuidad: 15

Alineación horizontal

☒ Izquierda

☐ Derecha

☐ Centro

Alineación vertical

☐ Superior

☐ Mitad

☒ Línea base

☐ Inferior

Aceptar Cancelar

NOTA: Recordemos poner a True la propiedad MultiLine de la casilla de edición en la que introducimos el texto.

DOCE.5.7. Objetos de polilínea

DOCE.5.7.1. Polilíneas de antigua definición

Para crear polilíneas no optimizadas de versiones anteriores a la 14 de **AutoCAD** utilizamos el método AddPolyline. En la siguiente sección se estudian las polilíneas optimizadas de la **AutoCAD**.

La sintaxis del método AddPolyline es la siguiente:

```
Set ObjPolilínea = ObjColección.AddPolyline(DblMatrizVértices)
```

Propiedades de los objetos de polilínea de antigua definición:

Application	Area	Closed	Color
Coordinates	EntityName	EntityType	Handle
Layer	LineType	LinetypeScale	Normal
ObjectID	Thickness	Type	Visible

Métodos de los objetos de polilínea de antigua definición:

AppendVertex	ArrayPolar	ArrayRectangular	Copy
Erase	Explode	GetBoundingBox	GetBulge
GetWidth	GetXData	Highlight	IntersectWith
Mirror	Mirror3D	Move	Offset
Rotate	Rotate3D	ScaleEntity	SetBulge
SetWidth	SetXData	TransformBy	Update

A continuación pasaremos a comentar las nuevas propiedades aún no estudiadas.

- **Closed.** Obtiene o asigna el estado de apertura o cierre de una polilínea (o spline). Es un valor de tipo Boolean que puesto a True dice de la polilínea que está abierta; por el contrario a False, indica que la polilínea está cerrada. La sintaxis para asignar esta propiedad es:

```
ObjGráfico.Closed = BooCerrada
```

Y la sintaxis para obtener la propiedad:

```
BooCerrada = ObjGráfico.Closed
```

- **Type.** Obtiene y/o asigna el tipo de adaptación a curva o amoldamiento de superficie de una polilínea, malla o directriz. Para asignar un tipo:

```
ObjGráfico.Type = IntTipo
```

y para obtenerlo:

```
IntTipo = ObjGráfico.Type
```

IntTipo es un valor Integer y que, además, admite las siguientes constantes:

Para polilíneas:

Tipo	Descripción
acSimplePoly	Simple
acFitCurvePoly	Adaptada a curva
acQuadSplinePoly	B-spline cuadrática
acCubicSplinePoly	B-spline cúbica

Para mallas:

Tipo	Descripción
acSimpleMesh	Simple
acQuadSurfaceMesh	Amoldada a superficie B-spline cuadrática
acCubicSurfaceMesh	Amoldada a superficie B-spline cúbica
acBezierSurfaceMesh	Amoldada a superficie Bezier

Para directrices:

Tipo	Descripción
acLineNoArrow	Una línea sin flecha
acLineWithArrow	Una línea con flecha
acSplineNoArrow	Una spline sin flecha
acSplineWithArrow	Una spline con flecha

Y con respecto a los nuevos métodos expuestos, su explicación detallada se proporciona a continuación.

- **AppendVertex.** Se aplica exclusivamente a polilíneas 2D, 3D y mallas poligonales. Este método añade un vértice adicional al final de dichas entidades. En el caso de las mallas se debe añadir realmente toda una fila de vértices de la matriz $M \times N$ de la superficie. La sintaxis es:

`ObjGráfico.AppendVertex(DblPunto)`

siendo *DblPunto* un punto de tipo de dato Double.

- **Explode.** Descompone una entidad compuesta en las entidades simples que la forman. Estas entidades simples se obtienen en una matriz de objetos que habrá sido definida como Variant. Después, y como ya hemos hecho en otros casos, podemos acceder a dichos objetos especificando un índice para la matriz. Su sintaxis es:

`VarMatrizObjetos = ObjGráfico.Explode`

- **GetBulge.** Obtiene el valor de curvatura de un vértice de la polilínea especificado por su índice (se empieza a contar desde el 0, como primer vértice). Sintaxis:

`DblCurvatura = ObjPolilínea.GetBulge(IntÍndice)`

La curvatura ha de almacenarse en una variable Double; el índice es evidentemente de tipo entero (Integer).

- **GetWidth.** Obtiene el grosor inicial y final (Double) del tramo que comienza en el vértice especificado por su índice (Integer). La sintaxis de este método es un poco distinta, ya que se llama a él con **Call** y se pasan las variables como argumentos del método:

```
Call ObjPolilínea.GetWidth(IntÍndice, DblGrosorInic, DblGrosorFin)
```

- **SetBulge.** Inversamente a **GetBulge**, **SetBulge** asigna un valor de curvatura a un vértice. Los tipos de datos son los mismos que para aquel:

```
ObjPolilínea.GetBulge(IntÍndice, DblCurvatura)
```

- **SetWidth.** De manera contraria a **GetWidth**, **SetWidth** asigna el grosor inicial y final de un tramo especificado por su vértice inicial. Los tipos de datos los mismo que para **GetWidth**:

```
Call ObjPolilínea.SetWidth(IntÍndice, DblGrosorInic, DblGrosorFin)
```

DOCE.5.7.2. Polilíneas optimizadas

Para crear polilíneas optimizadas de **AutoCAD** se utiliza el método **AddLightWeightPolyline**, cuya sintaxis es la que sigue:

```
Set ObjPolilínea14 = ObjColección.AddLightWeightPolyline(DblMatrizVértices)
```

Propiedades de los objetos de polilínea optimizada:

Application	Area	Closed	Color
Coordinates	EntityName	EntityType	Handle
Layer	LineType	LinetypeScale	Normal
ObjectId	Thickness	Type	Visible

Métodos de los objetos de polilínea optimizada:

AddVertex	ArrayPolar	ArrayRectangular	Copy
Erase	Explode	GetBoundingBox	GetBulge
GetWidth	GetXData	Highlight	IntersectWith
Mirror	Mirror3D	Move	Offset
Rotate	Rotate3D	ScaleEntity	SetBulge
SetWidth	SetXData	TransformBy	Update

Las propiedades de estas polilíneas son exactamente las mismas que las de las anteriores. Con respecto a los métodos, se cambia **AppendVertex** por **AddVertex**.

- **AddVertex.** Se aplica únicamente a polilínea optimizadas. Añade un nuevo vértice a la entidad, que habrá de ser el punto final de un nuevo segmento de la polilínea. Se indica el índice (Integer) con la posición del vértice a continuación del cual se añadirá uno nuevo (Double):

```
ObjPolilínea14.AddVertex(IntÍndice, DblPunto)
```

NOTA: Para crear polilíneas con tramos curvos o en arco, se crea la polilínea con segmentos rectos y, a continuación, se añade la curvatura a los tramos que interese mediante el método **SetBulge**.

DOCE.5.8. Polilíneas 3D

Para crear polilíneas 3D recurrimos al método Add3DPoly, cuya sintaxis es:

```
Set ObjPolilínea3D = ObjColección.Add3DPoly(DblMatrizVértices)
```

Propiedades de los objetos de polilínea 3D:

Application	Closed	Color	Coordinates
EntityName	EntityType	Handle	Layer
LineType	LinetypeScale	ObjectID	Visible

Métodos de los objetos de polilínea 3D:

AppendVertex	ArrayPolar	ArrayRectangular	Copy
Erase	Explode	GetBoundingBox	GetXData
Highlight	IntersectWith	Mirror	Mirror3D
Move	Rotate	Rotate3D	ScaleEntity
SetXData	TransformBy	Update	

Las propiedades y métodos expuestos están ya estudiados.

DOCE.5.9. Rayos

El método de creación de rayos es AddRay:

```
Set ObjRayo = ObjColección.AddRay(DblPtoPrimero, DblPtoSegundo)
```

Propiedades de los objetos de rayo:

Application	BasePoint	Color	DirectionVector
EntityName	EntityType	Handle	Layer
LineType	LinetypeScale	ObjectID	Visible

Métodos de los objetos de rayo:

ArrayPolar	ArrayRectangular	Copy	Erase
GetBoundingBox	GetXData	Highlight	IntersectWith
Mirror	Mirror3D	Move	Rotate
Rotate3D	ScaleEntity	SetXData	TransformBy
Update			

Se explican las nuevas propiedades seguidamente.

- **BasePoint**. Asigna u obtiene el punto (Double) a través del cual el rayo (o la línea auxiliar) pasará. Para asignar un punto de base utilizaremos la sintaxis siguiente:

```
ObjGráfico.BasePoint = DblPunto
```

Para obtener el punto de base de un objeto rayo (o línea auxiliar) dibujado:

```
VarPunto = ObjGráfico.BasePoint
```

- **DirectionVector**. Especifica (obtiene y/o asigna) una dirección según un vector de dirección.

Para asignar:

```
ObjGráfico.DirectionVector = DblVector
```

Para obtener:

```
VarVector = ObjGráfico.DirectionVector
```

DblVector es una matriz de tres elementos Double.

DOCE.5.10. Líneas auxiliares

El método de creación de líneas auxiliares es AddXLine:

```
Set ObjLíneaAux = ObjColección.AddXLine(DblPtoPrimero, DblPtoSegundo)
```

Propiedades de los objetos de línea auxiliar:

Application	BasePoint	Color	DirectionVector
EntityName	EntityType	Handle	Layer
LineType	LinetypeScale	ObjectID	Visible

Métodos de los objetos de línea auxiliar:

ArrayPolar	ArrayRectangular	Copy	Erase
GetBoundingBox	GetXData	Highlight	IntersectWith
Mirror	Mirror3D	Move	Rotate
Rotate3D	ScaleEntity	SetXData	TransformBy
Update			

Las propiedades y métodos expuestos están ya estudiados.

DOCE.5.11. Trazos

El método de creación de trazos es AddTrace:

```
Set ObjTrazo = ObjColección.AddTrace(DblMatrizVértices)
```

Propiedades de los objetos de trazo:

Application	Color	Coordinates	EntityName
EntityType	Handle	Layer	LineType
LinetypeScale	Normal	ObjectID	Thickness
Visible			

Métodos de los objetos de trazo:

ArrayPolar	ArrayRectangular	Copy	Erase
GetBoundingBox	GetXData	Highlight	IntersectWith
Mirror	Mirror3D	Move	Rotate
Rotate3D	ScaleEntity	SetXData	TransformBy
Update			

Como vemos hemos de indicar una matriz Double con los valores de los vértices o puntos del trazo.

Las propiedades y métodos expuestos están ya estudiados.

DOCE.5.12. Splines

El método de creación de splines es AddSpline:

<code>Set ObjSpline = ObjColección.AddSpline(DblMatrizVértices, DblTanIni, DblTanFin)</code>

Propiedades de los objetos de spline:

Application	Area	Closed	Color
Degree	EndTangent	EntityName	EntityType
FitTolerance	Handle	IsRational	Layer
LineType	LinetypeScale	NumberOfControlPoints	NumberOfFitPoints
ObjectID	StartTangent	Visible	

Métodos de los objetos de spline:

AddFitPoint	ArrayPolar	ArrayRectangular	Copy
DeleteFitPoint	ElevateOrder	Erase	GetBoundingBox
GetControlPoint	GetFitPoint	GetWeight	GetXData
Highlight	IntersectWith	Mirror	Mirror3D
Move	Offset	PurgeFitData	Reverse
Rotate	Rotate3D	ScaleEntity	SetControlPoint
SetFitPoint	SetWeight	SetXData	TransformBy
Update			

Como decimos, la manera de añadir splines a un dibujo es mediante el método AddSpline, pasándole como argumentos una matriz Double con los puntos de cada vértice (cada uno es una matriz de tres elementos Double), la tangente de inicio, que es un vector de dirección representado por un *array* de tres valores Double, y la tangente final, siendo esta también otro vector de dirección similar al anterior.

Por ejemplo, un programa simple que dibujara una spline podría ser la siguiente macro:

```
Option Explicit
Dim AcadDoc As Object
Dim AcadModel As Object
Dim DibujoSpl As Object

Dim Vértices(1 To 3 * 3) As Double

Dim TanInicial(1 To 3) As Double
Dim TanFinal(1 To 3) As Double

Sub DibujaSpline()
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument
    Set AcadModel = AcadDoc.ModelSpace

    Vértices(1) = 0
    Vértices(2) = 0
    Vértices(3) = 0
    Vértices(4) = 50
    Vértices(5) = 50
    Vértices(6) = 0
    Vértices(7) = 100
    Vértices(8) = 0
```

```
Vértices(9) = 0
TanInicial(1) = 100: TanInicial(2) = 100: TanInicial(1) = 0
TanFinal(1) = 100: TanFinal(2) = 100: TanFinal(1) = 0
Set DibujoSpl = AcadModel.AddSpline(Vértices, TanInicial, TanFinal)
End Sub
```

Existen multitud de nuevas propiedades y de nuevos métodos que se pasarán a detallar a continuación. Comencemos por las propiedades.

- Degree. Permite obtener la graduación de la representación polinomial de una spline. Su sintaxis es:

```
IntValor = ObjSpline.Degree
```

El valor es de sólo lectura y ha de almacenarse en una variable declarada como Integer (únicamente puede ser un número entre 1 y 25, ambos inclusive).

- EndTangent. Esta propiedad permite asignar y/o extraer el valor de la tangente final de la spline, esto es, una matriz de tres elementos Double que es un vector de dirección.

La sintaxis para asignar una tangente final es la que sigue:

```
ObjSpline.EndTangent = DblVector
```

Para obtenerla:

```
VarVector = ObjSpline.EndTangent
```

- FitTolerance. Obtiene el ajuste de tolerancia, o reajusta esta propiedad de las splines. Como hemos de saber, si el valor de la tolerancia es cero, la spline está obligada a pasar por los vértices; si es mayor de cero, tiende a ellos pero no los toca. El valor de tolerancia no puede ser negativo.

La sintaxis para usar esta propiedad para asignar un valor de tolerancia a una spline es:

```
ObjSpline.FitTolerance = DblValorTolerancia
```

Para obtenerlo:

```
DblValorTolerancia = ObjSpline.FitTolerance
```

La variable que guarde este valor habrá de ser declarada como Double.

- IsRational. Obtiene la condición de racionalidad de la spline dada, es decir, verdadero (True) o falso (False).

La sintaxis es:

```
BooRacional = ObjSpline.IsRational
```

La variable que guarde este valor habrá de ser declarada como Boolean.

- NumberOfControlPoints. Propiedad de sólo lectura que devuelve el número de puntos de control de una spline:


```
IntNumCtrl = ObjSpline.NumberOfControlPoints
```

La variable que guarde este valor habrá de ser declarada como Integer.

Si en el ejemplo anterior añadimos las siguientes líneas:

```
Dim NumControl As Integer  
NumControl = DibujoSpl.NumberOfControlPoints  
MsgBox (Str(NumControl))
```

Aparecerá un cuadro de mensaje (MsgBox) con el número de puntos de control de nuestra spline, en este caso 5 (podemos comprobarlo ejecutando el comando de **AutoCAD** EDITSPLINE, SPLINEDIT en inglés, y designando la spline en cuestión).

- NumberOfFitPoints. Parecida a la propiedad anterior, ésta devuelve el número de puntos de ajuste de una spline, es decir, sus vértices:

```
IntNumAjst = ObjSpline.NumberOfFitPoints
```

La variable que guarde este valor habrá de ser declarada como Integer también.

Si en al ejemplo anterior añadiéramos también lo siguiente:

```
Dim NumAjuste As Integer  
NumAjuste = DibujoSpl.NumberOfFitPoints  
MsgBox (Str(NumAjuste))
```

Aparecerá otro cuadro de mensaje con el número de puntos de ajuste de nuestra spline, en este caso 3.

- StartTangent. Así como EndTangent trabaja con la tangente final, StartTangent permite asignar y/o extraer el valor de la tangente inicial de la spline, esto es, una matriz de tres elementos Double que es un vector de dirección.

La sintaxis para asignar una tangente inicial es la que sigue:

```
ObjSpline.StartTangent = DblVector
```

Para obtenerla:

```
VarVector = ObjSpline.StartTangent
```

Y hasta aquí las nuevas propiedades de las splines. Pasemos ahora a ver los métodos nuevos.

- AddFitPoint. Este método nos permite añadir un nuevo punto de ajuste a una spline ya creada. Para ello deberemos proporcionar un índice de posición del nuevo vértice y sus coordenadas. Veamos la sintaxis de este método:

```
Call ObjSpline.AddFitPoint(IntÍndice, DblPtoAjuste)
```

Donde *IntÍndice* es la posición (Integer) en la que se insertará el nuevo vértice y *DblPtoAjuste* las coordenadas del punto en cuestión (matriz de tres valores Double). Deberemos llamar al método con Call.

Así por ejemplo, para añadir un nuevo punto inicial (índice 1) a la spline de nuestro ejemplo, agregaríamos a la macro:

```
Dim NuevoPunto(1 To 3) As Double
NuevoPunto(1) = 25
NuevoPunto(2) = 25
NuevoPunto(3) = 0
Call DibujoSpl.AddFitPoint(1, NuevoPunto)
```

Lo que hará que se añada el nuevo punto. Dependiendo de la situación puede ser necesario actualizar la spline con su método `Update`.

- `DeleteFitPoint`. De manera contraria al método anterior, `DeleteFitPoint` elimina un punto de ajuste o vértice de una spline. Su sintaxis:

```
Call ObjSpline.DeleteFitPoint(IntÍndice)
```

Como vemos en la sintaxis, únicamente hay que proporcionar como argumento el índice correspondiente al vértice que deseamos eliminar, que será, si es una variable, de tipo `Integer`.

- `ElevateOrder`. Eleva el orden de un spline al orden proporcionado como argumento del método:

```
Call ObjSpline.ElevateOrder(IntÍndice)
```

El orden de una spline es igual a su graduación polinomial más uno, de ahí que este valor sólo pueda ser como mucho 26.

- `GetControlPoint`. Permite obtener las coordenadas del punto de control especificado por el índice que se proporciona como argumento. La sintaxis de uso es:

```
VarPuntoCtrl = ObjSpline.GetControlPoint(IntÍndice)
```

Donde `VarPuntoCtrl` es una variable definida como `Variant` donde se guardarán las coordenadas del punto en cuestión. Como sabemos ya, si queremos utilizarlas posteriormente deberemos realizar un trasvase a una matriz de tres elementos `Double`.

`IntÍndice` es el índice `Integer` del punto de control del que queremos obtener sus coordenadas.

- `GetFitPoint`. Las mismas consideraciones que para el método anterior pero a la hora de obtener las coordenadas de un punto de ajuste:

```
VarPuntoAjst = ObjSpline.GetFitPoint(IntÍndice)
```

- `GetWeight`. Este método devuelve el peso del punto de control de una spline especificado por su índice. La sintaxis de utilización es la que sigue:

```
DblPeso = ObjSpline.GetWeight(IntÍndice)
```

`DblPeso` es una variable declarada como `Double` que almacenará el valor del peso del punto. `IntÍndice` como en los métodos anteriores.

- `PurgeFitData`. Limpia los datos de ajuste de una spline. Su sintaxis:

```
Call ObjSpline.PurgeFitData
```

- **Reverse.** Invierte la dirección de la curva spline (el vértice inicial se convierte en el final):

```
Call ObjSpline.Reverse
```

- **SetControlPoint.** Permite asignar coordenadas al punto de control especificado por el índice que se proporciona como argumento. La sintaxis de uso es:

```
Call ObjSpline.SetControlPoint(IntÍndice, DblCoordenadas)
```

Donde *IntÍndice* (Integer) es el índice que determina el punto de control y *DblCoordenadas* el array de tres elementos Double, que son las coordenadas del punto mencionado.

- **SetFitPoint.** Las mismas consideraciones que para el método anterior pero a la hora de asignar las coordenadas a un punto de ajuste:

```
Call ObjSpline.SetFitPoint(IntÍndice, DblCoordenadas)
```

- **SetWeight.** Este método asigna un peso al punto del control de una spline especificado por su índice. La sintaxis de utilización es la que sigue:

```
Call ObjSpline.SetWeight(IntÍndice, DblPeso)
```

IntÍndice es Integer; *DblPeso* es Double.

DOCE.5.13. Texto múltiple

Para añadir texto múltiple a la colección de objetos de Espacio Modelo, Espacio Papel o bloques, disponemos del método **AddMText**, cuya sintaxis es:

```
Set ObjTextoM = ObjColección.AddMText(DblPtoIns, DblAnchura, StrTexto)
```

Propiedades de los objetos de texto múltiple:

Application	AttachmentPoint	Color	DrawingDirection
EntityName	EntityType	Handle	Height
InsertionPoint	Layer	LineType	LinetypeScale
Normal	ObjectID	Rotation	StyleName
TextString	Visible	Width	

Métodos de los objetos de texto múltiple:

ArrayPolar	ArrayRectangular	Copy	Erase
GetBoundingBox	GetXData	Highlight	IntersectWith
Mirror	Mirror3D	Move	Rotate
Rotate3D	ScaleEntity	SetXData	TransformBy
Update			

Así como en el dibujo de textos en una línea, *DblPtoIns* es una matriz o tabla de tres elementos Double que son las coordenadas del punto de inserción, y *StrTexto* es la cadena literal de texto (String) que se imprimirá en pantalla. *DblAnchura* se refiere a la anchura de la caja de abarque del texto múltiple (es valor Double).

Podemos apreciar que los textos múltiples disponen de tres nuevas propiedades aún no vistas. Las vemos a continuación (los métodos están todos estudiados).

- **AttachmentPoint.** Esta propiedad, exclusiva de textos múltiples, permite asignar u obtener el tipo de justificación de un texto múltiple con respecto a su caja de abarque o rectángulo de reborde. Su sintaxis a la hora de asignar es:

```
ObjTextoM.AttachmentPoint = IntTipoJus
```

Para obtener:

```
IntTipoJus = ObjTextoM.AttachmentPoint
```

IntTipoJus es un valor Integer que además puede albergar las siguientes constantes:

acAttachmentPointTopLeft	acAttachmentPointTopCenter
acAttachmentPointTopRight	acAttachmentPointMidLeft
acAttachmentPointMidCenter	acAttachmentPointMidRight
acAttachmentPointBottomLeft	acAttachmentPointBottomCenter
acAttachmentPointBottomRight	

Refiriéndose todas ellas a las distintas justificaciones posibles del texto múltiple.

- **DrawingDirection.** Esta propiedad, también exclusiva de textos múltiples, permite asignar u obtener el tipo de orientación de un texto múltiple. Su sintaxis para asignar es:

```
ObjTextoM.DrawingDirection = IntTipoOr
```

Para obtener:

```
IntTipoOr = ObjTextoM.DrawingDirection
```

IntTipoOr es un valor Integer que puede albergar las siguientes constantes:

acLeftToRight	acRightToLeft	acTopToBottom	acBottomToTop
---------------	---------------	---------------	---------------

- **Width.** Dice referencia a la anchura del rectángulo de abarque de un objeto. Su sintaxis para asignar una anchura es:

```
ObjGráfico.Width = DblAnchura
```

La sintaxis para obtener una anchura es:

```
DblAnchura = ObjGráfico.Width
```

DOCE.5.14. Regiones

El método para la obtención de regiones es `AddRegion` y su sintaxis es:

```
Set ObjRegión = ObjColección.AddRegion(ObjListaObjetos)
```

Propiedades de los objetos de región:

Application	Area	Centroid	Color
EntityName	EntityType	Handle	Layer

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en Visual Basic orientada a AutoCAD (VBA)

LineType	LinetypeScale	MomentOfInertia	Normal
ObjectID	Perimeter	PrincipalDirections	PrincipalMoments
ProductOfInertia	RadiiOfGyration	Visible	

Métodos de los objetos de región:

ArrayPolar	ArrayRectangular	Boolean	Copy
Erase	Explode	GetBoundingBox	GetXData
Highlight	IntersectWith	Mirror	Mirror3D
Move	Rotate	Rotate3D	ScaleEntity
SetXData	TransformBy	Update	

Este método crea regiones a partir de una lista de objetos. Los objetos de esta lista deberán formar un área coplanar cerrada y sólo podrán ser líneas, arcos, círculos, arcos elípticos, polilíneas y/o splines.

La lista de objetos será una matriz o *array* declarada como *Object* y que contendrá todas la entidades gráficas a partir de las cuales se dibujará la región.

Veamos las propiedades nuevas:

- **Centroid.** Esta propiedad obtiene el centroide, es decir, el centro del área (o del volumen) de una región (o sólido). Su sintaxis es:

```
VarCentroide = ObjGráfico.Centroid
```

donde *VarCentroide* es una variable *Variant* que guardará las coordenadas del punto en cuestión. Para procesarlas —y como ya sabemos— deberemos realizar un trasvase, ya que así no podemos utilizarlas.

- **MomentOfInertia.** Esta propiedad obtiene el momento de inercia de una región (o sólido) y lo devuelve como tres coordenadas X, Y y Z. Su sintaxis es:

```
VarMomentoInercia = ObjGráfico.MomentOfInertia
```

La variable que habrá de guardarlo será también declarada como *Variant*, al igual que en el caso de la propiedad anterior.

- **Perimeter.** Esta propiedad (también de sólo lectura, como las dos anteriores) obtiene el perímetro de una región exclusivamente. Su sintaxis es:

```
DblPerímetro = ObjRegión.Perimeter
```

La variable que habrá de guardarlo será declarada como *Double*.

- **PrincipalDirections.** Permite obtener las principales direcciones (en X, Y y Z) de una región (o sólido). Su sintaxis es:

```
VarDireccionesP = ObjGráfico.PrincipalDirections
```

La variable que habrá de guardarlo será declarada como *Variant*.

- **PrincipalMoments.** Permite obtener los principales momentos (en X, Y y Z) de una región (o sólido). Su sintaxis es:

```
VarMomentosP = ObjGráfico.PrincipalMoments
```

La variable que habrá de guardarlo será declarada como Variant.

- **ProductOfInertia**. Devuelve el producto de inercias (en X, Y y Z) de una región (o sólido). Su sintaxis es:

```
VarProductoInercias = ObjGráfico.ProductOfInertia
```

La variable que habrá de guardarlo será declarada como Variant.

- **RadiiOfGyration**. Permite obtener el radio de giro (en X, Y y Z) de una región (o sólido). Su sintaxis es:

```
VarRadioGiro = ObjGráfico.RadiiOfGyration
```

La variable que habrá de guardarlo será declarada como Variant.

NOTA: Todos los cálculos que se realizan al utilizar estas propiedades se efectúan en el sistema de coordenadas actual.

Veamos ahora el nuevo método.

- **Boolean**. Permite realizar una operación booleana (unión, diferencia o intersección) entre el objeto al cual se le aplica y otro objeto, sea sólido o región.

Su sintaxis es la que sigue:

```
ObjGráfico3 = ObjGráfico1.Boolean(LngOperación, ObjGráfico2)
```

LngOperación es un valor tipo Long que indica la operación que se realizará entre ambos sólidos. Admite las siguientes constantes:

acUnion acIntersection acSubtraction

ObjGráfico2 es la entidad gráfica de **AutoCAD** (región o sólido) existente con la cual se realizará la operación booleana con el otro objeto (*ObjGráfico1*). Ambos objetos deberán estar ya creado, evidentemente.

ObjGráfico3 es el objeto gráfico resultado. Será una variable declarada como Object.

DOCE.5.15. Sólidos 3D

La obtención de sólidos primitivos de **AutoCAD** requiere más de un método para su ejecución. Veremos aquí todos y cada uno de ellos, además de sus propiedades y métodos, que son comunes a todos ellos.

Las diferentes sintaxis de creación de sólidos, con su explicación pertinente, se exponen a continuación.

DOCE.5.15.1. Prisma rectangular

```
Set ObjPrismaRect = ObjColección.AddBox(DblPtoOrg, DblLargo, DblAncho, DblAlto)
```

Los argumentos para este método son el punto de origen (*DblPtoOrg*), el cual es el centro de la caja de abarque, no una esquina (es una matriz de tres elementos *Double*); la dimensión de longitud (*Double*); la dimensión de anchura (*Double*); y la dimensión de altura (*Double*).

DOCE.5.15.2. Cono

```
Set ObjCono = ObjColección.AddCone(DblPtoCentro, DblAltura, DblRadioBase)
```

Los argumentos para este método son el centro, el cual es el centro de la caja de abarque, no el de la base (es una matriz de tres elementos *Double*); la altura del cono (*Double*); y el radio de la base (*Double*).

DOCE.5.15.3. Cilindro

```
Set ObjCilindro = ObjColección.AddCylinder(DblPtoCentro, DblAltura, DblRadio)
```

Los argumentos para este método son el centro, el cual es el centro de la caja de abarque, no el de la base (es una matriz de tres elementos *Double*); la altura del cilindro (*Double*); y el radio (*Double*).

DOCE.5.15.4. Cono elíptico

```
Set ObjConElp = ObjColección.AddEllipticalCone(DblPtoCentro, DblRadioMayor, DblRadioMenor, DblAltura)
```

Los argumentos para este método son el centro, el cual es el centro de la caja de abarque, no el de la base (es una matriz de tres elementos *Double*); el radio mayor de la elipse de la base (*Double*); el radio menor de la elipse de la base (*Double*); y la altura del cono elíptico (*Double*).

DOCE.5.15.5. Cilindro elíptico

```
Set ObjCilElp = ObjColección.AddEllipticalCylinder(DblPtoCentro, DblRadioMayor, DblRadioMenor, DblAltura)
```

Los argumentos para este método son el centro, el cual es el centro de la caja de abarque, no el de la base (es una matriz de tres elementos *Double*); el radio mayor de la elipse de la base (*Double*); el radio menor de la elipse de la base (*Double*); y la altura del cilindro elíptico (*Double*).

DOCE.5.15.6. Esfera

```
Set ObjEsfera = ObjColección.AddSphere(DblPtoCentro, DblRadio)
```

Los argumentos para este método son el centro, el cual es el centro de la caja de abarque, o sea el propio centro de la esfera (es una matriz de tres elementos *Double*); y el

radio de la esfera (Double).

DOCE.5.15.7. Toroide

```
Set ObjToroide = ObjColección.AddTorus(DblPtoCentro, DblRadToro, DblRadTub)
```

Los argumentos para este método son el centro, el cual es el centro de la caja de abarque, o sea el propio centro del toroide o toro (es una matriz de tres elementos Double); el radio de toroide (Double); y el radio del tubo (Double).

DOCE.5.15.8. Cuña

```
Set ObjCuña = ObjColección.AddWedge(DblPtoCentro, DblLargo, DblAncho, DblAlto)
```

Los argumentos para este método son el centro, el cual es el centro de la caja de abarque, no una esquina ni el centro de una cara (es una matriz de tres elementos Double); la dimensión de longitud (Double); la dimensión de anchura (Double); y la dimensión de altura (Double).

DOCE.5.15.9. Extrusión

```
Set ObjExtrusión = ObjColección.AddExtrudedSolid(ObjPerfil, DblAltura, DblÁngulo)
```

Con este método podemos crear extrusiones de cualquier perfil que cumpla las normas necesarias para poder ser perfil de extrusión (igual que desde la interfaz de **AutoCAD**). La única excepción quizá es que dicho perfil habrá de ser obligatoriamente una región, no es lícito desde VBA utilizar una polilínea. Por ello todos los perfiles originales tendrán información de superficie, lo que hará que **AutoCAD** cree sólidos cerrados (con "tapa").

ObjPerfil se refiere al objeto (Object) a partir del cual se creará la extrusión (una región, como hemos dicho); *DblAltura* es la altura de extrusión a lo largo del eje Z (Double); *DblÁngulo* es el ángulo de salida de la extrusión en radianes (Double).

DOCE.5.15.10. Extrusión con camino

```
Set ObjExtrusiónC = ObjColección.AddExtrudedSolidAlongPath(ObjPerfil, ObjCamino)
```

Con este método podemos crear extrusiones a lo largo de un camino de extrusión. *ObjPerfil* es el objeto (tipo Object) que será el perfil que se extruirá (será una región); *ObjCamino* es otro objeto (tipo Object también) que es el camino de extrusión (será una polilínea, un círculo, una elipse, una spline o un arco, exclusivamente).

DOCE.5.15.11. Revolución

```
Set ObjRevolución = ObjColección.AddRevolvedSolid(ObjPerfil, DblPtoInicial, DblDirección, DblÁngulo)
```


Con el método `AddRevolve` creamos objetos de revolución a partir de un perfil de revolución. Se pasan como argumentos el objeto (`Object`), que será perfil (sólo una región); el punto inicial del eje de revolución (matriz de tres elementos `Double`); la dirección del eje de revolución (matriz de tres elementos `Double`); y el número de ángulos cubiertos en radianes (`Double`).

DOCE.5.15.12. Propiedades y métodos de los sólidos 3D

Como ya hemos dicho las propiedades y los métodos son comunes a todos los mecanismos que acabamos de estudiar. Se detallan a continuación.

Propiedades de los objetos de sólido 3D:

<code>Application</code>	<code>Centroid</code>	<code>Color</code>	<code>EntityName</code>
<code>EntityType</code>	<code>Handle</code>	<code>Layer</code>	<code>LineType</code>
<code>LinetypeScale</code>	<code>MomentOfInertia</code>	<code>ObjectID</code>	<code>PrincipalDirections</code>
<code>PrincipalMoments</code>	<code>ProductOfInertia</code>	<code>RadiiOfGyration</code>	<code>Visible</code>
<code>Volume</code>			

Métodos de los objetos de sólido 3D:

<code>ArrayPolar</code>	<code>ArrayRectangular</code>	<code>Boolean</code>	<code>CheckInterference</code>
<code>Copy</code>	<code>Erase</code>	<code>GetBoundingBox</code>	<code>GetXData</code>
<code>Highlight</code>	<code>IntersectWith</code>	<code>Mirror</code>	<code>Mirror3D</code>
<code>Move</code>	<code>Rotate</code>	<code>Rotate3D</code>	<code>ScaleEntity</code>
<code>SectionSolid</code>	<code>SetXData</code>	<code>SliceSolid</code>	<code>TransformBy</code>
<code>Update</code>			

Pasaremos ahora a comentar las que aún no conocemos. Primero la nueva propiedad `Volume`.

- `Volume`. Esta propiedad de sólo lectura y exclusiva de los sólidos permite extraer el volumen de uno de ellos (tipo `Double`). Para esto utilizamos la sintaxis siguiente:

```
DblVolumen = ObjSólido.Volume
```

Veamos ahora los nuevos métodos `CheckInterference`, `SectionSolid` y `SliceSolid`.

- `CheckInterference`. Con este método disponemos de la posibilidad de averiguar si existe interferencia entre dos sólidos y, si así fuera y lo especificáramos, crear un nuevo sólido resultante. La sintaxis para su uso es:

```
ObjSólido3 = ObjSólido1.CheckInterference(ObjSólido2, BooCrearNuevo)
```

`ObjSólido2` es la entidad (tipo `Object`) con que se quiere comprobar la interferencia con `ObjSólido1`. `BooCrearNuevo` es un valor booleano (tipo `Boolean`), esto es `True` (verdadero) o `False` (falso), que especifica si se creará un sólido con la interferencia (si existiera). `ObjSólido3` es el sólido resultante (tipo `Object`) si hubiera interferencia y `BooCrearNuevo` estuviera puesto a `True`.

- `SectionSolid`. Crea un región, que es la sección del sólido al que se le aplica, determinada por un plano definido por tres puntos que se proporcionan como argumentos.

```
ObjSólido2 = ObjSólido1.SectionSolid(DblPto1, DblPto2, DblPto3)
```

ObjSólido2 es el objeto (tipo `Object`) resultante que será un región; *DblPto1*, *DblPto2* y *DblPto3* son los tres puntos (matriz de tres valores tipo `Double`) que definen el plano de sección.

- `SliceSolid`. Produce un corte, en el sólido al que se aplica, mediante un plano definido por tres puntos.

```
ObjSólido2 = ObjSólido1.SliceSolid(DblPto1, DblPto2, DblPto3, BooNeg)
```

ObjSólido2 es el objeto (tipo `Object`) resultante que será un sólido; *DblPto1*, *DblPto2* y *DblPto3* son los tres puntos (matriz de tres valores tipo `Double` cada uno) que definen el plano de corte; *BooNeg* es una valor tipo `Boolean` (`True` o `False`) que determina si se devuelve o no el sólido en la parte negativa del plano, o sea, que si se conserva la parte cortada en la zona negativa del plano o no. `True` conserva el sólido completo, pero cortado; `False` elimina dicha parte.

DOCE.5.16. Caras 3D

El método del que disponemos para la creación de caras 3D es `Add3DFace` y su sintaxis es la sigue a continuación:

```
Set ObjCara3D = ObjColección.Add3DFace(DblPto1, DblPto2, DblPto3, DblPto4)
```

Propiedades de los objetos de cara 3D:

<code>Application</code>	<code>Color</code>	<code>EntityName</code>	<code>EntityType</code>
<code>Handle</code>	<code>Layer</code>	<code>LineType</code>	<code>LinetypeScale</code>
<code>ObjectID</code>	<code>Visible</code>		

Métodos de los objetos de cara 3D:

<code>ArrayPolar</code>	<code>ArrayRectangular</code>	<code>Copy</code>	<code>Erase</code>
<code>GetBoundingBox</code>	<code>GetInvisibleEdge</code>	<code>GetXData</code>	<code>Highlight</code>
<code>IntersectWith</code>	<code>Mirror</code>	<code>Mirror3D</code>	<code>Move</code>
<code>Rotate</code>	<code>Rotate3D</code>	<code>ScaleEntity</code>	<code>SetInvisibleEdge</code>
<code>SetXData</code>	<code>TransformBy</code>	<code>Update</code>	

Al método `Add3DFace` debemos pasarle como argumentos los cuatro puntos de una cara 3D. Todos estos puntos serán *arrays* de tres elementos tipo `Double` cada uno. El último punto (*DblPto4*) es opcional, si no se indica se crea una cara 3D de tres aristas.

Veamos el nuevo par de métodos.

- `GetInvisibleEdge`. Devuelve el estado de visibilidad de la arista de la cara 3D especificada por su índice. Su sintaxis es la siguiente:

```
BooEstado = ObjCara3D.GetInvisibleEdge(IntÍndice)
```

La variable que almacenará el estado (*BooEstado*) será declarada como booleana (`Boolean`). Si recoge un valor `True` la arista es invisible; si recoge un valor `False` la arista es visible.

IntÍndice es el número de orden de la arista que se desea comprobar de la cara 3D. Ha de ser un valor entre 0 y 3, siendo 0 la primera arista y 3 la cuarta, si la hubiera.

- *SetInvisibleEdge*. Asigna un estado de visibilidad para una arista de una cara 3D, especificada por su índice. Su sintaxis es:

`Call ObjCara3D.SetInvisibleEdge(IntÍndice, BooEstado)`

El significado y tipo de dato para *IntÍndice* y *BooEstado* son los mismos que en el método anterior.

DOCE.5.17. Mallas poligonales

La sintaxis del método para dibujar mallas poligonales es:

`Set ObjMalla3D = ObjColección.Add3DMesh(IntM, IntN, DblMatrizPuntos)`

Propiedades de los objetos de malla poligonal:

Application	Color	Coordinates	EntityName
EntityType	Handle	Layer	LineType
LinetypeScale	MClose	MDensity	MVertexCount
NClose	NDensity	NVertexCount	ObjectID
Type	Visible		

Métodos de los objetos de malla poligonal:

AppendVertex	ArrayPolar	ArrayRectangular	Copy
Erase	Explode	GetBoundingBox	GetXData
Highlight	IntersectWith	Mirror	Mirror3D
Move	Rotate	Rotate3D	ScaleEntity
SetXData	TransformBy	Update	

Add3DMesh crea una malla tridimensional indicándole el número de vértices en las direcciones *M* y *N* (con el mismo sentido que desde la interfaz gráfica de **AutoCAD**), así como una matriz *M´N* de los puntos en cuestión. Los valores de *M* y *N* son valores enteros (si son variables serán declaradas como *Integer*) entre 2 y 256. La matriz de vértices contendrá todos los puntos, los cuales requieren tres coordenadas tipo *Double*. Veamos un ejemplo (macro):

```
Option Explicit
Dim AcadDoc As Object
Dim AcadModel As Object
```

`Dim Vértices(1 To 3 * 3) As Double`

```
Sub MallaDib()
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument
    Set AcadPapel = AcadDoc.PaperSpace
```

```
    Dim M As Integer, N As Integer
    Dim Malla as Object
    Dim Total As Integer
    M = 2
    N = 2
    Total = M * N
```

```
Dim Vértices(1 To Total * 3) As Double
```

```
Vértices(1) = 0  
Vértices(2) = 0  
Vértices(3) = 0  
Vértices(4) = 20  
Vértices(5) = 0  
Vértices(6) = 0  
Vértices(7) = 0  
Vértices (8) = 20  
Vértices (9) = 0  
Vértices (10) = 20  
Vértices (11) = 20  
Vértices (12) = 0
```

```
Set Malla = AcadPapel.Add3dMesh(M, N, Vértices)  
End Sub
```

Pasemos a la explicación pertinente de propiedades y métodos nuevos. Como siempre, primero las propiedades.

- **MClose**. Esta propiedad permite asignar u obtener la verificación de cerramiento de la malla en la dirección *M*.

La sintaxis que usaremos para asignar un cerramiento o apertura en esta dirección es:

```
ObjMalla3D.MClose = BooVerCerramiento
```

Y la sintaxis para obtener la propiedad:

```
BooVerCerramiento = ObjMalla3D.MClose
```

Siendo *BooVerCerramiento* una variable declarada como **Boolean**, la cual si guarda **True** significa que la malla está cerrada en *M*; si guarda **False** significa que está abierta en dicha dirección.

- **MDensity**. Esta propiedad permite asignar u obtener la densidad de la malla en la dirección *M*. La densidad es el número de vértices (en este caso en *M*) que posee la malla después de una operación de ajuste de la misma.

La sintaxis que usaremos para asignar una densidad en esta dirección es:

```
ObjMalla3D.MDensity = IntDensidad
```

Y la sintaxis para obtener la propiedad:

```
IntDensidad = ObjMalla3D.MDensity
```

Siendo *IntDensidad* una variable declarada como **Integer**. Por defecto este valor es igual a 6.

- **MVertexCount**. Esto es el número de vértices de una malla poligonal en la dirección *M* cuando la propiedad **Type** es igual a **acSimpleMesh** (véase la sección **DOCE.5.7.1.**, en el lugar en el que se muestran los valores de **Type** para las mallas).

Y la sintaxis que utilizaremos para obtener este valor de sólo lectura es:

```
IntNumVert = ObjMalla3D.MVertexCount
```

Siendo *IntNumVert* una variable declarada como *Integer*. Este valor sólo puede encontrarse entre 2 y 256.

- **NClose**. Lo mismo que para **MClose** pero en dirección *N*.
- **NDensity**. Lo mismo que para **MDensity** pero en dirección *N*.
- **NVertexCount**. Lo mismo que para **MVertexCount** pero en dirección *N*.

NOTA: Véase la sección **DOCE.5.7.1.**, donde se comenta la propiedad *Type* de las polilíneas, para observar los valores de esta propiedad en las mallas.

DOCE.5.18. Imágenes de trama

La sintaxis del método para insertar imágenes de trama (imágenes *raster*) es:

```
Set ObjImgTrama = ObjColección.AddRaster(StrNombArch, DblPtoIns, DblFacEscala, DblAngRotación)
```

Propiedades de los objetos de imagen de trama:

Application	Brightness	ClippingEnabled	Color
Contrast	EntityName	EntityType	Fade
Handle	Height	ImageFile	ImageVisibility
Layer	LineType	LinetypeScale	ObjectID
Origin	Transparency	Visible	Width

Métodos de los objetos de imagen de trama:

ArrayPolar	ArrayRectangular	ClipBoundary	Copy
Erase	GetBoundingBox	GetXData	Highlight
IntersectWith	Mirror	Mirror3D	Move
Rotate	Rotate3D	ScaleEntity	SetXData
TransformBy	Update		

Para poder introducir una imagen *raster* desde VBA necesitamos proporcionar el nombre y ruta completa del archivo de imagen (*StrNombArch*), que si es variable será del tipo *String*; el punto de inserción (matriz de tres elementos *Double*); el factor de escala (*Double*), cuyo valor por defecto es 1, y habrá de ser siempre positivo; y el ángulo de rotación en radianes (*Double*).

Los tipos de formatos gráficos admitidos son los mismos que en la inserción de imágenes de trama desde la interfaz gráfica.

Veamos las nuevas propiedades.

- **Brightness**. Establece u obtiene el brillo de una imagen. Este valor es *Integer* y debe estar comprendido entre 0 y 100. Por defecto vale 50.

La sintaxis que usaremos para asignar brillo es:

```
ObjRaster.Brightness = IntBrillo
```

Y la sintaxis para obtener el brillo:

```
IntBrillo = ObjRaster.Brightness
```

- **ClippingEnabled.** Habilita o no un contorno delimitador creado con el método **ClipBoundary** (que veremos luego). También se puede utilizar para devolver un valor booleano que nos informe de si cierta imagen posee o no esta propiedad activada.

La sintaxis que usaremos para habilitar o no el contorno delimitador es:

```
ObjRaster.ClippingEnabled = BooEstadoMarco
```

Si hacemos esta propiedad igual a **True** el contorno se habilita; con **False** se deshabilita.

Y la sintaxis para obtener es:

```
BooEstadoMarco = ObjRaster.ClippingEnabled
```

BooEstadoMarco habrá sido definida como **Boolean**.

- **Contrast.** Establece u obtiene el contraste de la imagen. Este valor **Integer** habrá de estar entre 0 y 100. Por defecto es 50.

La sintaxis que usaremos para asignar contraste es:

```
ObjRaster.Contrast = IntContraste
```

Y la sintaxis para obtener el contraste:

```
IntContraste = ObjRaster.Contrast
```

- **Fade.** Establece u obtiene el difuminado de la imagen. Este valor **Integer** habrá de estar entre 0 y 100. Por defecto es 0.

La sintaxis que usaremos para asignar difuminado es:

```
ObjRaster.Fade = IntDifuminado
```

Y la sintaxis para obtener el difuminado existente en una imagen:

```
IntDifuminado = ObjRaster.Fade
```

- **ImageFile.** Establece u obtiene el archivo de imagen de la inserción *raster*.

La sintaxis que usaremos para asignar un nombre es:

```
ObjRaster.ImageFile = StrNombArch
```

Y la sintaxis para obtener el nombre de archivo de una imagen:

```
StrNombArch = ObjRaster.ImageFile
```

StrNombArch es una cadena declarada como **String**.

- `ImageVisibility`. Establece u obtiene si la imagen es visible o no.

La sintaxis que usaremos para asignar la condición de visibilidad:

```
ObjRaster.ImageVisibility = BooVisibilidad
```

Y la sintaxis para obtener la condición de visibilidad:

```
BooVisibilidad = ObjRaster.ImageVisibility
```

La variable que guarde este valor estará declarada como `Boolean`. Esta variable es booleana y además puede contener las siguiente constantes:

`acOn` `acOff`

- `Origin`. Establece u obtiene el punto origen de un objeto (otros además de imágenes de trama) en coordenadas del SCU.

Utilizaremos la siguiente sintaxis para asignar un punto de origen:

```
ObjGráfico.Origin = DblPtoOrigen
```

Y podemos utilizar la siguiente para obtener el punto origen de un objeto que admita esta propiedad:

```
VarPtoOrigen = ObjGráfico.Origin
```

La variable que guarde este valor guardará un punto, es decir, será una matriz de tres elementos `Double` para asignar o un variable `Variant` para recoger.

- `Transparency`. Establece u obtiene la condición de transparencia para imágenes bitonales.

Para establecer o asignar:

```
ObjRaster.Trasparency = BooCondiciónTransp
```

Para obtener:

```
BooCondiciónTrans = ObjRaster.Transparency
```

`BooCondiciónTrans` es una variable del tipo `Boolean` que puede adquirir los siguientes valores constantes:

`acOn` `acOff`

El único método nuevo se detalla ahora:

- `ClipBoundary`. Establece una serie de puntos que determinan un contorno delimitador para la imagen de trama. Su sintaxis es:

```
Call ObjRaster.ClipBoundary(DblMatrizPtos)
```

donde `DblMatrizPtos` contiene los diferentes puntos (*array* de tres elementos `Double`).

NOTA: Recordemos que para hacer efectivo este contorno deberemos poner a True la propiedad ClippingEnabled del objeto de imagen de trama en cuestión.

DOCE.5.19. Sólidos 2D

Para crear sólidos 2D, esto es, polígonos de relleno sólido (comando SOLIDO, SOLID en inglés, de **AutoCAD**), el método de las colecciones de Espacio Modelo, Papel o de bloques que utilizamos es AddSolid.

La sintaxis del método AddSolid es la que sigue:

```
Set ObjSólido2D = ObjColección.AddSolid(DblPto1, DblPto2, DblPto3, DblPto4)
```

Propiedades de los objetos de sólido 2D:

Application	Color	Coordinates	EntityName
EntityType	Handle	Layer	LineType
LinetypeScale	Normal	ObjectID	Thickness
Visible			

Métodos de los objetos de sólido 2D:

ArrayPolar	ArrayRectangular	Copy	Erase
GetBoundingBox	GetXData	Highlight	IntersectWith
Mirror	Mirror3D	Move	Rotate
Rotate3D	ScaleEntity	SetXData	TransformBy
Update			

Las propiedades y métodos expuestos están ya estudiados.

DOCE.5.20. Formas

El siguiente método permite añadir formas al dibujo. Para ello el archivo de formas contenedor correspondiente .SHX habrá de estar cargado, lo que aprenderemos a hacer más adelante.

La sintaxis del método AddShape es:

```
Set ObjForma = ObjColección.AddShape(StrNombre, DblPtoIns, DblFacEscal, DblRot)
```

Propiedades de los objetos de forma:

Application	Color	EntityName	EntityType
Handle	Height	InsertionPoint	Layer
LineType	LinetypeScale	Name	Normal
ObjectID	ObliqueAngle	Rotation	ScaleFactor
Thickness	Visible		

Métodos de los objetos de forma:

ArrayPolar	ArrayRectangular	Copy	Erase
GetBoundingBox	GetXData	Highlight	IntersectWith
Mirror	Mirror3D	Move	Rotate
Rotate3D	ScaleEntity	SetXData	TransformBy
Update			

Veremos la única propiedad nueva.

- **Name.** Obtiene y/o asigna el nombre del objeto en cuestión, en el caso que nos ocupa el de la forma (también en el de documento activo, aplicación... que veremos más adelante).

La sintaxis para asignar es:

```
Objeto.Name = StrNombre
```

La sintaxis para obtener es:

```
StrNombre = Objeto.Name
```

Ejemplo:

Option Explicit

```
Dim AcadDoc As Object  
Dim AcadModel As Object
```

```
Dim Forma As Object  
Dim PtoIns(1 To 3) As Double
```

```
Sub Macro()  
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument  
    Set AcadModel = AcadDoc.ModelSpace  
  
    PtoIns(1) = 100: PtoIns(2) = 100: PtoIns(3) = 0  
    Set Forma = AcadModel.AddShape("box", PtoIns, 1, 0)  
    MsgBox Forma.Name  
End Sub
```

Esta macro inserta una forma llamada BOX (del archivo LTYPE\$HP.SHX) en el punto 100,100,0 del Espacio Modelo del dibujo actual. A continuación muestra en un cuadro de mensaje el nombre (extraído con Name) de la forma insertada (Forma), que en este caso será BOX. Como ya hemos comentado, el archivo LTYPE\$HP.SHX —en este caso— deberá estar cargado.

DOCE.5.21. Acotación, directrices y tolerancias

A continuación estudiaremos los diversos métodos de los que disponemos para agregar cotas a nuestros dibujos mediante programación VBA. Así también veremos dos aspectos muy relacionados con la acotación, esto es, la directriz y la tolerancia. Como viene siendo habitual también se expondrán, y estudiarán en su caso, las diferentes propiedades y los diferentes métodos de cada elemento de **AutoCAD**.

DOCE.5.21.1. Cotas alineadas

El método de adición de cotas alineadas es AddDimAligned:

```
Set ObjCotaAlineada = ObjColección.AddDimAligned(DblPtoLinExt1, DblPtoLinExt2,  
DblPosTexto)
```

Propiedades de los objetos de cota alineada:

Application	Color	EntityName	EntityType
ExtLine1Point	ExtLine2Point	Handle	Layer
LineType	LinetypeScale	Normal	ObjectID
Rotation	StyleName	TextPosition	TextRotation
TextString	Visible		

Métodos de los objetos de cota alineada:

ArrayPolar	ArrayRectangular	Copy	Erase
GetBoundingBox	GetXData	Highlight	IntersectWith
Mirror	Mirror3D	Move	Rotate
Rotate3D	ScaleEntity	SetXData	TransformBy
Update			

En las cotas alineadas, como debemos saber, la línea de cota es paralela a la línea que une los puntos de origen de las líneas de extensión o auxiliares. Las líneas de extensión comienzan en dos puntos (Double) que son los que indicaremos en la sintaxis de `AddDimAligned` como dos primeros argumentos. La posición del texto, que también es el valor de un punto (matriz de tres elementos Double), indicará la longitud de las líneas auxiliares, teniendo en cuenta por supuesto la configuración del estilo de acotación actual.

Veremos las cuatro nuevas propiedades.

- `ExtLine1Point`. Obtiene y/o asigna un punto para la primera línea de extensión.

La sintaxis para asignar es:

```
ObjetoCotaAlineada.ExtLine1Point = DbfPunto1
```

La sintaxis para obtener es:

```
VarPunto1 = ObjetoCotaAlineada.ExtLine1Point
```

Como siempre, para recoger un punto utilizaremos una variable `Variant`. Después podremos acceder a cada valor unitario mediante índices, como ya se explicó.

- `ExtLine2Point`. Obtiene y/o asigna un punto para la segunda línea de extensión.

La sintaxis para asignar es:

```
ObjetoCotaAlineada.ExtLine2Point = DbfPunto2
```

La sintaxis para obtener es:

```
VarPunto2 = ObjetoCotaAlineada.ExtLine2Point
```

Las mismas consideraciones que para la propiedad anterior.

- `TextPosition`. Obtiene y/o asigna un punto para la posición del texto.

La sintaxis para asignar es:

```
ObjetoCota.TextPosition = DbfPuntoTexto
```

La sintaxis para obtener es:

```
VarPuntoTexto = ObjetoCota.TextPosition
```

Como siempre, para recoger un punto utilizaremos una variable `Variant`. Después podremos acceder a cada valor unitario mediante índices, como ya explicó.

- `TextRotation`. Obtiene y/o asigna el ángulo de rotación en radianes del texto de cota.

La sintaxis para asignar es:

```
ObjetoCota.TextRotation = DblRotación
```

La sintaxis para obtener es:

```
DblRotación = ObjetoCota.TextRotation
```

`DblRotación` ha de ser un valor `Double`.

DOCE.5.21.2. Cotas angulares

El método de adición de cotas angulares es `AddDimAngular`:

```
Set ObjCotaAngular = ObjColección.AddDimAngular(DblVérticeÁngulo,  
DblPtoFinal1, DblPtoFinal2, DblPosTexto)
```

Propiedades de los objetos de cota angular:

<code>Application</code>	<code>Color</code>	<code>EntityName</code>
<code>EntityType</code>	<code>ExtLine1EndPoint</code>	<code>ExtLine1StartPoint</code>
<code>ExtLine2EndPoint</code>	<code>ExtLine1StartPoint</code>	<code>Handle</code>
<code>Layer</code>	<code>LineType</code>	<code>LinetypeScale</code>
<code>Normal</code>	<code>ObjectID</code>	<code>Rotation</code>
<code>StyleName</code>	<code>TextPosition</code>	<code>TextRotation</code>
<code>TextString</code>	<code>Visible</code>	

Métodos de los objetos de cota angular:

<code>ArrayPolar</code>	<code>ArrayRectangular</code>	<code>Copy</code>	<code>Erase</code>
<code>GetBoundingBox</code>	<code>GetXData</code>	<code>Highlight</code>	<code>IntersectWith</code>
<code>Mirror</code>	<code>Mirror3D</code>	<code>Move</code>	<code>Rotate</code>
<code>Rotate3D</code>	<code>ScaleEntity</code>	<code>SetXData</code>	<code>TransformBy</code>
<code>Update</code>			

En el método `AddDimAngular` primero se indica como argumento el centro del arco o el vértice común entre las dos líneas auxiliares (`Double`). Los dos siguientes argumentos especifican los puntos (también matriz de tres elementos `Double`) por los que pasan las líneas de extensión o auxiliares. El último argumento indica la posición del texto; es también un punto (`Double`).

Las cuatro nuevas propiedades a continuación.

- `ExtLine1EndPoint`. Obtiene y/o asigna un punto final para la primera línea de extensión.

La sintaxis para asignar es:

```
ObjetoCotaAngular.ExtLine1EndPoint = DbIPuntoFinal1
```

La sintaxis para obtener es:

```
VarPuntoFinal1 = ObjetoCotaAngular.ExtLine1EndPoint
```

- ExtLine1StartPoint. Obtiene y/o asigna un punto inicial para la primera línea de extensión.

La sintaxis para asignar es:

```
ObjetoCotaAngular.ExtLine1StartPoint = DbIPuntoInicial1
```

La sintaxis para obtener es:

```
VarPuntoInicial1 = ObjetoCotaAngular.ExtLine1StartPoint
```

- ExtLine2EndPoint. Obtiene y/o asigna un punto final para la segunda línea de extensión.

La sintaxis para asignar es:

```
ObjetoCotaAngular.ExtLine2EndPoint = DbIPuntoFinal2
```

La sintaxis para obtener es:

```
VarPuntoFinal2 = ObjetoCotaAngular.ExtLine2EndPoint
```

- ExtLine2StartPoint. Obtiene y/o asigna un punto inicial para la segunda línea de extensión.

La sintaxis para asignar es:

```
ObjetoCotaAngular.ExtLine2StartPoint = DbIPuntoInicial2
```

La sintaxis para obtener es:

```
VarPuntoInicial2 = ObjetoCotaAngular.ExtLine2StartPoint
```

DOCE.5.21.3. Cotas diamétricas

El método de adición de cotas diamétricas es AddDimDiametric:

```
Set ObjCotaDiamétrica = ObjColección.AddDimDiametric(DbIDiámetroPto1,  
DbIDiámetroPto2, DbLongDirectriz)
```

Propiedades de los objetos de cota diamétrica:

Application	Color	EntityName	EntityType
Handle	Layer	LeaderLength	LineType
LinetypeScale	Normal	ObjectID	Rotation
StyleName	TextPosition	TextRotation	TextString

Visible

Métodos de los objetos de cota diamétrica:

ArrayPolar	ArrayRectangular	Copy	Erase
GetBoundingBox	GetXData	Highlight	IntersectWith
Mirror	Mirror3D	Move	Rotate
Rotate3D	ScaleEntity	SetXData	TransformBy
Update			

El método `AddDimDiametric` crea una cota diámetro para un círculo o un arco dados los dos puntos del diámetro (matrices de tres elementos `Double`) y la longitud de la directriz de la línea de cota (`Double`).

Se crean diferentes tipos de cotas diámetros dependiendo del tamaño del círculo o del arco, del argumento de longitud de directriz y de los valores de las variables de acotación `DIMUPT`, `DIMTOFL`, `DIMFIT`, `DIMTIH`, `DIMTOH`, `DIMJUST` y `DIMTAD`.

La nueva propiedad:

- `LeaderLength`. Obtiene y/o asigna una longitud (`Double`) para la línea directriz.

La sintaxis para asignar es:

```
ObjetoCota.LeaderLength = DblLongitud
```

La sintaxis para obtener es:

```
DblLongitud = ObjetoCota.LeaderLength
```

NOTA: La asignación de una longitud de directriz debe realizarse únicamente a la hora de dibujar la cota. Después de que ésta haya sido guardada, cambiar el valor de `LeaderLength` no afectará a cómo se muestra cota, pero el nuevo valor se reflejará en un `.DXF` y en aplicaciones `AutoLISP` y `ADS`.

DOCE.5.21.4. Cotras por coordenadas

El método de adición de cotas de coordenadas es `AddDimOrdinate`:

```
Set ObjCotaCoordenada = ObjColección.AddDimOrdinate(DblPtoDefinición,  
DblPtoFinalDirectriz, BooUsarEjeX)
```

Propiedades de los objetos de cota de coordenadas:

Application	Color	EntityName	EntityType
Handle	Layer	LineType	LinetypeScale
Normal	ObjectID	Rotation	StyleName
TextPosition	TextRotation	TextString	Visible

Métodos de los objetos de cota de coordenadas:

ArrayPolar	ArrayRectangular	Copy	Erase
GetBoundingBox	GetXData	Highlight	IntersectWith
Mirror	Mirror3D	Move	Rotate
Rotate3D	ScaleEntity	SetXData	TransformBy
Update			

El método `AddDimOrdinate` crea una cota de coordenadas. Hay que indicar un punto (`Double`) de definición que será el acotado, del que parte de línea de cota. Después se indica otro punto (`Double`) final para la línea de cota. En este segundo punto será donde se sitúe el texto. Por último, un valor `Boolean` que especifica si el valor del texto será el del eje X o el del eje Y: `True` crea una cota de coordenadas mostrando el valor del eje X; `False` mostrando el valor de Y.

Las propiedades y métodos expuestos están ya estudiados.

DOCE.5.21.5. Cotas radiales

El método de adición de cotas radiales es `AddDimRadial`:

```
Set ObjCotaRadial = ObjColección.AddDimRadial(DblPtoCentro, DblPtoCruce, DblLongDirectriz)
```

Propiedades de los objetos de cota radial:

Application	Color	EntityName	EntityType
Handle	Layer	LeaderLength	LineType
LinetypeScale	Normal	ObjectID	Rotation
StyleName	TextPosition	TextRotation	TextString
Visible			

Métodos de los objetos de cota radial:

ArrayPolar	ArrayRectangular	Copy	Erase
GetBoundingBox	GetXData	Highlight	IntersectWith
Mirror	Mirror3D	Move	Rotate
Rotate3D	ScaleEntity	SetXData	TransformBy
Update			

Al método `AddDimRadial` hay que proporcionarle como argumentos el punto central (*array* de tres elementos `Double`) del arco o círculo que acotar, un punto (*array* de tres elementos `Double`) de dicho arco o círculo por el que pasará la línea de cota o directriz y una longitud (tipo de dato `Double`) para dicha directriz.

Las propiedades y métodos expuestos están ya estudiados.

DOCE.5.21.6. Cotas giradas

El método de adición de cotas giradas es `AddDimRotated`:

```
Set ObjCotaGirada = ObjColección.AddDimRotated(DblPtoLinExt1, DblPtoLinExt2, DblPosLíneaCota, DblRotación)
```

Propiedades de los objetos de cota girada:

Application	Color	EntityName	EntityType
Handle	Layer	LineType	LinetypeScale
Normal	ObjectID	Rotation	StyleName
TextPosition	TextRotation	TextString	Visible

Métodos de los objetos de cota girada:

ArrayPolar	ArrayRectangular	Copy	Erase
GetBoundingBox	GetXData	Highlight	IntersectWith
Mirror	Mirror3D	Move	Rotate
Rotate3D	ScaleEntity	SetXData	TransformBy
Update			

DblPtoLinExt1 es un valor que indica el punto (matriz de tres elementos Double) de la primera línea auxiliar o de extensión, es decir, uno de los extremos de la línea que será medida. *DblPtoLinExt2* lo mismo que *DblPtoLinExt1* pero para la segunda línea auxiliar. *DblPosLíneaCota* es también un punto (matriz de tres valores Double) que especifica la posición de la línea de cota y, por ende, la del texto (según variables). Y *DblRotación* es un valor Double que indica, en radianes, al ángulo de rotación de la cota girada.

Las propiedades y métodos expuestos están ya estudiados.

DOCE.5.21.7. Directrices

El método `AddLeader` nos ofrece la posibilidad de agregar directrices de **AutoCAD** a nuestro dibujo actual. La sintaxis del método es la que sigue:

```
Set ObjDirectriz = ObjColección.AddLeader(DblMatrizPtos, ObjAnotación, IntTipo)
```

Propiedades de los objetos de directriz:

Application	Color	Coordinates	EntityName
EntityType	Handle	Layer	LineType
LinetypeScale	Normal	ObjectID	StyleName
Type	Visible		

Métodos de los objetos de directriz:

ArrayPolar	ArrayRectangular	Copy	Erase
Evaluate	GetBoundingBox	GetXData	Highlight
IntersectWith	Mirror	Mirror3D	Move
Rotate	Rotate3D	ScaleEntity	SetXData
TransformBy	Update		

DblMatrizPtos ha de ser una matriz de elementos de punto (matriz también de tres elementos Double) que contenga los vértices (cuántos sean) de la directriz. *ObjAnotación* es un objeto (Object) que solamente puede ser de tolerancia (que enseguida veremos), de texto múltiple (ya estudiado) o de referencia a bloque.

IntTipo es un valor entero que además puede contener las constantes siguientes:

`AcLineNoArrow` `acLineWithArrow` `acSplineNoArrow` `acSplineWithArrow`

El valor que se devuelve (*ObjDirectriz*) es un objeto de directriz que ha de almacenarse en una variable declarada evidentemente como Object.

Ahora comentamos el método `Evaluate` de los objetos de directriz.

```
ObjGráfico.Evaluate
```

Este método exclusivo de directrices y sombreados evalúa los objetos y, si fuera necesario, los actualizaría. En el caso de las directrices se comprueba la relación de la misma con su anotación asociada.

NOTA: Véase la sección **DOCE.5.7.1.**, donde se comenta la propiedad `Type` de las polilíneas, para observar los valores de esta propiedad en las directrices.

DOCE.5.21.8. Tolerancias

El método `AddTolerance` permite crear objetos de tolerancia. La sintaxis del método es la siguiente:

<code>Set ObjTolerancia = ObjColección.AddTolerance(StrTexto, DblPtoIns, DblVectDir)</code>

Propiedades de los objetos de tolerancia:

<code>Application</code>	<code>Color</code>	<code>DirectionVector</code>	<code>EntityName</code>
<code>EntityType</code>	<code>Handle</code>	<code>InsertionPoint</code>	<code>Layer</code>
<code>LineType</code>	<code>LinetypeScale</code>	<code>Normal</code>	<code>ObjectID</code>
<code>StyleName</code>	<code>TextString</code>	<code>Visible</code>	

Métodos de los objetos de tolerancia:

<code>ArrayPolar</code>	<code>ArrayRectangular</code>	<code>Copy</code>	<code>Erase</code>
<code>GetBoundingBox</code>	<code>GetXData</code>	<code>Highlight</code>	<code>IntersectWith</code>
<code>Mirror</code>	<code>Mirror3D</code>	<code>Move</code>	<code>Rotate</code>
<code>Rotate3D</code>	<code>ScaleEntity</code>	<code>SetXData</code>	<code>TransformBy</code>
<code>Update</code>			

A `AddTolerance` se le suministra como primer argumento una cadena (`String`) de texto que será la que se escriba en la tolerancia. Como segundo argumento un valor que es matriz de tres elementos `Double`; esto es un punto, que será el punto de inserción de la tolerancia (del símbolo). Por último, un vector de dirección (tres elementos `Double`) que especifican la dirección del símbolo de tolerancia.

Las propiedades y métodos expuestos están ya estudiados.

A continuación veremos una macro de ejemplo la cual, tras dibujar un rectángulo con cuatro líneas y pasarlas a color azul (para luego distinguir las cotas), agrega tres cotas distintas: una alineada, otra angular y otra girada. A modo pues de ejemplo:

Option Explicit

```
Dim AcadDoc As Object
Dim AcadModel As Object
```

```
Dim Línea As Object
Dim PtoLin1(1 To 3) As Double
Dim PtoLin2(1 To 3) As Double
Dim PtoLin3(1 To 3) As Double
Dim PtoLin4(1 To 3) As Double
Dim Cota As Object
Dim PtoTexto(1 To 3) As Double
```

```
Sub Macro()
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument
    Set AcadModel = AcadDoc.ModelSpace
```



```

PtoLin1(1) = 10: PtoLin1(2) = 10: PtoLin1(3) = 0
PtoLin2(1) = 10: PtoLin2(2) = 20: PtoLin2(3) = 0
PtoLin3(1) = 40: PtoLin3(2) = 20: PtoLin3(3) = 0
PtoLin4(1) = 40: PtoLin4(2) = 10: PtoLin4(3) = 0
Set Línea = AcadModel.AddLine(PtoLin1, PtoLin2)
Línea.Color = 5
Set Línea = AcadModel.AddLine(PtoLin2, PtoLin3)
Línea.Color = 5
Set Línea = AcadModel.AddLine(PtoLin3, PtoLin4)
Línea.Color = 5
Set Línea = AcadModel.AddLine(PtoLin4, PtoLin1)
Línea.Color = 5

PtoTexto(1) = 25: PtoTexto(2) = 27: PtoTexto(3) = 0
Set Cota = AcadModel.AddDimAligned(PtoLin2, PtoLin3, PtoTexto)

Dim PtoPasa1(1 To 3) As Double
Dim PtoPasa2(1 To 3) As Double
PtoPasa1(1) = 40: PtoPasa1(2) = 15: PtoPasa1(3) = 0
PtoPasa2(1) = 30: PtoPasa2(2) = 10: PtoPasa2(3) = 0
PtoTexto(1) = 35: PtoTexto(2) = 12: PtoTexto(3) = 0
Set Cota = AcadModel.AddDimAngular(PtoLin4, PtoPasa1, PtoPasa2, PtoTexto)

PtoTexto(1) = 30: PtoTexto(2) = 3: PtoTexto(3) = 0
Set Cota = AcadModel.AddDimRotated(PtoLin1, PtoLin4, PtoTexto, 60)
End Sub

```

DOCE.5.22. Sombreado

Un sombreado es (y siempre ha sido), en realidad, un objeto más de dibujo de **AutoCAD** que se puede añadir sin ningún problema desde VBA. Veremos aquí cómo hacerlo.

El método para dibujar sombreados es `AddHatch`. La manera de agregar un sombreado en cualquiera de las colecciones que admiten este método (Espacio Modelo, Espacio Papel y bloques) es mediante la sintaxis siguiente:

```
Set ObjSombreado = ObjColección.AddHatch(IntTipoPatrón, StrNombrePatrón,
BooAsociatividad)
```

Propiedades de los objetos de sombreado:

Application	AssociativeHatch	Color	Elevation
EntityName	EntityType	Handle	HatchStyle
InsertionPoint	Layer	LineType	LinetypeScale
Normal	NumberOfLoops	ObjectID	PatternAngle
PatternDouble	PatternName	PatternScale	PatternSpace
PatternType	Visible		

Métodos de los objetos de sombreado:

AppendInnerLoop	AppendOuterLoop	ArrayPolar	ArrayRectangular
Copy	Erase	Evaluate	GetBoundingBox
GetLoopAt	GetXData	Highlight	InsertLoopAt
IntersectWith	Mirror	Mirror3D	Move
Rotate	Rotate3D	ScaleEntity	SetPattern
SetXData	TransformBy	Update	

IntTipoPatrón es un valor Integer que se refiere al tipo de patrón de sombreado que se utilizará. Además acepta las siguientes constantes:

`acHatchPatternTypePreDefined` `acHatchPatternTypeUserDefined`
`acHatchPatternTypeCustomDefined`

Cada una de ellas dice referencia a los distintos tipos de patrón que también podemos utilizar desde la interfaz gráfica de **AutoCAD**.

StrNombrePatrón es un valor de cadena alfanumérica (String) que representa el nombre del patrón que se usará. Y *BooAsociatividad* es un valor Boolean que determina la condición de asociatividad del patrón de sombreado: True significa que el patrón es asociativo y False que no es asociativo.

Pasemos ahora a comentar las propiedades nuevas no estudiadas.

- **AsociativeHatch**. Esta propiedad obtiene exclusivamente el carácter de asociatividad de un patrón de sombreado ya dibujado. El valor de retorno habrá de ser recogido en una variable declarada como Boolean. Si esta variable guarda un valor True significará que el patrón es asociativo; si guarda, por el contrario, un valor False, significará que el patrón no es asociativo. Veamos su sintaxis:

```
BooAsociatividad = ObjetoSombreado.AsociativeHatch
```

- **Elevation**. Obtiene y también asigna la elevación (valor de la coordenada Z) de un sombreado.

La sintaxis para asignar es:

```
ObjetoSombreado.Elevation = DblElevación
```

La sintaxis para obtener es:

```
DblElevación = ObjetoSombreado.Elevation
```

La elevación es un valor Double (doble precisión).

- **HatchStyle**. Dice relación al estilo del patrón de sombreado. Esta propiedad se puede extraer de un sombreado y también se puede asignar. La sintaxis que utilizaremos para asignar un estilo a un sombreado es:

```
ObjetoSombreado.HatchStyle = IntEstilo
```

La sintaxis para obtener un estilo de un sombreado es:

```
IntEstilo = ObjetoSombreado.HatchStyle
```

Donde *IntEstilo* es un valor Integer que además admite las siguientes constantes:

`acHatchStyleNormal` `acHatchStyleOuter` `acHatchStyleIgnore`

Todas ellas se corresponden con las opciones de *Estilo de contorno* de las *Opciones avanzadas* del sombreado en **AutoCAD**: *Normal*, *Exterior* e *Ignorar*.

- **NumberOfLoops**. Devuelve el número de contornos de un sombreado:

```
IntNúmContornos = ObjetoSombreado.NumberOfLoops
```

IntNúmContornos habrá sido declarada como Integer, evidentemente.

- *PatternAngle*. Especifica el ángulo en radianes del patrón de sombreado.

La sintaxis para asignar es:

```
ObjetoSombreado.PatternAngle = DblÁngulo
```

La sintaxis para obtener es:

```
DblÁngulo = ObjetoSombreado.PatternAngle
```

DblÁngulo es el valor del ángulo en radianes (Double), por lo tanto estará entre 0 y 6.28. Este valor es también controlado por la variable de sistema HPANG.

- *PatternDouble*. Especifica si el sombreado es doble o no. El valor asignado u obtenido será del tipo Boolean: True es doble, False no es doble.

La sintaxis para asignar es:

```
ObjetoSombreado.PatternDouble = BooDoble
```

La sintaxis para obtener es:

```
BooDoble = ObjetoSombreado.PatternDouble
```

Este valor es también controlado por la variable de sistema HPDOUBLE.

- *PatternName*. Especifica el nombre del patrón de sombreado. El valor asignado u obtenido será del tipo String.

La sintaxis para asignar es:

```
ObjetoSombreado.PatternName = StrNombre
```

La sintaxis para obtener es:

```
StrNombre = ObjetoSombreado.PatternName
```

Este valor es también controlado por la variable de sistema HPNAME.

- *PatternScale*. Especifica la escala del patrón de sombreado. El valor asignado u obtenido será del tipo Double.

La sintaxis para asignar es:

```
ObjetoSombreado.PatternScale = DblEscala
```

La sintaxis para obtener es:

```
DblEscala = ObjetoSombreado.PatternScale
```

Este valor es también controlado por la variable de sistema `HPSCALE`.

- `PatternSpace`. Especifica el espaciado entre líneas en los sombreados definidos por el usuario. El tipo de dato de este valor es `Double`.

La sintaxis para asignar es:

```
ObjetoSombreado.PatternSpace = DblEspaciado
```

La sintaxis para obtener es:

```
DblEspaciado = ObjetoSombreado.PatternSpace
```

Este valor es también controlado por la variable de sistema `HPSPACE`.

- `PatternType`. Obtiene el tipo de patrón utilizado en un sombreado. Es una propiedad de sólo lectura. El tipo de dato para este valor es `Integer`, aunque también admite las constantes explicadas para el argumento `IntTipoPatrón` del método `AddHatch`.

La sintaxis es:

```
IntTipoPatrón = ObjetoSombreado.PatternType
```

Vamos a repasar ahora los nuevos métodos propuestos.

- `AppendInnerLoop`. Este método añade contornos interiores al sombreado. Su sintaxis es la que sigue:

```
Call ObjSombreado.AppendInnerLoop(ObjContornoInterior)
```

`ObjContornoInterior` será una matriz de objetos que formen un contorno cerrado. Estos objetos podrán ser sólo líneas, polilíneas, círculos, elipses, splines y/o regiones.

Antes de añadir objetos interiores de contorno, habremos de añadir los objetos exteriores evidentemente. Para ello utilizaremos el siguiente método.

- `AppendOuterLoop`. Este método añade contornos exteriores al sombreado. Su sintaxis es la que sigue:

```
Call ObjSombreado.AppendOuterLoop(ObjContornoExterior)
```

`ObjContornoExterior` igual que para el método anterior `ObjetoContornoInterior`.

La forma correcta de crear un sombreado desde VBA consiste en definir primero el patrón y después encerrarlo en un contorno. Veamos la siguiente macro como ejemplo:

Option Explicit

```
Dim AcadDoc As Object  
Dim AcadModel As Object  
Dim Sombreado As Object  
Dim Círculo1 As Object  
Dim MatrizObjeto(0) As Object
```

```
Sub Macro()  
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument  
    Set AcadModel = AcadDoc.ModelSpace
```

```
Dim PtoCentrol(1 To 3) As Double
PtoCentrol(1) = 10: PtoCentrol(2) = 10: PtoCentrol(3) = 0
Set Círculo1 = AcadModel.AddCircle(PtoCentrol, 10)
Set MatrizObjeto(0) = Círculo1

Set Sombreado = AcadModel.AddHatch(1, "angle", True)
Call Sombreado.AppendOuterLoop(MatrizObjeto)
End Sub
```

Como podemos observar, lo primero que se hace es crear un círculo que será el objeto que se va a sombreadar. Seguidamente creamos el sombreado o patrón con las características que nos interesen. Por último, encerramos el sombreado dentro del círculo, añadiendo éste como contorno exterior con *AppendOuterLoop*.

NOTA: Nótese que el argumento que requiere *AppendOuterLoop* (y también *AppendInnerLoop*) ha de ser una matriz de objetos. En este caso en el que sólo deseamos sombreadar un círculo, creamos un matriz de un único elemento con dicho círculo. Si no lo hiciéramos así, este método no funcionaría.

- *GetLoopAt*. Obtiene el contorno de sombreado en el índice dado:

```
Call ObjSombreado.GetLoopAt(IntÍndice, VarContorno)
```

IntÍndice es una variable *Integer* que guardará el índice. El primer índice es el 0. *VarContorno* es una variable *Variant* que guardará una matriz de los diversos objetos obtenidos.

- *InsertLoopAt*. Este método inserta un contorno en la localización indicada por un índice:

```
Call ObjSombreado.InsertLoopAt(IntÍndice, IntTipoContorno, ObjContorno)
```

IntÍndice es un valor *Integer* que representa un índice de localización en la matriz de vértices que forman el contorno del sombreado. El primer índice de la matriz es el 0.

IntTipoContorno es un valor *Integer*, también, que representa el tipo de contorno. Este argumento además admite las siguientes constantes, que no son sino los tipos de contornos aceptados:

<i>acHatchLoopTypeDefault</i>	<i>acHatchLoopTypeExternal</i>
<i>acHatchLoopTypePolyline</i>	<i>acHatchLoopTypeDerived</i>
<i>acHatchLoopTypeTextbox</i>	

ObjContorno igual que para métodos anteriores en *ObjContornoExterior* y *ObjetoContornoInterior*.

- *SetPattern*. Asigna un nombre y un tipo de patrón a un sombreado. Su sintaxis es:

```
Call ObjSombreado.SetPattern(IntTipoPatrón, StrNombrePatrón)
```

Ambos argumentos de *SetPattern* tienen el mismo significado y los mismos valores que los argumentos del mismo nombre en la sintaxis del método *AddHatch*.

DOCE.5.23. Referencias a bloques

La manera de crear bloques la veremos posteriormente al hablar de la colección de bloques y del objeto de bloque (sección **DOCE.8.3.**), así como el método para insertar bloques ya creados que se explica en la colección de objetos de Espacio Modelo (sección **DOCE.8.1.**). Aquí se trata únicamente de las referencias o inserciones de bloques.

Es decir, los bloques primero hay que crearlos (crear sus objetos componentes y después el propio bloque). Esto se tratará, como ya se ha dicho, más adelante. La manera de tratar esos objetos unitarios de bloque también se verá después. Posteriormente los bloques se insertan, dando lugar a una inserción de bloque o referencia a bloque. De esto es de lo que se habla en esta sección.

A continuación veremos las propiedades y métodos de las inserciones de bloque.

Propiedades de los objetos de referencia a bloque:

Application	Color	EntityName	EntityType
Handle	InsertionPoint	Layer	LineType
LinetypeScale	Name	Normal	ObjectID
Rotation	Visible	XScaleFactor	YScaleFactor

Métodos de los objetos de referencia a bloque:

ArrayPolar	ArrayRectangular	Copy	Erase
Explode	GetAttributes	GetBoundingBox	GetXData
HasAttributes	Highlight	IntersectWith	Mirror
Mirror3D	Move	Rotate	Rotate3D
ScaleEntity	SetXData	TransformBy	Update

Veamos pues las nuevas propiedades inherentes a estos objetos de referencia a bloque.

- **XScaleFactor**. Asigna u obtiene el factor de escala en el eje X de la inserción de un bloque.

La sintaxis para asignar es:

```
ObjetoRefBloque.XScaleFactor = DblFactorEscalaX
```

La sintaxis para obtener es:

```
DblFactorEscalaX = ObjetoRefBloque.XScaleFactor
```

El valor es un valor Double que es 1.0 por defecto.

- **YScaleFactor**. Asigna u obtiene el factor escala en el eje Y de la inserción de un bloque.

La sintaxis para asignar es:

```
ObjetoRefBloque.YScaleFactor = DblFactorEscalaY
```

La sintaxis para obtener es:

```
DblFactorEscalaY = ObjetoRefBloque.YScaleFactor
```

El valor es un valor `Double` que es 1.0 por defecto.

Y veamos ahora los dos nuevos métodos.

- `GetAttributes`. Este método obtiene los atributos de una inserción de bloque. Para recogerlos habremos de declarar una variable como `Variant`. La sintaxis de uso es:

```
VarAtributos = ObjRefBloque.GetAttributes
```

El retorno de este método (como ya hemos dicho en una variable tipo `Variant`) será una matriz de objetos de atributo —que veremos enseguida—. Como ya sabemos podemos hacer uso de índices para extraer cada uno de ellos de la matriz.

- `HasAttributes`. Este método especifica si el bloque (por medio de su inserción) contiene atributos o no. El valor de retorno ha de recogerlo una variable declarada como `Boolean` que, si recoge `True` significará que el bloque tiene atributos y, si recoge `False`, que no tiene. La sintaxis de utilización es la siguiente:

```
BoolTieneAtributos = ObjRefBloque.HasAttributes
```

DOCE.5.24. Atributos de bloques

Para poder introducir un atributo en un bloque evidentemente hemos primero de crear el bloque (cosa que ya estudiaremos como se ha comentado anteriormente). El entorno VBA para **AutoCAD** dispone de dos objetos relacionados con los atributos: el objeto de atributo y el objeto de referencia de atributo.

Aunque pudiera parecer lo contrario, el atributo en sí lo representa el objeto denominado referencia de atributo. Éste es el atributo propiamente dicho, o sea, un objeto que contiene texto enlazado con un bloque. El objeto de atributo, por su lado, es el llamado en **AutoCAD** definición de atributo, y es un objeto que aparece como una cadena de texto y describe las características de un objeto de referencia de atributo. No nos equivoquemos.

DOCE.5.24.1. Referencias de atributos

Como hemos explicado ya, el objeto de referencia de atributo en sí el propio atributo. El método que utilizamos para añadir atributos a bloques ya creados es `AddAttribute`, y tiene la siguiente sintaxis de uso:

```
Set ObjRefAtributo = ObjColección.AddAttribute(DblAltura, IntModo, StrMensaje, DblPtoIns, StrIdentificador, StrValorDefecto)
```

Propiedades de los objetos de referencia de atributo:

<code>Application</code>	<code>Color</code>	<code>EntityName</code>	<code>EntityType</code>
<code>FieldLength</code>	<code>Handle</code>	<code>Height</code>	<code>HorizontalAlignment</code>
<code>InsertionPoint</code>	<code>Layer</code>	<code>LineType</code>	<code>LinetypeScale</code>
<code>Normal</code>	<code>ObjectID</code>	<code>ObliqueAngle</code>	<code>Rotation</code>
<code>ScaleFactor</code>	<code>StyleName</code>	<code>TagString</code>	<code>TextAlignmentPoint</code>
<code>TextGenerationFlag</code>	<code>TextString</code>	<code>Thickness</code>	<code>VerticalAlignment</code>
<code>Visible</code>			

Métodos de los objetos de referencia de atributo:

Erase	GetBoundingBox	GetXData	Highlight
IntersectWith	Move	Rotate	Rotate3D
ScaleEntity	SetXData	TransformBy	Update

DblAltura es el primer argumento que hemos de suministrar al método *AddAttribute*. Dice relación a la altura del texto del atributo en las unidades del documento activo actual, esto es del dibujo actual.

IntModo es un argumento opcional que indica el modo del atributo (invisible, constante, verificable y/o predefinido). Ha de ser un valor entero (*Integer*), pero también admite las siguientes constantes:

<i>acAttributeModeInvisible</i>	<i>acAttributeModeConstant</i>
<i>acAttributeModeVerify</i>	<i>acAttributeModePreset</i>

Puede ser combinada más una constante utilizando el operador booleano *OR*. Hemos de prestar especial atención a no combinar modos incompatibles.

StrMensaje es una cadena (*String*) que representa el mensaje que aparece en línea de comandos o en el cuadro de petición de atributos (dependiendo del valor de la variable de sistema de **AutoCAD** *ATTDIR*) al insertar el bloque que contiene el atributo. El valor por defecto para este argumento es la cadena indicada en el argumento *StrIdentificador*. Si el modo del atributo es *acAttributeModeConstant*, este argumento se inhabilita.

DblPtoIns es una matriz de tres valores *Double* (X, Y y Z) que representa las coordenadas del punto de inserción para el atributo el SCU.

StrIdentificador es una cadena (*String*) que representa al identificador del atributo. Como ya debemos saber en esta cadena deberemos obviar los espacios y los signos de exclamación. Además, decir que **AutoCAD** cambia automáticamente las minúsculas a mayúsculas.

StrValorDefecto es una cadena (*String*) que representa el valor por defecto del atributo al ser insertado.

Explicaremos ahora las dos propiedades no estudiadas.

- *FieldLength*. Asigna u obtiene la longitud de campo (*Integer*) en número de caracteres para el atributo.

La sintaxis para asignar es:

```
ObjetoAtributo.FieldLength = IntLongitudCampo
```

La sintaxis para obtener es:

```
IntLongitudCampo = ObjetoAtributo.FieldLength
```

- *TagString*. Asigna u obtiene el identificador (*String*) de un atributo.

La sintaxis para asignar es:


```
ObjetoAtributo.TagString = StrIdentificador
```

La sintaxis para obtener es:

```
StrIdentificador = ObjetoAtributo.TagString
```

DOCE.5.24.2. Objeto de atributo

Como ya hemos dicho este objeto aparece como una cadena de texto que describe las características de los objetos de referencia de atributo (definición de atributo en **AutoCAD**).

Propiedades de los objetos de atributo:

Application	Color	EntityName
EntityType	FieldLength	Handle
Height	HorizontalAlignment	InsertionPoint
Layer	LineType	LinetypeScale
Mode	Normal	ObjectID
ObliqueAngle	PromptString	Rotation
ScaleFactor	StyleName	TagString
TextAlignmentPoint	TextGenerationFlag	TextString
Thickness	VerticalAlignment	Visible

Métodos de los objetos de atributo:

ArrayPolar	ArrayRectangular	Copy	Erase
GetBoundingBox	GetXData	Highlight	IntersectWith
Mirror	Mirror3D	Move	Rotate
Rotate3D	ScaleEntity	SetXData	TransformBy
Update			

Veamos las dos nuevas propiedades.

- **Mode**. Obtiene y/o asigna el modo de un atributo. Es un valor *Integer* que admite además las constantes especificadas en la sección anterior para el argumento *IntModo* del método `AddAttribute`.

La sintaxis para asignar un modo es:

```
ObjetoAtributo.Mode = IntModo
```

La sintaxis para obtener un modo es:

```
IntModo = ObjetoAtributo.Mode
```

- **PromptString**. Obtiene y/o asigna la cadena (*String*) de mensaje de petición de un atributo (véase la explicación del argumento *StrMensaje* del método `AddAttribute` en la sección anterior).

La sintaxis para asignar una cadena es:

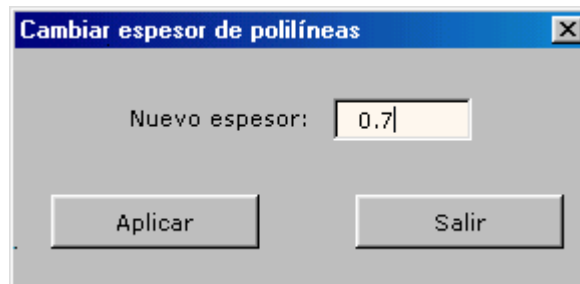
```
ObjetoAtributo.PromptString = StrMensaje
```

La sintaxis para obtener una cadena es:

```
StrMensaje = ObjetoAtributo.PromptString
```

1ª fase intermedia de ejercicios

- Realizar una macro VBA que dibuje una línea, un círculo y una elipse en Espacio Papel (el objeto de la elipse sombreado con un patrón cualquiera).
- Crear un programa VBA, con formulario, que sea capaz de cambiar el espesor de todas las polilíneas dibujadas en el Espacio Modelo de la sesión actual de dibujo. El nuevo espesor de polilínea se introducirá en el cuadro de diálogo. Este cuadro puede ser el siguiente:



DOCE.6. LA APLICACIÓN AutoCAD

La aplicación **AutoCAD** como tal es uno de los objetos definidos en VBA. Si nos fijamos en la plantilla de objetos proporcionada al principio de este **MÓDULO**, nos daremos cuenta de que se encuentra en el más alto escalafón de la jerarquía y por él vamos a comenzar.

Este objeto recibe el nombre de `Application` y dice relación a la ventana de la aplicación principal, es decir, el propio programa **AutoCAD** en sí. Si recordamos, a este objeto hemos hecho referencia en multitud de ocasiones cuando escribimos, por ejemplo:

```
Set AcadDoc = GetObject(, "AutoCAD.Application")
```

Como objeto que es, posee propiedades y métodos. A continuación veremos como siempre la relación completa de ambos y pasaremos seguidamente a estudiarlos.

Propiedades del objeto de aplicación:

ActiveDocument	Application	Caption	FullName
Height	Left	LocaleID	Name
Path	Preferences	Top	Version
Visible	Width		

Métodos del objeto de aplicación:

GetInterfaceObject	ListADS	ListARX	LoadADS
LoadARX	Quit	UnloadADS	UnloadARX
Update			

Pasemos pues a comentar las propiedades primero.

- `ActiveDocument`. Obtiene el objeto de documento activo. La sintaxis es:

```
Set ObjDocumento = ObjAplicación.ActiveDocument
```

La sintaxis la conocemos de sobra, ya que la utilizamos en todos los programas al hacer:

```
Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument
```

Nos preguntaremos la necesidad de esta propiedad cuando en **AutoCAD** únicamente se puede tener un documento abierto cada vez (es una aplicación SDI). Y es que `ActiveDocument` está preparada para futuras versiones de **AutoCAD**, en el que el entorno sea MDI (múltiples documentos abiertos) y el programa afecte únicamente al activo actualmente (uno cada vez evidentemente).

El retorno de esta propiedad aplicada a la aplicación es un objeto de documento, del que se hablará en la siguiente sección.

- `Caption`. Obtiene el texto de la barra de título (generalmente azul) de la aplicación. La sintaxis es:

```
StrTítulo = ObjAplicación.Caption
```

Siendo `StrTítulo` declarado como `String`.

Veamos el siguiente ejemplo de una macro VBA:

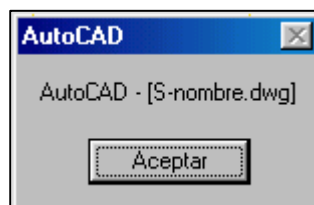
```
Option Explicit
```

```
Dim Acad As Object  
Dim AcadDoc As Object  
Dim AcadModel As Object
```

```
Sub Macro()  
    Set Acad = GetObject(, "AutoCAD.Application")  
    Set AcadDoc = Acad.ActiveDocument  
    Set AcadModel = AcadDoc.ModelSpace  
  
    Dim TitApli As String  
    TitApli = Acad.Caption  
    MsgBox TitApli  
End Sub
```

Como vemos hemos introducido un objeto nuevo, hemos desglosado la llamada a los objetos principales de **AutoCAD** en una rama más: la aplicación en sí. De esta manera, podemos utilizar este nuevo objeto (`Acad`) para extraer el texto de la ventana de título de **AutoCAD** en un momento concreto, para después mostrarlo con un `MsgBox`.

Un resultado posible de esta macro podría ser:



- `FullName`. Obtiene el nombre de la aplicación y su camino —o ruta de acceso— completo. La sintaxis es:

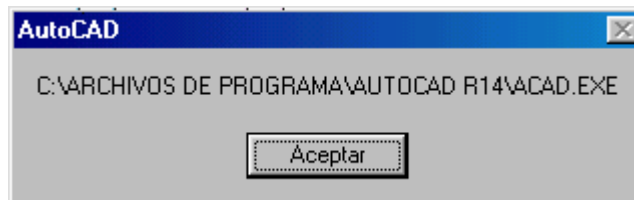
```
StrNombre = ObjAplicación.FullName
```

Siendo *StrTítulo* declarado como *String*.

Si las tres últimas líneas de la macro de ejemplo anterior se cambiaran por las siguientes:

```
Dim NomApli As String
NomApli = Acad.FullName
MsgBox NomApli
```

un resultado posible sería:



- **Left**. Esta propiedad permite obtener y/o asignar la distancia desde el lado izquierdo de la ventana principal de la aplicación hasta el lado izquierdo de la pantalla; se refiere pues a la coordenada X de la esquina superior izquierda de la ventana de la aplicación.

NOTA: El origen de coordenadas en los monitores comienza en la esquina superior izquierda de la pantalla (coordenada 0,0).

La sintaxis para asignar una distancia con *Left* es:

```
ObjAplicación.Left = IntDistanciaL
```

y para obtener la actual:

```
IntDistanciaL = ObjAplicación.Left
```

- **LocaleID**. Obtiene el ID (identificador) local para la sesión actual de **AutoCAD**. El ID local es definido por Microsoft para los entornos Windows 95/98 y Windows NT.

La sintaxis es:

```
LngID = ObjAplicación.LocaleID
```

Una respuesta coherente puede ser, por ejemplo: 1033.

- **Path**. Obtiene el camino o ruta de acceso a la aplicación (o documento), sin el nombre de archivo incluido. La sintaxis es:

```
StrCamino = ObjNoGráfico.Path
```

StrCamino se declarará como *String* evidentemente.

- **Preferences**. Obtiene el objeto de preferencias de aplicación (del que ya hablaremos). La sintaxis es:

```
ObjPreferencias = ObjAplicación.Preferences
```

- **Top**. Esta propiedad permite obtener y/o asignar la distancia desde el lado superior de la ventana principal de la aplicación hasta el lado superior de la pantalla; se refiere pues a la coordenada Y de la esquina superior izquierda de la ventana de la aplicación.

NOTA: El origen de coordenadas en los monitores comienza en la esquina superior izquierda de la pantalla (coordenada 0,0).

La sintaxis para asignar una distancia con **Top** es:

```
ObjAplicación.Top = IntDistanciaT
```

y para obtener la actual:

```
IntDistanciaT = ObjAplicación.Top
```

- **Version**. Devuelve la versión de **AutoCAD** que se está utilizando. Es igual a la variable de sistema **ACADVER** del programa. Su sintaxis:

```
StrVersión = ObjAplicación.Version
```

Veremos ahora los métodos de este objeto **Application**.

- **GetInterfaceObject**. Acepta el ID de un programa e intenta cargarlo en **AutoCAD** como un *InProcServer*.

Esta propiedad permite conectar un servidor ActiveX Automation. Su sintaxis:

```
Set ObjInterfaz = ObjAplicación.GetInterfaceObject(IDPrograma)
```

ObjInterfaz es un objeto declarado como **Object**.

```
Dim Poly as Object  
Set Poly = Acad.GetInterfaceObject("Polycad.Application")
```

- **ListADS**. Devuelve una lista de las aplicaciones ADS cargadas. Su sintaxis es:

```
VarListaADS = ObjAplicación.ListADS
```

VarListaADS estará declarada como **Variant**. Guardará una matriz de elementos con cada uno de los valores de la lista. Para acceder a ellos recordemos que podemos utilizar índices.

- **ListARX**. Devuelve una lista de las aplicaciones ARX cargadas. Su sintaxis es:

```
VarListaARX = ObjAplicación.ListARX
```

Las mismas consideraciones que para el método anterior.

- **LoadADS**. Carga una aplicación ADS. Debemos proporcionarle el nombre y ruta (si fuera necesaria) de la aplicación en cuestión. Veamos la sintaxis.

```
Call ObjAplicación.LoadADS(StrNombreAplicADS)
```

Nombre y ruta habrán de ser una cadena.

- LoadARX. Carga una aplicación ARX. Deberemos proporcionarle el nombre y ruta (si fuera necesaria) de la aplicación en cuestión. Veamos la sintaxis.

`Call ObjAplicación.LoadARX(StrNombreAplicARX)`

Nombre y ruta habrán de ser una cadena.

- Quit. Cierra la aplicación **AutoCAD**. Sintaxis:

`ObjAplicación.Quit`

- UnloadADS. Las mismas consideraciones que para LoadADS pero para descargar la aplicación ADS.

- UnloadARX. Las mismas consideraciones que para LoadARX pero para descargar la aplicación ARX.

DOCE.7. EL DOCUMENTO ACTUAL ACTIVO

Tras acceder al más alto escalafón que es la aplicación (así como a sus propiedades y métodos) descendemos un escalón para adentrarnos en el documento actual activo: ActiveDocument. Según la sintaxis de declaraciones y asignaciones que vemos manejando desde el principio del **MÓDULO**:

```
Dim AcadDoc As Object
Dim AcadModel As Object

y

Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument
Set AcadModel = AcadDoc.ModelSpace
```

y dejando aparte la utilizada anteriormente para acceder a la propia aplicación, AcadDoc es como vemos nuestra variable para acceder al documento actual activo.

Al documento actual como tal también pueden aplicársele una serie de propiedades y métodos que operarán con él; son los puntos que estudiaremos bajo este epígrafe. Vamos pues.

Propiedades del objeto de documento actual:

ActiveDimStyle	ActiveLayer	ActiveLinetype
ActivePViewport	ActiveSelectionSet	ActiveSpace
ActiveTextStyle	ActiveUCS	ActiveViewport
Application	Blocks	Dictionaries
DimStyles	ElevationModelSpace	ElevationPapersSpace
FullName	Groups	Layers
Limits	Linetypes	ModelSpace
Path	Plot	ReadOnly
RegisteredApplications	Saved	SelectionSets
TextStyles	UserCoordinateSystems	Utility
ViewPorts	Views	

Métodos del objeto de documento actual:

AuditInfo	Export	GetVariable	HandleToObject
Import	LoadShapeFile	New	ObjectIDToObject

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en Visual Basic orientada a AutoCAD (VBA)

Open
SaveAs

PurgeAll
Setvariable

Regen
WBlock

Save

Veamos primero, como siempre, las propiedades.

- **ActiveDimStyle**. Asigna u obtiene el estilo de acotación actual. Se estudiarán más detalles sobre los estilos de acotación en su sección correspondiente.

La sintaxis para asignar:

```
ObjDocumento.ActiveDimStyle = ObjEstiloAcotación
```

La sintaxis para obtener:

```
Set ObjEstiloAcotación = ObjDocumento.ActiveDimStyle
```

ObjEstiloAcotación es un objeto de estilo de acotación del que ya se hablará.

- **ActiveLayer**. Asigna u obtiene la capa actual. Se estudiarán más detalles sobre las capas en su sección correspondiente.

La sintaxis para asignar:

```
ObjDocumento.ActiveLayer = ObjCapa
```

La sintaxis para obtener:

```
Set ObjCapa = ObjDocumento.ActiveLayer
```

ObjCapa es un objeto de capa del que ya se hablará.

- **ActiveLinetype**. Asigna u obtiene el tipo de línea actual. Se estudiarán más detalles sobre los tipos de línea en su sección correspondiente.

La sintaxis para asignar:

```
ObjDocumento.ActiveLinetype = ObjTipoLínea
```

La sintaxis para obtener:

```
Set ObjTipoLínea = ObjDocumento.ActiveLinetype
```

ObjTipoLínea es un objeto de tipo de línea del que ya se hablará.

- **ActivePViewport**. Asigna u obtiene la ventana de Espacio Papel actual. Se estudiarán más detalles sobre las ventanas de Espacio Papel en su sección correspondiente.

La sintaxis para asignar:

```
ObjDocumento.ActivePViewport = ObjVentanaPapel
```

La sintaxis para obtener:

```
Set ObjVentanaPapel = ObjDocumento.ActivePViewport
```

ObjVentanaPapel es un objeto de ventana de Espacio Papel del que ya se hablará.

Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

- `ActiveSelectionSet`. Obtiene exclusivamente el conjunto de selección actual. Se estudiarán más detalles sobre los conjuntos de selección en su sección correspondiente.

La sintaxis para asignar:

```
ObjDocumento.ActiveSelectionSet = ObjConjuntoSel
```

La sintaxis para obtener:

```
Set ObjConjuntoSel = ObjDocumento.ActiveSelectionSet
```

`ObjConjuntoSel` es un objeto de conjunto de selección del que ya se hablará.

- `ActiveSpace`. Conmuta el espacio actual entre Espacio Modelo Mosaico y Espacio Papel/Modelo Flotante.

La sintaxis para realizar la conmutación:

```
ObjDocumento.ActiveSpace = IntEspacio
```

Si el actual es Espacio Modelo Mosaico se cambia a Espacio Papel/Modelo Flotante, y viceversa.

La sintaxis para obtener el valor del espacio actual:

```
IntEspacio = ObjDocumento.ActiveSpace
```

`IntEspacio` es un valor Integer que puede almacenar las siguientes constantes:

`acModelSpace` `acPaperSpace`

- `ActiveTextStyle`. Asigna u obtiene el estilo de texto actual. Se estudiarán más detalles sobre los estilos de texto en su sección correspondiente.

La sintaxis para asignar:

```
ObjDocumento.ActiveTextstyle = ObjEstiloText
```

La sintaxis para obtener:

```
Set ObjEstiloTexto = ObjDocumento.ActiveTextStyle
```

`ObjEstiloTexto` es un objeto de estilo de texto del que ya se hablará.

- `ActiveUCS`. Asigna u obtiene el SCP actual. Se estudiarán más detalles sobre los SCP en su sección correspondiente.

La sintaxis para asignar:

```
ObjDocumento.ActiveUCS = ObjSCP
```

La sintaxis para obtener:

```
Set ObjSCP = ObjDocumento.ActiveUCS
```


Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

ObjSCU es un objeto de SCP (Sistema de Coordenadas Personal) del que ya se hablará.

- *ActiveViewport*. Asigna u obtiene la ventana mosaico actual. Se estudiarán más detalles sobre las ventanas de Espacio Modelo Mosaico en su sección correspondiente.

La sintaxis para asignar:

```
ObjDocumento.ActiveViewport = ObjVentanaModelo
```

La sintaxis para obtener:

```
Set ObjVentanaModelo = ObjDocumento.ActiveViewport
```

ObjVentanaModelo es un objeto de ventana de Espacio Modelo Mosaico del que ya se hablará.

- *Blocks*. Obtiene exclusivamente el objeto de colección de bloques del dibujo. Acerca de los bloques hablaremos largo y tendido en su momento.

La sintaxis para utilizar esta propiedad es:

```
Set ObjColBloques = ObjDocumento.Blocks
```

ObjColBloques es un objeto de colección de bloques del que ya se hablará.

- *Dictionaries*. Obtiene exclusivamente el objeto de colección de diccionarios del documento activo. Acerca de los diccionarios hablaremos en su momento.

La sintaxis para utilizar esta propiedad es:

```
Set ObjColDiccionarios = ObjDocumento.Dictionaries
```

ObjColDiccionarios es un objeto de colección de diccionario del que ya se hablará.

- *DimStyles*. Obtiene exclusivamente el objeto de colección de estilos de acotación del dibujo. Acerca de los estilos de acotación hablaremos en su momento.

La sintaxis para utilizar esta propiedad es:

```
Set ObjColEstilosAcotación = ObjDocumento.DimStyles
```

ObjColEstilosAcotación es un objeto de colección de estilos de acotación del que ya se hablará.

- *ElevationModelSpace*. Obtiene y/o asigna la elevación en el Espacio Modelo.

La sintaxis para asignar:

```
ObjDocumento.ElevationModelSpace = DblElevaciónM
```

La sintaxis para obtener:

```
DblElevaciónM = ObjDocumento.ElevationModelSpace
```

DblElevaciónM será declarada como Double (doble precisión).

Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

- **ElevationPaperSpace**. Obtiene y/o asigna la elevación en el Espacio Papel. La sintaxis para asignar:

```
ObjDocumento.ElevationPaperSpace = DblElevaciónP
```

La sintaxis para obtener:

```
DblElevaciónP = ObjDocumento.ElevationPaperSpace
```

DblElevaciónP será declarada como Double.

- **Groups**. Obtiene el objeto de colección de grupos. Sobre los grupos se hablará más adelante.

La sintaxis para utilizar esta propiedad es:

```
Set ObjColGrupos = ObjDocumento.Groups
```

ObjColGrupos es un objeto de colección de estilos de acotación del que ya se hablará.

- **Layers**. Obtiene el objeto de colección de capas. Sobre las capas se hablará más adelante.

La sintaxis para utilizar esta propiedad es:

```
Set ObjColCapas = ObjDocumento.Layers
```

ObjColCapas es un objeto de colección de capas del que ya se hablará.

- **Limits**. Permite asignar y obtener los límites del dibujo actual.

La sintaxis para asignar unos límites es:

```
ObjDocumento.Limits = DblLímites
```

La sintaxis para obtener unos límites es:

```
VarLímites = ObjDocumento.Limits
```

Para asignar límites al dibujo deberemos crear una matriz de cuatro valores tipo Double. Para recoger unos límites habilitaremos una variable definida como Variant. Sabemos que después mediante índices podemos acceder a cada valor. El primer par de valores define las coordenadas X e Y del límite inferior izquierdo y, el segundo par de valores, las coordenadas X e Y del límite superior derecho.

- **Linetypes**. Devuelve el objeto de colección de tipos de línea. De los tipos de línea hablaremos más adelante.

La sintaxis para utilizar esta propiedad es:

```
Set ObjColTiposLínea = ObjDocumento.Linetypes
```

ObjColTiposLínea es un objeto de colección de tipos de línea del que ya se hablará.

- **ModelSpace**. Devuelve el objeto de colección de Espacio Modelo. De esta colección ya hablaremos más adelante, sin embargo es una sintaxis que nos sonará de utilizarla al principio de los programas:

```
Set ObjColEspacioM = ObjDocumento.ModelSpace
```

- **Plot**. Obtiene el objeto de trazado del documento actual. Este objeto será estudiado más adelante. Su sintaxis de uso es:

```
Set ObjTrazado = ObjDocumento.Plot
```

ObjTrazado es un objeto de trazado del que ya se hablará.

- **ReadOnly**. Obtiene la condición que tiene un documento de ser de sólo lectura o de lectura y escritura. La sintaxis para usar esta propiedad, por cierto de sólo lectura, es:

```
BooCondición = ObjDocumento.ReadOnly
```

BooCondición habrá sido declarada como Boolean. Si almacena True el documento es de sólo lectura; si almacena False el documento es de lectura y escritura.

- **RegisteredApplications**. Obtiene la colección de aplicaciones registradas. Esta colección será estudiada más adelante. Su sintaxis de uso es:

```
Set ObjColAplicReg = ObjDocumento.RegisteredApplications
```

- **Saved**. Especifica si el documento actual tiene algún cambio realizado sin guardar. Se recoge en una variable Boolean: si el valor es True el documento posee cambios sin guardar; si es False el documento no posee cambios sin guardar.

```
BooCambios = ObjDocumento.Saved
```

- **SelectionSets**. Obtiene el objeto de colección de conjuntos de selección. Este objeto será estudiado más adelante. Su sintaxis de uso es:

```
Set ObjColConjuntosSel = ObjDocumento.SelectionSets
```

ObjColConjuntosSel es un objeto de colección conjuntos de selección del que ya se hablará.

- **TextStyles**. Obtiene el objeto de colección de estilos de texto. Este objeto será estudiado más adelante. Su sintaxis de uso es:

```
Set ObjColEstilosTexto = ObjDocumento.TextStyles
```

ObjColEstilosTexto es un objeto de colección de estilos de texto del que ya se hablará.

- **UserCoordinateSystems**. Obtiene el objeto de colección de SCPs. Este objeto será estudiado más adelante. Su sintaxis de uso es:

```
Set ObjColSCPs = ObjDocumento.UserCoordinateSystems
```

ObjColSCPs es un objeto de colección de Sistemas de Coordenadas Personales del que ya se hablará.

- **Utility.** Obtiene el objeto de utilidades para el documento actual activo. De este objeto hablaremos largo y tendido en su momento. Su sintaxis de uso es:

```
Set ObjUtil = ObjDocumento.Utility
```

ObjUtil es un objeto de utilidades del que ya se hablará.

- **Viewports.** Obtiene el objeto de colección de ventanas del documento actual activo. Su sintaxis de uso es:

```
Set ObjColVentanas = ObjDocumento.Viewports
```

ObjVentanas es un objeto de colección de ventanas del que ya se hablará.

- **Views.** Obtiene el objeto de colección de vistas del documento actual activo. Su sintaxis de uso es:

```
Set ObjColVistas = ObjDocumento.Views
```

ObjVistas es un objeto de colección de vistas del que ya se hablará.

Probablemente no logremos comprender ahora para que nos sirven todas estas propiedades que devuelven objetos y colecciones de objetos, pero en cuanto nos introduzcamos en la materia de las distintas colecciones se nos hará el camino más sencillo.

Básicamente lo que viene a significar lo podemos deducir de una de las propiedades que ya conocíamos: *ModelSpace*. Lo que en realidad hacemos al llamar a esta propiedad es llamar al objeto de colección de Espacio Modelo (llamado *ModelSpace*), que se encuentra bajo el documento actual activo (*ActiveDocument*) y bajo la aplicación (*AutoCAD.Application*). Tras la llamada podremos utilizar multitud de métodos y propiedades de este objeto de colección. Concretamente, y como ya dijimos, todos los métodos de dibujo de entidades (*AddLine*, *AddCircle*, *AddRaster*...) son métodos de la colección de Espacio Modelo, así como de Espacio Papel y bloques (se puede ver el la jerarquía del cuadro del inicio del **MÓDULO**). También existen propiedades que ya veremos.

Con los demás objetos y colecciones ocurrirá lo mismo: realizaremos la llamada al objeto de colección de capas, por ejemplo, para crear una nueva capa o para averiguar el número de capas que hay; o realizaremos la llamada a una capa concreta para cambiarle el color, averiguar su nombre, bloquearla... Digamos que todo está basado en algo parecido a las rutas de acceso o caminos de directorios: necesitamos incluir en ellas todas las carpetas y subcarpetas (niveles jerárquicos) en el orden correcto hasta llegar al objetivo final (propiedad o método). Por ello es muy importante entender (que no memorizar) el cuadro jerárquico del principio del **MÓDULO**; es esencial comprenderlo completa y correctamente.

A continuación estudiaremos los distintos métodos del documento activo.

- **AuditInfo.** Este método evalúa la integridad del dibujo actual. Para cada error **AutoCAD** provee de un mensaje de error al usuario. Además se puede elegir la opción de hacer que los errores se corrijan automáticamente.

La sintaxis de este método es:

```
Call ObjDocumento.AuditInfo(StrNombArch, BooCorregir)
```

donde *StrNombArch* es una cadena (tipo de dato *String*) que contendrá el nombre del archivo que se desea comprobar, y *BooCorregir* es un valor *Boolean* que determina si se desea la

corrección de errores automática: True corrige automáticamente; False no corrige automáticamente.

- **Export.** Exporta el dibujo actual de **AutoCAD** a formato SAT, WMF, EPS, DXF, 3DS y/o BMP.

La sintaxis de este método es:

`Call ObjDocumento.Export(StrNombArch, StrExtensión, ObjConjuntoSel)`

donde *StrNombArch* es una cadena (tipo de dato String) que contendrá el nombre del archivo, *StrExtensión* es otra cadena que contendrá tres caracteres especificando la extensión, y puede ser (mayúsculas o minúsculas):

SAT	WMF	EPS	DXF	3DS	BMP
-----	-----	-----	-----	-----	-----

Y *ObjConjuntoSel* exporta sólo los objetos contenidos en el conjunto de selección especificado. A crear conjuntos de selección aprenderemos más adelante. Este conjunto de selección no puede estar vacío. Si este valor es Null se exporta el dibujo completo.

- **GetVariable.** Obtiene el valor de la variable de sistema de **AutoCAD** especificada. La sintaxis es:

`VarValor = ObjDocumento.GetVariable(StrNombVar)`

Siendo *StrNombVar* una cadena (String) que representa el nombre de la variable en cuestión (mayúsculas o minúsculas), y *VarValor* una variable Variant donde se guardará el valor.

- **HandleToObject.** Obtiene el objeto que se corresponde con el código de objeto dado. La sintaxis es:

`ObjNoGráfico = ObjDocumento.HandleToObject(StrCódigoObj)`

ObjNoGráfico es Object y *StrCódigoObj* es String.

- **Import.** Funciona de manera inversa a **Export**, es decir, importa un dibujo en determinado formato gráfico (SAT, WMF, EPS, DXF, 3DS o BMP) a un dibujo de **AutoCAD** .DWG. La sintaxis de **Import** es:

`Set ObjDocumento2 = ObjDocumento1.Import(StrNombArch, DblPtoIns, DblEsc)`

ObjDocumento2 es un objeto (declarado como Object) que es el valor de salida del método. Este objeto es de tipo de documento evidentemente. *StrNombArch* es una cadena con el nombre y ruta del archivo en cuestión, *DblPtoIns* un array de tres valores Double que representan las coordenadas del punto de inserción, y *DblEsc* el factor de escala de inserción (Double también).

- **LoadShapeFile.** Carga un archivo de formas compilado .SHX. La sintaxis de este método es:

`Call ObjDocumento.LoadShapeFile(StrNombArch)`

StrNombArch como en métodos anteriores.

Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

NOTA: Para insertar formas de un archivo de formas cargado recordemos el método AddShape de las colecciones de Espacio Modelo, Espacio Papel y bloques.

- New. Crea un nuevo documento y lo establece como documento activo. Para utilizar este método usaremos la sintaxis siguiente:

```
Set ObjDocumento2 = ObjDocumento1.New(StrNombrePlantilla)
```

ObjDocumento2 es el valor de salida (un objeto de documento). *StrNombrePlantilla* es una cadena String que contiene el camino completo y nombre de la plantilla (recordemos: archivos .DWT) con la que se abrirá el dibujo nuevo.

NOTA: Las plantillas por defecto (las que se abren en **AutoCAD** cuando no se indica plantilla alguna) son ACADISO.DWT o ACAD.DWT, dependiendo de si trabajamos en milímetros (unidades métricas) o pulgadas (unidades inglesas).

- ObjectIDToObject. Obtiene el objeto que se corresponde con el ID de objeto dado. La sintaxis es:

```
ObjNoGráfico = ObjDocumento.ObjectIDToObject(LngIDObjeto)
```

ObjNoGráfico es Object y *LngIDObjeto* es Long.

- Open. Abre un dibujo existente que se convierte en el documento actual. Hay que pasarle como argumento String la ruta completa y nombre del archivo. La sintaxis de este método es:

```
Call ObjDocumento.Open(StrNombArch)
```

- PurgeAll. Limpia todo lo no utilizado en el dibujo. El método realiza la misma acción que utilizar el comando LIMPIA de **AutoCAD** con la opción TODO (PURGE, en inglés, con la opción ALL). Su sintaxis es:

```
ObjDocumento.PurgeAll
```

- Regen. Regenera el dibujo completo y recalcula las coordenadas de la pantalla y la resolución de la vista para todos los objetos. También reindexa la Base de Datos de dibujo para una óptima representación y selección de los objetos. Su sintaxis es:

```
ObjDocumento.Regen(IntVentanas)
```

IntVentanas es un valor Integer que admite las siguientes constantes:

acActiveViewport acAllViewports

haciendo referencia a una regeneración en la ventana gráfica actual (REGEN) o en todas las ventanas gráficas (REGENT, REGENALL en inglés).

- Save. Guarda los cambios del documento actual activo. La sintaxis de uso es:

```
ObjDocumento.Save
```

- SaveAs. Guarda los cambios del documento actual activo, pasando como argumento el nombre de archivo y camino de acceso completo. El nuevo documento es el documento activo. La sintaxis de uso es:

```
ObjDocumento.SaveAs(StrNombArch)
```

StrNombArch es una cadena (tipo String).

- **SetVariable**. Asigna el valor especificado como segundo argumento a la variable de sistema de **AutoCAD** especificada como primer argumento. La sintaxis es:

```
Call ObjDocumento.SetVariable(StrNombVar, VarValor)
```

Siendo *StrNombVar* una cadena (String) que representa el nombre de la variable en cuestión (mayúsculas o minúsculas), y *VarValor* una variable Variant donde se guardará el valor.

NOTA: Tras el siguiente método —que es el último— se estudia un ejemplo completo en el que podremos observar un trato correcto de las variables (el más típico quizá) a la hora de utilizar **GetVariable** y **SetVariable**.

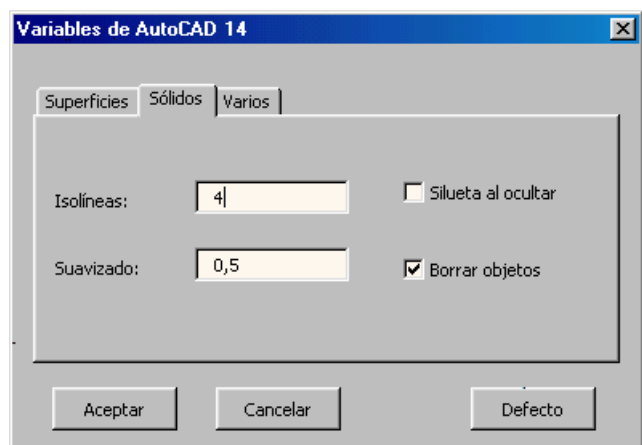
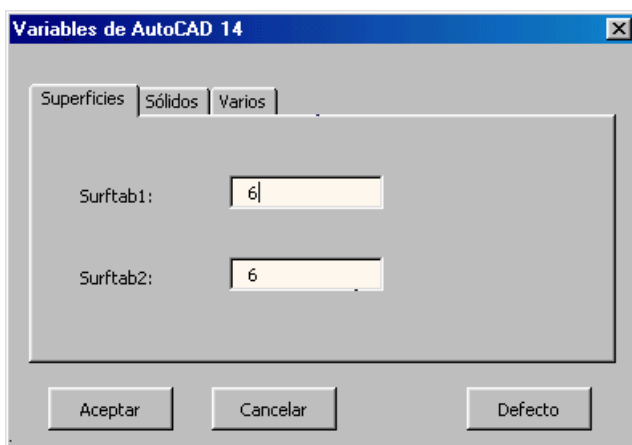
- **WBlock**. Crea un nuevo archivo de dibujo con el contenido del conjunto de selección indicado (al estilo de BLOQUEDISC, WBLOCK en inglés). La sintaxis es:

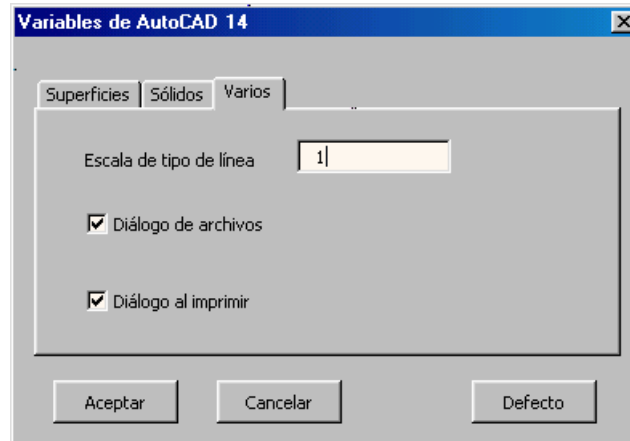
```
Call ObjDocumento.WBlock(StrNombArch, ObjConjuntoSel)
```

Siendo *StrNombVar* una cadena (String) que representa el nombre del archivo en cuestión (y su ruta), y *ObjConjuntoSel* un objeto de conjunto de selección que guarda las entidades con las que se creará un nuevo archivo. Este nuevo archivo no pasa a ser el documento activo, evidentemente.

Y una vez vistas propiedades y vistos métodos, exponemos un ejemplo muy interesante en el que, aunque no se encuentren muchas de las características vistas hasta este momento —ya que hemos de estudiar las colecciones para avanzar más—, nos ayudará a comprender un poco más la programación en VBA para **AutoCAD**.

El programa en cuestión consta de un cuadro de diálogo con varias pestañas en las que existen diferentes características de **AutoCAD** referidas a variables de sistema. Desde este cuadro podremos actuar sobre dichas variables de una forma versátil y rápida:





Y ahora se proporciona el listado completo del programa:

Option Explicit

Dim AcadDoc As Object

```
Private Sub buttonAceptar_Click()  
    Call AcadDoc.SetVariable("surftab1", CInt(Val(boxSurftab1.Text)))  
    Call AcadDoc.SetVariable("surftab2", CInt(Val(boxSurftab2.Text)))  
    Call AcadDoc.SetVariable("isolines", CInt(Val(boxIsolíneas.Text)))  
    Call AcadDoc.SetVariable("facetres", Val(boxSuavizado.Text))  
    Call AcadDoc.SetVariable("dispsilh", Abs(checkSilueta.Value))  
    Call AcadDoc.SetVariable("delobj", Abs(checkBorrar.Value))  
    Call AcadDoc.SetVariable("ltscale", Val(boxEscala.Text))  
    Call AcadDoc.SetVariable("filedia", Abs(checkArchivos.Value))  
    Call AcadDoc.SetVariable("cmddia", Abs(checkImprimir.Value))  
End  
End Sub
```

```
Private Sub buttonCancelar_Click()  
    End  
End Sub
```

```
Private Sub buttonDefecto_Click()  
    boxSurftab1.Text = "16"  
    boxSurftab2.Text = "16"  
    boxIsolíneas.Text = "4"  
    boxSuavizado.Text = "2.5"  
    checkSilueta.Value = 0  
    checkBorrar.Value = 1  
    boxEscala.Text = "1"  
    checkArchivos.Value = 1  
    checkImprimir.Value = 1  
End Sub
```

```
Private Sub UserForm_Initialize()  
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument  
  
    boxSurftab1.Text = Str(AcadDoc.GetVariable("surftab1"))  
    boxSurftab2.Text = Str(AcadDoc.GetVariable("surftab2"))  
    boxIsolíneas.Text = Str(AcadDoc.GetVariable("isolines"))  
    boxSuavizado.Text = Str(AcadDoc.GetVariable("facetres"))
```



```
checkSilueta.Value = AcadDoc.GetVariable("dispsilh")
checkBorrar.Value = AcadDoc.GetVariable("delobj")
boxEscala.Text = Str(AcadDoc.GetVariable("ltscale"))
checkArchivos.Value = AcadDoc.GetVariable("filedia")
checkImprimir.Value = AcadDoc.GetVariable("cmddia")
End Sub
```

Comentémoslo un poco. Tras declarar e inicializar únicamente un objeto para el documento actual (no necesitamos más), se escriben en las distintas casillas (al inicializar el formulario) los diferentes valores de las variables de sistema. Para ello se convierten a cadena con `Str`. Es conveniente realizar esto, ya que si no utilizamos `Str` vamos a tener posteriores problemas con la coma (,) y el punto (.) como separadores decimales; convirtiendo con `Str` pasará todo como tal, es decir, con un punto decimal que es lo que se utiliza normalmente.

Las variables que tienen un valor booleano (0 ó 1) se pueden igualar directamente a la propiedad `Value` de las casillas de verificación, porque éstas reconocen sin problema alguno los valores y establecen la casilla dependiendo de ellos (1 marcada y 0 sin marcar).

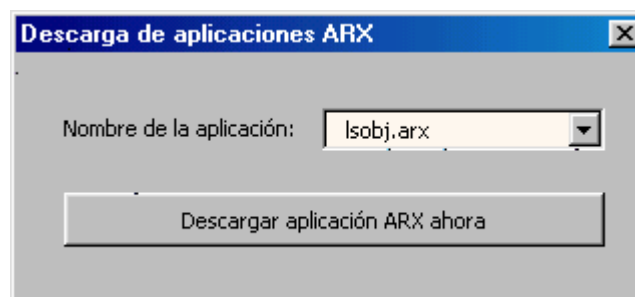
A la hora de realizar la operación inversa, es decir, leer los valores de las casillas e introducirlos en las variables de sistema, deberemos observar algunas consideraciones. Las variables que necesiten un valor `Double` (tipo `FACETRES`) pueden ser convertidas a valor numérico directamente con `Val`; las que necesiten un valor entero (`SURFTAB1` por ejemplo) se convierte con `Val` y se transforman a `Integer` con `Cint`; las variables que necesiten un valor booleano (tipo `FILEDIA`) se suministran directamente leídas de la casilla de verificación. Pero cuidado, cuando una casilla está activada, al leerla para `SetVariable` se devuelve el valor -1, por lo que habremos de asegurarnos siempre introduciendo una función `Abs` por delante, que devuelve el valor absoluto sin signo.

Por último y como vemos, el botón *Defecto* rellena las casillas con unos valores que se estiman por defecto, y el botón *Cancelar* simplemente termina el programa.

Evidentemente faltaría un pequeño control de errores en el programa que evitara introducir valores decimales donde no se puede y demás. Con ello y algunos detalles más el programa estaría completo.

2ª fase intermedia de ejercicios

- Programar un ejercicio que permita descargar las aplicaciones ARX cargadas actualmente. El letrero de diálogo presentará una lista desplegable con las aplicaciones cargadas y, tras escoger una de ellas se procederá a su descarga haciendo clic en un botón de acción. El cuadro puede ser este:



DOCE.8. LAS COLECCIONES Y SUS OBJETOS

Pasaremos ahora al estudio más profundo de las diversas colecciones de objetos que posee VBA para **AutoCAD**, mencionadas en diversos puntos de la sección anterior.

Un objeto *colección* es un conjunto de objetos a los que se puede hacer referencia como grupo. Bajo esta sección veremos las distintas maneras de tratar las colecciones VBA para **AutoCAD**, que son: objetos del Espacio Modelo, objetos del Espacio Papel, bloques, diccionarios, estilos de acotación, grupos, capas, tipos de línea, aplicaciones registradas, conjuntos de selección, estilos de texto, SCPs, vistas y ventanas.

Cada colección será explicada con sus métodos, sus propiedades y sus elementos simples. Esto último se refiere a los elementos integrantes de cada colección; así por ejemplo, la colección de capas está formada por objetos simples que son, evidentemente, las capas. Tanto la colección de capas, como cada una de ellas dispondrán de propiedades y métodos propios.

NOTA: Las colecciones nacen directamente del documento actual activo, como se puede ver en la plantilla del comienzo de este **MÓDULO**.

DOCE.8.1. Colección de objetos de Espacio Modelo

La colección de objetos de Espacio Modelo (ModelSpace) es en realidad una colección especial de bloques que contiene todas las entidades del Espacio Modelo de **AutoCAD**. Veamos sus propiedades y métodos.

Propiedades de la colección de objetos de Espacio Modelo:

Application	Count	Name
-------------	-------	------

Métodos de la colección de objetos de Espacio Modelo:

Add3DFace	Add3DMesh	Add3DPoly	AddArc
AddAttribute	AddBox	AddCircle	AddCone
AddCustomObject	AddCylinder	AddDimAligned	AddDimAngular
AddDimDiametric	AddDimOrdinate	AddDimRadial	AddDimRotated
AddEllipse	AddEllipticalCone	AddEllipticalCylinder	AddExtrudedSolid
AddExtrudedSolidAlongPath		AddHatch	AddLeader
AddLightWeightPolyline		AddLine	AddMText
AddPoint	AddPolyline	AddRaster	AddRay
AddRegion	AddRevolvedSolid	AddShape	AddSolid
AddSphere	AddSpline	AddText	AddTolerance
AddTorus	AddTrace	AddWedge	AddXLine
InsertBlock	Item		

Como vemos, al menos los métodos nos suenan casi todos, y es que no son otros que los explicados en la sección **DOCE.5.** de este **MÓDULO**. Veamos lo nuevo.

En cuestión de propiedades aparece una nueva:

- **Count.** Obtiene el número de objetos existentes en una colección (grupo y conjunto de selección también).

Esta propiedad de sólo lectura tiene como sintaxis:

```
IntNúmero = ObjNoGráfico.Count
```

Donde *IntNúmero* será una variable declarada evidentemente como *Integer*.

La ejecución de la siguiente macro devolvería el número de objetos existente actualmente en el Espacio Modelo:

```
Option Explicit
```

```
Dim Acad As Object  
Dim AcadDoc As Object  
Dim AcadModel As Object
```

```
Sub Macro()  
    Set Acad = GetObject(, "AutoCAD.Application")  
    Set AcadDoc = Acad.ActiveDocument  
    Set AcadModel = AcadDoc.ModelSpace  
  
    Dim Num As Integer  
    Num = AcadModel.Count  
    MsgBox Num  
End Sub
```

Probemos a dibujar y borrar entidades para ver los distintos resultados de la propiedad *Count* y comprenderla bien.

Veamos ahora los dos nuevos métodos.

- **InsertBlock**. Este método inserta un bloque en una colección (Modelo, Papel o bloques) que se encuentre definido en la sesión actual de dibujo.

NOTA: Si se inserta un bloque dentro de otro (en colección de bloques) se crearán bloques anidados.

La sintaxis para utilizar este método es:

```
Set ObjRefBloque = ObjNoGráfico.InsertBlock(DblPtoIns, StrNombre, DblEscalaX,  
DblEscalaY, DblRotación)
```

DblPtoIns es una matriz de tres valores *Double* que especifica el punto de inserción para el bloque. *StrNombre* se refiere al nombre del bloque, si es una variable habrá sido declarada como *String*. *DblEscalaX* y *DblEscalaY* son dos valores *Double* que representan la escala de inserción del bloque en X e Y. *DblRotación* es un valor *Double* que dice relación al ángulo de rotación de inserción en radianes.

Veamos la siguiente macro:

```
Option Explicit
```

```
Dim Acad As Object  
Dim AcadDoc As Object  
Dim AcadModel As Object
```

```
Sub Macro()  
    Set Acad = GetObject(, "AutoCAD.Application")  
    Set AcadDoc = Acad.ActiveDocument  
    Set AcadModel = AcadDoc.ModelSpace
```

```
Dim Bloq As Object
Dim PtoIns(1 To 3) As Double
PtoIns(1) = 10: PtoIns(2) = 10: PtoIns(3) = 0
Set Bloq = AcadModel.InsertBlock(PtoIns, "circl", 1, 1, 0)
End Sub
```

El resultado será la inserción en el punto 10,10,0 (a escala 1:1 y con un ángulo de rotación de 0 radianes) del bloque definido con el nombre de `circl`. Este bloque deberá existir en el dibujo actual. Este resultado es un objeto de referencia a bloque, que ya vimos, por lo que podrá ser tratado como tal.

- Item. Este método obtiene el objeto especificado por un índice dentro de una colección (o también grupo o conjunto de selección). Su sintaxis es:

```
Set ObjGráfico = ObjNoGráfico.Item(VarÍndice)
```

Donde *VarÍndice* es un valor Variant que puede ser un entero o una cadena. Si es entero va desde 0 (como primer objeto) hasta *N-1* (como último objeto), siendo *N* el total de objetos en la colección (o grupo o conjunto de selección). Si *VarÍndice* es una cadena será el nombre de un bloque creado o similar.

El resultado es un objeto gráfico, definido como `Object`.

DOCE.8.2. Colección de objetos de Espacio Papel

La colección de objetos de Espacio Papel (`PaperSpace`) es, así como la de Espacio Modelo, en realidad una colección especial de bloques que contiene todas las entidades del Espacio Papel de **AutoCAD**. Las propiedades y los métodos son iguales que para Espacio Modelo.

Propiedades de la colección de objetos de Espacio Papel:

Application	Count	Name
-------------	-------	------

Métodos de la colección de objetos de Espacio Papel:

Add3DFace	Add3DMesh	Add3DPoly	AddArc
AddAttribute	AddBox	AddCircle	AddCone
AddCustomObject	AddCylinder	AddDimAligned	AddDimAngular
AddDimDiametric	AddDimOrdinate	AddDimRadial	AddDimRotated
AddEllipse	AddEllipticalCone	AddEllipticalCylinder	AddExtrudedSolid
AddExtrudedSolidAlongPath		AddHatch	AddLeader
AddLightWeightPolyline		AddLine	AddMText
AddPoint	AddPolyline	AddPViewport (*)	AddRaster
AddRay	AddRegion	AddRevolvedSolid	AddShape
AddSolid	AddSphere	AddSpline	AddText
AddTolerance	AddTorus	AddTrace	AddWedge
AddXLine	InsertBlock	Item	

Las propiedades y métodos expuestos están ya estudiados.

(*) NOTA: Únicamente un método aparece aquí nuevo (`AddPViewport`), pero vamos a posponer su estudio a la sección **DOCE.8.14.1.**, pues allí se habla de las ventanas de Espacio Modelo y este método está relacionado, ya que sirve para gestionar ventanas en Espacio Papel.

Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

Veamos un pequeño programa que dibuja una línea desde el punto 100,-50,0 hasta el punto 0,0,0 en Espacio Modelo o Espacio Papel, dependiendo de lo que el usuario elija. El listado es:

Option Explicit

```
Dim Acad As Object
Dim AcadDoc As Object
Dim AcadModel As Object
Dim AcadPaper As Object
Dim Línea As Object
```

```
Dim Pto1(1 To 3) As Double
Dim Pto2(1 To 3) As Double
```

```
Private Sub cbDibujar_Click()
    If obModelo.Value = True Then
        Set Línea = AcadModel.AddLine(Pto1, Pto2)
    Else
        Set Línea = AcadPaper.AddLine(Pto1, Pto2)
    End If
End Sub
```

```
Private Sub cbSalir_Click()
    End
End Sub
```

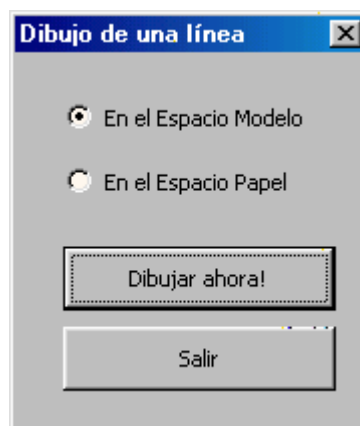
```
Private Sub UserForm_Initialize()
    Set Acad = GetObject(, "AutoCAD.Application")
    Set AcadDoc = Acad.ActiveDocument
    Set AcadModel = AcadDoc.ModelSpace
    Set AcadPaper = AcadDoc.PaperSpace

    Pto1(1) = 100: Pto1(2) = -50: Pto1(3) = 0
    Pto2(1) = 0: Pto2(2) = 0: Pto2(3) = 0
End Sub
```

El cuadro de diálogo (formulario) correspondiente podría ser el que se muestra a continuación.

La manera de utilizarlo es simple: sólo hay que escoger a cuál de las dos colecciones se van a añadir los objetos gráficos, a la colección de objetos de Espacio Modelo o a la colección de objetos de Espacio Papel. Tras esto se pulsa el botón *Dibujar ahora!* Para trazar la línea en el espacio correspondiente.

El botón *Salir* simplemente termina la ejecución del programa.



De estas dos colecciones vistas hasta ahora los objetos componentes simples son todos los ya estudiados: líneas, círculos, arcos, referencias de bloque...

DOCE.8.3. Colección de bloques y el objeto bloque

La colección de bloques (Blocks) la constituyen todos los bloques del documento actual activo. Las propiedades y métodos de esta colección las vemos seguidamente.

Propiedades de la colección de bloques:

Application Count

Métodos de la colección de bloques:

Add Item

Sólo un nuevo método hemos de explicar:

- Add. Este método añade un nuevo objeto a la colección indicada. Su sintaxis específica para la colección de bloques es:

```
Set ObjBloque = ObjColBloques.Add(DblPtoIns, StrNombre)
```

DblPtoIns es un valor Double para el punto de inserción (matriz de tres valores Double) y *StrNombre* es una cadena (String) para el nombre del bloque.

El objeto resultante de esta operación es un objeto de bloque, esto es, un bloque. Los objetos de bloque poseen las siguientes propiedades y métodos:

Propiedades de los objetos de bloque:

Application Name Origin

Métodos de los objetos de bloque:

Add3DFace	Add3DMesh	Add3DPoly	AddArc
AddAttribute	AddBox	AddCircle	AddCone
AddCustomObject	AddCylinder	AddDimAligned	AddDimAngular
AddDimDiametric	AddDimOrdinate	AddDimRadial	AddDimRotated
AddEllipse	AddEllipticalCone	AddEllipticalCylinder	AddExtrudedSolid
AddExtrudedSolidAlongPath		AddHatch	AddLeader
AddLightWeightPolyline		AddLine	AddMText
AddPoint	AddPolyline	AddRaster	AddRay
AddRegion	AddRevolvedSolid	AddShape	AddSolid
AddSphere	AddSpline	AddText	AddTolerance
AddTorus	AddTrace	AddWedge	AddXLine
Delete	InsertBlock	Item	

Veamos un ejemplo de macro VBA en el que se crea un bloque con una sola línea:

Option Explicit

```
Dim Acad As Object
Dim AcadDoc As Object
Dim AcadModel As Object
Dim AcadBloq As Object
Dim Lin As Object
```

```
Sub Macro()  
    Set Acad = GetObject(, "AutoCAD.Application")  
    Set AcadDoc = Acad.ActiveDocument  
    Set AcadModel = AcadDoc.ModelSpace  
    Set AcadBloq = AcadDoc.Blocks  
  
    Dim Bloq As Object  
    Dim PtoIns(1 To 3) As Double  
    Dim Pto2(1 To 3) As Double  
    PtoIns(1) = 10: PtoIns(2) = 10: PtoIns(3) = 0  
    Pto2(1) = 100: Pto2(2) = 100: Pto2(3) = 0  
    Set Bloq = AcadBloq.Add(PtoIns, "pppq")  
    Set Bloq = Bloq.AddLine(PtoIns, Pto2)  
End Sub
```

Lo que se debe hacer en primera instancia es crear el objeto de bloque en la colección de bloques (`Set Bloq = AcadBloq.Add(PtoIns, "pppq")`); es como si creáramos un bloque vacío. Después se le añaden entidades de dibujo, como en este caso una línea (`Set Bloq = Bloq.AddLine(PtoIns, Pto2)`) con los métodos ya estudiados, ya que estos métodos, como hemos visto, son propios también de la colección de objetos de bloque.

El nuevo método no estudiado aún de los objetos que son bloques es:

- **Delete.** Delete borra o elimina el objeto indicado. Hemos de indicar el nombre de dicho objeto (bloque, estilo de acotación, SCP...) como una cadena (String), pero esto no es necesario más que para la colección de ventanas:

`ObjNoGrafico.Delete(StrNombreObj)`

DOCE.8.4. Colección de diccionarios y el objeto diccionario

La colección de diccionarios (Dictionaries) la constituyen todos los objetos de diccionario del documento actual activo.

Propiedades de la colección de diccionarios:

Application	Count
-------------	-------

Métodos de la colección de diccionarios:

Add	Item
-----	------

Las propiedades y métodos expuestos están ya estudiados.

NOTA: El método Add hemos visto que tenía una sintaxis especial para la colección de bloques. Pues bien, para ella y para la colección de SCPs existen estas sintaxis especiales (esta última ya se verá en su momento). Para el resto de las colecciones la sintaxis del método Add es la siguiente:

`Set ObjComponenteCol = ObjColección.Add(StrNombre)`

StrNombre es el nombre (String) del objeto que se añadirá a la colección.

Sigamos. El objeto simple de la colección de diccionarios es un objeto de diccionario. Los objetos de diccionario poseen las siguientes propiedades y métodos:

Propiedades de los objetos de diccionario:

Application	Name
-------------	------

Métodos de los objetos de diccionario:

AddObject	Delete	GetName	GetObject
GetXData	Remove	Rename	Replace
SetXData			

Un diccionario es un sistema para almacenar y recuperar objetos con una cadena de palabras clave asociada. El objeto puede ser referenciado en un diccionario mediante su palabra clave. Un diccionario puede contener cualquier tipo de objeto, incluso otro diccionario.

NOTA: Este objeto de diccionario no tiene nada que ver con el diccionario ortográfico de corrección de **AutoCAD**. De éste hablaremos cuando estudiemos el objeto de preferencias.

Los nuevos métodos de los objetos de diccionario que hemos de estudiar son los que siguen a continuación.

- **AddObject.** Añade un nuevo objeto a un diccionario. Si la entrada ya existe es sustituida por el nuevo objeto, si no simplemente se agrega. Los dos argumentos para este método (nombre de objeto y palabra clave) han de ser cadenas (String). Veamos la sintaxis:

```
Set ObjNoGráfico = ObjDiccionario.AddObject(StrPalClave, StrNomObjeto)
```

- **GetName.** Devuelve el nombre (palabra clave) de un objeto de un diccionario:

```
StrNombre = ObjDiccionario.GetName(ObjEnDiccionario)
```

- **GetObject.** Devuelve el objeto que se corresponde con una palabra clave en un diccionario:

```
ObjEnDiccionario = ObjDiccionario.GetObject(StrPalClave)
```

- **Remove.** Elimina el objeto de un diccionario indicado por su palabra clave:

```
Call ObjDiccionario.Remove(StrPalClave)
```

- **Rename.** Cambia el nombre (palabra clave) del objeto de un diccionario indicado por su palabra clave:

```
Call ObjDiccionario.Rename(StrNombreAntiguo, StrNombreNuevo)
```

- **Replace.** Cambia una entrada en un diccionario por otra dada:

```
Call ObjDiccionario.Replace(StrPalClave, ObjObjetoNuevo)
```


DOCE.8.5. Colección de estilos de acotación y el objeto estilo de acotación

La colección de estilos de acotación (`DimStyles`) la constituyen todos los objetos de estilos de cota del documento actual activo, es decir, los estilos de acotación que todos conocemos.

Propiedades de la colección de estilos de acotación:

`Application` `Count`

Métodos de la colección de estilos de acotación:

`Add` `Item`

Las propiedades y métodos expuestos están ya estudiados.

El objeto simple de esta colección es un objeto de estilo de acotación. Los objetos de estilo de acotación poseen las siguientes propiedades y métodos:

Propiedades de los objetos de estilo de acotación:

`Application` `Name`

Métodos de los objetos de estilo de acotación:

`Delete` `GetXData` `SetXData`

Las propiedades y métodos expuestos están ya estudiados.

DOCE.8.6. Colección de grupos y el objeto grupo

La colección de grupos (`Groups`) la constituyen todos los objetos de grupo del documento actual activo.

Propiedades de la colección de grupos:

`Application` `Count`

Métodos de la colección de grupos:

`Add` `Item`

Las propiedades y métodos expuestos están ya estudiados.

El objeto simple de esta colección es un objeto de grupo. Los objetos de grupo son conjuntos de selección con nombre y poseen las siguientes propiedades y métodos:

Propiedades de los objetos de grupo:

<code>Application</code>	<code>Color</code>	<code>Count</code>	<code>EntityName</code>	<code>EntityType</code>
<code>Handle</code>	<code>Layer</code>	<code>LineType</code>	<code>LinetypeScale</code>	<code>Name</code>
<code>ObjectID</code>	<code>Visible</code>			

Métodos de los objetos de grupo:

AppendItems	Delete	GetXData	Highlight
Item	RemoveItems	SetXData	Update

Los métodos nuevos son:

- AppendItems. Añade una o más entidades gráficas al grupo:

`Call ObjGrupo.AppendItems(ObjGráficos)`

ObjGráficos es una matriz de objetos gráficos de **AutoCAD**.

- RemoveItems. Elimina los objetos especificados de un grupo (o también de un conjunto de selección):

`Call ObjGrupo.RemoveItems(ObjGráficos)`

ObjGráficos es una matriz de objetos gráficos de **AutoCAD**.

DOCE.8.7. Colección de capas y el objeto capa

La colección de capas (*Layers*) la constituyen todos los objetos de capa del documento actual activo, es decir, las capas del dibujo.

Propiedades de la colección de capas:

Application Count

Métodos de la colección de capas:

Add Item

Las propiedades y métodos expuestos están ya estudiados.

Por ejemplo, la siguiente macro crearía una nueva capa llamadas **EJES**:

Option Explicit

Dim AcadDoc As Object

Dim AcadModel As Object

Sub Macro()

 Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument

 Set AcadModel = AcadDoc.ModelSpace

 Call AcadDoc.Layers.Add("ejes")

End Sub

El objeto simple de esta colección es un objeto de capa. Los objetos de capa poseen las siguientes propiedades y métodos:

Propiedades de los objetos de capa:

Application	Color	Freeze	LayerOn	LineType
Lock	Name			

Métodos de los objetos de capa:

Delete GetXData SetXData

Las nuevas propiedades se explican a continuación.

- **Freeze.** La propiedad **Freeze** congela o descongela una capa. Es lo que se denomina en la versión castellana de **AutoCAD** la inutilización y reutilización de capas (representado por un Sol o un símbolo de hielo, dependiendo de la condición, en el cuadro de diálogo de capas). También nos permite obtener el estado en que se encuentra una capa —con respecto a este aspecto—.

La sintaxis para asignar un estado con **Freeze** a una capa es:

```
ObjCapa.Freeze = BooEstadoUtilización
```

BooEstadoUtilización es un valor Boolean que puesto a **True** indica que la capa se inutiliza (se congela) y puesto a **False** indica que la capa se reutiliza o utiliza (se descongela).

La sintaxis para obtener un estado con **Freeze** de una capa es:

```
BooEstadoUtilización = ObjCapa.Freeze
```

BooEstadoUtilización es un valor Boolean que si almacena **True** indica que la capa está inutilizada y si almacena **False** indica que la capa está reutilizada o utilizada.

Recordemos que la ventaja de inutilizar o congelar una capa es que es excluida de la regeneración, lo que no ocurre por ejemplo al desactivarla simplemente. También decir que una capa inutilizada no es trazada (por *plotter* o impresora).

- **LayerOn.** La propiedad **LayerOn** activa o desactiva una capa (representado por una bombilla encendida o apagada, dependiendo de la condición, en el cuadro de diálogo de capas). También nos permite obtener el estado en que se encuentra una capa —con respecto a este aspecto—.

La sintaxis para asignar un estado con **LayerOn** a una capa es:

```
ObjCapa.LayerOn = BooEstadoActivación
```

BooEstadoActivación es un valor Boolean que puesto a **True** indica que la capa se activa y puesto a **False** indica que la capa se desactiva.

La sintaxis para obtener un estado con **LayerOn** de una capa es:

```
BooEstadoActivación = ObjCapa.LayerOn
```

BooEstadoActivación es un valor Boolean que si almacena **True** indica que la capa está activada y si almacena **False** indica que la capa está desactivada.

Al desactivar una capa no aparecerá en pantalla ni en trazador, pero sus objetos seguirán respondiendo a las regeneraciones del dibujo.

- **Lock.** La propiedad **Lock** bloquea o desbloquea una capa (representado por un candado abierto o cerrado, dependiendo de la condición, en el cuadro de diálogo de capas).

También nos permite obtener el estado en que se encuentra una capa —con respecto a este aspecto—.

La sintaxis para asignar un estado con Lock a una capa es:

```
ObjCapa.Lock = BooEstadoBloqueo
```

BooEstadoBloqueo es un valor Boolean que puesto a True indica que la capa se bloquea y puesto a False indica que la capa de desbloquea.

La sintaxis para obtener un estado con Lock de una capa es:

```
BooEstadoBloqueo = ObjCapa.Lock
```

BooEstadoBloqueo es un valor Boolean que si almacena True indica que la capa está bloqueada y si almacena False indica que la capa está desbloqueada.

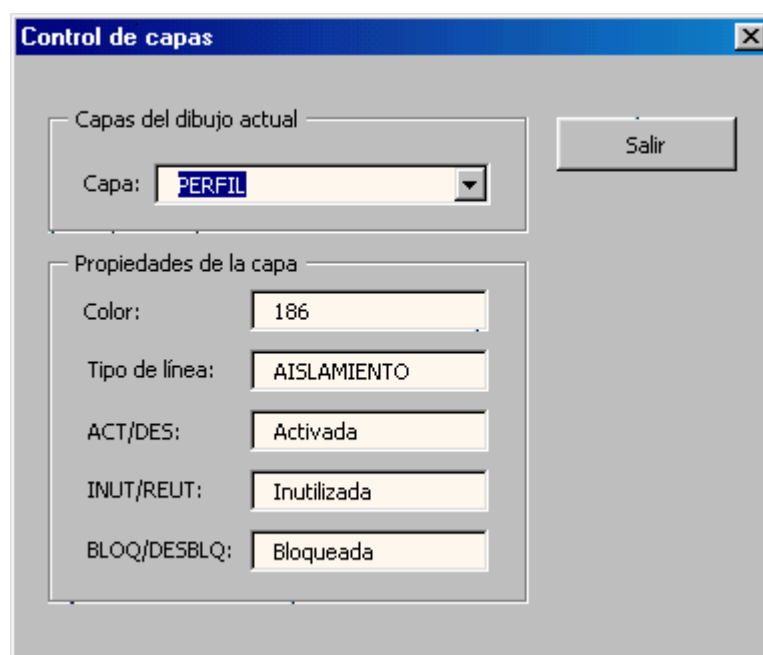
Al bloquear una capa no se podrá modificar, pero seguirá visible y respondiendo a acciones como, por ejemplo, la utilización de modos de referencia en ella.

Veamos un programa de ejemplo de manejo de capas. Este ejercicio maneja el cuadro de diálogo que se puede observar en esta misma página. Su misión es únicamente informativa, es decir, sólo informa del nombre y propiedades de las diferente capas del dibujo actual.

Al ejecutar el programa deberá rellenar la lista desplegable con las capas de la sesión de **AutoCAD** en curso y mostrará la primera de ellas (siempre la capa 0) escribiendo en los distintos cuadros de edición sus respectivas propiedades.

Al actuar sobre la lista desplegable eligiendo una capa, las casillas de edición se actualizarán con los nuevos valores.

Veamos pues el letrero de diálogo comentado y, a continuación, el listado del programa propiamente dicho.



El listado es el que sigue:

```
Option Explicit
Dim AcadDoc As Object
Dim AcadModel As Object
Dim AcadCapas As Object

Dim NumCapas As Integer

Private Sub cbSalir_Click()
    End
End Sub

Private Sub cbCapas_Change()
    Dim Índice As Integer
    Índice = cbCapas.ListIndex
    tbColor.Text = AcadCapas.Item(Índice).Color
    tbTipolin.Text = AcadCapas.Item(Índice).Linetype
    If AcadCapas.Item(Índice).LayerOn = True Then
        tbAD.Text = "Activada"
    Else
        tbAD.Text = "Desactivada"
    End If
    If AcadCapas.Item(Índice).Freeze = True Then
        tbIR.Text = "Inutilizada"
    Else
        tbIR.Text = "Utilizada"
    End If
    If AcadCapas.Item(Índice).Lock = True Then
        tbBD.Text = "Bloqueada"
    Else
        tbBD.Text = "Desbloqueada"
    End If
End Sub

Private Sub UserForm_Initialize()
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument
    Set AcadModel = AcadDoc.ModelSpace
    Set AcadCapas = AcadDoc.Layers

    NumCapas = AcadCapas.Count
    Dim i As Integer
    For i = 0 To Val(NumCapas - 1)
        cbCapas.AddItem AcadCapas.Item(i).Name
    Next i
    cbCapas.Text = AcadCapas.Item(0).Name
End Sub
```

Vemos como en cuanto se corre el programa se hace una llamada al procedimiento `cbCapas_Change`. Esto es debido a la línea `cbCapas.Text = AcadCapas.Item(0).Name`, que en sí ya realiza un cambio en la lista desplegable, por eso se ejecuta el procedimiento anteriormente dicho. Esta línea escribe en la casilla de la lista desplegable el primero de los elementos (la primera capa, la capa 0).

Antes de esto se introducen todos los nombres de capas en la lista. Vemos que no tiene ningún misterio y que simplemente es limitarse a seguir las pautas y diversas sintaxis de uso que vamos estudiando.

Con las extracción del color, tipo de línea y demás propiedades, vemos que el proceso es similar. La única variante es que se ha introducido un control para averiguar cuando

determinados valores son `True` o `False` y escribir textos en las casillas que resulten más lógicos al usuario que *Verdadero* y/o *Falso*.

NOTA: Como el cuadro es sólo informativo no nos olvidemos de establecer la propiedad `Style` de la lista desplegable a 2 (`fmStyleDropDownList`) y las propiedades `Locked` de las casillas de edición a `True`. Todo ello para no poder editar valores manualmente. Si quisiéramos que el usuario pudiera entrar en las casillas y variar los valores para cambiar las propiedades de las capas, únicamente habríamos de dejar estas propiedades especificadas como están y añadir algo más de código al programa con las diferentes sintaxis para asignar —en lugar de obtener— las propiedades de las capas.

DOCE.8.8. Colección de tipos de línea y el objeto tipo de línea

La colección de tipos de línea (`Linetypes`) la constituyen todos los objetos de tipo de línea del documento actual activo, es decir, los tipos de línea cargados en el dibujo.

Propiedades de la colección de tipos de línea:

<code>Application</code>	<code>Count</code>
--------------------------	--------------------

Métodos de la colección de tipos de línea:

<code>Add</code>	<code>Item</code>	<code>Load</code>
------------------	-------------------	-------------------

Como vemos hay un nuevo método:

- `Load`. Carga en el dibujo actual el tipo de línea especificado (`String`) del archivo de definición de tipos de línea (`.LIN`) especificado (`String`):

`Call ObjColTiposDeLínea.Load(StrNombreTipoLínea, StrNombreArchivo)`

NOTA: Si utilizamos el método `Add` se creará un nuevo tipo de línea con las propiedades por defecto.

El objeto simple de esta colección es un objeto de tipo de línea. Los objetos de tipo de línea poseen las siguientes propiedades y métodos:

Propiedades de los objetos de tipo de línea:

<code>Application</code>	<code>Description</code>	<code>Name</code>
--------------------------	--------------------------	-------------------

Métodos de los objetos de tipo de línea:

<code>Delete</code>	<code>GetXData</code>	<code>SetXData</code>
---------------------	-----------------------	-----------------------

La nueva propiedad es:

- `Description`. Asigna u obtiene una descripción (`String`) para el tipo de línea. Esta descripción es la que aparece en el archivo de definición `.LIN` y en el cuadro de gestión de tipos de línea, es la representación de la línea con caracteres ASCII (puntos, guiones, texto, etc...). Como máximo puede tener 47 caracteres.

La sintaxis para asignar una descripción es:

```
ObjTipoLínea.Description = StrDescripción
```

La sintaxis para obtener una descripción es:

```
StrDescripción = ObjTipoLínea.Description
```

Veamos la siguiente macro de ejemplo en la que se carga un tipo de línea y se obtiene su descripción:

```
Option Explicit
```

```
Dim AcadDoc As Object
```

```
Sub Macro()  
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument  
  
    If IsEmpty(AcadDoc.Linetypes.Item("trazo_y_punto")) Then  
        Call AcadDoc.Linetypes.Load("trazo_y_punto", "acadiso.lin")  
    End If  
    Dim TipoLin As Object  
    Set TipoLin = AcadDoc.Linetypes.Item("trazo_y_punto")  
    Dim Descr As String  
    Descr = TipoLin.Description  
    MsgBox Descr  
End Sub
```

NOTA: Podemos comprobar que con la única definición de `AcadDoc` nos sirve para este programa; no hay por qué declarar más variables de objeto de **AutoCAD** si no es necesario.

Si intentamos cargar un tipo de línea y este ya está cargado (o crear un capa que ya existe, un estilo de texto, etcétera), VBA devuelve un error. Para evitar esto hemos de comprobar primero si dicho tipo de línea —en este caso— está ya cargado. Para estos menesteres utilizamos la función de Visual Basic `IsEmpty` como se puede observar en el listado.

Tras cargar el tipo de línea, si no estaba ya cargado, se extrae su descripción y se muestra con un `MsgBox`. El resultado es el siguiente:



NOTA: Si el archivo `ACADISO.LIN` no hubiera estado en uno de los directorios de soporte de **AutoCAD**, habríamos de haber indicado la ruta completa al fichero; al igual que ocurría con los programas en AutoLISP, si recordamos.

DOCE.8.9. Colección de aplicaciones registradas y el objeto aplicación registrada

La colección de aplicaciones registradas (`RegisteredApplications`) la constituyen todos los objetos de aplicación registrada del documento actual activo.

Propiedades de la colección de aplicaciones registradas:

Application Count

Métodos de la colección de aplicaciones registradas:

Add Item

Las propiedades y métodos expuestos están ya estudiados.

El objeto simple de esta colección es un objeto de aplicación registrada. Los objetos de aplicación registrada poseen las siguientes propiedades y métodos:

Propiedades de los objetos de aplicación registrada:

Application Name

Métodos de los objetos de tipo de aplicación registrada:

Delete GetXData SetXData

Las propiedades y métodos expuestos están ya estudiados.

DOCE.8.10. Colección de conjuntos de selección y el objeto conjunto de selección

La colección de conjuntos de selección (*SelectionSets*) la constituyen todos los objetos de conjunto de selección del documento actual activo, o sea, los conjuntos actuales.

Propiedades de la colección de conjuntos de selección:

Application Count

Métodos de la colección de conjuntos de selección:

Add Item

Las propiedades y métodos expuestos están ya estudiados.

El objeto simple de esta colección es un objeto de conjunto de selección. Los objetos de conjunto de selección poseen las siguientes propiedades y métodos:

Propiedades de los objetos de conjunto de selección:

Application Count Name

Métodos de los objetos de conjunto de selección:

AddItems	Clear	Delete	Erase	Highlight
Item	RemoveItems	ScaleEntity	Select	SelectAtPoint
SelectByPolygon	SelectOnScreen	Update		

Veamos estos nuevos métodos aplicables únicamente a conjuntos de selección.

- **AddItems**. Añade objetos al conjunto de selección:

`Call ObjConjuntoSel.AddItem(ObjGráficos)`

ObjGráficos es una matriz de objetos gráficos.

NOTA: No confundir `AddItems` con el método `AppendItems` de los objetos de grupo ni con el método `AddItem` de las listas desplegables y cuadros de lista de Visual Basic.

El siguiente ejemplo de macro dibuja una línea, un círculo, crea un conjunto de selección y le añade ambos objetos creados:

Option Explicit

```
Dim AcadDoc As Object
Dim AcadModel As Object
Dim Línea As Object
Dim Círculo As Object
Dim Grupo1 As Object
```

```
Sub Macro()
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument
    Set AcadModel = AcadDoc.ModelSpace
    Dim Pto1(2) As Double
    Dim Pto2(2) As Double
    Pto1(0) = 100: Pto1(1) = 100: Pto1(2) = 0
    Pto2(0) = 200: Pto2(1) = 200: Pto2(2) = 0
    Set Línea = AcadModel.AddLine(Pto1, Pto2)
    Set Círculo = AcadModel.AddCircle(Pto1, 10)
    Set Grupo1 = AcadDoc.SelectionSets.Add("Prueba")
    Dim Objeto(1) As Object
    Set Objeto(0) = Línea: Set Objeto(1) = Círculo
    Call Grupo1.AddItem(Objeto)
End Sub
```

Si ahora yo quisiera eliminar —por ejemplo— ambos objetos guardados en el conjunto de selección `Grupo1`, simplemente haría:

```
Grupo1.Erase
```

- **Clear.** Elimina todos los elementos de un conjunto de selección:

`Call ObjConjuntoSel.Clear`

Esto es, vacía el conjunto.

- **Select.** Selecciona los objetos designados por el modo indicado y los introduce en un conjunto de selección. Su sintaxis de utilización es:

`Call ObjConjuntoSel.Select(IntModo, DblPto1, DblPto2, VarTipoFiltro, VarDatoFiltro)`

El argumento *IntModo* es un valor `Integer` que define el tipo o modo de selección que se ejecutará sobre los objetos. Admite también las siguientes constantes:

<code>acSelectionSetAll</code>	<code>acSelectionSetCrossing</code>
<code>acSelectionSetCrossingPolygon</code>	<code>acSelectionSetFence</code>
<code>acSelectionSetLast</code>	<code>acSelectionSetPrevious</code>
<code>acSelectionSetWindows</code>	<code>acSelectionSetWindowPolygon</code>

Las cuales se corresponden con los diferentes modos de designación de entidades de **AutoCAD**, esto es: *Todo*, *Captura*, *Polígono-Captura*, *Borde*, *Último*, *Previo*, *Ventana* y *Polígono-Ventana*.

Los dos argumentos de punto siguientes (matriz de tres valores Double) son obligatorios. En el caso de *Captura* y *Ventana* representan las dos esquinas opuestas por la diagonal. En el caso de *Polígono* (*Captura* o *Ventana*) y *Borde*, el primer argumento *DblPto1* debe contener una matriz con todos los puntos que definen el polígono o el borde, y se debe suministrar un argumento *DblPto2* ficticio; en este caso resulta preferible utilizar el método *SelectByPolygon* (que luego veremos). En el caso de modos que no necesitan puntos, como *Todo*, *Último* y *Previo*, los dos puntos se deben suministrar como argumentos ficticios que luego no se van a utilizar.

El argumento *VarTipoFiltro* (Variant) permite especificar un filtro de selección, de manera similar a la función de AutoLISP *SSGET*. El último argumento permite especificar filtros para los datos extendidos.

- *SelectAtPoint*. Selecciona el objeto que pasa por el punto indicado y lo añade a un conjunto de selección. Su sintaxis de utilización es:

```
Call ObjConjuntoSel.SelectAtPoint(DblPto, VarTipoFiltro, VarDatoFiltro)
```

DblPto es una matriz de tres valores Double. Los dos últimos argumentos como en el método *Select*.

- *SelectByPolygon*. Selecciona entidades incluidas dentro de un polígono definido por una lista de puntos suministrada (y las añade a un conjunto de selección). Su sintaxis de utilización es:

```
Call ObjConjuntoSel.SelectByPolygon(IntModo, DblListaPtos, VarTipoFiltro, VarDatoFiltro)
```

El argumento *IntModo* es un valor Integer que define el tipo o modo de selección que se ejecutará sobre los objetos. Admite también las siguientes constantes:

acSelectionSetCrossing
acSelectionSetFence

acSelectionSetCrossingPolygon

DblListaPtos es una matriz de puntos con las esquinas del polígono. Los dos últimos argumentos como en el método *Select*.

- *SelectOnScreen*. Permite seleccionar objetos en pantalla (tras el *prompt* *Designar objetos:*, o *Select objects:* en inglés) que serán añadidos a un conjunto de selección. Su sintaxis de utilización es:

```
Call ObjConjuntoSel.SelectOnScreen(VarTipoFiltro, VarDatoFiltro)
```

Los dos argumentos como en el método *Select*.

NOTA: En todos estos métodos de selección, los argumentos *VarTipoFiltro* y *VarDatoFiltro* son opcionales.

Veamos el siguiente ejemplo:

Option Explicit

```
Dim AcadDoc As Object
Dim AcadModel As Object
Dim Línea As Object
Dim Círculo As Object
Dim Grupo1 As Object
```

```
Sub Macro()
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument
    Set AcadModel = AcadDoc.ModelSpace

    Dim Pto1(2) As Double
    Dim Pto2(2) As Double
    Pto1(0) = 100: Pto1(1) = 100: Pto1(2) = 0
    Pto2(0) = 200: Pto2(1) = 200: Pto2(2) = 0
    Set Línea = AcadModel.AddLine(Pto1, Pto2)
    Set Círculo = AcadModel.AddCircle(Pto1, 10)
    Set Grupo1 = AcadDoc.SelectionSets.Add("Prueba")

    Call Grupo1.SelectOnScreen
    Grupo1.Erase
End Sub
```

Esta macro dibuja una línea y un círculo en pantalla. Tras crear un nuevo conjunto de selección (Grupo1) se permite seleccionar en pantalla los objetos que se insertarán en él. Después de esta designación los objetos se borran.

NOTA: Más adelante veremos como seleccionar también en pantalla puntos, ángulos, distancias, etcétera.

DOCE.8.11. Colección de estilos de texto y el objeto estilo de texto

La colección de estilos de texto (TextStyles) la constituyen todos los objetos de estilo de texto del documento actual activo, o sea, los estilos de texto creados actualmente.

Propiedades de la colección de estilos de texto:

Application Count

Métodos de la colección de estilos de texto:

Add Item

Las propiedades y métodos expuestos están ya estudiados.

El objeto simple de esta colección es un objeto de estilo de texto. Los objetos de estilo de texto poseen las siguientes propiedades y métodos:

Propiedades de los objetos de estilo de texto:

Application	BigFontFile	FontFile	Height
LastHeight	Name	ObliqueAngle	TextGenerationFlag
Width			

Métodos de los objetos de estilo de texto:

Delete GetXData SetXData

Veamos estas nuevas propiedades aplicables únicamente a estilos de texto.

- **BigFontFile.** Permite especificar u obtener el nombre de archivo de fuente grande (para tipos de letra asiáticos, por ejemplo) asociado con el estilo de texto. La sintaxis para asignar un archivo es:

```
ObjEstiloTexto.BigFontFile = StrNombreArch
```

donde *StrNombreArch* es una cadena (tipo de dato String).

La sintaxis para obtener o extraer el archivo de texto de fuente grande de un estilo de texto es:

```
StrNombreArch = ObjEstiloTexto.BigFontFile
```

donde *StrNombreArch* es una cadena (tipo de dato String).

NOTA: El único tipo de archivo válido es el de extensión .SHX.

- **FontFile.** Permite especificar u obtener el nombre de archivo de fuente principal o primario asociado con el estilo de texto. La sintaxis para asignar un archivo es:

```
ObjEstiloTexto.FontFile = StrNombreArch
```

donde *StrNombreArch* es una cadena (tipo de dato String).

La sintaxis para obtener o extraer el archivo de texto de fuente principal de un estilo de texto es:

```
StrNombreArch = ObjEstiloTexto.FontFile
```

donde *StrNombreArch* es una cadena (tipo de dato String).

NOTA: Los tipos de archivo válidos son .PFA, .PFB, .SHX, .TTF.

- **LastHeight.** Asigna u obtiene la última altura de texto utilizada. Para asignar:

```
ObjEstiloTexto.LastHeight = DblÚltimaAltura
```

donde *DblÚltimaAltura* es un valor Double.

La sintaxis para obtener:

```
DblÚltimaAltura = ObjEstiloTexto.LastHeight
```

donde *DblÚltimaAltura* es un valor Double.

DOCE.8.12. Colección de SCPs y el objeto SCP

La colección de SCPs (`UserCoordinateSystems`) la constituyen todos los objetos de SCP del documento actual activo, esto es, los SCPs guardados.

Propiedades de la colección de SCPs:

`Application` `Count`

Métodos de la colección de SCPs:

`Add` `Item`

Las propiedades y métodos expuestos están ya estudiados.

NOTA: El método `Add` hemos visto que tenía una sintaxis especial para la colección de bloques. Veremos ahora la sintaxis especial para la colección de SCPs.

```
Set ObjSCP = ObjColección.Add(DblOrigen, DblPtoEjeX, DblPtoEjeY, StrNombre)
```

`DblOrigen` es una coordenada 3D (matriz de tres valores `Double`) en el SCU que especifica dónde se añadirá el SCP. `DblPtoEjeX` y `DblPtoEjeY` son dos coordenadas 3D también (matrices de tres valores `Double` cada una) en el SCU que indican sendos puntos en la parte positiva de los ejes X e Y para el SCP; la dirección positiva del eje Z queda definida por la regla de la "mano derecha". `StrNombre` es el nombre (`String`) del SCP que se añadirá a la colección.

Si nos damos cuenta lo que hacemos es definir un SCP mediante tres puntos: el punto de origen, un punto en el eje X y un punto en el eje Y.

El objeto simple de la colección de SCPs es un objeto de SCP. Los objetos de SCP poseen las siguientes propiedades y métodos:

Propiedades de los objetos de SCP:

`Application` `Name` `Origin` **`XVector`** **`YVector`**

Métodos de los objetos de SCP:

`Delete` **`GetUCSMatrix`** `GetXData` `SetXData`

Observamos que existen dos nuevas propiedades y un método nuevo que aún no conocemos. Pasaremos pues a estudiar esto.

Primero las propiedades:

- `XVector`. Asigna u obtiene la dirección X del SCP. Para asignar:

```
ObjSCP.XVector = DblVectorX
```

donde `DblVectorX` es un *array* de tres valores `Double`. Esta característica está controlada también por la variable de sistema de **AutoCAD** `UCSXDIRE`.

Para obtener:

```
VarVectorX = ObjSCP.XVector
```

- `YVector`. Asigna u obtiene la dirección Y del SCP. Para asignar:

```
ObjSCP.YVector = DblVectorY
```

donde `DblVectorY` es un *array* de tres valores `Double`. Esta característica está controlada por la variable de sistema de **AutoCAD** `UCSYDIR`.

Para obtener:

```
VarVectorY = ObjSCP.YVector
```

Ahora el método:

- `GetUCSMatrix`. Obtiene la matriz de transformación del objeto de SCP al que se aplica el método. Esta matriz de 4×4 define las rotaciones y traslaciones de cada eje del SCP respecto al Sistema Universal. Sintaxis:

```
VarMatriz = ObjSCP.GetUCSMatrix
```

donde `VarMatriz` es una variable que habrá sido declarada como `Variant`, aunque recogerá una matriz de 4×4 elementos `Double`. Después podremos acceder a cada valor por medio de índices, como hemos hecho otras veces.

DOCE.8.13. Colección de vistas y el objeto vista

La colección de vistas (`Views`) la constituyen todos los objetos de vista del documento actual activo, esto es, las vistas guardadas.

Propiedades de la colección de vistas:

<code>Application</code>	<code>Count</code>
--------------------------	--------------------

Métodos de la colección de vistas:

<code>Add</code>	<code>Item</code>
------------------	-------------------

Las propiedades y métodos expuestos están ya estudiados.

El objeto simple de esta colección es un objeto de vista. Los objetos de vista poseen las siguientes propiedades y métodos:

Propiedades de los objetos de vista:

<code>Application</code>	<code>Center</code>	<code>Direction</code>	<code>Height</code>
<code>Name</code>	<code>Target</code>	<code>Width</code>	

Métodos de los objetos de vista:

<code>Delete</code>	<code>GetXData</code>	<code>SetXData</code>
---------------------	-----------------------	-----------------------

Las nuevas propiedades `Direction` y `Target` se explican a continuación.

- `Direction`. Asigna u obtiene la dirección de la vista para una visualización tridimensional. Para asignar:

```
ObjNoGráfico.Direction = DblVector
```

Para obtener:

```
VarVector = ObjNoGráfico.Direction
```

donde *DblVector* es un *array* de tres valores *Double*. Es un vector de dirección desde donde el dibujo se verá. Esta propiedad es similar al comando *PTOVISTA* (*VPOINT* en inglés) de **AutoCAD**.

- *Target*. Asigna u obtiene el punto objetivo de una vista (en este caso). Para asignar:

```
ObjNoGráfico.Target = DblPuntoObjetivo
```

Para obtener:

```
VarPuntoObjetivo = ObjNoGráfico.Target
```

donde *DblPuntoObjetivo* es un *array* de tres valores *Double*. Como siempre, para recoger utilizaremos una variable declarada como *Variant*.

DOCE.8.14. Colección de ventanas y el objeto ventana

La colección de ventanas (*Viewports*) la constituyen todos los objetos de ventana del documento actual activo en el Espacio Modelo, esto es, las ventanas gráficas en mosaico activas actualmente.

Propiedades de la colección de ventanas:

Application *Count*

Métodos de la colección de ventanas:

Add *Delete* *Item*

Las propiedades y métodos expuestos están ya estudiados.

El objeto simple de esta colección es un objeto de ventana gráfica. Los objetos de ventana gráfica poseen las siguientes propiedades y métodos:

Propiedades de los objetos de ventana:

<i>Application</i>	<i>Center</i>	<i>Direction</i>	<i>GridOn</i>
<i>Height</i>	<i>LowerLeftCorner</i>	<i>Name</i>	<i>OrthoOn</i>
<i>SnapBasePoint</i>	<i>SnapOn</i>	<i>SnapRotationAngle</i>	<i>StatusID</i>
<i>Target</i>	<i>UCSIconAtOrigin</i>	<i>UCSIconOn</i>	<i>UpperRightCorner</i>
<i>Width</i>			

Métodos de los objetos de ventana:

<i>GetGridSpacing</i>	<i>GetSnapSpacing</i>	<i>GetXData</i>	<i>SetGridSpacing</i>
<i>SetSnapSpacing</i>	<i>SetView</i>	<i>SetXData</i>	<i>Split</i>
<i>ZoomAll</i>	<i>ZoomCenter</i>	<i>ZoomExtents</i>	<i>ZoomPickWindow</i>
<i>ZoomScaled</i>	<i>ZoomWindow</i>		

Comencemos por las nuevas propiedades.

- **GridOn**. Obtiene y asigna el valor de estado de la rejilla en una ventana. Es un valor booleano (Boolean) que si es True especifica que la rejilla está activada y si, por el contrario, es False, especifica que la rejilla está desactivada.

La sintaxis para asignar este valor es:

```
ObjVentana.GridOn = BooRejilla
```

Y la sintaxis para obtenerlo:

```
BooRejilla = ObjVentana.GridOn
```

- **LowerLeftCorner**. Obtiene el vértice inferior izquierdo de la ventana gráfica, en porcentaje de ancho y alto de pantalla, tal como se almacenan las configuraciones de ventanas mosaico en **AutoCAD**. Sintaxis:

```
VarEsquinaII = ObjVentana.LowerLeftCorner
```

VarEsquinaII recogerá una matriz de valores Double.

- **OrthoOn**. Obtiene y asigna la condición del modo Orto en la ventana activa actualmente. Sintaxis para asignar:

```
ObjVentana.OrthoOn = BooModoOrto
```

Sintaxis para obtener:

```
BooModoOrto = ObjVentana.OrthoOn
```

BooModoOrto tomará el valor True si el modo Orto está activado —o se quiere activar— ; tomará el valor False si el modo Orto está desactivado o se quiere desactivar.

- **SnapBasePoint**. Especifica el punto base para el forzado de cursor/rejilla en la ventana actual.

La sintaxis para asignar este valor es:

```
ObjVentana.SnapBasePoint = DblPtoBase
```

Y la sintaxis para obtenerlo:

```
VarPtoBase = ObjVentana.SnapBasePoint
```

Evidentemente, *DblPtoBase*, al ser un punto, será una matriz de tres valores Double. Si es para obtener, recordar que habremos de guardarlo en una variable *Variant*.

- **SnapOn**. Especifica el estado del forzado de cursor.

La sintaxis para asignar este valor es:

```
ObjVentana.SnapOn = BooEstadoForzcursor
```

Y la sintaxis para obtenerlo:


```
BooEstadoForzcursor = ObjVentana.SnapOn
```

`BooEstadoForzcursor` será un valor Boolean que además admite las siguientes constantes:

`acOn` `acOff`

que especifican la activación (`acOn`) o no activación (`acOff`) del `Forzcursor`.

- `SnapRotationAngle`. Es el ángulo de rotación en radianes (valor `Double` de 0 a 6.28) del forzado de cursor en la ventana con respecto al SCU.

La sintaxis para asignar este valor es:

```
ObjVentana.SnapRotationAngle = DblÁnguloRotación
```

Y la sintaxis para obtenerlo:

```
DblÁnguloRotación = ObjVentana.SnapRotationAngle
```

- `StatusID`. Obtiene únicamente el estado de activación de una ventana. El valor se recoge en una variable declarada como Boolean: si guarda `True` la ventana está activada, si guarda `False` la ventana está desactivada:

```
BooEstadoActivación = ObjVentana.StatusID
```

- `UCSIconAtOrigin`. Especifica si el icono o símbolo del SCP se muestra en el origen de coordenadas o no. Todo ello es controlado por un valor Boolean que, si es `True` quiere decir que el SCP aparece en el origen y, si es `False`, que no.

La sintaxis para asignar este valor es:

```
ObjVentana.UCSIconAtOrigin = BooSimbSCPOrigen
```

Y la sintaxis para obtenerlo:

```
BooSimbSCPOrigen = ObjVentana.UCSIconAtOrigin
```

- `UCSIconOn`. Especifica si el icono o símbolo del SCP se visualiza o no. Todo ello es controlado por un valor Boolean que, si es `True` quiere decir que el SCP se visualiza y, si es `False`, que no se visualiza.

La sintaxis para asignar este valor es:

```
ObjVentana.UCSIconOn = BooSimbSCPActivo
```

Y la sintaxis para obtenerlo:

```
BooSimbSCPActivo = ObjVentana.UCSIconOn
```

- `UpperRightCorner`. Obtiene el vértice superior derecho de la ventana gráfica, en porcentaje de ancho y alto de pantalla, tal como se almacenan las configuraciones de ventanas mosaico en **AutoCAD**. Sintaxis:

```
VarEsquinaSD = ObjVentana.UpperRightCorner
```

VarEsquinaSD será una matriz de valores Double.

Veamos ahora los nuevos métodos.

- *GetGridSpacing*. Obtiene el espaciado de la rejilla para la ventana:

```
Call ObjVentana.GetGridSpacing(DblEspXRejilla, DblEspYRejilla)
```

DblEspXRejilla y *DblEspYRejilla* son dos variables declaradas como Double que guardarán el espaciado en X y en Y respectivamente.

- *GetSnapSpacing*. Obtiene el espaciado del forzado de cursor para la ventana:

```
Call ObjVentana.GetSnapSpacing(DblEspXForzcursor, DblEspYForzcursor)
```

DblEspXForzcursor y *DblEspYForzcursor* son dos variables declaradas como Double que guardarán el espaciado en X y en Y respectivamente.

- *SetGridSpacing*. Asigna el espaciado de la rejilla para la ventana:

```
Call ObjVentana.SetGridSpacing(DblEspXRejilla, DblEspYRejilla)
```

DblEspXRejilla y *DblEspYRejilla* son dos valores Double que representan el espaciado en X y en Y respectivamente.

- *SetSnapSpacing*. Asigna el espaciado del forzado de cursor para la ventana:

```
Call ObjVentana.SetSnapSpacing(DblEspXForzcursor, DblEspYForzcursor)
```

DblEspXForzcursor y *DblEspYForzcursor* son dos valores Double que representan el espaciado en X y en Y respectivamente. El valor inicial para X e Y es 1.0000.

- *SetView*. Establece en la ventana gráfica la visualización de un objeto de vista. El argumento suministrado deberá ser de tipo Object, conteniendo una vista existente en el dibujo:

```
Call ObjVentana.SetView(ObjVista)
```

ObjVista es un objeto de vista de los explicados en la sección anterior.

- *Split*. Divide la ventana gráfica actual en el número de partes indicado:

```
Call ObjVentana.Split(IntNúmVentanas)
```

IntNúmVentanas ha de ser un número entero, aunque también se admiten las siguientes constantes (las cuales dicen relación a las diferentes formas predeterminadas de **AutoCAD** de dividir una ventana):

<i>acViewport2Horizontal</i>	<i>acViewport2Vertical</i>
<i>acViewport3Left</i>	<i>acViewport3Right</i>
<i>acViewport3Horizontal</i>	<i>acViewport3Vertical</i>
<i>acViewport3Above</i>	<i>acViewport3Below</i>
<i>acViewport4</i>	

La siguiente macro muestra la manera de dividir la ventana gráfica actual en tres ventanas con una posición predefinida de **AutoCAD**:

Option Explicit

Dim AcadDoc As Object
Dim AcadModel As Object

Dim Ventanal As Object

```
Sub Macro()  
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument  
    Set AcadModel = AcadDoc.ModelSpace  
  
    Set Ventanal = AcadDoc.Viewports.Add("Nueva")  
    Call Ventanal.Split(acViewport3Left)  
    Set AcadDoc.ActiveViewport = Ventanal  
End Sub
```

Primero hemos de crear un nuevo objeto de ventana en la colección de ventanas, después dividirlo a nuestro gusto y, por último, definirlo como la ventana actual (recordemos la propiedad `ActiveViewport` del objeto de documento activo).

- `ZoomAll`. Realiza un *Zoom Todo* en la ventana actual:

`ObjVentana.ZoomAll`

- `ZoomCenter`. Realiza un *Zoom Centro* en la ventana actual. Hay que proporcionar el centro en cuestión (matriz de tres valores `Double`) y el factor de ampliación (`Double`).

`Call ObjVentana.ZoomCenter(DblPtoCentro, DblFactorAmpl)`

- `ZoomExtens`. Realiza un *Zoom Extensión* en la ventana actual:

`ObjVentana.ZoomExtens`

- `ZoomPickWindow`. Realiza un *Zoom Ventana* en la ventana actual. Los puntos diagonales de la ventana se marcan en pantalla con el dispositivo señalador al mensaje habitual:

`ObjVentana.ZoomPickWindow`

- `ZoomScaled`. Realiza un *Zoom Factor* en la ventana actual. Hay que proporcionar el factor de escala y el tipo de escala:

`Call ObjVentana.ZoomScaled(DblFactorEscala, IntTipoEscala)`

`DblFactorEscala` es un valor `Double` que representa el factor de escala. `IntTipoEscala` es un valor `Integer` que además admite las tres siguientes constantes (absoluta, relativa y relativa al Espacio Papel):

`acZoomScaledAbsolute acZoomScaledRelative acZoomScaledRelativePSPACE`

- `ZoomWindow`. Realiza un *Zoom Ventana* en el que se proporcionan los puntos diagonales como argumentos del método:

```
Call ObjVentana.ZoomWindow(DblPtoInfIzq, DblPtoSupDcha)
```

DblPtoInfIzq y *DblPtoSupDcha* son los puntos (matriz de tres valores Double) inferior izquierda y superior derecha, respectivamente, de la ventana.

DOCE.8.14.1. Ventanas del Espacio Papel

Todo lo visto hasta aquí en cuestión de ventanas se refiere a las ventanas del Espacio Modelo de **AutoCAD**. A la hora de crear y gestionar ventanas de Espacio Papel se recurre a un método exclusivo de la colección de objetos de Espacio Papel: `AddPViewport`. Si recordamos, emplazamos el estudio de este método a esta sección.

La sintaxis de `AddPViewport` es:

```
Set ObjVentanaPapel = ObjColEspacioPapel.AddPViewport (DblCentro, DblAltura, DblAnchura)
```

Propiedades de los objetos de ventana de Espacio Papel:

Application	Center	Direction	EntityName
EntityType	GridOn	Handle	Height
Layer	LensLength	Linetype	LinetypeScale
ObjectID	RemoveHiddenLines	SnapBasePoint	SnapOn
SnapRotationAngle	Target	TwistAngle	UCSIconAtOrigin
UCSIconOn	Visible	Width	

Métodos de los objetos de ventana de Espacio Papel:

ArrayPolar	ArrayRectangular	Copy	Display
Erase	GetBoundingBox	GetGridSpacing	GetSnapSpacing
GetXData	Highlight	IntersectWith	Mirror
Mirror3D	Move	Rotate	Rotate3D
ScaleEntity	SetGridSpacing	SetSnapSpacing	SetXData
TransformBy	Update	ZoomAll	ZoomCenter
ZoomExtents	ZoomPickWindow	ZoomScaled	ZoomWindow

DblCentro es un valor que especifica el punto (array de tres valores Double) central de la ventana. *DblAltura* es un valor Double que especifica la altura de la ventana. *DblAnchura* es un valor Double que especifica la anchura de la ventana. Estos dos últimos valores han de ser positivos.

Pasemos a explicar las nuevas propiedades y el nuevo método.

- **LensLength**. Especifica la longitud de la lente (en milímetros) para la vista en perspectiva. Este valor Double está también controlado por la variable de sistema `LENSLENGTH`.

La sintaxis para asignar este valor es:

```
ObjVentanaPapel.LensLength = DblLongLente
```

Y la sintaxis para obtenerlo:

```
DblLongLente = ObjVentanaPapel.LensLength
```

Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

• `RemoveHiddenLines`. Especifica si las líneas ocultas han de ser trazadas o imprimidas en determinada ventana. Su sintaxis para asignar un valor es la que sigue:

```
ObjVentanaPapel.RemoveHiddenLines = BooLíneasOcultas
```

Y la sintaxis para obtenerlo:

```
BooLíneasOcultas = ObjVentanaPapel.RemoveHiddenLines
```

`BooLíneasOcultas` es un valor Boolean; si es `True` las líneas ocultas no aparecen trazadas, si es `False` sí.

• `TwistAngle`. Obtiene el ángulo de ladeo de una ventana. Su sintaxis es la que sigue:

```
DblÁngLadeo = ObjVentanaPapel.TwistAngle
```

`DblÁngLadeo` es un valor Double.

El método `Display` a continuación.

• `Display`. Este método activa o desactiva una ventana de Espacio Papel:

```
ObjVentanaPapel.Display(BooEstado)
```

`BooEstado` es un valor Boolean; `True` activa la ventana, `False` la desactiva.

Veamos el ejemplo que sigue:

Option Explicit

```
Dim AcadDoc As Object
```

```
Dim AcadPapel As Object
```

```
Dim VentanaPapel As Object
```

```
Dim CentroVentana(1 To 3) As Double
```

```
Sub Macro()
```

```
Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument
```

```
Set AcadPapel = AcadDoc.PaperSpace
```

```
CentroVentana(1) = 100: CentroVentana(2) = 100: CentroVentana(3) = 0
```

```
Set VentanaPapel = AcadPapel.AddPViewport(CentroVentana, 100, 100)
```

```
VentanaPapel.Display (True)
```

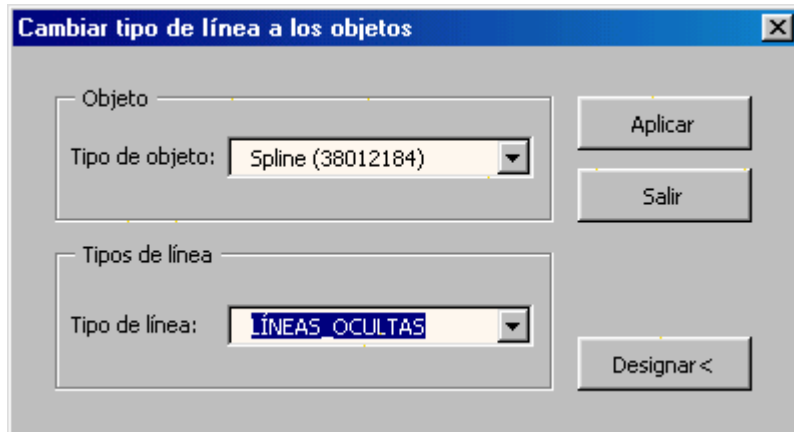
```
End Sub
```

Con esta macro creamos una ventana flotante de Espacio Papel. Al crearla, por defecto aparece desactivada. Para activarla utilizamos el método `Display`.

Hasta aquí todo lo referente a las colecciones de objetos. Hemos aprendido muchos detalles que ya nos harán programar medianamente bien. Sin embargo, para optar por una programación aceptable necesitamos conocer algunos mecanismos más, sobre todo los que veremos en la sección siguiente.

3ª fase intermedia de ejercicios

- Escribir una programa en el módulo VBA que permita cambiar el tipo de línea de uno o varios objetos. El letrero de diálogo que manejará el programa es el siguiente (por ejemplo):



El funcionamiento del programa es el siguiente. Al iniciar el letrero se rellenará la lista desplegable *Tipo de línea* con los nombres de los tipos de línea actualmente cargados en el dibujo. La lista *Tipo de objeto* permanecerá vacía por el momento.

Al pulsar el botón *Designar <* se cerrará el cuadro y se permitirá designar en pantalla uno o varios objetos. Tras el final de la designación o selección, se retornará al cuadro en el que se rellenará la lista desplegable *Tipo de objeto* con el tipo (línea, círculo, spline...) y el número ID de identificación, entre paréntesis, de cada uno de los objetos seleccionados.

La manera de aplicar ahora tipos de línea a los objetos es escogiendo de la lista superior la entidad en concreto y, de la lista inferior, el tipo de línea que será asignado a ella. Tras esto se pulsa el botón *Aplicar* para aplicar los cambios. Los cambios deben ser evidentes de inmediato (al pulsar *Aplicar*) pero el letrero no ha de cerrarse; por si acaso se quieren realizar más modificaciones o cambios.

El botón *Salir* cierra el letrero de diálogo y termina la aplicación.

DOCE.9. UTILIDADES VARIAS (EL OBJETO `Utility`)

Como podemos comprobar al observar la tabla jerárquica de objetos y colecciones del principio del **MÓDULO**, existe un objeto de utilidad que es descendiente directo del documento actual activo. Este objeto llamado `Utility`, posee una serie de métodos que nos harán la vida más fácil a la hora de programar, ya que podremos acceder a ellos para solicitar puntos, convertir coordenadas, utilizar marcas de *inicio* y *fin* de DESHACER, etcétera.

La manera de llamar a este objeto y a sus métodos es idéntica a la ya utilizada para los demás objetos. Lo lógico será, si vamos a utilizarlo mucho, que creamos una variable para acceder a él de forma simple, como hemos venido haciendo. Por ejemplo, podemos declarar así una serie de variables de objeto:

```
Option Explicit  
  
Dim AcadDoc As Object  
Dim AcadModel As Object
```

```
Dim AcadUtil As Object
```

y luego inicializarlas así:

```
Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument  
Set AcadModel = AcadDoc.ModelSpace  
Set AcadUtil = AcadDoc.Utility
```

Como vemos, el objeto de utilidad desciende directamente del de documento activo, por lo que la forma de inicializarlo es similar a la de la colección de objetos de Espacio Modelo por ejemplo, como se ve (están en el mismo nivel).

Pasemos directamente a ver la lista de propiedades y métodos de este objeto, para después estudiarlos detenidamente.

Propiedades del objeto de utilidad:

Application

Métodos del objeto de utilidad:

AngleFromXAxis	AngleToReal	AngleToString	DistanceToReal
EndUndoMark	GetAngle	GetCorner	GetDistance
GetInput	GetInteger	GetKeyword	GetOrientation
GetPoint	GetReal	GetString	InitializeUserInput
PolarPoint	RealToString	StartUndoMark	TranslateCoordinates

Como propiedad tenemos, como siempre, *Application*, que es común a todos los objetos VBA. Los diversos métodos serán los que se expliquen a continuación, como venimos haciendo hasta ahora.

- **AngleFromXAxis**. Este método obtiene el ángulo comprendido entre la línea o vector determinado por los dos puntos que se suministran como argumentos, y el eje X actual del dibujo, siempre en sentido trigonométrico o antihorario y en radianes. La sintaxis para **AngleFromXAxis** es:

```
DblÁnguloDesdeX = ObjUtilidad.AngleFromXAxis(DblPtoInic, DblPtoFinal)
```

Tanto *DblPtoInic* como *DblPtoFinal* son los dos puntos de una línea —o de un vector que no tiene por qué estar dibujado—, por lo que serán matrices o *arrays* de tres elementos (coordenada X, coordenada Y y coordenada Z) de tipo de dato *Double*.

No es indiferente el orden en que se introducen los puntos, ya que no es igual el ángulo desde el eje X a una línea que va desde un punto 1 hasta un punto 2, que el ángulo desde el eje X a una línea que va desde 2 hasta 1, evidentemente.

NOTA: La variable que recoja este valor (*DblÁnguloDesdeX*) habrá sido declarada como *Double*.

- **AngleToReal**. Este método obtiene la conversión en radianes de un ángulo en formato de texto a un número real de doble precisión. Veamos la sintaxis de este método:

```
DblÁnguloReal = ObjUtilidad.AngleToReal(StrÁngulo, IntUnidades)
```

Como vemos, el ángulo ha de ser en formato de cadena de texto —normalmente introducido por el usuario—. *IntUnidades*, por su lado, es un valor *Integer* que determina en qué unidades aparece el ángulo del argumento anterior, procediendo a su conversión en

número real según convenga. Este segundo argumento admite también las siguientes constantes:

acDegrees	acDegreeMinuteSeconds	acGrads
acRadians	acSurveyorUnits	

Éstas se corresponden con los diferentes tipos de ángulos que maneja **AutoCAD** (grados sexagesimales decimales, grados/minutos/segundos, grados centesimales, radianes, unidades geodésicas).

Así pues, una operación como la que sigue (siguiendo con la notación especificada al principio de esta sección):

```
Dim Resultado As Double
Resultado = AcadUtil.AngleToReal("180", acDegrees)
MsgBox Resultado
```

devolverá 3.14159265358979, esto es, el resultado de pasar a radianes la cantidad obtenida de convertir el texto "180" considerado como grados en notación decimal. El resultado (PI) sería el mismo al hacer lo siguiente:

```
Dim Resultado As Double
Resultado = AcadUtil.AngleToReal("100", acGrads)
MsgBox Resultado
```

- **AngleToString.** AngleToString convierte el ángulo proporcionado, el cual se considera siempre en radianes, a cadena de texto de acuerdo con las unidades y los decimales de precisión indicados:

```
StrÁnguloCadena = ObjUtilidad.AngleToString(DblÁngulo, IntUnidades, IntPrecis)
```

DblÁngulo es un valor Double que, como decimos, estará en radianes, ya que VBA así siempre lo interpreta. *IntUnidades* es un valor Integer que admite las mismas constantes explicadas en el método anterior y que representa las unidades a las que se convertirán los radianes especificados. *IntPrecis* especifica la precisión del ángulo, que se convertirá a cadena, en número de decimales. Este último argumento admite un valor entre 0 y 8 (siempre entero).

Veamos una rutina de ejemplo:

```
Dim Resultado As String
Const PI = 3.14159265358979
Resultado = AcadUtil.AngleToString(PI / 2, acDegrees, 3)
MsgBox Resultado
```

El resultado sería la cadena "90". Como vemos se ha transformado en grados sexagesimales en formato decimal (acDegrees), en cadena (AngleToString) y, aunque se indicó un precisión de tres decimales (3), no se ha tenido en cuenta al resultar un valor exacto.

- **DistanceToReal.** Convierte un distancia en formato de texto —normalmente indicada por el usuario— a un valor real, de acuerdo con el tipo de unidades especificado:

```
DblDistReal = ObjUtilidad.DistanceToReal(StrDistancia, IntUnidades)
```

StrDistancia es un valor tipo String que, como decimos, será una cadena. *IntUnidades* es un valor entero (Integer) que también admite las siguientes constantes:

Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

acScientific
acArchitectural

acDecimal
acFractional

acEngineering

Este último argumento indica las unidades de conversión, cuyas constantes se corresponden con las unidades manejadas por **AutoCAD** (científicas, decimales, pies y pulgadas I, pies y pulgadas II y fraccionarias).

NOTA: El valor de la distancia obtenido es siempre en unidades de dibujo.

- EndUndoMark. Este método coloca una señal de *fin* de DESHACER en el lugar del programa que se utilice. Su sintaxis es:

`ObjUtilidad.EndUndoMark`

Si recordamos, las marcas de *inicio* y *fin* del comando de **AutoCAD** DESHACER las utilizábamos mucho en los programas de AutoLISP, por lo que no hay motivo para no hacerlo en VBA también.

Cuando realizamos un programa o una macro que dibuja varias entidades, por ejemplo, si al término de su ejecución no estamos contentos con el resultado y utilizamos el comando H para deshacer el dibujo, únicamente se deshacerá la última entidad dibujada. Si antes de comenzar el dibujo del conjunto introducimos una marca de *inicio* de DESHACER y, tras terminar el dibujo completo, otra de *fin* de DESHACER, al introducir H al final de la ejecución del programa o de la macro, el conjunto de entidades se deshacerá por completo como un todo, cosa que nos interesa por estética y por funcionalidad.

Esta característica se corresponde con la opción Fin (End en inglés) del comando DESHACER (UNDO en inglés) de **AutoCAD**.

NOTA: Más adelante se explicará el método StartUndoMark que coloca marcas de *inicio* de DESHACER.

- GetAngle. Acepta el valor de un ángulo indicado por el usuario. Sintaxis:

`DblÁngulo = ObjUtilidad.GetAngle(DblPtoBase, StrMensaje)`

Este valor se puede introducir directamente desde el teclado (en el formato actual de unidades en **AutoCAD** o con los sufijos permitidos para radianes, centesimales, etc.), o señalando puntos en pantalla. Si se especifica un punto de base (es opcional) Double, se muestra un cursor elástico “enganchado” a dicho punto y el ángulo es el formado por la línea desde ese punto hasta el señalado por el usuario. Si no se especifica un punto de base, el usuario puede señalar dos puntos en pantalla para indicar el ángulo. El ángulo se mide siempre en dos dimensiones, ignorándose las coordenadas Z de los puntos.

En cualquiera de los supuestos, el ángulo se mide a partir del origen actualmente establecido en **AutoCAD** (variable de ANGBASE), siempre en sentido antihorario. El valor devuelto es siempre en radianes. Si se introduce el valor por teclado, se considerará como radianes. A diferencia del método GetOrientation —explicado más adelante—, GetAngle se emplea sobre todo para medir ángulos relativos.

El mensaje también es opcional (tipo String), y especifica el texto que aparecerá como solicitud del ángulo en la línea de comandos.

Comentemos la macro siguiente:

Option Explicit

Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

```
Dim AcadDoc As Object
Dim AcadModel As Object
Dim AcadUtil As Object
```

```
Sub Macro()
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument
    Set AcadModel = AcadDoc.ModelSpace
    Set AcadUtil = AcadDoc.Utility

    Dim Resultado As Double
    Dim PtoBase(1 To 3) As Double
    PtoBase(1) = 10: PtoBase(2) = 18
    Resultado = AcadUtil.GetAngle(PtoBase, "Ángulo: ")
    MsgBox Resultado
End Sub
```

Veamos que aquí se solicita un ángulo al usuario y luego se muestra con un `MsgBox`. Percatémonos también de que la coordenada Z del punto base no es necesario que tenga valor, ya que como hemos dicho no se toma en cuenta. Sin embargo, a la hora de declarar la variable habremos de hacerlo como un *array* de tres valores (aunque el último luego lo dejemos vacío), ya que de otro modo no funcionará el método.

- `GetCorner`. Este método acepta el valor de un punto indicado por el usuario, mientras existe otro punto “enganchado”, lo que forma un rectángulo. Sintaxis:

```
VarPuntoEsquina = ObjUtilidad.GetCorner(DblPtoBase, StrMensaje)
```

En este caso *DblPtoBase* es obligatorio (`Double`). *StrMensaje* sigue siendo opcional, al igual que con el método anterior, y es una cadena alfanumérica (`String`). El resultado de este método será un punto (matriz de tres elementos `Double`), por lo que habrá de recogerse en una variable tipo `Variant` para luego, y si se quieren utilizar las coordenadas de dicho punto, extraer los valores mediante índices y hacer un trasvase.

- `GetDistance`. Este método acepta el valor de una distancia indicada por el usuario. Este valor (`Double`) podrá ser introducido mediante el teclado o directamente en pantalla marcando dos puntos. En este caso da lo mismo el orden de introducción de puntos.

La sintaxis para `GetDistance` es la siguiente:

```
DblDistancia = ObjUtilidad.GetDistance(DblPtoBase, StrMensaje)
```

Si se indica un punto base, que habrá de ser `Double`, el cursor se “enganchará” a él y la distancia será medida desde ahí hasta el siguiente punto indicado por el usuario. *StrMensaje* (`String`), como viene siendo habitual, es un valor opcional.

NOTA: En principio la distancia medida con `GetDistance` es una distancia 3D. Aprenderemos al ver el método `InitializeUserInput` que esto puede modificarse.

- `GetInput`. Este método Se utiliza inmediatamente después de alguno de los otros métodos de solicitud (*Get...*), para detectar si el usuario ha dado alguna respuesta textual desde el teclado. Devuelve el texto introducido desde el teclado. Su sintaxis es:

```
StrCadenaDevuelta = ObjUtilidad.GetInput
```

NOTA: Veremos más adelante algún ejemplo de este método (después de `InitializeUserInput`) que nos lo aclarará más.

- **GetInteger.** Solicita del usuario que indique un número entero. Si no es así, rechaza el dato introducido y vuelve a solicitar un número entero. Se permiten valores enteros negativos y el valor 0, a no ser que se especifique lo contrario mediante el método `InitializeUserInput`.

La sintaxis de `GetInteger` es:

`IntValorEntero = ObjUtilidad.GetInteger(StrMensaje)`

StrMensaje es opcional y funciona como en los métodos explicados anteriormente.

NOTA: Más aclaraciones sobre este y otros métodos tras `InitializeUserInput`.

- **GetKeyword.** Solicita del usuario que indique la palabra clave que le interesa. El mensaje de solicitud ofrecerá, lógicamente, las opciones posibles con las abreviaturas en mayúsculas para que el usuario sepa a qué atenerse. Previamente, se habrá utilizado el método `InitializeUserInput` para establecer las palabras clave permitidas.

Su sintaxis:

`StrPalabraClave = ObjUtilidad.GetKeyword(StrMensaje)`

StrMensaje es opcional y funciona como en los métodos explicados anteriormente.

- **GetOrientation.** Funciona de manera muy similar a `GetAngle`, con la única diferencia de que los ángulos se miden siempre desde el origen por defecto (posición de las 3 en el reloj o punto cardinal Este), independientemente del establecido en **AutoCAD** (variable de `ANGBASE`). Este método se emplea sobre todo para mediar ángulos absolutos y su sintaxis es:

`DblOrientación = ObjUtilidad.GetOrientation(DblPtoBase, StrMensaje)`

Los argumentos funcionan igual que en `GetAngle`.

- **GetPoint.** `GetPoint` solicita al usuario un punto que podrá ser marcado en pantalla o introducido por teclado:

`VarPunto = ObjUtilidad.GetPoint(DblPtoBase, StrMensaje)`

Si se indica un punto `Double` de base (opcional), el cursor aparece “enganchado” mediante una línea elástica a dicho punto. *StrMensaje* funciona igual que en métodos anteriores y también es opcional.

La siguiente macro de ejemplo se utiliza para dibujar rectángulos mediante polilíneas con sólo marcar dos puntos en pantalla: el primero controlado por un `GetPoint` y el segundo con un `GetCorner`, para poder ver el rectángulo final en tiempo real antes de ser dibujado:

Option Explicit

```
Dim AcadDoc As Object
Dim AcadModel As Object
Dim AcadUtil As Object
```

```
Sub Macro()
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument
    Set AcadModel = AcadDoc.ModelSpace
```

```
Set AcadUtil = AcadDoc.Utility

Dim Punto1, Punto2
Dim PuntoTras(2) As Double
Dim PuntosPol(9) As Double
Punto1 = AcadUtil.GetPoint(, "Primera esquina: ")
PuntoTras(0) = Punto1(0): PuntoTras(1) = Punto1(1): PuntoTras(2) = Punto1(2)
Punto2 = AcadUtil.GetCorner(PuntoTras, "Esquina opuesta: ")
PuntosPol(0) = Punto1(0): PuntosPol(1) = Punto1(1)
PuntosPol(2) = Punto2(0): PuntosPol(3) = Punto2(1)
PuntosPol(4) = Punto2(0): PuntosPol(5) = Punto2(1)
PuntosPol(6) = Punto1(0): PuntosPol(7) = Punto2(1)
PuntosPol(8) = Punto1(0): PuntosPol(9) = Punto1(1)
Call AcadDoc.ModelSpace.AddLightWeightPolyline(PuntosPol)
End Sub
```

Lo primero que se hace, tras declarar las variables, es solicitar el primer punto del rectángulo. Se realiza ahora un trasvase de coordenadas (de Variant a Double) para poderse las suministrar al método `GetCorner` como punto base. Se pide el segundo punto (el cursor permanecerá enganchado al primero mediante un rectángulo elástico) y se calculan los puntos para la polilínea que dibujará el rectángulo.

Una nota importante que debemos reseñar de este ejemplo es la manera de dibujar la polilínea. Nótese que por primera vez en estas páginas, en lugar de utilizar el método como explicamos en su momento, lo hemos usado con `Call`. Es momento ahora de decir que esto es perfectamente factible con todos los métodos de dibujo de entidades. Lo que ocurre, es que normalmente se utiliza la otra manera (guardando el objeto resultante en una variable de objeto) para después tener acceso absoluto a la entidad dibujada: cambiar su color, su tipo de línea, etcétera o utilizar cualquiera de las propiedades o métodos de ella.

Ahora bien, en momentos en los que no nos interese de una entidad más que su puro dibujo, se puede utilizar esta técnica.

NOTA: Percatémonos que declarar un matriz con (2) elementos es igual que hacerlo con (1 To 3) elementos. En este segundo caso los índices variarían de 1 a 3, y en el primero de 0 a 2; lo que da un total de tres elementos en ambos casos.

- `GetReal`. Solicita del usuario que indique un número real. Si no es así, rechaza el dato introducido y vuelve a solicitar un número real. Si se indica un número entero, es aceptado como real.

La sintaxis de `GetReal` es:

`DblValorReal = ObjUtilidad.GetReal (StrMensaje)`

`StrMensaje` es opcional y funciona como en los métodos explicados anteriormente.

- `GetString`. Acepta un cadena de texto introducida por el usuario. Si contiene más de 132 caracteres, sólo devuelve los primeros 132 caracteres. Sintaxis:

`StrCadena = ObjUtilidad.GetString (BooModo, StrMensaje)`

El modo es un valor Boolean (True o False) que indica si la cadena de texto puede contener espacios en blanco. Si es verdadero se admiten espacios y el texto introducido por teclado debe terminarse con `INTRO`. Si es falso, el primer espacio se considerará como un `INTRO` y terminará el texto. Si no se introduce ningún texto y se pulsa directamente `INTRO`, se devuelve una cadena vacía.

Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

- `InitializeUserInput`. Este método establece limitaciones para aceptar los datos introducidos por el usuario, y también permite especificar palabras clave para ser aceptadas como nombres de opción. Veamos la sintaxis de utilización:

```
Call ObjUtilidad.InitializeUserInput (IntModo, StrPalabrasClave)
```

La mayoría de los métodos `Get...` que hemos visto se parecen enormemente (hasta algunos en los nombres) a las funciones `GET...` de AutoLISP que realizaban los mismos cometidos. Evidentemente necesitaremos pues un método como `InitializeUserInput` que haya opción de añadir justo antes de cualquier `Get...` para filtrar sus resultados; es lo que hacíamos en AutoLISP con `INITGET`.

`IntModo` es un valor entero (Integer) con código de bits que determina las limitaciones impuestas al usuario. Los modos posibles coinciden con los de la función `INITGET` de AutoLISP y se encuentran en la siguiente tabla:

Valor de bit	Modo
1	No admite valores nulos, es decir, <code>INTRO</code> como respuesta.
2	No admite el valor cero (0).
4	No admite valores negativos.
8	No verifica límites, aunque estén activados.
16	(No se utiliza).
32	Dibuja la línea o el rectángulo elásticos con línea de trazos en lugar de continua.
64	Hace que la función <code>GETDIST</code> devuelva distancias 2D.
128	Permite introducir datos arbitrarios por teclado. Tiene prioridad sobre el valor 1.

Al indicar un modo se pueden sumar varios de los bits. Por ejemplo, para impedir que el usuario indique un valor cero, nulo (es decir `INTRO`) y/o negativo, el modo que se especificará será 7 ($1 + 2 + 4$).

Este método debe invocarse justo antes del método `Get...` que limita. Los modos que tienen sentido para cada método `Get...` también coinciden con los correspondientes de AutoLISP y se encuentran en la siguiente tabla:

Método	Valores de bits de modo con sentido para el método					
<code>GetInteger</code>	1	2	4			128
<code>GetReal</code>	1	2	4			128
<code>GetDistance</code>	1	2	4	32	64	128
<code>GetAngle</code>	1	2		32		128
<code>GetOrientation</code>	1	2		32		128
<code>GetPoint</code>	1		8	32		128
<code>GetCorner</code>	1		8	32		128
<code>GetString</code>						
<code>GetKeyword</code>	1					128

NOTA: El modo no es opcional, y si no se desea ninguno hay que especificar un valor 0.

El segundo parámetro (`String`) es una cadena que define las palabras clave válidas como nombres de opciones. Estas se indican entre comillas, separadas por un espacio en blanco, y con la abreviatura en mayúsculas. La abreviatura es el mínimo número de caracteres en que debe coincidir la respuesta del usuario con una de las palabras clave válidas. El método

siempre devuelve la palabra tal y como está escrita en `InitializeUserInput`. La solicitud de palabra clave se realiza mediante el método `GetKeyword`. Por ejemplo:

```
Call AcadUtil.InitializeUserInput (7, "Alta Baja Normal")
Op = AcadUtil.GetKeyword ("Precisión Alta/Baja/Normal: ")
```

En el ejemplo se supone que las variables `AcadUtil` y `Op` ya han sido definidas. El método `GetKeyword` solicita del usuario una opción. Si éste desea la opción `Alta`, puede indicar `a`, `al`, `alt` o `alta`, y en todos los casos la variable `Op` almacena el valor `Alta`.

Aceptación de valores por defecto

Estudiemos ahora un mecanismo para aceptar valores por defecto, tan típicos en línea de comandos.

Cuando desde un programa en VBA se introducen valores no esperados por teclado, se produce un error de VBA. En estos casos, VBA ejecuta automáticamente una sentencia especial llamada `On Error`, si se ha incluido en el programa. Si no es así, detiene automáticamente el programa informando del error. Si se indica la sentencia `On Error Resume Next`, el programa se reanuda en la línea siguiente a la que ha producido el error, sin detenerse.

Además, existe un objeto específico denominado `Err`, que tiene una propiedad `Number` que almacena un número de error. Cuando no hay errores, ese número o índice de error es 0. Cuando VBA recibe un tipo de dato inesperado (es lo que ocurre al pulsar `INTRO` en la solicitud de número entero —por ejemplo—, o al cancelar con `ESC`), el número de error es diferente de 0. Una vez que se detecta que ha habido error, para averiguar el tipo de dato inesperado causante del mismo, se puede examinar la propiedad `Description` del mismo objeto `Err`. Si el usuario ha introducido una letra o un texto por teclado ante la solicitud de un entero, real o punto, la descripción de error es "La entrada de usuario es una palabra clave". Si el usuario cancela, mediante `ESC` por ejemplo, la descripción de error será diferente.

El método `GetInput` estudiado, devuelve el texto introducido por teclado que ha producido el error. Cuando el usuario pulsa `INTRO` para aceptar una opción por defecto, y VBA espera un número o un punto, lo considera un texto vacío y por lo tanto produce un error con la misma descripción expuesta más arriba. En este caso, `GetInput` devuelve una cadena vacía "".

Para que VBA considere `INTRO` como un error, podría pensarse en establecer un modo 1 en `InitializeUserInput`. Pero en este caso, simplemente se impediría el `INTRO` mostrándose un mensaje y solicitando de nuevo el dato. Si no se establece un modo 1, el `INTRO` es aceptado, pero entonces no se produce error. El resultado es que cada tipo de solicitud acepta un valor diferente. Así, `GetInteger` podría considerar `INTRO` como 0 (depende del diseño del programa y de la definición de variable asignada), `GetReal` como un valor muy pequeño prácticamente 0 y `GetPoint` tomaría la posición del cursor en pantalla en el momento de pulsar `INTRO` o el punto de base si se ha especificado.

La solución a este problema es utilizar el modo 128 en `InitializeUserInput`. Este modo acepta datos arbitrarios por teclado y tiene prioridad sobre el modo 1. Por lo tanto, si se indica el modo 129 ($1 + 128$), se está impidiendo el `INTRO` a causa del modo 1 pero el modo 128 fuerza a aceptarlo. VBA lo considera entonces un error, y lo acepta como palabra clave no esperada, con valor de cadena vacía.

En resumen, un mecanismo general para detectar el `INTRO` pulsado por el usuario, comprende los siguientes pasos:

- Establecer la sentencia `On Error Resume Next` para que el programa no se detenga al producirse un error.
- Establecer un modo en `InitializeUserInput` con 129 como sumando.
- Detectar si ha habido error, examinando si `Err.Number` es diferente de 0.
- Detectar si el error se debe a texto del teclado, examinando si `Err.Description = "La entrada de usuario es una palabra clave"`
- Recuperar el texto introducido por teclado, mediante `GetInput`.
- Examinar si ese texto es una cadena vacía `""`.

El siguiente ejemplo muestra cómo aceptar una opción por defecto desde `GetInteger`.

Option Explicit

```
Dim AcadDoc As Object
Dim AcadModel As Object
Dim AcadUtil As Object
```

```
Sub Macro()
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument
    Set AcadModel = AcadDoc.ModelSpace
    Set AcadUtil = AcadDoc.Utility

    Dim Prec As Integer
    Dim HayClave As Boolean
    Dim ValorClave As String

    On Error Resume Next
    Call AcadUtil.InitializeUserInput(135)
    Prec = AcadUtil.GetInteger("Valor de precisión <50>: ")
    If Err.Description = "La entrada de usuario es una palabra clave" Then
        HayClave = True
    Else
        HayClave = False
    End If
    ValorClave = AcadUtil.GetInput
    If Err.Number <> 0 Then
        If HayClave And ValorClave = "" Then Prec = 50 Else GoTo Errores
    End If
    Err.Clear

    ... resto del código ...

End

Errores:
    MsgBox "*NO VALE*"
End Sub
```

El modo empleado en `InitializeUserInput` es 135 ($1 + 2 + 4 + 128$). Esto impide valores negativos ó 0, pero no INTRO, debido a que el modo 128 prevalece sobre el 1.

Mediante `GetInteger` se solicita un número entero. La variable `Prec` se define como `Integer` y acepta la respuesta del usuario. Si éste indica un valor negativo o cero, se rechaza y se vuelve a solicitar. Si indica un valor entero positivo éste se almacena en `Prec`, el valor de `Err.Number` es 0 por lo que no se ejecuta la sentencia dentro del `If`, y `Prec` mantiene su valor para el resto del programa.

Si el usuario cancela mediante `ESC`, se produce un error. La variable `HayClave`, definida como `Boolean`, almacena el resultado verdadero o falso de comparar `Err.Description` con el texto de descripción, en este caso `False`. Por eso dentro de la sentencia de `If` no se cumplirá la condición y el programa saltará a la subrutina `Errores`, que por simplificar consiste simplemente en mostrar un cuadro de diálogo de aviso con el mensaje `*NO VALE*`.

Si el usuario pulsa `INTRO`, también se produce un error al haberse sumado 1 al modo de `InitializeUserInput`. La descripción del error sí coincide con el texto indicado en el código del programa. La variable `ValorClave`, definida como `String`, almacena el valor devuelto por `GetInput` que en este caso es una cadena vacía. Por lo tanto, se cumple la sentencia dentro del `If` y la variable `Prec` se iguala al valor por defecto 50.

Si el usuario pulsa cualquier otra combinación (un valor numérico con decimales o un texto), se produce un error con la descripción de palabra clave. Pero `GetInput` devuelve el texto introducido en vez de cadena vacía, por lo que la sentencia dentro del `If` no cumplirá su condición, y se producirá un salto a la subrutina de `Errores`.

En todos los casos, la sentencia `Err.Clear` elimina la descripción de error para que no se quede almacenada y pueda originar mal funcionamiento en la siguiente ejecución del programa.

Con métodos de aceptación de cadenas de texto (`GetKeyword` y `GetString`)

Los métodos de solicitud de cadenas de texto como `GetKeyword` y `GetString` no producen error al pulsar el usuario `INTRO` y lo entienden como cadena vacía `""`. El mecanismo para aceptar opciones por defecto difiere del explicado anteriormente. Si se utiliza el modo 128 de `InitializeUserInput` todos los textos introducidos por teclado se aceptan como palabras clave, sin producir error. Esto obliga a examinar esos textos para ver si coinciden con las palabras claves permitidas y se desvirtúa la finalidad de `GetKeyword`. Por lo tanto, un mecanismo sencillo para aceptar opciones por defecto podría ser:

```
Dim Op As String
On Error Resume Next
Call AcadUtil.InitializeUserInput(0, "Alta Baja Normal")
Op = AcadUtil.GetKeyword("Precisión Alta/Baja/<Normal>: ")
If Err.Number = 0 Then
    If Op = "" Then Op = "Normal" Else GoTo Errores
End If
```

El modo indicado en `InitializeUserInput` es 0, y se incluyen tres palabras clave como nombres de opciones permitidas. El propio método `GetKeyword` impide indicar valores numéricos o textos que no correspondan con las tres palabras clave. Si se produce cualquier error inesperado, la sentencia `Else` dentro del `If` hace que el programa salte a la subrutina de control de errores. Si el usuario introduce una de las opciones por teclado, no se produce error y la variable `Op` almacena la palabra clave de la opción. Por lo tanto, la sentencia dentro del `If` no se cumple y la variable `Op` sigue con su valor. Si el usuario pulsa `INTRO`, no se produce tampoco error, pero la variable `Op` almacena una cadena vacía y eso hace que la sentencia dentro del `If` se cumpla. El resultado es asignar a la variable `Op` el valor correspondiente a la opción por defecto, en este caso `Normal`.

Con el método de aceptación de números enteros (`GetInteger`)

Para aceptar un valor por defecto se puede utilizar el mecanismo ya explicado. Para aceptar palabras clave además de valores numéricos, el mecanismo se explica un poco más adelante. Estos mecanismos tienen la ventaja de que se pueden aplicar con mínimas modificaciones a todos los métodos que solicitan valores numéricos y puntos. No obstante, es

posible utilizar otros mecanismos específicos dependiendo del diseño del programa y conociendo los tipos de errores producidos.

Por ejemplo, cuando se define una variable como `Integer` y se le asigna el valor devuelto por `GetInteger`, si el usuario pulsa `INTRO`, VBA lo considera un tipo de dato inesperado y origina un error de desbordamiento con un valor de `Err.Number` igual a 6. Cualquier otro error, por ejemplo al cancelar mediante `ESC`, produce otro número diferente. Por lo tanto, un mecanismo sencillo para aceptar opciones por defecto es:

```
Dim N As Integer
Call AcadUtil.InitializeUserInput(6)
On Error Resume Next
N = AcadUtil.GetInteger("Precisión <3>: ")
If Err.Number <> 0 Then
    If Err.Number = 6 Then N = 3 Else GoTo Errores
End If
```

Mediante el modo 6, el método `InitializeUserInput` impide valores negativos y 0. La sentencia `On Error Resume Next`, hace que el programa no se detenga al producirse un error y continúe normalmente. El método `GetInteger` solicita un número entero. Si el usuario indica uno que no sea negativo ni 0, se acepta, el valor de `Err.Number` es 0, la condicional `If` no se cumple y el programa continúa sin problemas. Si el usuario indica `INTRO`, se produce un error aunque el programa continúa sin detenerse, el valor de `Err.Number` es 6, y entonces en la variable `N` se almacena el valor por defecto 3.

Si se origina cualquier otro error (por ejemplo el usuario cancela mediante `ESC`), el valor de `Err.Number` es diferente de 0 y 6, y entonces la sentencia `GoTo` salta a una subrutina `Errores`, donde habrá especificadas una serie de actuaciones y después se abortará el programa.

Con el método de aceptación de números reales (`GetReal`)

Para la aceptación de valores por defecto se puede utilizar el mismo mecanismo, sustituyendo simplemente la declaración de variable como `Double` en vez de `Integer`, y empleando lógicamente el método `GetReal` en lugar de `GetInteger`. Para aceptar palabras clave además de valores numéricos, el mecanismo se explica un poco más adelante —en esta misma página—.

Si se desea un mecanismo específico para números reales, cuando se define una variable como `Double` y se le asigna el valor devuelto por `GetReal`, si el usuario pulsa `INTRO`, VBA lo considera un valor residual muy pequeño próximo a 0. Como ese valor es un número real, `GetReal` no produce error. Por lo tanto, el mecanismo de aceptación no va a ser detectar un número de error, sino un valor devuelto muy pequeño.

```
Dim Prec As Double
On Error GoTo Errores
Call AcadUtil.InitializeUserInput(6)
Prec = AcadUtil.GetReal("Precisión <2.5>: ")
If Prec < 0.00000001 Then Prec = 2.5
```

En este caso, la sentencia `On Error` envía el programa directamente a la subrutina `Errores`, porque el `INTRO` como respuesta no va a ser considerado un error sino como un valor muy pequeño próximo a 0. Mediante el modo 6, el método `InitializeUserInput` impide valores negativos y 0. El propio método `GetReal` impide valores no numéricos. Mediante `If` se analiza el valor introducido por el usuario. Si es más pequeño que cualquiera que hubiera podido indicar por teclado, entonces es que ha pulsado `INTRO` y se asigna a la variable el valor

por defecto. En caso contrario, en `If` no se realiza ninguna acción y el programa continúa normalmente.

Otros métodos

Crear una rutina de aceptación de valores por defecto para los demás métodos resulta sencillo, ya que sólo hay que reflejarse en los ejemplos vistos hasta aquí y adecuar el más preciso.

Combinación de solicitudes numéricas con opciones textuales y `GetInput`

Si se ha indicado un modo 128 en `InitializeUserInput`, el método de solicitud empleado a continuación aceptará la entrada de teclado como palabra clave y `GetInput` devolverá dicha entrada como un texto. Si la entrada ha sido `INTRO`, lo devolverá como cadena vacía. Esto permite establecer un mecanismo de aceptación de valores por defecto cuando se emplean funciones de solicitud de datos numéricos o puntos, tal como se ha explicado anteriormente.

Pero si la entrada de teclado no es `INTRO` se puede emplear `GetInput` para combinar solicitudes numéricas con opciones de texto. Por ejemplo:

```
Dim Prec As Integer
Dim HayClave As Boolean
Dim ValorClave As String
On Error Resume Next
Call AcadUtil.InitializeUserInput(135, "Alta Baja Normal")
Prec = AcadUtil.GetInteger("Valor de precisión o Alta/Baja/Normal: ")
If Err.Description = "La entrada de usuario es una palabra clave" Then
    HayClave = True
Else
    HayClave = False
End If
ValorClave = AcadUtil.GetInput
If Err.Number <> 0 Then
    If HayClave And ValorClave <> "" Then GoSub Precision Else GoTo Errores
End If
Err.Clear

... resto del código ...
End
```

```
Precision:
    If ValorClave = "Normal" Then Prec = 10: Return
    If ValorClave = "Alta" Then Prec = 100: Return
    If ValorClave = "Baja" Then Prec = 1: Return
    GoTo Errores
```

El mecanismo es similar al explicado ya. Si se indica un valor negativo o cero se rechaza. Si se indica un número entero positivo se acepta porque no produce error. Si se indica un valor no esperado por teclado, se produce un error y el modo 128 como sumando en `InitializeUserInput` acepta la entrada como palabra clave. `GetInput` devuelve esa entrada como texto y lo almacena en `ValorClave`. La sentencia dentro del `If` examina si `HayClave` es verdadera y si `ValorClave` no es una cadena vacía. En caso de ser así, llama a la subrutina `Precision`, donde se analiza el texto aceptado como palabra clave y se asigna a la variable `Prec` el valor entero correspondiente a cada precisión, continuando la ejecución del programa mediante `Return`. Si la palabra clave no es ninguna de las tres admitidas, se salta a la subrutina de errores.

Si se produce un error inesperado, su descripción no corresponderá a la de palabra clave, y la sentencia dentro del `If` no se cumplirá por lo que `Else` saltará a la subrutina Errores. Si se pulsa `INTRO` (en el ejemplo no se admite una opción por defecto), `ValorClave` será una cadena vacía, la sentencia dentro del `If` tampoco se cumplirá y se saltará a la subrutina de Errores.

Sigamos pues ahora con la explicación de los métodos que faltan del objeto `Utility` de utilidad.

- **PolarPoint.** Este método obtiene el punto (matriz de tres elementos `Double`) a partir de otro punto dado (matriz de tres elementos `Double`), según un ángulo en radianes (`Double`) y una distancia en las unidades actuales (`Double`). Es decir, obtiene un punto por coordenadas polares a partir de otro dado.

La sintaxis de este método es:

<code>DblPunto2 = ObjUtilidad.PolarPoint (DblPunto1, DblÁngulo, DblDistancia)</code>

En el siguiente ejemplo, se dibuja una línea perpendicular desde el punto medio entre otros dos puntos, con una longitud especificada:

Option Explicit

```
Dim AcadDoc As Object
Dim AcadModel As Object
Dim AcadUtil As Object
```

```
Sub Macro()
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument
    Set AcadModel = AcadDoc.ModelSpace
    Set AcadUtil = AcadDoc.Utility

    Dim VPunto1, VPunto2, VPuntoFinal
    Dim Punto1(2) As Double, Punto2(2) As Double
    Dim PuntoMedio(2) As Double, PuntoFinal(2) As Double
    Dim Ángulo As Double, Distancia As Double
    Const PI = 3.1415926

    Call AcadUtil.InitializeUserInput(1)
    VPunto1 = AcadUtil.GetPoint(, "Primer punto: ")
    Punto1(0) = VPunto1(0): Punto1(1) = VPunto1(1): Punto1(2) = VPunto1(2)
    Call AcadUtil.InitializeUserInput(1)
    VPunto2 = AcadUtil.GetPoint(Punto1, "Segundo punto: ")
    Punto2(0) = VPunto2(0): Punto2(1) = VPunto2(1): Punto2(2) = VPunto2(2)
    PuntoMedio(0) = ((Punto1(0) + Punto2(0)) / 2)
    PuntoMedio(1) = ((Punto1(1) + Punto2(1)) / 2): PuntoMedio(2) = Punto1(2)
    Ángulo = AcadUtil.AngleFromXAxis(Punto1, Punto2)
    Distancia = AcadUtil.GetDistance(PuntoMedio, "Distancia en perpendicular: ")
    VPuntoFinal = AcadUtil.PolarPoint(PuntoMedio, Ángulo + PI / 2, Distancia)
    PuntoFinal(0) = VPuntoFinal(0)
    PuntoFinal(1) = VPuntoFinal(1)
    PuntoFinal(2) = VPuntoFinal(2)
    Call AcadDoc.ModelSpace.AddLine(PuntoMedio, PuntoFinal)
End Sub
```

Los puntos `Punto1` y `Punto2` (`VPunto1` y `VPunto2` en su definición `Variant`) son solicitados por el programa, sin permitir `INTRO` como respuesta nula. Pueden ser los extremos de una línea ya dibujada o dos puntos cualesquiera. A continuación, el programa calcula el

punto medio `PuntoMedio` haciendo medias aritméticas con las coordenadas X e Y. Mediante el método `AngleFromXAxis` calcula el ángulo absoluto entre los puntos 1 y 2. Después solicita la distancia en perpendicular y calcula el punto `PuntoFinal` a partir del punto medio, llevando la distancia a un ángulo que resulta de sumar $\pi / 2$ al ángulo absoluto entre 1 y 2. La última operación es dibujar una línea entre los dos últimos puntos.

- `RealToString`. `RealToString` convierte el valor proporcionado, el cual será real (`Double`), a cadena de texto de acuerdo con las unidades y los decimales de precisión indicados:

```
StrRealCadena = ObjUtilidad.RealToString(DblValorReal, IntUnidades, IntPrecis)
```

`DblValorReal` es, como hemos dicho, un valor `Double`. `IntUnidades` es un valor `Integer` que admite las mismas constantes explicadas en el método `DistanceToReal` y que representa las unidades a las que se convertirá el valor real. `IntPrecis` especifica la precisión en decimales del número real, el cual se convertirá a cadena. Este último argumento admite un valor entre 0 y 8 (siempre entero).

En la siguiente rutina:

```
Dim TxValor As String
TxValor = AcadUtil.RealToString(326.7539, acFractional, 2)
MsgBox TxValor
```

el valor devuelto será la cadena "326 3/4".

- `StartUndoMark`. Este método coloca una señal de *inicio* del comando `DESHACER` en el lugar del programa que se utilice. Su sintaxis es:

```
ObjUtilidad.StartUndoMark
```

Esta característica se corresponde con la opción *Inicio* (`BEGIN` en inglés) del comando `DESHACER` (`UNDO` en inglés) de **AutoCAD**.

NOTA: Véase en esta misma sección el método `EndUndoMark` que coloca marcas de *fin* de `DESHACER`.

- `TranslateCoordinates`. Convierte un punto o vector de desplazamiento de un sistema de coordenadas de origen a otro de destino:

```
VarPtoConvertido = ObjUtilidad.TranslateCoordinates(DblPtoOriginal, IntSisOrigen, IntSisDestino, BooDesplazamiento)
```

`BooDesplazamiento` es un valor `Boolean` que indica si la matriz de tres valores `Double` que es `DblPtoOriginal` se considera un vector de desplazamiento. Si es verdadero, se considera un vector de desplazamiento. Si es falso, se considera un punto.

Los sistemas de coordenadas de origen y destino se indican mediante un código de número entero (`Integer`). Para mayor facilidad y comodidad existen cuatro constantes que también se pueden especificar:

<code>acWorld</code>	<code>acUCS</code>	<code>acDisplayDCS</code>	<code>acPaperSpaceDCS</code>
----------------------	--------------------	---------------------------	------------------------------

Estas constantes se corresponden con los distintos sistemas de coordenadas que se utilizan en la interfaz gráfica de **AutoCAD** (Sistema de Coordenadas Universal o SCU, Sistema

de Coordenadas Personal o SCP, Sistema de Coordenadas de Visualización o SCV y Sistema de Coordenadas de Espacio Papel o SCEP).

En esta rutina (utilizando las convenciones que arrastramos desde el inicio de esta sección):

```
Dim VarPtoOr, VarPtoDest
VarPtoOr = AcadUtil.GetPoint(, "Punto que convertir: ")
VarPtoDest = AcadUtil.TranslateCoordinates(VarPtoOr, acUCS, acWorld, False)
```

el método `GetPoint` solicita indicar un punto, éste se acepta en la variable (definida como `Variant`) `VarPtoOr` y después se convierte desde el SCP actual al SCU, indicando un valor `False` de desplazamiento para que no se considere un vector.

NOTA: Recuérdese que no declarar una variable con un tipo concreto es lo mismo que declararla como `Variant`.

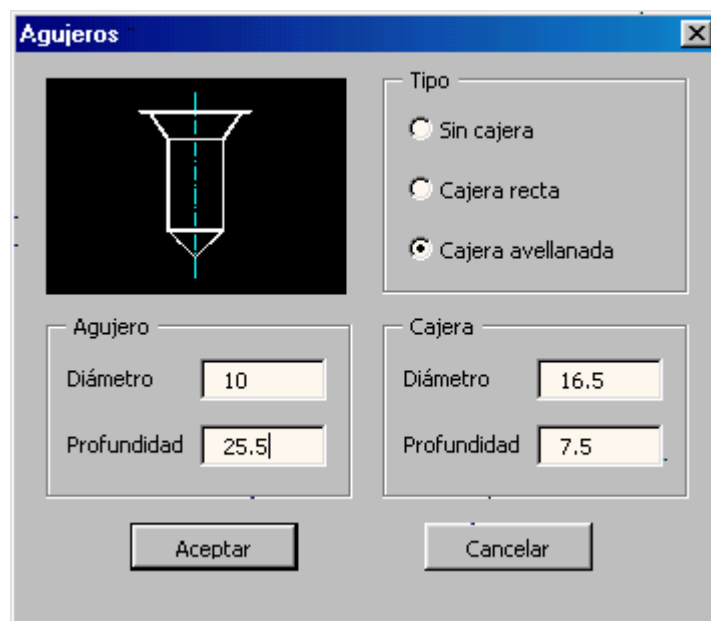
A continuación estudiaremos un programa que resulta muy jugoso como repaso de todo lo que hemos estudiado hasta ahora. Este programa maneja un cuadro de diálogo (formulario) que es el que se observa en la página siguiente.

Como vemos, resulta ser un programa para dibujar agujeros para tornillos en alzado. Se indica primero el tipo de agujero (con cajera recta, con cajera avellanada o sin cajera). Después hemos de introducir en las diferentes casillas los distintos valores necesarios para dibujar el agujero.

Al pulsar el botón *Aceptar* se nos solicitará el punto y el ángulo de inserción, tras lo cual se dibujará el agujero con la línea de ejes en una capa llamada `EJES`, con tipo de línea `TRAZO_Y_PUNTO` y color rojo.

El botón *Cancelar* termina el programa.

Veamos, tras el diseño del letrero, el código VBA de este programa —ya un poco complejo—.



Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en Visual Basic orientada a AutoCAD (VBA)

Option Explicit

```
Dim AcadDoc As Object, AcadUtil As Object
Dim AcadObj As Object, AcadEje As Object, AcadCapa As Object
Dim VPt0, VPt1, VPt2, VPt3, VPt4, VPt5
Dim VPtgj, VPt1Eje, VPt2Eje
Dim DiamAgujero0, ProfAgujero0, DiamCajera0, ProfCajera0
Dim Pt0(2) As Double
Dim Pt1(2) As Double, Pt2(2) As Double, Pt3(2) As Double
Dim Pt4(2) As Double, Pt5(2) As Double
Dim Ptgj(2) As Double, Pt1Eje(2) As Double, Pt2Eje(2) As Double
Dim Refent0 As Integer, Angulo As Variant, PI As Double
```

```
Private Sub CajeraAvellanada_Click()
    DiamCajera.Enabled = True
    ProfCajera.Enabled = True
    Label3.Enabled = True
    Label4.Enabled = True
    Imagen.Picture = LoadPicture("cajera_avellanada.bmp")
End Sub
```

```
Private Sub CajeraRecta_Click()
    DiamCajera.Enabled = True
    ProfCajera.Enabled = True
    Label3.Enabled = True
    Label4.Enabled = True
    Imagen.Picture = LoadPicture("cajera_recta.bmp")
End Sub
```

```
Private Sub Cancelar_Click()
    End
End Sub
```

```
Private Sub Dibujar_Click()
    On Error GoTo Error
    Chequear
    If Errores.Caption = "" Then Else Exit Sub
    formAgujeros.Hide
    Call AcadUtil.InitializeUserInput(1)
    VPt0 = AcadUtil.GetPoint(, "Punto: ")
    Pt0(0) = VPt0(0): Pt0(1) = VPt0(1): Pt0(2) = VPt0(2)
    Call AcadDoc.SetVariable("osmode", 512)
    Call AcadUtil.InitializeUserInput(1)
    Angulo = AcadDoc.Utility.GetAngle(VPt0, "Angulo (Cerca de): ")
    PI = 3.14159265359
    If SinCajera.Value = True Then
        VPtgj = VPt0
        DibAgujero
    Else
        DibCajera
        DibAgujero
    End If
    VPt1Eje = AcadDoc.Utility.PolarPoint(Pt0, Angulo + PI, 5)
    Pt1Eje(0) = VPt1Eje(0): Pt1Eje(1) = VPt1Eje(1): Pt1Eje(2) = VPt1Eje(2):
    VPt2Eje = AcadDoc.Utility.PolarPoint(Pt3, Angulo, 5)
    Pt2Eje(0) = VPt2Eje(0): Pt2Eje(1) = VPt2Eje(1): Pt2Eje(2) = VPt2Eje(2):
    Set AcadEje = AcadObj.AddLine(Pt1Eje, Pt2Eje)
    On Error Resume Next
    If IsEmpty(AcadDoc.Linetypes.Item("trazo_y_punto")) Then
        Call AcadDoc.Linetypes.Load("trazo_y_punto", "acadiso.lin")
    End If
```

Curso Práctico de Personalización y Programación bajo AutoCAD
Programación en Visual Basic orientada a AutoCAD (VBA)

```
If IsEmpty(AcadDoc.Layers.Item("ejes")) Then
    Set AcadCapa = AcadDoc.Layers.Add("ejes")
    AcadCapa.Linetype = "trazo_y_punto"
    AcadCapa.Color = 1
End If
AcadEje.Layer = "ejes"
Call AcadDoc.SetVariable("osmode", Refent0)

Open "agujeros.$vr" For Output As #1
    Write #1, DiamAgujero, ProfAgujero, DiamCajera, ProfCajera
Close #1
```

```
Error:
    MsgBox "¡NO VALE!", , "Mensaje de error"
End Sub
```

```
Private Sub DibCajera()
    VPt1 = AcadUtil.PolarPoint(Pt0, Angulo - (PI / 2), Val(DiamCajera) / 2)
    Pt1(0) = VPt1(0): Pt1(1) = VPt1(1): Pt1(2) = VPt1(2)
    VPt2 = AcadUtil.PolarPoint(Pt1, Angulo, Val(ProfCajera))
    Pt2(0) = VPt2(0): Pt2(1) = VPt2(1): Pt2(2) = VPt2(2)
    VPt3 = AcadUtil.PolarPoint(Pt2, Angulo + (PI / 2), Val(DiamCajera))
    Pt3(0) = VPt3(0): Pt3(1) = VPt3(1): Pt3(2) = VPt3(2)
    VPt4 = AcadUtil.PolarPoint(Pt3, Angulo + PI, Val(ProfCajera))
    Pt4(0) = VPt4(0): Pt4(1) = VPt4(1): Pt4(2) = VPt4(2)

    If CajeraAvellanada.Value = True Then
        VPt1 = AcadUtil.PolarPoint(Pt1, Angulo - (PI / 2), Val(ProfCajera) / 2): _
        Pt1(0) = VPt1(0): Pt1(1) = VPt1(1): Pt1(2) = VPt1(2)
    End If
    If CajeraAvellanada.Value = True Then
        VPt4 = AcadUtil.PolarPoint(Pt4, Angulo + (PI / 2), Val(ProfCajera) / 2): _
        Pt4(0) = VPt4(0): Pt4(1) = VPt4(1): Pt4(2) = VPt4(2)
    End If

    Call AcadObj.AddLine(Pt1, Pt2)
    Call AcadObj.AddLine(Pt2, Pt3)
    Call AcadObj.AddLine(Pt3, Pt4)
    VPtgj = AcadUtil.PolarPoint(Pt0, Angulo, Val(ProfCajera))
End Sub
```

```
Private Sub DibAgujero()
    Ptgj(0) = VPtgj(0): Ptgj(1) = VPtgj(1): Ptgj(2) = VPtgj(2)
    VPt1 = AcadUtil.PolarPoint(Ptgj, Angulo - (PI / 2), Val(DiamAgujero) / 2)
    Pt1(0) = VPt1(0): Pt1(1) = VPt1(1): Pt1(2) = VPt1(2)
    VPt2 = AcadUtil.PolarPoint(Pt1, Angulo, Val(ProfAgujero))
    Pt2(0) = VPt2(0): Pt2(1) = VPt2(1): Pt2(2) = VPt2(2)
    VPt3 = AcadUtil.PolarPoint(Pt2, Angulo + (PI / 2), Val(DiamAgujero) / 2)
    Pt3(0) = VPt3(0): Pt3(1) = VPt3(1): Pt3(2) = VPt3(2)
    VPt3 = AcadUtil.PolarPoint(Pt3, Angulo, Val(DiamAgujero) / 4)
    Pt3(0) = VPt3(0): Pt3(1) = VPt3(1): Pt3(2) = VPt3(2)
    VPt4 = AcadUtil.PolarPoint(Pt2, Angulo + (PI / 2), Val(DiamAgujero))
    Pt4(0) = VPt4(0): Pt4(1) = VPt4(1): Pt4(2) = VPt4(2)
    VPt5 = AcadUtil.PolarPoint(Pt4, Angulo + PI, Val(ProfAgujero))
    Pt5(0) = VPt5(0): Pt5(1) = VPt5(1): Pt5(2) = VPt5(2)

    Call AcadObj.AddLine(Pt1, Pt2)
    Call AcadObj.AddLine(Pt2, Pt4)
    Call AcadObj.AddLine(Pt4, Pt5)
    Call AcadObj.AddLine(Pt2, Pt3)
    Call AcadObj.AddLine(Pt3, Pt4)
```

End Sub

```
Private Sub Chequear()  
    If Val(DiamAgujero) <= 0 Then  
        Errores.Caption = "Diámetro de agujero debe ser mayor o igual que 0"  
        DiamAgujero.SelStart = 0: DiamAgujero.SelLength = Len(DiamAgujero)  
        DiamAgujero.SetFocus: Exit Sub  
    End If  
    If Val(ProfAgujero) <= 0 Then  
        Errores.Caption = "Profundidad de agujero debe ser mayor o igual que 0"  
        ProfAgujero.SelStart = 0: ProfAgujero.SelLength = Len(ProfAgujero)  
        ProfAgujero.SetFocus: Exit Sub  
    End If  
    If Cajera.Enabled = True Then  
        If Val(DiamCajera) <= Val(DiamAgujero) Then  
            Errores.Caption = "Diámetro de cajera debe ser mayor que el de agujero"  
            DiamCajera.SelStart = 0: DiamCajera.SelLength = Len(DiamCajera)  
            DiamCajera.SetFocus: Exit Sub  
        End If  
    End If  
    If Cajera.Enabled = True Then  
        If Val(ProfCajera) <= 0 Then  
            Errores.Caption = "Profundidad de cajera debe ser mayor o igual que 0"  
            ProfCajera.SelStart = 0: ProfCajera.SelLength = Len(ProfCajera)  
            ProfCajera.SetFocus: Exit Sub  
        End If  
    End If  
    Errores.Caption = ""  
End Sub
```

```
Private Sub UserForm_Initialize()  
    Set AcadDoc = GetObject(, "Autocad.Application").ActiveDocument  
    Set AcadUtil = AcadDoc.Utility  
    Set AcadObj = AcadDoc.ModelSpace  
  
    On Error Resume Next  
    Refent0 = AcadDoc.GetVariable("osmode")  
    Open "agujeros.$vr" For Input As #1  
        If Err.Description = "No se encontró el archivo" Then GoSub Defecto  
        Input #1, DiamAgujero0, ProfAgujero0, DiamCajera0, ProfCajera0  
        DiamAgujero = DiamAgujero0  
        ProfAgujero = ProfAgujero0  
        DiamCajera = DiamCajera0  
        ProfCajera = ProfCajera0  
    Close #1  
    Exit Sub
```

```
Defecto:  
    Open "agujeros.$vr" For Append As #1  
        Write #1, "10", "10", "20", "5"  
    Close #1  
    Open "agujeros.$vr" For Input As #1  
    Return  
End Sub
```

```
Private Sub SinCajera_Click()  
    DiamCajera.Enabled = False  
    ProfCajera.Enabled = False  
    Label3.Enabled = False  
    Label4.Enabled = False  
    Imagen.Picture = LoadPicture("sin_cajera.bmp")
```


End Sub

Iremos comentando cada procedimiento Sub por separado y no en el orden en que están en el listado, sino en uno quizá más lógico.

— (General)_(Declaraciones)

Aquí como siempre se declaran todas las variables que luego utilizaremos, tanto las de objeto (Object), como Variant y Double. Recordar la necesidad de tener un doble juego de variables, unas Variant y otras Double, para hacer trasvase al obtener un punto y luego querer utilizarlo.

— UserForm_Initialize()

Este es el procedimiento que se ejecuta nada más correr el programa, al inicializarse el formulario. Lo primero es lo de siempre, esto es, asignar a cada objeto de **AutoCAD** que vamos a necesitar su valor correspondiente. Después se utiliza la sentencia On Error Resume Next para controlar la apertura del archivo que vamos a explicar ahora. Luego se guarda en Refent0 el valor de la variable OSMODE de **AutoCAD**; ya veremos para qué.

Así como AutoLISP guarda los valores de las variables globales utilizadas hasta cerrar **AutoCAD**, con VBA no disponemos de esa ventaja. Es por ello que, dada la característica de los programas para **AutoCAD** que poseen la capacidad de almacenar los últimos valores utilizados como valores por defecto, nos vamos a inventar un método para que esto suceda también en nuestros programas VBA.

El sencillo método consiste simplemente en crear un archivo de texto donde se almacenarán, en cada ejecución del programa, los últimos valores utilizados. De este modo, al correr de nuevo el programa, se leerán dichos valores y se introducirán en el cuadro para ofrecerlos por defecto.

De esta manera intentamos leer el archivo que almacenará los valores (AGUJEROS.\$VR) en el directorio actual. Si no existiera se produciría un error, por lo que la ejecución sigue en la siguiente línea (recordemos el On Error Resume Next). En esta línea se compara el texto del error con el que significa que el archivo no se ha encontrado y, si fueran iguales, la ejecución se dirige a la subrutina Defecto donde se crea y se le añaden unos valores por defecto.

Tanto si existiera como si estuviera recién creado, se continúa la ejecución leyendo los valores del archivo e introduciéndolos en el cuadro.

NOTA: Es más lógico utilizar los valores numéricos de Err en lugar de sus descripciones, ya que podría utilizarse así en cualquier plataforma idiomática VBA. Para hallar el número de un error (si no disponemos de una lista), sólo hemos de provocarlo y extraer el valor con Err.Number.

— SinCajera_Click()

Este procedimiento y los dos siguientes dicen referencia a la hora de hacer clic en alguno de los tres botones excluyentes para elegir el tipo de agujero. Este concretamente responde al evento de hacer clic en el botón excluyente *Sin cajera*. Al hacerlo, tanto la casillas de profundidad de cajera como la de diámetro de la cajera, así como sus etiquetas, deben inhabilitarse. También se refleja en el cuadro de imagen el archivo .BMP correspondiente.

— CajeraRecta_Click()

Al igual que en el anterior procedimiento explicado, en `CajeraRecta_Click()` se muestra la imagen correspondiente en el cuadro de imagen y se establecen como habilitadas las casillas y etiquetas de profundidad y diámetro de cajera por si al hacer clic en *Cajera recta* se proviniera de *Sin cajera*, la cual las desactiva como sabemos.

— CajeraAvellanada_Click()

Así también, en este procedimiento `Sub` se muestra la imagen correspondiente con una cajera avellanada en el cuadro de imagen, y también se establecen como habilitadas las casillas y etiquetas de profundidad y diámetro de cajera por si al hacer clic en *Cajera avellanada* se proviniera de *Sin cajera*.

— Dibujar_Click()

Este es el procedimiento que arranca al ser pulsado el botón *Aceptar*. Lo primero que hace es definir una rutina de errores que controlará salidas no deseadas, por ejemplo (como al pulsar `ESC`), u otros errores no deseados. Después se llama al procedimiento de chequeo de casillas, el cual se comentará seguido de éste.

Tras ocultar el formulario (letrero de diálogo) se pregunta por el punto de inserción del agujero (sin admitir `INTRO` como respuesta) y se guardan sus coordenadas en `VPt0`. Seguidamente se hace el trasvase de variables con `Pto0`, que será la que se utilice para el dibujo.

Se establece el valor de `OSMODE` a 512 (*Cercano*) y se pide el ángulo de inserción. Y tras establecer el valor de `PI` se comprueba si el agujero tiene cajera o no. Si no tuviera se llama únicamente a la rutina de dibujo del agujero y, si tuviera cajera (recta o avellanada), se llama primero a la rutina que dibuja la cajera y luego a la del agujero. Además, el hecho de no tener cajera hace que el punto de inserción sea igual al primer punto de dibujo del agujero

Las sentencias siguientes se corresponden con los cálculos de los puntos del eje de simetría. Además se carga el tipo de línea —si no está cargado— y se crea la capa —si no existe— que pertenecerán al eje. A esta última se le asigna el tipo de línea cargado y el color rojo.

Por último se reasigna a la variable `OSMODE` su valor original (por eso lo guardamos en `Refent0`) y se guardan los valores utilizados en el archivo de valores por defecto.

NOTA: Recuérdese que antes de finalizar este procedimiento ha habido que pasar por otros dos o tres: el de chequeo, el de dibujo de cajera y el de dibujo de agujero. Estos se estudian ahora por ese orden.

— Chequear()

Aquí se comprueban los valores de cada casilla y, si alguno estuviera errado, se muestra un mensaje en una línea de errores inferior (que es una etiqueta), se selecciona el texto de la casilla y se sale de procedimiento.

Al haber error el texto de la línea inferior es diferente que una cadena vacía. Si no hay error este texto es igual a la cadena vacía (" "). Repásese el código del procedimiento anterior para ver cómo se controla después esto.

— DibCajera()

Para dibujar la cajera se calculan todos los puntos necesarios y se dibuja. También se

tiene en cuenta si es recta o avellanada.

— DibAgujero()

Para dibujar el agujero se calculan todos los puntos y se dibuja.

NOTA: Recuérdese que tras este procedimiento *Sub* se sigue en *Dibujar_Click()*.

— Cancelar_Click()

Este *Sub* únicamente dice que al pulsar el botón *Cancelar* se acabe el programa, sin más.

NOTA: Evidentemente, para que este programa funcione, habrán de estar los archivos .BMP en el directorio actual de trabajo.

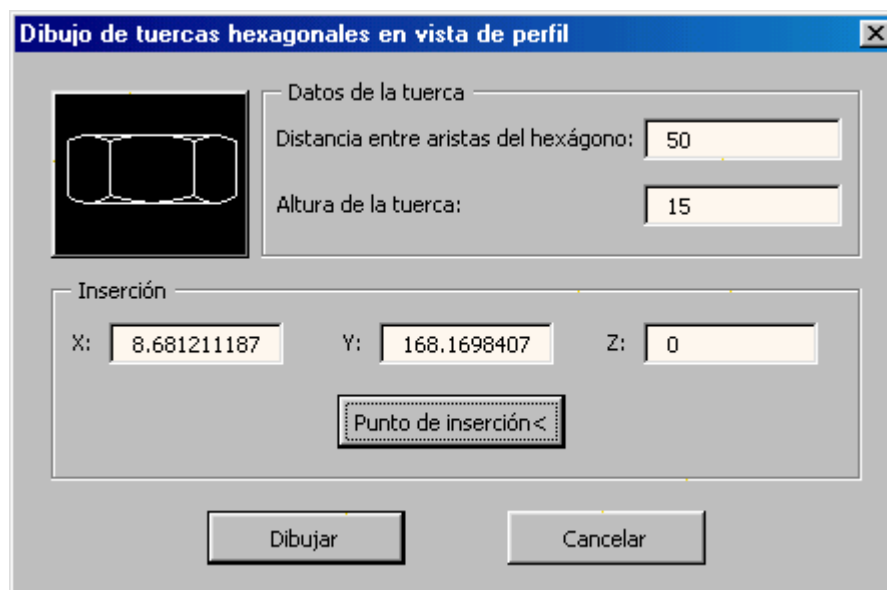
4ª fase intermedia de ejercicios

- Escribir una programa que maneje el cuadro de diálogo que se puede ver en la página siguiente.

El programa dibuja tuercas hexagonales en vista de perfil. Los datos de la distancia entre aristas y la altura de la tuerca se introducen en las casillas correspondientes. El botón *Punto de inserción<* sale del letrero para marcar un punto en pantalla. Al hacerlo vuelve al letrero y escribe las coordenadas en sus casillas correspondientes.

Una vez introducidos todos los datos necesarios, el botón *Dibujar* realiza el dibujo y acaba el programa. El botón *Cancelar* simplemente termina la aplicación.

Introdúzcase algún control de entrada de datos del usuario, así como unos valores por defecto con los que arranque el cuadro (se puede hacer en tiempo de diseño), y también marcas de *inicio* y *fin* de DESHACER.



DOCE.10. EL TRAZADO

El trazado o *plotteado* de dibujos desde el VBA de **AutoCAD** se maneja mediante el objeto de trazado llamado `Plot`. Si comprobamos la lista jerárquica de objetos, éste es descendiente directo del documento actual activo (`ActiveDocument`).

Como siempre, si vamos a utilizarlo mucho en un programa podemos declarar una variable para él, y después asignarle su valor, por ejemplo así:

```
Option Explicit
```

```
Dim AcadDoc As Object  
Dim AcadPlot As Object
```

y luego:

```
Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument  
Set AcadPlot = AcadDoc.Plot
```

Propiedades del objeto de trazado:

AdjustAreaFill	Application	HideLines	Origin
PaperSize	PlotOrientation	PlotScale	PlotUnits
Rotation			

Métodos del objeto de trazado:

LoadPC2	PlotExtents	PlotLimits	PlotPreview
PlotToDevice	PlotToFile	PlotView	PlotWindow
PlotWithConfigFile	SavePC2		

Vemos que este objeto posee diferentes propiedades (algunas ya conocidas) y métodos que habremos de utilizar con conveniencia para asegurar un trazado fidedigno. Por ello vamos a pasar directamente a su explicación.

- **AdjustAreaFill**. Obtiene y/o asigna el estado actual de activación de ajuste de área de relleno para el trazado. La variable que lo recoja o el valor que se asigne serán ambos tipo `Boolean`; siendo `True` la opción para activado y `False` la opción para desactivado.

La sintaxis para asignar un valor es la siguiente:

```
ObjetoTrazado.AdjustAreaFill = BooAjusteRelleno
```

La sintaxis para obtener el valor actual es la que sigue:

```
BooAjusteRelleno = ObjetoTrazado.AdjustAreaFill
```

- **HideLines**. Obtiene y/o asigna el estado actual de ocultación de líneas para el trazado. La variable que lo recoja o el valor que se asigne serán ambos tipo `Boolean`; siendo `True` la opción para activado y `False` la opción para desactivado.

La sintaxis para asignar un valor es la siguiente:

```
ObjetoTrazado.HideLines = BooLíneasOcultas
```

La sintaxis para obtener el valor actual es la que sigue:

```
BoolLineasOCultas = ObjetoTrazado.HideLine
```

NOTA: Esta propiedad se refiere a los objetos del Espacio Modelo. Para ocultar líneas en las ventanas de Espacio Papel utilizamos la propiedad ya estudiada `RemoveHiddenLines` del objeto de ventana de Espacio Papel.

- `PaperSize`. Obtiene el tamaño actual para el papel de impresión o trazado. También permite asignar un tamaño. La sintaxis para ello es:

```
ObjetoTrazado.PaperSize = DblMatrizTamaños
```

La sintaxis para obtener el tamaño de papel actual asignado:

```
VarMatrizTamaños = ObjetoTrazado.PaperSize
```

`DblMatrizTamaños` es una matriz, tabla o *array* de dos elementos `Double`. El primero de ellos dice relación a la anchura del papel, y el segundo a la altura del mismo.

- `PlotOrientation`. Esta propiedad `PlotOrientation` permite especificar y obtener una orientación para el papel de trazado.

Para asignar:

```
ObjetoTrazado.PlotOrientation = IntOrientación
```

Para obtener:

```
IntOrientación = ObjetoTrazado.PlotOrientation
```

`IntOrientation` es un valor `Integer` que además admite las constantes siguientes:

`acPlotOrientationPortrait`

`acPlotOrientationLandscape`

Ambas constantes se refieren a la orientación vertical (*portrait*) o a la orientación apaisada (*landscape*).

- `PlotScale`. `PlotScale` permite especificar y obtener una escala para el trazado. Para asignar:

```
ObjetoTrazado.PlotScale = DblMatrizEscala
```

Para obtener:

```
VarMatrizEscala = ObjetoTrazado.PlotScale
```

`DblMatrizEscala` es una matriz de dos valores tipo `Double`. El primero indica las unidades de trazado y el segundo las unidades de dibujo, es decir, unidades en el papel y unidades en el dibujo.

NOTA: Si ambos valores son igual a 0, la escala se corresponde con *Escala hasta ajustar* del letrero de trazado/impresión de **AutoCAD**.

Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

- **PlotUnits.** `PlotUnits` permite obtener y/o asignar las unidades del trazado, es decir o milímetros (métricas) o pulgadas (inglesas). Para asignar:

```
ObjetoTrazado.PlotUnits = IntUnidades
```

Para obtener:

```
IntUnidades = ObjetoTrazado.PlotUnits
```

`IntUnidades` es un valor `Integer` que también admite las siguientes constantes:

`acEnglish` `acMetric`

Las demás propiedades ya están estudiadas. Decir que, obviamente, la rotación y el origen como propiedades en este objeto `Plot` (`Rotation` y `Origin`), se refieren a la rotación y al origen del trazado —con el mismo significado que en el cuadro de diálogo de **AutoCAD**—. Pasemos a estudiar los métodos.

- **LoadPC2.** Carga las especificaciones de trazado del archivo `.PC2` especificado:

```
ObjTrazado.LoadPC2(StrNombrePC2)
```

`StrNombrePC2` es un cadena (`String`) que indica el nombre del archivo y su ubicación, por ejemplo:

`Option Explicit`

```
Dim AcadDoc As Object  
Dim AcadPlot As Object
```

```
Sub Macro()  
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument  
    Set AcadPlot = AcadDoc.Plot  
  
    AcadPlot.LoadPC2 ("c:\autocad\xtras\cfgplot.pc2")  
End Sub
```

NOTA: Estos archivos `.PC2` tienen información del dispositivo de trazado, además de todo el resto de características del trazado en sí. No así ocurría con los archivos `.PCP` (de versiones anteriores a la 14 de **AutoCAD**), que únicamente guardaban (y guardan, si se quieren utilizar) información del trazado; y se materializaban en el dispositivo actualmente establecido.

- **PlotExtents.** Define como área de trazado la extensión del dibujo. Hay que tener en cuenta que esta extensión no se actualiza cuando hay reducciones en el dibujo, por lo que es recomendable hacer un *Zoom Extensión* antes de trazar el dibujo:

```
ObjTrazado.PlotExtents
```

- **PlotLimits.** Define como área de trazado los límites del dibujo. Equivale, como sabemos, a realizar un *Zoom Todo*:

```
ObjTrazado.PlotLimits
```

Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

- **PlotPreview.** Muestra la presentación preliminar típica de **AutoCAD** en cualquiera de sus dos versiones (según se indique), parcial o total:

```
ObjTrazado.PlotPreview(IntModoPresPrel)
```

IntModoPresPrel es un valor Integer que admite también las siguientes constantes:

acPartialPreview *acFullPreview*

- **PlotToDevice.** Realiza el trazado del dibujo en el dispositivo cuyo nombre se indica (String). Este nombre es aquel con el que se ha configurado dicho dispositivo. La lista de todos los dispositivos de impresión se encuentra tras el comando **PREFERENCIAS** (**PREFERENCES** en versiones sajonas) de **AutoCAD**, en la pestaña *Impresora*:

```
ObjTrazado.PlotToDevice(StrDispositivo)
```

NOTA: Si *StrDispositivo* es una cadena vacía o nula (""), el trazado se envía al dispositivo trazador configurado por defecto actualmente.

- **PlotToFile.** Realiza el trazado del dibujo en el archivo cuyo nombre se indica (String). Por defecto, la extensión de los archivos de trazado es **.PLT**.

```
ObjTrazado.PlotToFile(StrArchivo)
```

- **PlotView.** Define como área de trazado la vista cuyo nombre se indica como cadena (String):

```
ObjTrazado.PlotView(StrVista)
```

- **PlotWindow.** Define como área de trazado la ventana cuyas esquinas se especifican:

```
ObjTrazado.PlotWindow(DblEsquinaSupIz, DblEsquinaInfDcha)
```

Las dos esquinas (superior izquierda e inferior derecha) son matrices de tres elementos Double cada una (X, Y y Z en el SCU).

- **PlotWithConfigFile.** Realiza el trazado del dibujo de acuerdo con las especificaciones que en el archivo de configuración se indican. Este archivo puede ser un **.PCP** o un **.PC2**:

```
ObjTrazado.PlotWithConfigFile(StrArchivoConfiguración)
```

StrArchivoConfiguración es un valor de cadena (String) con el nombre y la ruta del archivo en cuestión.

- **SavePC2.** Como contraposición a **LoadPC2**, **SavePC2** guarda las especificaciones de trazado en el archivo **.PC2** especificado:

```
ObjTrazado.SavePC2(StrNombrePC2)
```

StrNombrePC2 es un cadena (String) que indica el nombre del archivo y su ubicación.

El procedimiento típico para imprimir un dibujo desde VBA puede parecer extraño, pero no lo es tanto. Teniendo en cuenta que únicamente los métodos **PlotToDevice**, **PlotToFile** y

`PlotWithConfigFile` envían realmente el documento al trazador o a la impresora, hemos de diseñar un algoritmo que nos permita seguir determinados pasos para la impresión típica de un dibujo de **AutoCAD**.

Evidentemente si deseamos enviar el trazado a un archivo, utilizaremos el método `PlotToFile` del objeto de trazado `Plot`. Si ya disponemos de un archivo de configuración de trazado `.PCP` o `.PC2` a nuestro gusto, utilizaremos el método `PlotWithConfigFile`.

Pero lo lógico será que nuestro programa se ejecute en máquinas de las cuales no conocemos su configuración y, por lo tanto, no sabremos qué impresoras o qué *plotters* tienen instalados sus usuarios; o incluso si disponen de alguno instalado.

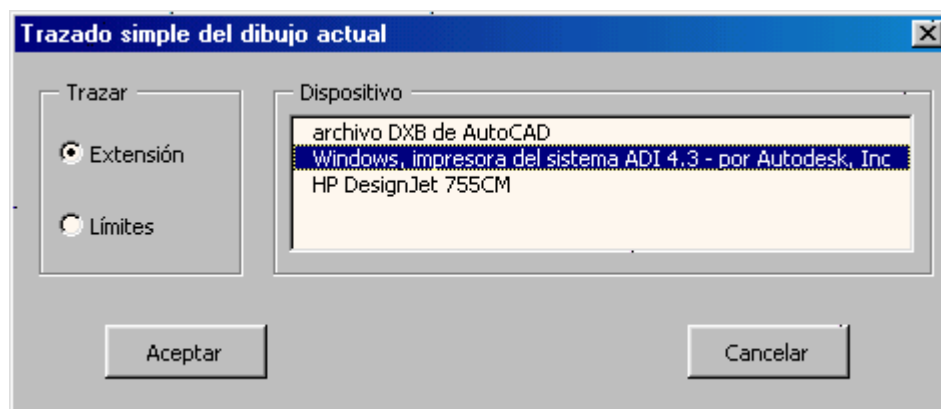
Por lo tanto, los pasos lógicos a la hora de imprimir un dibujo serán los siguientes:

- Averiguar qué dispositivos de trazado hay instalados.
- Permitir al usuario elegir entre cualquiera de ellos.
- Permitir al usuario elegir las preferencias para la impresión o trazado.
- Enviar el dibujo al dispositivo elegido.

Para averiguar los dispositivos sólo tenemos que extraer en un bucle todos los valores de la variable `PLOTID`, que guarda la descripción del trazador configurado por defecto, apoyándonos en la variable `PLOTTER`, que cambia el trazador por defecto.

Con este pequeño truco, la mayor parte del problema está resuelto. Pero para verlo correctamente, comentaremos un programa que permite imprimir el dibujo actual, ya sea en su extensión o en sus límites.

Veamos cómo sería el letrero de diálogo (formulario en VBA) que manejaría este programa, por ejemplo:



Y ahora se muestra el código VBA de susodicho:

Option Explicit

```
Dim AcadDoc As Object  
Dim AcadPlot As Object
```

```
Dim StrPlot As String  
Dim NumPlot As Integer
```

```
Private Sub UserForm_Initialize()  
    On Error Resume Next
```


Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

```
Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument
Set AcadPlot = AcadDoc.Plot

NumPlot = 0
Do
    Call AcadDoc.SetVariable("PLOTTER", NumPlot)
    If Err.Number <> 0 Then Exit Do
    StrPlot = AcadDoc.GetVariable("PLOTID")
    Dispositivo.AddItem StrPlot, NumPlot
    NumPlot = NumPlot + 1
Loop
End Sub
```

```
Private Sub Aceptar_Click()
    Dim Índice As Integer
    Dim Papel(1 To 2) As Double
    Dim Escala(1 To 2) As Double
    Dim Origen(1 To 3) As Double

    If Dispositivo.ListIndex = -1 Then MsgBox "Escoja un dispositivo": Exit Sub
    Índice = Dispositivo.ListIndex

    AcadPlot.PlotUnits = acMetric
    Papel(1) = 210: Papel(2) = 297
    AcadPlot.PaperSize = Papel
    AcadPlot.PlotOrientation = acPlotOrientationPortrait
    Escala(1) = 0: Escala(2) = 0
    AcadPlot.PlotScale = Escala
    Origen(1) = 0: Origen(2) = 0: Origen(3) = 0
    AcadPlot.Origin = Origen
    AcadPlot.Rotation = 0
    AcadPlot.AdjustAreaFill = True
    AcadPlot.HideLines = True
    If Extensión.Value = True Then
        AcadPlot.PlotExtents
    Else
        AcadPlot.PlotLimits
    End If

    Call AcadDoc.SetVariable("PLOTTER", Índice)
    StrPlot = AcadDoc.GetVariable("PLOTID")
    AcadPlot.PlotToDevice (StrPlot)
End Sub
```

```
Private Sub Cancelar_Click()
    End
End Sub
```

Comentaremos el programa por procedimientos Sub.

— (General)_(Declaraciones)

Se declaran las variables de objeto típicas (objetos de **AutoCAD**) y alguna más que nos será útil a lo largo del programa.

— UserForm_Initialize()

Aquí, lo primero que se hace es establecer la condición que dice que si existiera algún error, se continuaría en la línea siguiente (veremos para qué). Inmediatamente después se

inician las variables de los objetos de **AutoCAD** (se les da valor).

Las dos variables que hemos declarado al principio las vamos a utilizar aquí. `NumPlot` va a guardar un valor entero que va a ser el número de trazador, además del índice del cuadro de lista. `StrPlot` será una variable de cadena que almacenará las distintas descripciones de los diferentes trazadores. Recordemos que al método `PlotToDevice` hay que darle un nombre de descripción de trazador.

Lo que hace la siguiente serie de sentencias de este `Sub` es rellenar el cuadro de lista con las descripciones de los trazadores, impresoras u otros dispositivos (archivo en formato DXB, por ejemplo). Se inicializa `NumPlot` a 0. Se sigue con una estructura repetitiva aparentemente infinita; pero no es así. Veamos:

- Se introduce el valor de `NumPlot` (0) en la variable de sistema `PLOTTER` que, como hemos dicho, almacena un número que se corresponde con el trazador actual configurado. El valor de `PLOTTER` va desde 0 hasta el número de configuraciones de trazador menos uno (si son tres como en el ejemplo, serán 0, 1 y 2); hasta un máximo de 29. El truco está en que, si damos un valor por encima de dicho número, `PLOTTER` devuelve un mensaje de error. Este mensaje es el que queremos captar con el `On Error Resume Next` del principio y con la siguiente sentencia `If Err.Number <> 0 Then Exit Do`.
- Por lo tanto, si ha habido error se sale del bucle. Si no hay error se continúa extrayendo la descripción del trazador actual y guardándola en `StrPlot`.
- A continuación se añade esa descripción al cuadro de lista con el índice igual al número de trazador.
- Se incrementa el valor de `NumPlot` en una unidad con el contador-suma (`NumPlot = NumPlot + 1`) y se vuelve a empezar. Ahora el trazador por defecto será otro y la descripción otra. Y así sucesivamente.
- En el momento en que se hayan añadido todos los dispositivos configurados, se produce el error controlado y se sale del bucle.

Sigamos con lo que ocurre al pulsar el botón *Aceptar*.

— `Aceptar_Click()`

Se dimensionan algunas variables que vamos a utilizar. A continuación se controla si la propiedad `ListIndex` del cuadro de lista es igual a -1, lo que querría decir que no se ha seleccionado trazador alguno. En este caso se muestra un mensaje y se sale del procedimiento `Sub`, permitiendo así al usuario que elija un trazador.

En el momento en que se elija uno se hace su índice igual a la variable *Índice*. Recordemos que el índice era igual al número del trazador.

Se establece toda una serie de valores para la escala, las líneas ocultas, etcétera. Algunos de ellos no pueden estar dispuestos en cualquier orden. Por norma general conviene hacer caso a la lógica y dar valores según se haría en **AutoCAD**. Así, resulta lógico establecer primero las unidades, luego el tamaño del papel, la orientación, la escala, el origen, la rotación...

Se controla también si el usuario eligió un *plotteo* de la extensión o de los límites, actuando en consecuencia con el método `PlotExtents` o `PlotLimits`.

Por fin se establece en la variable de sistema de **AutoCAD** `PLOTTER` el valor del trazador (o índice) escogido y se extrae de `PLOTID` la descripción de ese trazador. Esta descripción se proporciona al método `PlotToDevice` y el proceso ha concluido.

— Cancelar_Click()

Aquí simplemente se termina el programa, sin más.

5ª fase intermedia de ejercicios

- Compréndase perfectamente el método de impresión explicado y aplíquese a cualquiera de los ejercicios anteriores que dibujen objetos.

DOCE.11. EL OBJETO DE PREFERENCIAS

Este es el último objeto que estudiaremos. El objeto de preferencias, llamado *Preferences*, controla todas las preferencias u opciones de **AutoCAD**. Podríamos decir que es el equivalente al cuadro que despliega el comando *PREFERENCIAS* (*PREFERENCES* en versión inglesa) de **AutoCAD**, más o menos.

Como comprobamos, si retomamos la tabla jerárquica de objetos del principio de este **MÓDULO**, el objeto *Preferences* “cuelga” directamente del objeto raíz de aplicación (*Application*). Es por ello que si, como siempre, deseamos utilizarlo mucho, podemos crear una variable de acceso directo de la siguiente forma, por ejemplo (continuando la notación que venimos utilizando):

```
Option Explicit
```

```
Dim AcadApp As Object  
Dim AcadPref As Object
```

para luego dar valores así:

```
Set AcadApp = GetObject(, "AutoCAD.Application")  
Set AcadPref = AcadApp.Preferences
```

Propiedades del objeto de preferencias:

ActiveProfile	AltFontFile	AltTabletMenuFile
Application	ArcSmoothness	AutoAudit
AutoSaveFile	AutoSaveInterval	BeepOnError
ConfigFile	ContourLinesPerSurface	CreateBackup
CrosshairColor	CursorSize	CustomDictionary
DefaultInternetURL	DemandLoadARXApp	DisplayDraggedObject
DisplayScreenMenu	DisplayScrollBars	DisplaySilhouette
DockedVisibleLines	DriversPath	EnableStartupDialog
FontFileMap	FullCrcValidation	GraphicFont
GraphicFontSize	GraphicFontStyle	GraphicsTextBackgrndColor
GraphicsTextColor	GraphicsWinBackgrndColor	HistoryLines
HelpFilePath	IncrementalSavePercent	KeyboardAccelerator
KeyboardPriority	LicenseServer	LogFileName
LogFileOn	MainDictionary	MaxActiveViewports
MaxAutoCADWindow	MaxNumOfSymbols	MeasurementUnits
MenuFile	MonochromeVectors	PersistentLISP
PostScriptPrologFile	PrintFile	PrintSpoolerPath
PrintSpoolExecutable	ProxyImage	RenderSmoothness
SavePreviewThumbnail	SegmentPerPolyline	ShowProxyDialogBox
ShowRasterImage	SupportPath	TempFileExtension
TempFilePath	TemplateDWGPath	TempXRefPath

TextEditor	TextFont	TextFontSize
TextFontStyle	TextFrameDisplay	TextureMapPath
TextWinBackgrndColor	TextWinTextColor	XRefDemandLoad

Métodos del objeto de preferencias:

DeleteProfile	ExportProfile	GetProjectFilePath
ImportProfile	ResetProfile	SetProjectFilePath

La ingente cantidad de propiedades que hemos de explicar la dividiremos en grupos temáticos para su óptimo aprendizaje. Estos grupos se corresponden (la mayoría) con las diferentes pestañas del cuadro *Preferencias* de **AutoCAD**. Comencemos pues sin más dilación.

DOCE.11.1. Preferencias de archivos

- **AltFontFile**. Especifica la localización de una fuente (tipo de letra) alternativa para el caso en que **AutoCAD** no encuentre la fuente original y en el mapeado del archivo no esté especificada otra.

La sintaxis para asignar:

```
ObjetoPreferencias.AltFontFile = StrArchivoFuente
```

La sintaxis para obtener:

```
StrArchivoFuente = ObjetoPreferencias.AltFontFile
```

StrArchivoFuente (nombre y ruta al archivo) es tipo de dato *String*.

- **AltTabletMenuFile**. Especifica la localización de un archivo de menú de tableta alternativo para intercambiar con el estándar de **AutoCAD**.

La sintaxis para asignar:

```
ObjetoPreferencias.AltTabletMenuFile = StrArchivoMenúTablero
```

La sintaxis para obtener:

```
StrArchivoMenúTablero = ObjetoPreferencias.AltTabletMenuFile
```

StrArchivoMenúTablero (nombre y ruta al archivo) es tipo de dato *String*.

- **AutoSaveFile**. Especifica la localización del archivo de guardado automático.

La sintaxis para asignar:

```
ObjetoPreferencias.AutoSaveFile = StrArchivoGuardAut
```

La sintaxis para obtener:

```
StrArchivoGuardAut = ObjetoPreferencias.AltFontFile
```

StrArchivoGuardAut (nombre y ruta al archivo) es tipo de dato *String*.

Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

- `ConfigFile`. Obtiene la localización del archivo de configuración de **AutoCAD** (por defecto `ACAD14.CFG`).

Su sintaxis es:

```
StrArchivoConfig = ObjetoPreferencias.ConfigFile
```

StrArchivoConfig (nombre y ruta al archivo) es tipo de dato `String`.

NOTA: La manera de asignar otra ubicación a este archivo de configuración se explicará en el **APÉNDICE H**, sobre la configuración general del programa, ya que esto se realiza desde las propiedades del acceso directo al ejecutable principal.

- `CustomDictionary`. Especifica la localización de un diccionario personalizado si existe.

La sintaxis para asignar:

```
ObjetoPreferencias.CustomDictionary = StrArchivoDiccionario
```

La sintaxis para obtener:

```
StrArchivoDiccionario = ObjetoPreferencias.CustomDictionary
```

StrArchivoDiccionario (nombre y ruta al archivo) es tipo de dato `String`.

- `DefaultInternetURL`. Especifica la localización URL del sitio WWW configurado por defecto.

La sintaxis para asignar:

```
ObjetoPreferencias.DefaultInternetURL = StrURL
```

La sintaxis para obtener:

```
StrURL = ObjetoPreferencias.DefaultInternetURL
```

StrURL es tipo de dato `String`.

- `DriversPath`. Especifica el directorio en el cual **AutoCAD** buscará controladores (*drivers*) de dispositivos ADI para el monitor, dispositivos señaladores, impresoras y trazadores.

La sintaxis para asignar:

```
ObjetoPreferencias.DriversPath = StrDirectorioControladores
```

La sintaxis para obtener:

```
StrDirectorioControladores = ObjetoPreferencias.DriversPath
```

StrDirectorioControladores es tipo de dato `String`.

- `FontFileMap`. Especifica la localización del archivo de representación de tipos de letra o fuentes, es decir, el archivo que define cómo **AutoCAD** convertirá las fuentes que no pueda encontrar.

La sintaxis para asignar:

```
ObjetoPreferencias.FontFileMap = StrArchivoRepFuentes
```

La sintaxis para obtener:

```
StrArchivoRepFuentes = ObjetoPreferencias.FontFileMap
```

StrArchivoRepFuentes (nombre y ruta al archivo) es tipo de dato String.

- *HelpFilePath*. Especifica la localización del archivo de ayuda de **AutoCAD**.

La sintaxis para asignar:

```
ObjetoPreferencias.HelpFilePath = StrDirectorioAyuda
```

La sintaxis para obtener:

```
StrDirectorioAyuda = ObjetoPreferencias.HelpFilePath
```

StrDirectorioAyuda es tipo de dato String.

- *LicenseServer*. Obtiene una lista de licencias de proveedor disponibles.

Su sintaxis es:

```
StrListaLicencias = ObjetoPreferencias.LicenseServer
```

StrListaLicencias es tipo de dato String.

- *LogFileName*. Especifica la localización del archivo de registro de **AutoCAD** (por defecto *ACAD.LOG*).

La sintaxis para asignar:

```
ObjetoPreferencias.LogFileName = StrArchivoRegistro
```

La sintaxis para obtener:

```
StrArchivoRegistro = ObjetoPreferencias.LogFileName
```

StrArchivoRegistro (nombre y ruta al archivo) es tipo de dato String.

- *MainDictionary*. Especifica la localización del diccionario de corrección ortográfica principal.

La sintaxis para asignar:

```
ObjetoPreferencias.MainDictionary = StrArchivoDiccPrincipal
```

La sintaxis para obtener:

```
StrArchivoDiccPrincipal = ObjetoPreferencias.MainDictionary
```

StrArchivoDiccPrincipal (nombre y ruta al archivo) es tipo de dato String.

- `MenuFile`. Especifica la localización del archivo de menú principal.

La sintaxis para asignar:

```
ObjetoPreferencias.MenuFile = StrArchivoMenúPrincipal
```

La sintaxis para obtener:

```
StrArchivoMenúPrincipal = ObjetoPreferencias.MenuFile
```

`StrArchivoMenúPrincipal` (nombre y ruta al archivo) es tipo de dato `String`.

- `PostScriptPrologFile`. Especifica el nombre de una sección de prólogo personalizada en el archivo `ACAD.PSF`.

La sintaxis para asignar:

```
ObjetoPreferencias.PostScriptPrologFile = StrNomSecciónPrólogo
```

La sintaxis para obtener:

```
StrNomSecciónPrólogo = ObjetoPreferencias.PostScriptPrologFile
```

`StrNomSecciónPrólogo` es tipo de dato `String`.

- `PrintFile`. Especifica un nombre para el archivo temporal de trazado. Por defecto es el nombre del dibujo más la extensión `.PLT`.

La sintaxis para asignar:

```
ObjetoPreferencias.PrintFile = StrArchivoTrazado
```

La sintaxis para obtener:

```
StrArchivoTrazado = ObjetoPreferencias.PrintFile
```

`StrArchivo` es tipo de dato `String`.

NOTA: Para aceptar el nombre por defecto (nombre de archivo más extensión `.PLT`) se introduce un punto (.) como `StrArchivoTrazado`.

- `PrintSpoolerPath`. Especifica el directorio para los archivos de trazado diferido.

La sintaxis para asignar:

```
ObjetoPreferencias.PrintSpoolerPath = StrDirectorioTrazadoDif
```

La sintaxis para obtener:

```
StrDirectorioTrazadoDif = ObjetoPreferencias.PrintSpoolerPath
```

`StrDirectorioTrazadoDif` es tipo de dato `String`.

- `PrintSpoolExecutable`. Especifica el nombre del ejecutable para el trazado diferido.

La sintaxis para asignar:

```
ObjetoPreferencias.PrintSpoolExecutable = StrEjecTrazadoDif
```

La sintaxis para obtener:

```
StrEjecTrazadoDif = ObjetoPreferencias.PrintSpoolExecutable
```

StrEjecDrazadoDif es tipo de dato String.

- **SupportPath**. Especifica las diversas localizaciones —si hay más de una han de ir separadas por caracteres de punto y coma (;)— donde **AutoCAD** buscará archivos de soporte.

La sintaxis para asignar:

```
ObjetoPreferencias.SupportPath = StrDirectorioSoporte
```

La sintaxis para obtener:

```
StrDirectorioSoporte = ObjetoPreferencias.SupportPath
```

StrDirectorioSoporte es tipo de dato String.

- **TempFilePath**. Especifica la localización de archivos temporales.

La sintaxis para asignar:

```
ObjetoPreferencias.TempFilePath = StrDirectorioTemp
```

La sintaxis para obtener:

```
StrDirectorioTemp = ObjetoPreferencias.TempFilePath
```

StrDirectorioTemp es tipo de dato String.

- **TemplateDWGPath**. Especifica la localización de archivos de plantilla, utilizados por los asistentes de inicio.

La sintaxis para asignar:

```
ObjetoPreferencias.TemplateDWGPath = StrDirectorioPlantillas
```

La sintaxis para obtener:

```
StrDirectorioPlantillas = ObjetoPreferencias.TemplateDWGPath
```

StrDirectorioPlantillas es tipo de dato String.

- **TempXRefPath**. Especifica la localización de archivos de referencia externa. Esta localización se utiliza si elegimos la constante `acDemandLoadEnabledWithCopy` con la propiedad `XRefDemandLoad` que veremos al término de la próxima sección.

La sintaxis para asignar:

```
ObjetoPreferencias.TempXRefPath = StrDirectorioTempRefX
```


La sintaxis para obtener:

```
StrDirectorioTempRefX = ObjetoPreferencias.TempXRefPath
```

StrDirectorioTempRefX es tipo de dato String.

- *TextEditor*. Especifica el nombre para un editor de texto externo con el comando TEXTOM (MTEXT en inglés).

La sintaxis para asignar:

```
ObjetoPreferencias.TextEditor = StrNombreEditor
```

La sintaxis para obtener:

```
StrNombreEditor = ObjetoPreferencias.TextEditor
```

StrNombreEditor es tipo de dato String.

- *TextureMapPath*. Especifica la localización del directorio contenedor de archivos de mapas de texturas para renderizar.

La sintaxis para asignar:

```
ObjetoPreferencias.TextureMapPath = StrDirectorioTexturas
```

La sintaxis para obtener:

```
StrDirectorioTexturas = ObjetoPreferencias.TextureMapPath
```

StrDirectorioTexturas es tipo de dato String.

DOCE.11.2. Preferencias de rendimiento

- *ArcSmoothness*. Especifica la resolución de arcos, círculos y elipses. El valor ha de estar entre 1 y 20000.

La sintaxis para asignar:

```
ObjetoPreferencias.ArcSmoothness = IntResolución
```

La sintaxis para obtener:

```
IntResolución = ObjetoPreferencias.ArcSmoothness
```

IntResolución es tipo de dato Integer.

- *ContourLinesPerSurface*. Especifica el valor que representa el número de isolíneas en sólidos. El valor ha de estar entre 0 y 2047.

La sintaxis para asignar:

```
ObjetoPreferencias.ContourLinesPerSurface = IntIsolíneas
```

La sintaxis para obtener:

```
IntIsolíneas = ObjetoPreferencias.ContourLinesPerSurface
```

IntIsolíneas es tipo de dato Integer.

- *DisplayDraggedObject*. Especifica el modo de arrastre dinámico.

La sintaxis para asignar:

```
ObjetoPreferencias.DisplayDraggedObject = IntModoArrastre
```

La sintaxis para obtener:

```
IntModoArrastre = ObjetoPreferencias.DisplayDraggedObject
```

IntModoArrastre es tipo de dato Integer, que además admite las constantes siguientes:

<i>acDragDoNotDisplay</i>	<i>acDragDisplayOnRequest</i>
<i>acDragDisplayAutomatically</i>	

- *DisplaySilhouette*. Especifica el estado de activación de la silueta al ocultar sólidos.

La sintaxis para asignar:

```
ObjetoPreferencias.DisplaySilhouette = BooSilueta
```

La sintaxis para obtener:

```
BooSilueta = ObjetoPreferencias.DisplaySilhouette
```

BooSilueta es tipo de dato Boolean: True activa la silueta, False la desactiva.

- *IncrementalSavePercent*. Especifica el valor del guardado progresivo. Ha de ser un valor porcentual, es decir, entre 0 y 100.

La sintaxis para asignar:

```
ObjetoPreferencias.IncrementalSavePercent = IntGuardProgresivo
```

La sintaxis para obtener:

```
IntGuardProgresivo = ObjetoPreferencias.IncrementalSavePercent
```

IntGuardProgresivo es tipo de dato Integer.

- *MaxActiveViewports*. Especifica el valor del máximo de ventanas activas. Ha de ser un valor entre 2 y 48.

La sintaxis para asignar:

```
ObjetoPreferencias.MaxActiveViewports = IntMáxVentanas
```

La sintaxis para obtener:

```
IntMáxVentanas = ObjetoPreferencias.MaxActiveViewports
```

IntMaxVentanas es tipo de dato Integer.

- *RenderSmoothness*. Especifica el valor de suavizado de sólidos o resolución de faceteado. Ha de ser una valor entre 0.01 y 10.0.

La sintaxis para asignar:

```
ObjetoPreferencias.RenderSmoothness = DblResFaceteado
```

La sintaxis para obtener:

```
DblResFaceteado = ObjetoPreferencias.RenderSmoothness
```

DblResFaceteado es tipo de dato Double.

- *SegmentPerPolyline*. Especifica el valor de número de segmentos por curva polilíneal. Este valor estará entre 0 y 42950.

La sintaxis para asignar:

```
ObjetoPreferencias.SegmentPerPolyline = DblSegmentosPol
```

La sintaxis para obtener:

```
DblSegmentosPol = ObjetoPreferencias.SegmentPerPolyline
```

DblSegmentosPol es tipo de dato Double.

- *ShowRasterImage*. Especifica la activación de mostrar o no el contenido de imágenes de trama o *raster* cuando son movidas con un encuadre o un zoom en tiempo real.

La sintaxis para asignar:

```
ObjetoPreferencias.ShowRasterImage = BooMostrarRaster
```

La sintaxis para obtener:

```
BooMostrarRaster = ObjetoPreferencias.ShowRasterImage
```

BooMostrarRaster es tipo de dato Boolean: True activa la muestra de imágenes y False la desactiva. También está admitido el siguiente par de constantes:

acOn *acOff*

- *TextFrameDisplay*. Especifica la activación de mostrar o no sólo el marco de contorno de los texto o el texto en sí.

La sintaxis para asignar:

```
ObjetoPreferencias.TextFrameDisplay = BooMostrarMarcoTextos
```

La sintaxis para obtener:

```
BooMostrarMarcoTextos = ObjetoPreferencias.TextFrameDisplay
```

BooMostrarMarcoTextos es tipo de dato Boolean: True activa la muestra del marco y False la desactiva.

- *XRefDemandLoad*. Especifica la activación o desactivación de la carga bajo demanda de referencias externas.

La sintaxis para asignar:

```
ObjetoPreferencias.XRefDemandLoad = IntCargaBajoDemanda
```

La sintaxis para obtener:

```
IntCargaBajoDemanda = ObjetoPreferencias.XRefDemandLoad
```

IntCargaBajoDemanda es tipo de dato Integer, aunque también admite las siguientes constantes:

acDemandLoadDisabled

acDemandLoadEnabled

acDemandLoadEnabledWithCopy

DOCE.11.3. Preferencias de compatibilidad

- *DemandLoadARXApp*. Especifica la activación de la carga bajo demanda de aplicaciones ARX.

La sintaxis para asignar:

```
ObjetoPreferencias.DemandLoadARXApp = IntCargaBajoDemandaARX
```

La sintaxis para obtener:

```
IntCargaBajoDemandaARX = ObjetoPreferencias.DemandLoadARXApp
```

IntDemandLoadARXApp es tipo de dato Integer, que además admite las constantes siguientes:

acDemandLoadDisabled

acDemandLoadOnObjectDetect

acDemandLoadCmdInvoke

- *EnableStartupDialog*. Especifica si se muestra el letrero de diálogo de inicio en una sesión de **AutoCAD**.

La sintaxis para asignar:

```
ObjetoPreferencias.EnableStartupDialog = BooCuadroInicio
```

La sintaxis para obtener:

```
BooCuadroInicio = ObjetoPreferencias.EnableStartupDialog
```

Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

EnableStartupDialog es tipo de dato Boolean: True activa el cuadro de inicio y False lo desactiva.

- *KeyboardAccelerator*. Especifica la activación o desactivación de la prioridad de teclas de aceleración.

La sintaxis para asignar:

```
ObjetoPreferencias.KeyboardAccelerator = BooTeclasAcel
```

La sintaxis para obtener:

```
BooTeclasAcel = ObjetoPreferencias.KeyboardAccelerator
```

BooTeclasAcel es tipo de dato Boolean, pero que además admite las siguientes constates:

acPreferenceClassic

acPreferenceCustom

las cuales hacen referencia al modo **AutoCAD** y al modo Windows respectivamente.

- *KeyboardPriority*. Controla la prioridad de coordenadas.

La sintaxis para asignar:

```
ObjetoPreferencias.KeyboardPriority = IntPrioridadCoordenadas
```

La sintaxis para obtener:

```
IntPrioridadCoordenadas = ObjetoPreferencias.KeyboardPriority
```

IntPrioridadCoordenadas es tipo de dato Integer, pero que además admite las siguientes constates:

acKeyboardRunningObjSnap
acKeyboardProxyNoScripts

acKeyboardEntry

- *PersistentLISP*. Especifica la activación o desactivación de la característica de volver a cargar programas o rutinas AutoLISP entre dibujos.

La sintaxis para asignar:

```
ObjetoPreferencias.PersistentLISP = BooLISP
```

La sintaxis para obtener:

```
BooLISP = ObjetoPreferencias.PersistentLISP
```

BooLISP es tipo de dato Boolean.

- *ProxyImage*. Obtiene y/o especifica el modo de mostrar objetos *proxy*.

La sintaxis para asignar:

```
ObjetoPreferencias.ProxyImage = IntObjProxy
```

La sintaxis para obtener:

```
IntObjProxy = ObjetoPreferencias.ProxyImage
```

IntObjProxy es tipo de dato Integer, pero que además admite las siguientes constates:

acProxyNotShow *acProxyShow* *acProxyBoundingBox*

- *ShowProxyDialog*. Obtiene y/o especifica si **AutoCAD** debe mostrar un mensaje cuando se abra un dibujo con objetos *proxy*.

La sintaxis para asignar:

```
ObjetoPreferencias.ShowProxyDialog = BooMensajeProxy
```

La sintaxis para obtener:

```
BooMensajeProxy = ObjetoPreferencias.ShowProxyDialog
```

BooMensajeProxy es tipo de dato Boolean.

DOCE.11.4. Preferencias generales

- *AutoAudit*. Especifica la activación de la revisión de dibujos al cargar un *.DXF* o un *.DXB*.

La sintaxis para asignar:

```
ObjetoPreferencias.AutoAudit = BooRevisión
```

La sintaxis para obtener:

```
BooRevisión = ObjetoPreferencias.AutoAudit
```

BooRevisión es tipo de dato Boolean.

- *AutoSaveInterval*. Especifica la duración, en minutos, entre guardados automáticos. Ha de ser un valor entre 0 y 600.

La sintaxis para asignar:

```
ObjetoPreferencias.AutoSaveInterval = IntMinutos
```

La sintaxis para obtener:

```
IntMinutos = ObjetoPreferencias.AutoSaveInterval
```

IntMinutos es tipo de dato Integer.

- *BeepOnError*. Especifica la activación del sonido en caso de error.

La sintaxis para asignar:

```
ObjetoPreferencias.BeepOnError = BooSonidoError
```

La sintaxis para obtener:

```
BooSonidoError = ObjetoPreferencias.BeepOnError
```

BooSonidoError es tipo de dato Boolean.

- **CreateBackup.** Especifica la activación de crear o no copia de seguridad al guardar (archivos .BAK).

La sintaxis para asignar:

```
ObjetoPreferencias.CreateBackup = BooCopiaSeguridad
```

La sintaxis para obtener:

```
BooCopiaSeguridad = ObjetoPreferencias.CreateBackup
```

BooCopiaSeguridad es tipo de dato Boolean.

- **FullCrcValidation.** Especifica la activación de la validación CRC (comprobación cíclica de redundancia) continua.

La sintaxis para asignar:

```
ObjetoPreferencias.FullCrcValidation = BooCRC
```

La sintaxis para obtener:

```
BooCRC = ObjetoPreferencias.FullCrcValidation
```

BooCRC es tipo de dato Boolean.

- **LogFileOn.** Especifica si se escribe en un archivo de revisión lo que va apareciendo en la ventana de texto (por defecto ACAD.LOG).

La sintaxis para asignar:

```
ObjetoPreferencias.LogFileOn = BooArchivoLOG
```

La sintaxis para obtener:

```
BooArchivoLOG = ObjetoPreferencias.LogFileOn
```

BooArchivoLOG es tipo de dato Boolean.

- **MaxNumOfSymbols.** Especifica el máximo de elementos que ordenar (por defecto es igual a 200).

La sintaxis para asignar:

```
ObjetoPreferencias.MaxNumOfSymbols = IntNumElementos
```

La sintaxis para obtener:

```
IntNumElementos = ObjetoPreferencias.MaxNumOfSymbols
```

MaxNumOfSymbols es tipo de dato Integer.

- *MeasurementUnits*. Especifica el tipo de unidades del dibujo.

La sintaxis para asignar:

```
ObjetoPreferencias.MeasurementUnits = IntUnidadesDibujo
```

La sintaxis para obtener:

```
IntUnidadesDibujo = ObjetoPreferencias.MeasurementUnits
```

IntUnidadesDibujo es tipo de dato Integer.

- *SavePreviewThumbnail*. Especifica la activación o desactivación de guardar imagen preliminar con el dibujo.

La sintaxis para asignar:

```
ObjetoPreferencias.SavePreviewThumbnail = IntImgPreliminar
```

La sintaxis para obtener:

```
IntImgPreliminar = ObjetoPreferencias.SavePreviewThumbnail
```

IntImgPreliminar es tipo de dato Integer.

- *TempFileExtension*. Especifica la extensión de los archivos temporales de **AutoCAD** (por defecto *.AC\$*).

La sintaxis para asignar:

```
ObjetoPreferencias.TempFileExtension = StrExtArchivoTemp
```

La sintaxis para obtener:

```
StrExtArchivoTemp = ObjetoPreferencias.TempFileExtension
```

StrExtArchivoTemp es tipo de dato String.

DOCE.11.5. Preferencias de visualización

- *CrosshairColor*. Especifica el color para la cruceta del cursor (también cambia el cuadro de designación y el símbolo del SCP). Habrá de ser un valor entre 1 y 255.

La sintaxis para asignar:

```
ObjetoPreferencias.CrosshairColor = IntColorCursor
```


La sintaxis para obtener:

```
IntColorCursor = ObjetoPreferencias.CrosshairColor
```

IntColorCursor es tipo de dato Integer, aunque también se admiten las siguientes constantes:

AcRed	AcBlue	AcYellow	AcMagenta
AcCyan	AcGreen	AcWhite	

- *DisplayScreenMenu*. Especifica si se muestra o no el menú de pantalla.

La sintaxis para asignar:

```
ObjetoPreferencias.DisplayScreenMenu = BooMenúPantalla
```

La sintaxis para obtener:

```
BooMenúPantalla = ObjetoPreferencias.DisplayScreenMenu
```

BooMenúPantalla es tipo de dato Boolean.

- *DisplayScrollBars*. Especifica si se muestran o no las barras de desplazamiento (horizontal y vertical).

La sintaxis para asignar:

```
ObjetoPreferencias.DisplayScrollBars = BooBarrasDespl
```

La sintaxis para obtener:

```
BooBarrasDespl = ObjetoPreferencias.DisplayScrollBars
```

BooBarrasDespl es tipo de dato Boolean.

- *DockedVisibleLines*. Especifica el número de líneas de comando ancladas (valor por defecto 5).

La sintaxis para asignar:

```
ObjetoPreferencias.DockedVisibleLines = IntNumLíneasComando
```

La sintaxis para obtener:

```
IntNumLíneasComando = ObjetoPreferencias.DockedVisibleLines
```

IntNumLíneasComando es tipo de dato Integer.

- *HistoryLines*. Especifica el número de líneas en la ventana de texto que se guardan en memoria (valor por defecto 400).

La sintaxis para asignar:

```
ObjetoPreferencias.HistoryLines = IntNumLíneasHistorial
```

La sintaxis para obtener:

```
IntNumLíneasHistorial = ObjetoPreferencias.HistoryLines
```

IntNumLíneasHistorial es tipo de dato Integer.

- **GraphicFont**. Especifica la fuente que se utiliza para gráficos, es decir, el tipo de letra con el que se escriben las opciones de menú, textos en línea de comandos, etcétera.

La sintaxis para asignar:

```
ObjetoPreferencias.GraphicFont = StrFuenteGráficos
```

La sintaxis para obtener:

```
StrFuenteGráficos = ObjetoPreferencias.GraphicFont
```

StrFuenteGráficos es tipo de dato String.

- **GraphicFontSize**. Especifica el tamaño de la fuente que se utiliza para gráficos.

La sintaxis para asignar:

```
ObjetoPreferencias.GraphicFontSize = IntTamañoFuenteGráficos
```

La sintaxis para obtener:

```
IntTamañoFuenteGráficos = ObjetoPreferencias.GraphicFontSize
```

IntTamañoFuenteGráficos es tipo de dato Integer.

- **GraphicFontStyle**. Especifica el estilo de la fuente que se utiliza para gráficos.

La sintaxis para asignar:

```
ObjetoPreferencias.GraphicFontStyle = IntEstiloFuenteGráficos
```

La sintaxis para obtener:

```
IntEstiloFuenteGráficos = ObjetoPreferencias.GraphicFontStyle
```

IntEstiloFuenteGráficos es tipo de dato Integer, aunque admite las constantes siguientes:

acFontRegular acFontItalic acFontBold acFontBoldItalic

- **GraphicsTextBackgrndColor**. Especifica el color para el fondo del texto gráfico.

La sintaxis para asignar:

```
ObjetoPreferencias.GraphicsTextBackgrndColor = IntColorFondoTextoGráficos
```

La sintaxis para obtener:

```
IntColorFondoTextoGráficos = ObjetoPreferencias.GraphicsTextBackgrndColor
```

Curso Práctico de Personalización y Programación bajo AutoCAD

Programación en Visual Basic orientada a AutoCAD (VBA)

IntColorFondoTextoGráficos es tipo de dato Integer. El intervalo y las constantes son las mismas que para *CrosshairColor*.

- *GraphicsTextColor*. Especifica el color para el texto gráfico.

La sintaxis para asignar:

```
ObjetoPreferencias.GraphicsTextColor = IntColorTextoGráficos
```

La sintaxis para obtener:

```
IntColorTextoGráficos = ObjetoPreferencias.GraphicsTextColor
```

IntColorTextoGráficos es tipo de dato Integer. El intervalo y las constantes son las mismas que para *CrosshairColor*.

- *GraphicsWinBackgrndColor*. Especifica el color para el fondo de la ventana gráfica.

La sintaxis para asignar:

```
ObjetoPreferencias.GraphicsWinBackgrndColor = IntColorFondoVentanaGráfica
```

La sintaxis para obtener:

```
IntColorFondoVentanaGráfica = ObjetoPreferencias.GraphicsWinBackgrndColor
```

IntColorFondoVentanaGráfica es tipo de dato Integer. El intervalo y las constantes son las mismas que para *CrosshairColor*.

- *MaxAutoCADWindow*. Especifica la activación de maximizar o no **AutoCAD** al iniciar.

La sintaxis para asignar:

```
ObjetoPreferencias.MaxAutoCADWindow = BooMaximizar
```

La sintaxis para obtener:

```
BooMaximizar = ObjetoPreferencias.MaxAutoCADWindow
```

BooMaximizar es tipo de dato Boolean.

- *MonochromeVectors*. Especifica la activación o desactivación de los vectores monocromo.

La sintaxis para asignar:

```
ObjetoPreferencias.MonochromeVectors = BooVectoresMonocromo
```

La sintaxis para obtener:

```
BooVectoresMonocromo = ObjetoPreferencias.MonochromeVectors
```

BooVectoresMonocromo es tipo de dato Boolean.

- *TextFont*. Especifica la fuente que se utiliza para textos.

La sintaxis para asignar:

```
ObjetoPreferencias.TextFont = StrFuenteTextos
```

La sintaxis para obtener:

```
StrFuenteTextos = ObjetoPreferencias.TextFont
```

StrFuenteTextos es tipo de dato String.

- *TextFontSize*. Especifica el tamaño de la fuente que se utiliza para textos.

La sintaxis para asignar:

```
ObjetoPreferencias.TextFontSize = IntTamañoFuenteTextos
```

La sintaxis para obtener:

```
IntTamañoFuenteTextos = ObjetoPreferencias.TextFontSize
```

IntTamañoFuenteTextos es tipo de dato Integer.

- *TextFontStyle*. Especifica el estilo de la fuente que se utiliza para textos.

La sintaxis para asignar:

```
ObjetoPreferencias.TextFontStyle = IntEstiloFuenteTextos
```

La sintaxis para obtener:

```
IntEstiloFuenteTextos = ObjetoPreferencias.TextFontStyle
```

IntEstiloFuenteTextos es tipo de dato Integer, aunque admite las constantes siguientes:

acFontRegular *acFontItalic* *acFontBold* *acFontBoldItalic*

- *TextWinBackgrndColor*. Especifica el color para el fondo de la ventana de texto.

La sintaxis para asignar:

```
ObjetoPreferencias.TextWinBackgrndColor = IntColorFondoVentanaTexto
```

La sintaxis para obtener:

```
IntColorFondoVentanaTexto = ObjetoPreferencias.TextWinBackgrndColor
```

IntColorFondoVentanaTexto es tipo de dato Integer. El intervalo y las constantes son las mismas que para *CrosshairColor*.

- *TextWinTextColor*. Especifica el color para el texto de la ventana de texto.

La sintaxis para asignar:

```
ObjetoPreferencias.TextWinTextColor = IntColorTextoVentanaTexto
```

La sintaxis para obtener:

```
IntColorTextoVentanaTexto = ObjetoPreferencias.TextWinTextColor
```

IntColorTextoVentanaTexto es tipo de dato Integer. El intervalo y las constantes son las mismas que para *CrosshairColor*.

DOCE.11.6. Preferencia de dispositivo

- *CursorSize*. Especifica el tamaño del cursor gráfico (como porcentaje respecto a la pantalla). Será un valor de 0 a 100.

La sintaxis para asignar:

```
ObjetoPreferencias.CursorSize = IntTamañoCursor
```

La sintaxis para obtener:

```
IntTamañoCursor = ObjetoPreferencias.CursorSize
```

IntTamañoCursor es tipo de dato Integer.

DOCE.11.7. Preferencia de perfil

- *ActiveProfile*. Especifica el perfil de usuario activo.

La sintaxis para asignar:

```
ObjetoPreferencias.ActiveProfile = StrPerfil
```

La sintaxis para obtener:

```
StrPerfil = ObjetoPreferencias.ActiveProfile
```

StrPerfil es tipo de dato String.

DOCE.11.8. Métodos del objeto de preferencias

Una vez vistas las propiedades pasamos a los métodos. Los métodos siguientes se corresponden con las acciones de los botones existentes en el cuadro de la pestaña *Perfil* del cuadro *Preferencias* de **AutoCAD**.

- *DeleteProfile*. Elimina un perfil de usuario de **AutoCAD**.

```
ObjPreferencias.DeleteProfile(StrPerfilElim)
```

StrPerfilElim es un valor String que especifica el nombre del perfil que se desea eliminar.

- *ExportProfile*. Exporta un perfil de usuario de **AutoCAD**.

```
ObjPreferencias.DeleteProfile(StrPerfilExport, StrArchivoARG)
```

StrPerfilExport es un valor *String* que especifica el nombre del perfil que se desea exportar. *StrArchivoARG* es un valor de tipo *String* también, el cual indica la ruta y el nombre del archivo donde se exporta el perfil para su posterior uso. El archivo será de extensión *.ARG*.

- *GetProjectFilePath*. Obtiene el camino de búsqueda de referencias externas asociado con el proyecto cuyo nombre se indica.

```
StrDirectorio = ObjPreferencias.GetProjectFilePath(StrNombreProyecto)
```

StrNombreProyecto es un valor *String* que especifica el nombre del proyecto. *StrDirectorio* es una variable de tipo *String* que recogerá el directorio donde **AutoCAD** buscará referencias externas.

- *ImportProfile*. La sintaxis de este método es:

```
ObjPreferencias.DeleteProfile(StrPerfilImport, StrArchivoARG, IncluirCamino)
```

ImportProfile importa el perfil de usuario cuyo nombre se indica de un archivo de registro de perfiles con la extensión *.ARG*. El último argumento es un valor booleano (*Boolean*) que determina si se incluye el camino de acceso en la información del perfil. Si el valor es *True* se incluirá dicho camino; en caso contrario (*False*) no.

- *ResetProfile*. La sintaxis de este método es:

```
ObjPreferencias.ResetProfile(StrPerfilInic)
```

ResetProfile inicializa el perfil que se indica (como cadena *String*), estableciendo todas las características con los valores por defecto.

- *SetProjectFilePath*. Crea un nuevo proyecto con el nombre indicado y le asigna o asocia un camino de búsqueda para referencias externas cuando se trabaje con él.

La sintaxis de este método es:

```
ObjPreferencias.SetProjectFilePath(StrNombreProyecto, StrDirectorioRefX)
```

StrNombreProyecto es un valor *String* que especifica el nombre del proyecto. *StrDirectorioRefX* es una variable de tipo *String* que recogerá el directorio donde **AutoCAD** buscará referencias externas.

6ª fase intermedia de ejercicios

- Prográmesse algún ejemplo que actúe sobre las preferencias de **AutoCAD**.

DOCE.12. ALGUNOS TRUCOS ActiveX Automation PARA AutoCAD

En esta sección simplemente se quieren mostrar algunos trucos o técnicas útiles a la

hora de programar en VBA para **AutoCAD**. Existe a veces un vacío en el programador cuando se encuentra con situaciones que no sabe resolver. La mayor parte de estas situaciones tienen solución, sólo hay que saber encontrarla y, para ello, hacen mucha falta horas de programación y experiencia.

Aquí se explica un pequeño grupo de técnicas que pueden resolver algunos problemas a la hora de hacer programas para **AutoCAD**.

DOCE.12.1. Compilación de programas con un compilador de Visual Basic externo

Como nos habremos dado cuenta a la hora de programar a lo largo de este **MÓDULO DOCE**, resulta un poco fastidioso el mecanismo de VBA en el momento de ejecutar un programa. Si este dispone de letrero de diálogo se cambia automáticamente a la ventana de **AutoCAD** (siempre que no esté minimizada), pero si es una macro simplemente, hemos de cambiar manualmente.

Además de ello, en cuanto el programa termina se vuelve al editor VBA, ya que hemos de tenerlo siempre abierto para ejecutar un programa. Esto no es nada elegante si nuestro deseo es distribuir o vender un programa. Con el consabido temor a que alguien además haga suyo nuestro código fuente.

Para evitar todos estos problemas podemos recurrir a un compilador externo de Visual Basic, como Visual Basic 5.0 o Visual Basic 6.0. De esta manera, únicamente hemos de compilar el programa para crear una aplicación .EXE independiente (pero que sólo correrá bajo **AutoCAD**).

Sin embargo, compilar un proyecto VBA en un editor externo de Visual Basic no es un camino de rosas. Tampoco pensemos que es un infierno, no, es sencillo, pero hay que seguir una serie de pasos y unas normas para que todo funcione correctamente. Veamos cuáles son.

Primero. El primer paso consiste, evidentemente, en conseguir una distribución de Visual Basic. Sin el compilador no podremos hacer nada, como es lógico. Aquí se explicarán los pasos refiriéndonos a Visual Basic 5.0.

Segundo. Un compilador de Visual Basic no lee archivos binarios .DVB (los del VBA de **AutoCAD**), por lo que habrá que exportar los archivos a un formato comprensible. Para ello, y desde la ventana de proyecto de VBA, seleccionamos uno por uno los archivos de nuestro proyecto (los de formulario, módulo...) y, pulsando sobre ellos con el botón derecho del ratón elegimos la opción Exportar archivo... del menú contextual.

Todos los archivos resultantes los guardaremos en una sola carpeta en el disco duro para mayor organización final. Los elementos, según su condición, se exportan a un archivo diferente según la siguiente tabla:

Elemento VBA	Al exportar es un archivo de Visual Basic
Formulario	.FRM
Módulo	.BAS
Módulo de clase	.CLS

Tercero. Exportados y guardados como archivos todos los elementos, hemos de pasarlos a Visual Basic. Aquí abriremos un nuevo proyecto y lo dejaremos vacío de formularios y módulos de código (eliminando el formulario que aparece por defecto).

Ahora hemos de pulsar con el botón derecho en un espacio vacío (blanco) de la ventana de proyecto y elegir *Agregar>Formulario...* En el cuadro de diálogo buscaremos y abriremos el archivo `.FRM` generado del proyecto en VBA. Lo mismo para los módulos de código y demás. Es decir, crearemos un sólo proyecto de Visual Basic `.VBP` agregando o anexando todos los trozos extraídos del proyecto de VBA. Al final podemos guardarlo todo.

Cuarto. Muy importante es una cosa. Puede que al intentar añadir un formulario nos dé un error Visual Basic. Procuremos antes de nada elegir la referencia a la librería de objetos de **AutoCAD** (AutoCAD Object Library) en el menú *Proyecto>Referencias...*, además de las dos siguientes referencias también: Visual Basic For Applications y OLE Automation (otras que haya escogidas ya hemos de dejarlas como están). Además, bajo *Proyectos>Componentes...* escogeremos Microsoft Forms 2.0 Object Library.

Este último componente es el que utiliza el VBA de **AutoCAD** como predeterminado; es la librería de objetos insertables en los formularios: cuadros de lista, listas desplegadas, botones, etcétera. Si en el formulario hay algún elemento que no se encuentra en la librería por defecto de Visual Basic, como por ejemplo la página múltiple (la ventana con pestañas), puede que se produzca un error al agregar el formulario.

Las referencias expuestas dicen relación a VBA. La más importante es AutoCAD Object Library, ya que contiene definidos los objetos de **AutoCAD** para VBA: Application, Document, ModelSpace, Plot, Preferences...

Quinto. Una vez hecho todo lo anterior hemos de repasar el código para encontrar incompatibilidades de VBA con Visual Basic. Por ejemplo, sabemos que el documento actual activo puede ser referenciado desde VBA con el término `ThisDrawing`. Esto es imposible desde un Visual Basic externo; a la hora de compilar nos dará un error y nos dirá que esa variable no está declarada.

Desde Visual Basic es indispensable utilizar la sintaxis que hemos venido usando hasta ahora, por ejemplo:

```
Dim AcadDoc as Object
Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument
```

De esta forma hacemos referencia a la aplicación **AutoCAD** y a su librería de objetos. A partir de aquí ningún método o propiedad será considerada por Visual Basic como un error de sintaxis o de falta de declaración de variables.

Otra serie de errores nos producen todas las constantes equivalentes a valores Integer que admiten algunos métodos y algunas propiedades: `acOn`, `acOff`, `acMetric`, `acSelectionSetAll`, `acPlotOrientationLandscape`... En estos casos se producirá también un error, al compilar, de variable no declarada. Para evitarlo únicamente hemos de sustituir estos valores constantes por sus correspondientes valores enteros (Integer).

NOTA IMPORTANTE DE SINTAXIS: No se ha comentado a lo largo de este **MÓDULO** ninguno de los valores enteros correspondientes con las diversas constantes, sino únicamente ellas. Cada lista de constantes en estas páginas está ordenada de izquierda a derecha y de arriba a abajo (sentido lógico de lectura). Pues bien, los valores enteros se encuentran también en dicho orden, comenzando siempre por el cero (0).

En última instancia, si no pudiéramos solucionar un error de esta índole por alguno de estos métodos, bastaría deshacernos de la sentencia `Option Explicit` para que todo funcione a la perfección.

Sexto. El proyecto a nivel estético en Visual Basic es mucho más agradecido que en VBA. Ahora, y antes de la compilación, podremos agregar un icono propio a cada formulario, anular o no los botones de maximizar y minimizar, etcétera. Esto va con el gusto del programador.

Séptimo. Este paso es el de la compilación. Simplemente consiste en dirigirnos a Archivo>Generar Proyecto1.exe... (esta opción cambia, pero la primera vez probablemente sea este el título), elegir la opción, darle un título al ejecutable y pulsar el botón *Aceptar*.

Si se nos ofrece algún error de compilación sin depurar de los mencionados en el paso anterior, simplemente habremos de seguir las instrucciones pertinentes ya comentadas para solucionarlo.

Octavo. Una vez en disposición de un ejecutable válido, se nos presenta el problema de la ejecución desde **AutoCAD**. El archivo objeto es .EXE, sin embargo no tiene nada que ver con aplicaciones ADS ni ARX.

La manera de cargar un ejecutable compilado con Visual Basic es con la función `STARTAPP` de AutoLISP. La explicación completa de esta función se encuentra en la sección **ONCE.15.5.**, sobre el inicio de aplicaciones Windows. Así por ejemplo, si nuestro nuevo ejecutable se llamara `TRAZADO.EXE` y se encontrará en una carpeta llamada `VBACOMP` en el directorio raíz del disco duro (`C:\`), podríamos hacer:

```
(STARTAPP "c:/vbacom/trazado.exe")
```

Esta línea puede formar parte de un programa AutoLISP, de la macro de un botón, de una opción de menú o de un archivo de guión, además de poder teclearla directamente en la línea de comandos.

De esta manera, queda así explicada la forma de crear aplicaciones con VBA y Visual Basic para **AutoCAD** que no necesiten el editor VBA para funcionar y también, que no tengan el código fuente a la vista.

Aún así, **AutoCAD** deberá estar ejecutándose para que estos programas funcionen correctamente, de forma evidente. También hemos de tener en cuenta a la hora de correr nuestros programas en otras máquinas, la necesidad de ciertos archivos (librerías de objetos y demás) para que todo funcione a la perfección.

Por último, reseñar la necesidad de comprobar los derechos de autor de ciertos archivos al distribuirlos de manera gratuita o por una cantidad dineraria.

DOCE.12.1.1. Objeto de aplicación en programas compilados

Al compilar un programa VBA con un Visual Basic externo, se puede dar el caso de que el usuario lo pretenda ejecutar sin tener **AutoCAD** arrancado. En este caso el programa ofrecería un error.

Para evitar este trance, se suele añadir al programa un código parecido al siguiente, para que se ejecute nada más empezar:

```
On Error Resume Next
Set AcadApp = GetObject(, "AutoCAD.Application")
If Err Then
    Err.Clear
```

```
Set AcadApp = CreateObject("AutoCAD.Application")
If Err Then
    MsgBox Err.Description
    Exit Sub
End If
End If
```

De esta manera, se intenta acceder con `GetObject` al objeto de aplicación de **AutoCAD** (previamente declarado como `Object`, al igual que siempre). Si se produce un error se procura crear un objeto de aplicación de **AutoCAD** (con `CreateObject`). Lo que hará este mecanismo es arrancar **AutoCAD** si lo encuentra y no se produce ningún error. Por último, y si esto falla, se presenta un mensaje de error con la descripción del mismo.

Por defecto la aplicación de **AutoCAD** se inicia con la propiedad de visibilidad desactivada, por lo que habrá que activarla así, por ejemplo:

```
AcadApp.Visible = True
```

Después, y si nos interesa, podremos darle tamaño a la ventana, por ejemplo así:

```
acadApp.Top = 0
acadApp.Left = 0
acadApp.Width = 400
acadApp.Height = 400
```

A menudo el orden de las propiedades del objeto no es importante, pero en este caso sí lo es. Dado que el tamaño de la ventana de aplicación se mide desde la esquina superior izquierda de la ventana, el ajuste de las propiedades `Width` o `Height` antes que `Top` o `Left` provoca un comportamiento no recomendable.

DOCE.12.2. Ejecución de programas VBA desde AutoLISP y en macros

Así como la función `STARTAPP`, comentada en la sección anterior, nos es útil para ejecutar aplicaciones Windows, puede que lo que nos interese es mantener el programa VBA sin compilar y necesitemos ejecutarlo sin necesidad de hacer pasar al usuario por todo el proceso de cargar un proyecto, etcétera; es decir, hacerlo de una manera automática.

Desde un programa en AutoLISP podremos utilizar sin ningún problema los comandos que se añaden al instalar VB, es decir `_VBALOAD`, `_VBARUN` y `_VBAUNLOAD`. Imaginemos que disponemos de una macro creada en un archivo que es de proyecto de VBA llamado `PRUEBA.DVB`, que se encuentra en el directorio `C:\VBA\`. Para cargarlo podríamos incluir las siguientes líneas en un archivo AutoLISP:

```
(COMMAND "_VBALOAD" "c:\\vba\\prueba.dvb")
(COMMAND "_VBARUN" "Módulo1.MacroPrueba")
(COMMAND "_VBAUNLOAD")
```

Lo primero es cargar el archivo con `VBLOAD` (incluimos el guión de subrayado por mayor compatibilidad, como ya sabemos).

Después hemos de ejecutar la macro. Para ello utilizamos el comando `VBARUN` con el guión normal (-) por delante para ejecutar su versión de línea de comandos. La manera de ejecutar una macro desde la línea de comandos es con la sintaxis que se indica, es decir: primero el nombre del módulo de código que la contiene, luego un punto (.) y por último el

nombre de la macro. Aquí el nombre del módulo es `Módulo1` (nombre por defecto) y el de la macro definida dentro de él es `MacroPrueba`.

Al final hemos de descargar el programa o proyecto. Para ello utilizamos `VBAUNLOAD`. El programa se descargará al finalizar su ejecución.

NOTA: A esta serie de funciones se le puede añadir código para controlar la existencia del archivo, de la macro, etcétera.

Si lo que deseamos ejecutar no es una macro dentro de un módulo sino un programa con su letrero de diálogo (formulario) y demás, la práctica lógica pasa por crear una macro en un módulo de dicho programa que muestre el formulario principal, ya que no se puede llamar a otro elemento que no sea una macro con `VBARUN`. Por ejemplo, en un programa que arranque con un formulario llamado `formPrincipal`, creamos una macro (en el módulo `Módulo1`) tal que así:

```
Sub Arranque()  
    formPrincipal.Show  
End Sub
```

De esta manera luego podemos dirigirnos desde AutoLISP a esta macro así:

```
(COMMAND "_-VBARUN" "Módulo1.Arranque")
```

lo que hará ejecutarse el programa normalmente.

A la hora de ejecutar programas VBA desde una macro (botón, menú o archivo de guión), podemos incluir el código anterior en AutoLISP, ya que resultaría como si lo estuviéramos entrando en línea de comandos.

Sin embargo, por las propias características de las macros, podemos incluir la serie de comandos sin necesidad de hacer intervenir a AutoLISP por ningún lado. Así por ejemplo, la macro de un botón de barra de herramientas que cargue un programa podría ser:

```
^C^C_VBALOAD c:/misdoc~1/autocad/vba/trazado.dvb;  
_-VBARUN Módulo1.Macro;_VBAUNLOAD
```

NOTA: Ténganse en cuenta las propias características de cada tipo de macro, sea de dentro de un botón, de un archivo de guión, etcétera.

Por último decir que en la máquina que se escriban estos comandos habrá de estar instalado VBA para **AutoCAD**, sino no funcionará. Es por esto, que a la hora de distribuir aplicaciones se recomienda la compilación explicada en la sección anterior (**DOCE.12.1.**).

DOCE.12.3. Enviar cadenas a la línea de comandos desde VBA

A veces quizá nos interese, desde un programa VBA, acceder a la línea de comandos de **AutoCAD** para ejecutar algún comando, por ejemplo. O nos sea necesario ejecutar una función o una secuencia de funciones en AutoLISP, porque su efecto no podemos —o no sabemos— conseguirlo con VBA.

Para ello disponemos de un pequeño truco, el cual consiste en utilizar la instrucción `SendKeys` de Visual Basic. Ésta permite enviar una o varias pulsaciones de teclas a la ventana activa como si lo estuviéramos haciendo desde el teclado. Su sintaxis es:

```
SendKeys Cadena[,Modo_espera]
```

Donde *Cadena* es la cadena que deseamos enviar. Esta puede ser también una pulsación de teclas especiales (ENTER, ALT...), pero nos centraremos en una cadena típica. *Modo_espera* es un argumento opcional booleano que, si se establece como *True* se procesan las pulsaciones antes de devolver el control al procedimiento *Sub* y, si es *False* se devuelve el control inmediatamente después de enviarse las pulsaciones. Normalmente utilizaremos esta última opción, que es la predeterminada, por lo que no hará falta que escribamos nada.

Para poder hacer funcionar este mecanismo habremos de ayudarnos de la instrucción *AppActivate*, cuya sintaxis es:

```
AppActivate Título[,Modo_espera]
```

AppActivate activa una ventana de aplicación. *Título* es el argumento obligatorio que hay que pasarle, y dice referencia al nombre actual en la barra de título (barra azul normalmente) de la ventana. *Modo_espera* es un valor booleano opcional que especifica si la aplicación que hace la llamada tiene el enfoque antes de activar otra. Si se especifica *False* (predeterminado), se activa inmediatamente la aplicación especificada, aunque la aplicación que hace la llamada no tenga el enfoque. Si se especifica *True*, la aplicación que hace la llamada espera hasta que tenga el enfoque y luego activa la aplicación especificada (normalmente utilizaremos esta opción *True*).

Pero veamos un ejemplo:

```
Option Explicit  
Dim AcadApp As Object  
Dim AcadCommand As String
```

```
Sub Macro()  
    Set AcadApp = GetObject(, "AutoCAD.Application")  
    AppActivate AcadApp.Caption  
    AcadCommand = "_open "  
    SendKeys AcadCommand, True  
End Sub
```

Esta macro define el objeto de aplicación de **AutoCAD** primero. Después la activa enviando el nombre de la barra de título actual. Como en **AutoCAD** (y en otras muchas aplicaciones) este título cambia según el nombre del archivo abierto, se extrae directamente el actual con la propiedad *Caption* del objeto *Application*.

A continuación se define y envía la cadena a la línea de comandos. En este caso es el comando para abrir un archivo. Nótese que ha de dejarse un espacio blanco al final para emular un *INTRO*, ya que es como si estuviéramos escribiendo en línea de comandos.

NOTA: Un *INTRO* puede también representarse como {INTRO} o con una tilde ~.

Como se ha dicho, *SendKeys* se utiliza para enviar pulsaciones de teclas (para pulsar botones desde código, ejecutar opciones de menú, etc.). En **AutoCAD**, como sabemos, si escribimos algo directamente —teniendo delante la interfaz gráfica— donde primero se refleja es en la línea de comandos. De ahí este mecanismo para este problema.

Podemos también enviar código AutoLISP o lo que nos apetezca. Incluso se podría hacer que un programa VBA leyera línea por línea un archivo *.LSP* y lo fuera enviando a la línea de comandos. Sin embargo esto es un poco peligroso, ya que un programa completo AutoLISP espera respuestas por parte del usuario, abre cuadros DCL, etc., y el programa VBA

seguiría mandando líneas y líneas sin parar. Aunque se puede conseguir que esto funcione, lo lógico es reservarlo para acciones simples y cortas, programando el resto del código en VBA.

Aún así, veremos un ejemplo más práctico que el anterior con código en AutoLISP. Sabemos que VBA no dispone (como AutoLISP) de un método o propiedad que llame al cuadro de color de **AutoCAD**. Pues mezclando algunas técnicas podremos hacer desde VBA que se llame a dicho cuadro utilizando AutoLISP para luego acabar con el resultado en una variable VBA. Veamos el código:

```
Option Explicit
```

```
Dim AcadApp As Object  
Dim AcadDoc As Object
```

```
Sub Macro()  
    Dim AcadCommand As String  
    Set AcadApp = GetObject(, "AutoCAD.Application")  
    Set AcadDoc = AcadApp.ActiveDocument  
  
    AppActivate AcadApp.Caption  
    AcadCommand = "{(}{SETVAR ""user11"" {(}{SETQ NumCol {(}{ACAD_COLORDLG  
                    256{}}{}}{}}}"  
    SendKeys AcadCommand, True  
End Sub
```

```
Sub Macro2()  
    Dim NúmeroColor As Integer  
  
    NúmeroColor = AcadDoc.GetVariable("user11")  
    MsgBox "Elegido color: " & NúmeroColor  
End Sub
```

Cómo podemos observar, este código consta de dos macros (Macro y Macro2). La primera de ellas es la encargada de enviar el código AutoLISP a la línea de comandos, como hacíamos en el ejemplo anterior. El único inconveniente viene dado por la utilización de paréntesis, y es que estos tienen un significado especial para `SendKeys`. Es por ello que, cuando queramos utilizar uno, habremos de encerrarlo entre llaves.

También debemos saber, que en Visual Basic para introducir comillas dentro de una cadena hemos de indicar estos caracteres como dobles (" "), para que no se confundan con las comillas propias de la cadena de texto.

Según todo esto, una secuencia AutoLISP que sería así:

```
(SETVAR ""user11"" (SETQ NumCol (ACAD_COLORDLG 256)))
```

se convierte en esto:

```
{(}{SETVAR ""user11"" {(}{SETQ NumCol {(}{ACAD_COLORDLG 256{}}{}}{}}}
```

Aquí lo que hacemos es introducir directamente el valor de la variable de AutoLISP que creamos, llamada `NumCol`, la cual recoge el color seleccionado en el cuadro, en una de las variables de usuario para valores enteros de **AutoCAD**, `USER11`.

NOTA: Repásese la sección **NUEVE.3.6.** para recordar el sentido de estas quince variables de diferentes tipos. Ahora les hemos encontrado otro quehacer.

Macro2 simplemente se limita a extraer el valor de `USERI1`, con el método `GetVariable` del objeto de documento activo, y a mostrarlo. Estas variables de usuario son accesibles del mismo modo que las demás variables de sistema.

NOTA: Primero ejecutaremos Macro y después Macro2.

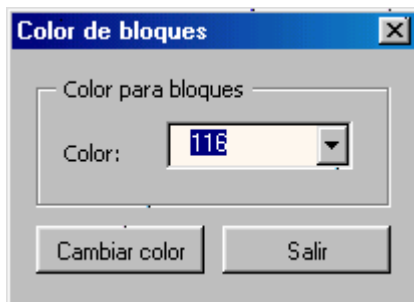
DOCE.13. COMO APUNTE FINAL

Como se advirtió al principio de este **MÓDULO**, la comprensión del mismo pasa por el conocimiento obligado por parte del lector de la programación en Visual Basic. Aquí, según se ha visto, se le pega simplemente un gran repaso a todos los métodos y propiedades que componen la inclusión de VBA en **AutoCAD**.

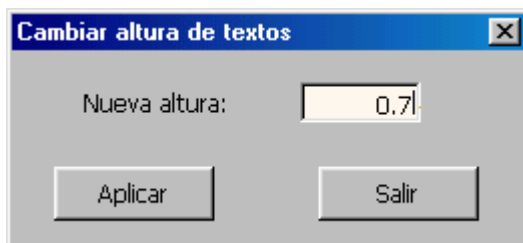
Aún así, el autor de este curso cree que no resulta harto complicado la comprensión del mismo, ya que los ejemplos, los ejercicios y las explicaciones detalladas pueden ayudarnos a comenzar a programar en un lenguaje tan sencillo y versátil como es Visual Basic, sin dejar de lado por supuesto a la parte que le toca al, todavía algo verde, VBA de **AutoCAD**.

DOCE.FIN. EJERCICIOS PROPUESTOS

- I. Crear un sencillo programa en VBA que permita cambiar el color a todos los bloques definidos en el dibujo de una sola vez. El letrero de diálogo que maneje puede ser el siguiente:



- II. Crear una macro VBA que permita añadir líneas de ejes a arcos, círculos y/o elipses previamente seleccionados.
- III. Programar un ejemplo en VBA que permita cambiar la altura a todos los textos del dibujo actual. El letrero de diálogo puede ser este mismo:



- IV. Realizar una serie de programas en AUTOLISP/DCL y en VBA los cuales permitan aprovechar al completo las capacidades de **AutoCAD** a la hora de trabajar con

él en cualquiera que sea el puesto de trabajo. Un proyecto de personalización completo puede incluir diseños de menús, barras de herramientas, etcétera; en fin todo lo que hemos estudiado a lo largo de este curso.

EJERCICIOS RESUELTOS DEL MÓDULO ONCE

1ª fase intermedia de ejercicios

PUNTO I

— (/ (+ 50 5) 2)
— (/ (- (* 200 5) 3) (/ 4 2))
— (* (- 10.23 (/ 12.03 3)) (- (+ 12 (* 2 -2)) (* (/ 12.5 2) 2.65)))
— (/ (+ 19 23) (+ 10 (/ 23 (/ 23 19))))
— (/ (- (/ -20 5) 1) (* 15.5 (/ (- 15.5 1) 12)))

2ª fase intermedia de ejercicios

PUNTO I

— (SQRT (* (- 20 3) (- 8 2)))
— (+ 1 78.8 (/ (EXPT 78.8 2) 2) (/ (EXPT 78.8 3) 3))
— (/ (SIN (- (SQRT (* 80 28.002)) (COS (/ PI 2))) (- PI (/ 1 2)))
— (- 1.5707633 (ATAN (/ (/ 100 2) (SQRT (- 1 (EXPT (/ 100 2) 2))))))
— (/ (* 124.6589 (EXP 2.3)) (* 7 (SQRT 2)))
— (LOG (* 45 (/ 7 2)))
— (EXPT (/ 23.009 78.743) (- 56.00123 1))

PUNTO II

(MIN (/ 7 2 3 2.5) 0.56)	devuelve 0.466667
(MIN (* 8 9) 67.002 45 (/ (SQRT 78)))	devuelve 8.83176
(MAX 12 12 12 12)	devuelve 12
(MAX 56.00001 56.00001)	devuelve 56.0
(GCD (/ 34 2) (* 34 2))	devuelve 17
(GCD 34 34)	devuelve 34

PUNTO III

(1- (* (* 38 2) (1+ (1- 10))))	devuelve 759.0
(1+ -1)	devuelve 0

3ª fase intermedia de ejercicios

PUNTO I

— T
— T

— nil
— T
— T
— nil
— T
— nil
— nil
— T
— T

4ª fase intermedia de ejercicios

PUNTO I

— nil
— T
— T
— nil
— T

5ª fase intermedia de ejercicios

PUNTO I

```
(DEFUN C:CalDiez ()  
  (+ 1 2 3 4 5 6 7 8 9 10)  
)
```

PUNTO II

```
(DEFUN C:CompVal ()  
  (> 20 0.7 10 0.9)  
)
```

PUNTO III

```
(DEFUN C:Variab ()  
  (SETQ w 5 x 45.5 u 89)  
  (* w x u)  
)
```

6ª fase intermedia de ejercicios

PUNTO I

```
(DEFUN ARO (/ Dint Dext Centro)  
  (SETQ Dint (GETDIST "Indica el diámetro interior: "))  
  (SETQ Dext (GETDIST "Indica el diámetro exterior: "))  
  (SETQ Centro (GETPOINT "Indica el centro del aro: "))  
  (COMMAND "_circle" Centro "_d" Dint)  
  (COMMAND "_circle" Centro "_d" Dext)
```



```
)  
  
(DEFUN C:Aro ()  
  (aro)  
)
```

PUNTO II

```
(DEFUN ARO2 (/ RadInt Grosor Centro Radiol Radio2)  
  (SETQ RadInt (GETDIST "Indica el radio intermedio: "))  
  (SETQ Grosor (GETDIST "Indica el grosor: "))  
  (SETQ Centro (GETPOINT "Indica el centro del aro: "))  
  (SETQ Radiol (- RadInt (/ Grosor 2)))  
  (SETQ Radio2 (+ RadInt (/ Grosor 2)))  
  (COMMAND "_circle" Centro Radiol); Se pueden operar aquí  
  (COMMAND "_circle" Centro Radio2)  
)  
  
(DEFUN C:Aro2 ()  
  (aro2)  
)
```

PUNTO III

```
(DEFUN C:Clave ()  
  (INITGET 1 "Girar Desplazar aCercar aLejar")  
  (GETPOINT "Girar/Desplazar/aCercar aLejar/<Punto>: ")  
  (INITGET 1 "ACTivar DESactivar")  
  (GETKEYWORD "Variable principal (ACT/DES): ")  
)
```

7ª fase intermedia de ejercicios

PUNTO I

```
(DEFUN DatosCir ()  
  (INITGET 7)  
  (SETQ Num (GETINT "Número de círculos concéntricos: "))(TERPRI)  
  (INITGET 1)  
  (SETQ Centro (GETPOINT "Centro: "))(TERPRI)  
  (INITGET 4)  
  (IF  
    (SETQ RadInt (GETDIST Centro "Radio interior <0>: "))  
    () ; Si RadInt=T (algo escrito) no hace nada.  
    (SETQ RadInt 0) ; Si RadInt=nil (INTRO) hace RadInt=0 (defecto).  
  )(TERPRI)  
  (INITGET 7)  
  (SETQ RadExt (GETDIST Centro "Radio exterior: "))(TERPRI)  
)  
  
(DEFUN CircCero (/ DRad N RadN)  
(SETQ DRad (/ RadExt Num))  
  (SETQ N 1)  
  (REPEAT Num  
    (SETQ RadN (* DRad N))  
    (COMMAND "_circle" Centro RadN)  
    (SETQ N (1+ N)); Contador suma.  
  )  
)
```

```
(DEFUN CircInt (/ DRad N RadN)
  (SETQ DRad (/ (- RadExt RadInt) (- Num 1)))
  (SETQ N 0)
  (REPEAT Num
    (SETQ RadN (+ RadInt (* DRad N)))
    (COMMAND "_circle" Centro RadN)
    (SETQ N (1+ N))
  )
)

(DEFUN C:Circul (/ Num Centro RadInt RadExt)
  (SETVAR "cmdecho" 0)
  (DatosCir)
  (SETQ Refnt0 (GETVAR "osmode"))(SETVAR "osmode" 0); Captura OSMODE
  (IF (= 0 RadInt) (CircCero) (CircInt))
  (SETVAR "cmdecho" 1) (SETVAR "osmode" refnt0) (PRIN1); Restaura
)

(PROMPT "Nuevo comando CIRCUL definido.")(PRIN1)
```

PUNTO II

```
(DEFUN Circul (/Centro Radio)
  (SETQ Centro (GETPOINT "Centro de los círculos: ")) (TERPRI)
  (SETQ Radio (GETDIST Centro "Radio del círculo base: ")) (TERPRI)
  (COMMAND "_circle" Centro Radio)
  (WHILE (/= Radio nil); Cuando se pulse INTRO, Radio=nil
    (SETQ Radio (GETDIST Centro "Radio del círculo (INTRO para
      terminar: ")) (TERPRI)
    (COMMAND "_circle" Centro Radio)
  )
)
(PRIN1)
)

(DEFUN C:Circul ()
  (Circul)
)
)
```

8ª fase intermedia de ejercicios

PUNTO I

```
(DEFUN Rectg_Datos (); Solicitud de datos
  (INITGET 5)
  (SETQ Grosor (GETREAL "Grosor del rectángulo: ")) (TERPRI)
  (INITGET 5)
  (SETQ Radio (GETREAL "Radio de redondeo: ")) (TERPRI)
  (INITGET 1)
  (SETQ Pto1 (GETPOINT "Primera esquina: ")) (TERPRI)
  (INITGET 1)
  (SETQ Pto3 (GETCORNER Pto1 "Segunda esquina: ")) (TERPRI); Elástico
)

(DEFUN Rectg_Dibujo (/ Pto2 Pto4); Función de dibujo
  (SETQ Pto2 (LIST (CAR Pto3) (CADR Pto1)))
  (SETQ Pto4 (LIST (CAR Pto1) (CADR Pto3)))
  (COMMAND "_pline" Pto1 "_w" Grosor "" "_non" Pto2 Pto3 "_non" Pto4
    "c"); Una modo de referencia "NINGUNO" por si acaso
  (COMMAND "_fillet" "_r" Radio) (COMMAND "_fillet" "_p" "_l")
)
)
```

```
(DEFUN C:Rectg (/ Grosor Radio Pto1 Pto3)
  (SETVAR "cmdecho" 0)
  (Rectg_Datos)
  (SETVAR "blipmode" 0)
  (Rectg_Dibujo)
  (SETVAR "blipmode" 1) (SETVAR "blipmode" 0) (PRIN1)
)

(PROMPT "Nueva orden RECTG definida.") (PRIN1)
```

PUNTO II

```
(DEFUN Datos_Ventana ()
  (INITGET 5)
  (SETQ Gr (GETREAL "Grosor del rectángulo exterior: "))(TERPRI)
  (INITGET 5)
  (SETQ Marco (GETREAL "Anchura para todos los marcos: "))(TERPRI)
  (INITGET 1)
  (SETQ Pt1 (GETPOINT "Vértice inferior izquierdo: "))(TERPRI)
  (INITGET 1)
  (SETQ Pt2 (GETCORNER pt1 "Vértice superior derecho: "))(TERPRI)
)

(DEFUN Ventana (/ x1 y1 x2 y2 x3 y3 LongT Alt LonInt AltInt)
  (SETQ x1 (CAR Pt1) y1 (CADR Pt1) x2 (CAR Pt2) y2 (CADR Pt2))
  (SETQ x3 (+ x1 Marco) y3 (+ y1 Marco))
  (SETQ LongT (- (CAR Pt2) (CAR Pt1)))
  (SETQ Alt (- (CADR Pt2) (CADR Pt1)))
  (SETQ LonInt (/ (- LongT (* 3 Marco)) 2))
  (SETQ AltInt (/ (- Alt (* 3 Marco)) 2))
  (SETQ Pt3 (LIST (+ x1 Marco) (+ y1 Marco)))
  (SETQ Pt4 (LIST (+ x3 LonInt) (+ y3 AltInt)))
  (COMMAND "_rectang" "_w" Gr "_non" Pt1 "_non" Pt2)
  (COMMAND "_rectang" "_w" 0 "_non" Pt3 "_non" Pt4)
  (COMMAND "_array" "_l" "" "_r" 2 2 (+ AltInt Marco) (+ LonInt Marco))
)

(DEFUN C:Ventana (/ Gr Marco Pt1 Pt2)
  (SETVAR "cmdecho" 0)
  (Datos_Ventana)
  (Ventana)
  (SETVAR "cmdecho" 1)(PRIN1)
)

(PROMPT "Nuevo comando VENTANA definido.")(PRIN1)
```

9ª fase intermedia de ejercicios

PUNTO I

(FLOAT -13)	devuelve -13.0
(FLOAT 3.5)	devuelve 3.5
(ITOA 10)	devuelve "10"
(ITOA -37)	devuelve "-37"
(RTOS 12.78 3)	devuelve "'-0.78' '"
(RTOS -4.1 4)	devuelve "-4 1/8"
(ANGTOS -1.2 1 2)	devuelve "2.91d15' '"

(ANGTOS (/ PI 4) 3 4)	devuelve "0.7854r"
(ATOI "-13.2")	devuelve -13
(ATOI "123")	devuelve 123
(ATOF "35.72")	devuelve 35.72
(ATOF "-512.67")	devuelve -512.67
(DISTOF "12.34" 2)	devuelve 12.34
(ANGTOF "34" 3)	devuelve 2.58407
(CVUNIT -32.5 "celsius" "kelvin")	devuelve 240.65
(CVUNIT 3600 "segundos" "horas")	devuelve 1.0
(CVUNIT 2 "m^2" "acres")	devuelve 0.000494211

PUNTO II

```
(DEFUN datos_helice ( / mens paso)
  (INITGET 1)
  (SETQ cen (GETPOINT "Centro de la hélice: "))(TERPRI)
  (IF radin0 () (SETQ radin0 0))
  (SETQ mens (STRCAT "Radio inicial <" (RTOS radin0 2 2) ">: "))
  (INITGET 4)
  (IF (SETQ radin (GETDIST cen mens)) () (SETQ radin radin0)) (TERPRI)
  (SETQ radin0 radin)
  (IF radfin0 () (SETQ radfin0 0))
  (SETQ mens (STRCAT "Radio final <" (RTOS radfin0 2 2) ">: "))
  (INITGET 4)
  (IF (SETQ radfin (GETDIST cen mens)) () (SETQ radfin radfin0))
  (TERPRI)
  (SETQ radfin0 radfin)
  (IF pv0 () (SETQ pv0 16))
  (SETQ mens (STRCAT "Precisión de puntos en cada vuelta <"
    (ITOA pv0) ">: "))
  (INITGET 6)
  (IF (SETQ pv (GETINT mens)) () (SETQ pv pv0)) (TERPRI)
  (SETQ pv0 pv)
  (IF nv0 () (SETQ nv0 1))
  (SETQ mens (STRCAT "Número de vueltas <" (ITOA nv0) ">: "))
  (INITGET 6)
  (IF (SETQ nv (GETINT mens)) () (SETQ nv nv0)) (TERPRI)
  (SETQ nv0 nv)
  (INITGET 1 "Paso Altura")
  (SETQ op (GETKEYWORD "Paso/Altura de la hélice: "))(TERPRI)
  (IF (= op "Paso")
    (PROGN
      (IF paso0 () (SETQ paso0 0))
      (SETQ mens (STRCAT "Paso de la hélice <"
        (RTOS paso0 2 2) ">: "))
      (IF (SETQ paso (GETDIST cen mens)) () (SETQ paso paso0))
      (TERPRI)
      (SETQ paso0 paso))
    )
  (IF (= op "Altura")
    (PROGN
      (IF alt0 () (SETQ alt0 0))
      (SETQ mens (STRCAT "Altura total de la hélice <"
        (RTOS alt0 2 2) ">: "))
      (IF (SETQ alt (GETDIST cen mens)) () (SETQ alt alt0)) (TERPRI)
      (SETQ alt0 alt paso (/ alt nv))
    )
  )
  (SETQ dang (/ (* 2 PI) pv))
  (SETQ dz (/ paso pv))
  (SETQ drad (/ (- radfin radin) (* pv nv)))
```

```
)

(DEFUN helice ( / n z0 z xy pto)
  (SETQ p0 (POLAR cen 0 radin))
  (SETQ z0 (CADDR p0))
  (COMMAND "_spline" p0)
  (SETQ n 1)
  (REPEAT (* pv nv)
    (SETQ z (+ z0 (* dz n)))
    (SETQ xy (POLAR cen (* dang n) (+ radin (* drad n))))
    (SETQ pto (list (car xy) (cadr xy) z))
    (COMMAND pto)
    (SETQ n (+ n 1))
  )
  (COMMAND "" "" "")
)

(DEFUN c:helice ( / cen radin radfin nv pv dz drad)
  (SETVAR "cmdecho" 0)
  (SETQ refnt0 (GETVAR "osmode"))(SETVAR "osmode" 0)
  (COMMAND "_undo" "_begin")
  (datos_helice)
  (helice)
  (COMMAND "_undo" "_end")
  (SETVAR "osmode" refnt0)(SETVAR "cmdecho" 1)(PRIN1)
)

(PROMPT "Nuevo comando HELICE definido.")(PRIN1)
```

10ª fase intermedia de ejercicios

PUNTO I

```
(DEFUN C:Edita (/ Mensaje)
  (SETVAR "cmdecho" 0)
  (IF ArchivoDefecto () (SETQ ArchivoDefecto ""))
  (SETQ Mensaje (STRCAT "Nombre de archivo <" ArchivoDefecto ">: "))
  (IF (= "" (SETQ Archivo (GETSTRING Mensaje)))
    (SETQ Archivo ArchivoDefecto)
  )
  (SETQ ArchivoDefecto Archivo)
  (STARTAPP "notepad" (STRCAT Archivo ".lsp"))
  (PRIN1)
)
```

PUNTO II

```
(DEFUN DatosTxArco (/ Mensaje)
  (INITGET 1)
  (SETQ Centro (GETPOINT "Centro del arco para el texto: "))(TERPRI)
  (INITGET 7)
  (SETQ Radio (GETDIST Centro "Radio del arco para el texto: "))(TERPRI)
  (IF AltX0 () (SETQ AltX0 1))
  (INITGET 6)
  (SETQ Mensaje (STRCAT "Altura del texto <" (RTOS AltX0 2 2) ">: "))
  (IF (SETQ AltX (GETDIST Mensaje)) () (SETQ AltX AltX0))(TERPRI)
  (SETQ AltX0 AltX)
  (IF AngTx0 () (SETQ AngTx0 (/ PI 2)))
  (SETQ Mensaje (STRCAT "Angulo de inicio <" (ANGTOS AngTx0 0 0)
    ">: "))
```

```
(IF (SETQ AngTx (GETANGLE Centro Mensaje)) () (SETQ AngTx AngTx0))
(TERPRI)
(SETQ AngTx0 AngTx)
(SETQ CadTx (GETSTRING T "Texto: "))
)

(DEFUN Txarco (/ N LonTx AngIns Tx XIns YIns PIns AngTxN Esp)
  (SETQ LonTx (STRLEN CadTx))
  (SETQ n 1)(SETQ AngIns AngTx)
  (REPEAT Lontx
    (SETQ Tx (SUBSTR CadTx n 1))
    (SETQ XIns (+ (CAR Centro) (* Radio (COS AngIns))))
    (SETQ YIns (+ (CADR Centro) (* Radio (SIN AngIns))))
    (SETQ PIns (LIST XIns YIns))
    (SETQ AngTxn (/ (* (- AngIns (/ PI 2)) 180) PI))
    (COMMAND "_text" PIns AlTx AngTxN Tx)
    (SETQ Esp AlTx)
    (IF (OR (= Tx "i") (= Tx "I") (= Tx "l") (= Tx "1") (= Tx "j")))
      (SETQ Esp (/ AlTx 2))
    )
    (IF (OR (= Tx "m") (= Tx "M"))) (SETQ Esp (* AlTx 1.5)))
    (SETQ AngIns (- AngIns (/ Esp Radio)))
    (SETQ N (1+ N))
  )
)

(DEFUN C:TxArco (/ Radio AlTx Centro AngTx CadTx)
  (SETVAR "cmdecho" 0)
  (SETQ Refnt0 (GETVAR "osmode"))(SETVAR "osmode" 0)
  (COMMAND "_undo" "_begin")
  (DatosTxArco)
  (TxArco)
  (COMMAND "_undo" "_end")
  (SETVAR "osmode" Refnt0)(SETVAR "cmdecho" 1)(PRIN1)
)

(PROMPT "Nuevo comando TXARCO definido.")(PRIN1)
```

11ª fase intermedia de ejercicios

PUNTO I

; Nuevo modo de referencia "Punto medio virtual".
; Puede ser utilizado de manera transparente entre comandos
; tecleando 'mv.
; Puede ser añadido al menu de cursor de Modos de Referencia
; de AutoCAD para tenerlo a mano y no tener que teclearlo.

```
(DEFUN MV (/ Pto1 Pto2)
  (SETQ Pto1 (GETPOINT "medio virtual entre "))
  (SETQ Pto2 (GETPOINT Pto1 "y "))
  (POLAR Pto1 (ANGLE Pto1 Pto2) (/ (DISTANCE Pto1 Pto2) 2.0))
)

(DEFUN C:MV ()
  (mv)
)
```

PUNTO II

```
(DEFUN datos_marca (/ Mensaje)
  (IF DiaP0 () (SETQ DiaP0 2))
  (SETQ Mensaje (STRCAT "\nDiámetro del punto <" (RTOS DiaP0 2 2)
    ">: "))
  (INITGET 4)
  (IF (SETQ DiaP (GETDIST Mensaje)) () (SETQ DiaP DiaP0))(TERPRI)
  (SETQ DiaP0 DiaP)
  (IF DiaM0 () (SETQ DiaM0 10))
  (SETQ Mensaje (STRCAT "Diámetro del círculo de marca <"
    (RTOS DiaM0 2 2) ">: "))
  (INITGET 6)
  (IF (SETQ DiaM (GETDIST Mensaje)) () (SETQ DiaM DiaM0))(TERPRI)
  (SETQ DiaM0 DiaM)
  (INITGET 1)
  (SETQ PtI (GETPOINT "Punto inicial: "))(TERPRI)
  (INITGET 1)
  (SETQ PtF (GETPOINT PtI "Punto de situación de la marca: "))
  (TERPRI)
)

(DEFUN marca (/ Ang Cen Mensaje Tx AlTx TxM)
  (COMMAND "_donut" "0" DiaP PtI "")
  (COMMAND "_line" PtI PtF "")
  (SETQ Ang (ANGLE PtI PtF))
  (SETQ Cen (POLAR PtF Ang (/ DiaM 2)))
  (COMMAND "_circle" Cen "_d" DiaM)
  (IF Tx0 () (SETQ Tx0 1))
  (SETQ Mensaje (STRCAT "Número de marca <" (ITOA Tx0) ">: "))
  (INITGET 6)
  (IF (SETQ Tx (GETINT Mensaje)) () (SETQ Tx Tx0))(TERPRI)
  (SETQ Tx0 (+ Tx 1))
  (SETQ AlTx (/ Diam 2))
  (SETQ TxM (ITOA Tx))
  (COMMAND "_text" "_m" Cen AlTx "0" TxM)
)

(DEFUN C:Marca (/ DiaP DiaM PtI PtF)
  (SETVAR "cmdecho" 0)
  (datos_marca)
  (SETQ refnt0 (GETVAR "osmode"))(SETVAR "osmode" 0)
  (COMMAND "_undo" "_begin")
  (marca)
  (COMMAND "_undo" "_end")
  (SETVAR "osmode" refnt0)(SETVAR "cmdecho" 1)(PRIN1)
)

(PROMPT "Nuevo comando MARCA definido.")(PRIN1)
```

12ª fase intermedia de ejercicios

PUNTO I

```
(DEFUN datos_cortetubo ()
  (SETQ pt1 (GETPOINT "\nPrimer punto del corte (modo FINAL
    activado): "))
  (SETVAR "osmode" 128)
  (SETQ pt2 (GETPOINT pt1 "\nSegundo punto del corte,
    PERpendicular a: "))
  (SETVAR "osmode" 512)
  (SETQ ang2 (ANGLE pt1 (GETPOINT pt1 "\nLado de tubería,
    CERcano de: ")))
```

```
(SETVAR "osmode" 0)
)

(DEFUN cortetubo ( / ang dis p1 p2 p3 p4 p5)
  (SETQ ang (ANGLE pt1 pt2) dis (DISTANCE pt1 pt2))
  (SETQ p1 (POLAR pt1 ang (/ dis 4)))
  (SETQ p1 (POLAR p1 (+ ang2 PI) (/ dis 6)))
  (SETQ p2 (POLAR pt1 ang (/ dis 2)))
  (SETQ p3 (POLAR p2 ang (/ dis 4)))
  (SETQ p4 (POLAR p3 (+ ang2 PI) (/ dis 6)))
  (SETQ p5 (POLAR p3 ang2 (/ dis 6)))
  (COMMAND "_arc" pt1 p1 p2)
  (COMMAND "_arc" p2 p4 pt2)
  (COMMAND "_arc" p2 p5 pt2)
  (COMMAND "_hatch" "_u" 45 2 "_n" p4 p5 "")
)

(DEFUN c:cortetubo ( / pt1 pt2 ang2)
  (SETVAR "cmdecho" 0)
  (SETQ error0 *error* *error* ControlErrores)
  (SETQ refnt0 (GETVAR "osmode"))(SETVAR "osmode" 1)
  (datos_cortetubo)
  (COMMAND "_undo" "_begin")
  (cortetubo)
  (COMMAND "_undo" "_end")
  (SETVAR "cmdecho" 1)(SETVAR "osmode" refnt0)
)
(DEFUN c:ct ()
  (c:cortetubo)
)

(DEFUN ControlErrores (CadenaError)
  (SETQ *error* error0)
  (PRINC (STRCAT "\nError: " CadenaError "."))
  (COMMAND "_undo" "_end")
  (SETVAR "osmode" refnt0)
  (SETVAR "cmdecho" 1)
  (COMMAND)
  (TERPRI)
  (PRIN1)
)

(PROMPT "\nNuevo comando CORTETUBO (abreviatura CT) creado.")(PRIN1)
```

PUNTO II

```
(DEFUN datos_puerta ( / ptang dist mens)
  (SETVAR "osmode" 1)
  (INITGET 1)
  (SETQ ptb (GETPOINT "\nSeñalar esquina (FINAl de): "))
  (SETVAR "osmode" 512)
  (INITGET 1)
  (SETQ ptang (GETPOINT ptb "\nSeñalar pared donde abrir el hueco
    (CERca de): "))
  (SETQ ang (ANGLE ptb ptang))
  (SETVAR "osmode" 0)
  (INITGET 1)
  (SETQ dist (GETDIST ptb "\nDistancia desde la esquina: "))
  (SETQ ptins (POLAR ptb ang dist))
  (IF marco0 ( ) (SETQ marco0 5))
  (SETQ mens (STRCAT "\nEspesor del marco <" (RTOS marco0 2 2)
    ">: "))
  (INITGET 6)
```



```
(IF (SETQ marco (GETDIST ptins mens)) () (SETQ marco marco0))
(SETQ marco0 marco)
(IF hoja0 ( ) (SETQ hoja0 5))
(SETQ mens (STRCAT "\nEspesor de la hoja <" (RTOS hoja0 2 2)
">: "))
(INITGET 6)
(IF (SETQ hoja (GETDIST mens)) () (SETQ hoja hoja0))
(SETQ hoja0 hoja)
(IF ancho0 ( ) (SETQ ancho0 90))
(SETQ mens (STRCAT "\nAnchura del hueco de la puerta <"
(RTOS ancho0 2 2) ">: "))
(INITGET 6)
(IF (SETQ ancho (GETDIST ptins mens)) () (SETQ ancho ancho0))
(SETQ ancho0 ancho)
(SETVAR "osmode" 128)
(INITGET 1)
(SETQ ptlado (GETPOINT ptins "\nSeñalar la pared opuesta del muro
o tabique (PER a): "))
(SETVAR "osmode" 0)
(INITGET "Actual Opuesta")
(IF (SETQ op (GETKEYWORD "\nDirección de apertura (Actual/Opuesta)
<A>: ")) () (SETQ op "Actual"))
)

(DEFUN hueco ( )
(SETQ ptop (POLAR ptins ang ancho))
(SETQ ptlop (POLAR ptlado ang ancho))
(SETVAR "pickbox" 1)
(COMMAND "_break" ptins ptop)
(COMMAND "_break" ptlado ptlop)
(COMMAND "_line" ptins ptlado "")
(COMMAND "_matchprop" ptb "_l" "")
(COMMAND "_line" ptop ptlop "")
(COMMAND "_matchprop" ptb "_l" "")
)

(DEFUN puerta ( / pt pt1 pt2 pt3 pt4 ptarc lhoja)
(IF (= op "Actual")
(PROGN
(SETQ pt ptins ptins ptop ptop pt)
(SETQ pt ptlado ptlado ptlop ptlop pt)
(SETQ ang (- ang PI))
)
)
(SETQ pt1 (POLAR ptins (ANGLE ptins ptlado) marco))
(SETQ pt2 (POLAR pt1 ang marco))
(SETQ pt3 (POLAR pt2 (ANGLE ptlado ptins) marco))
(SETQ ptarc pt3)
(COMMAND "_line" pt1 pt2 pt3 ptins "")
(SETQ pt1 (POLAR ptop (ANGLE ptins ptlado) marco))
(SETQ pt2 (POLAR pt1 (- ang PI) marco))
(SETQ pt3 (POLAR pt2 (ANGLE ptlado ptins) marco))
(COMMAND "_line" pt1 pt2 pt3 ptop "")
(SETQ lhoja (- ancho marco marco))
(SETQ pt2 (POLAR pt3 (- ang PI) hoja))
(SETQ pt1 (POLAR pt2 (ANGLE ptlado ptins) lhoja))
(SETQ pt4 (POLAR pt1 ang hoja))
(COMMAND "_line" pt3 pt2 pt1 pt4 pt3 "")
(SETQ angarc (- (ANGLE pt2 ptarc) (ANGLE pt2 pt1) ))
(SETQ angarc (/ (* angarc 180) PI))
(IF (>= angarc 180) (SETQ angarc (- angarc 360)))
(IF (<= angarc -180) (SETQ angarc (+ angarc 360)))
(COMMAND "_arc" "_c" pt3 pt4 "_a" angarc)
)
(DEFUN c:puerta ( / ptb ang ptins marco hoja ancho ptlado op ptop
```

```
        ptlop)
    (SETVAR "cmdecho" 0)
    (SETQ error0 *error* *error* ControlErrores)
    (SETQ refnt0 (GETVAR "osmode"))(SETQ pick0 (GETVAR "pickbox"))
    (datos_puerta)
    (COMMAND "_undo" "_begin")
    (hueco)
    (puerta)
    (COMMAND "_undo" "_end")
    (SETVAR "cmdecho" 1)(SETVAR "osmode" refnt0)(SETVAR "pickbox"
        pick0)(PRIN1)
)

(DEFUN ControlErrores (CadenaError)
    (SETQ *error* error0)
    (PRINC (STRCAT "\nError: " CadenaError "."))
    (COMMAND "_undo" "_end")
    (SETVAR "pickbox" pick0) (SETVAR "osmode" refnt0)
    (SETVAR "cmdecho" 1)
    (COMMAND)
    (TERPRI)
    (PRIN1)
)

(PROMPT "\nNuevo comando PUERTA definido.")(PRIN1)
```

13ª fase intermedia de ejercicios

PUNTO I

```
;;; Esto es un archivo ACAD.LSP personalizado.

;; Las siguientes funciones serán cargadas en memoria al comenzar

(DEFUN C:...

...

)

;; Las siguientes rutinas AutoLISP, ADS y ARX serán evaluadas en el inicio

(LOAD "c:\\\\...")
(LOAD "c:\\\\...")
(LOAD "d:\\\\...")
(XLOAD "c:\\\\...")
(XLOAD "d:\\\\...")
(ARXLOAD "c:\\\\...")
(ARXLOAD "c:\\\\...")

;; Las siguientes rutinas AutoLISP, ADS y ARX serán cargadas y
;; evaluadas al ser llamadas

(AUTOLOAD "c:/..." '("..."))
(AUTOLOAD "c:/..." '("..."))
(AUTOLOAD "d:/..." '("..."))
(AUTOLOAD "a:/..." '("..."))
(AUTOXLOAD "c:/..." '("..."))
(AUTOXLOAD "c:/..." '("..."))
(AUTOARXLOAD "c:/..." '("..."))
(AUTOARXLOAD "c:/..." '("..."))
;; Lo que sigue será evaluado directamente al comenzar
```

```
(DEFUN S::STARTUP ()  
  
...  
  
)  
  
;; Lo que sigue también será evaluado directamente al comenzar  
  
(...)  
  
;;; Fin del archivo ACAD.LSP personalizado.
```

14ª fase intermedia de ejercicios

PUNTO I

Letrero a)

```
(DEFUN defecto_agujeros ()  
  (START_IMAGE "imagen")  
  (SETQ animg (DIMX_TILE "imagen") altimg (DIMY_TILE "imagen"))  
  (FILL_IMAGE 0 0 animg altimg -2)  
  (SLIDE_IMAGE 0 0 animg altimg "sin.sld")  
  (END_IMAGE)  
  (SET_TILE "sin" "1")  
  (SETQ op "sin")  
  (IF diamagu () (SETQ diamagu 1))  
  (SET_TILE "diamagu" (RTOS diamagu 2 2))  
  (IF profagu () (SETQ profagu 1))  
  (SET_TILE "profagu" (RTOS profagu 2 2))  
  (IF diamcaj () (SETQ diamcaj 1))  
  (SET_TILE "diamcaj" (RTOS diamcaj 2 2))  
  (IF profcaj () (SETQ profcaj 1))  
  (SET_TILE "profcaj" (RTOS profcaj 2 2))  
  (MODE_TILE "cajera" 1)  
)  
  
(DEFUN acciones_cuadro ()  
  (ACTION_TILE "tipo" "(tipo)")  
  (ACTION_TILE "accept" "(aceptar_agujeros)")  
)  
  
(DEFUN tipo ()  
  (SETQ op $value)  
  (IF (= op "sin") (MODE_TILE "cajera" 1) (MODE_TILE "cajera" 0))  
  (COND ((= op "sin") (START_IMAGE "imagen")(FILL_IMAGE 0 0 animg  
                                                    altimg -2)  
                (SLIDE_IMAGE 0 0 animg altimg "sin.sld")  
                (END_IMAGE))  
        ((= op "recta") (START_IMAGE "imagen")(FILL_IMAGE 0 0 animg  
                                                    altimg -2)  
                (SLIDE_IMAGE 0 0 animg altimg "recta.sld")  
                (END_IMAGE))  
        ((= op "avella") (START_IMAGE "imagen")(FILL_IMAGE 0 0 animg  
                                                    altimg -2)  
                (SLIDE_IMAGE 0 0 animg altimg "avella.sld")  
                (END_IMAGE))  
  )
```

```
)

(DEFUN aceptar_agujeros ( )
  (SETQ diamagu (ATOF (GET_TILE "diamagu")))
  (SETQ profagu (ATOF (GET_TILE "profagu")))
  (SETQ diamcaj (ATOF (GET_TILE "diamcaj")))
  (SETQ profcaj (ATOF (GET_TILE "profcaj")))
  (COND ((<= diamagu 0)(MODE_TILE "diamagu" 2)
    (SET_TILE "error" "Diámetro de agujero debe ser mayor que 0"))
    ((<= profagu 0)(MODE_TILE "profagu" 2)
    (SET_TILE "error" "Profundidad de agujero debe ser mayor que 0"))
    ((AND (<= diamcaj 0) (/= op "sin"))(MODE_TILE "diamcaj" 2)
    (SET_TILE "error" "Diámetro de cajera debe ser mayor que 0"))
    ((AND (<= diamcaj diamagu) (/= op "sin"))(MODE_TILE "diamcaj" 2)
    (SET_TILE "error" "Diámetro de cajera debe ser mayor que agujero
    "))
    ((AND (<= profcaj 0) (/= op "sin"))(MODE_TILE "profcaj" 2)
    (SET_TILE "error" "Profundidad de cajera debe ser mayor que 0"))
    (T (DONE_DIALOG 1))
  )
)

(DEFUN insertar_agujeros ( / mens pticj ptigj ang)
  (INITGET 1)
  (SETQ pticj (GETPOINT "Punto de inserción: ")) (TERPRI)
  (IF ang0 () (SETQ ang0 0))
  (SETQ mens (STRCAT "Angulo para la inserción <" (ANGTOS ang0 0 2)
    ">: "))
  (IF (SETQ ang (GETANGLE pticj mens)) () (SETQ ang ang0)) (TERPRI)
  (SETQ ang0 ang)
  (IF (/= op "sin")(cajera)(SETQ ptigj pticj))
  (agujero)
  (eje)
)

(DEFUN cajera ( / pt1 pt2 pt3 pt4)
  (SETQ pt1 (POLAR pticj (- ang (/ PI 2)) (/ diamcaj 2)))
  (SETQ pt2 (POLAR pt1 ang profcaj))
  (SETQ pt3 (POLAR pt2 (+ ang (/ PI 2)) diamcaj))
  (SETQ pt4 (POLAR pt3 (+ ang PI) profcaj))
  (IF (= op "avella") (PROGN (SETQ pt1 (POLAR pt1 (- ang (/ PI 2))
    profcaj))
    (SETQ pt4 (POLAR pt4 (+ ang (/ PI 2))
    profcaj)))
  )
  (COMMAND "_line" pt1 pt2 pt3 pt4 "")
  (SETQ ptigj (POLAR pticj ang profcaj))
)

(DEFUN agujero ( / pt1 pt2 pt3 pt4 pt5)
  (SETQ pt1 (POLAR ptigj (- ang (/ PI 2)) (/ diamagu 2)))
  (SETQ pt2 (POLAR pt1 ang profagu))
  (SETQ pt3 (POLAR pt2 (+ ang (/ PI 2)) (/ diamagu 2)))
  (SETQ pt3 (POLAR pt3 ang (/ diamagu 2)))
  (SETQ pt4 (POLAR pt2 (+ ang (/ PI 2)) diamagu))
  (SETQ pt5 (POLAR pt4 (+ ang PI) profagu))
  (COMMAND "_line" pt1 pt2 pt4 pt5 "")
  (COMMAND "_line" pt2 pt3 pt4 "")
)

(DEFUN eje (/ lon lonj ptij capj ptfj)
```

```
(SETQ lon (+ profcaj profagu (/ diamagu 2)))
(SETQ lonj (* lon 1.25))
(SETQ ptij (POLAR pticj (+ ang PI) (* lon 0.125)))
(IF (= "" (SETQ capj (GETSTRING "Nombre de capa para el eje <EJES>:
"))))
    (SETQ capj "EJES")
)(TERPRI)
(SETQ ptfj (POLAR ptij ang lonj))
(COMMAND "_layer" "_make" capj "")
(COMMAND "_line" ptij ptfj "")
(COMMAND "_layer" "_set" cap0 "")
)

(DEFUN c:ddagujeros ( / animg alting op)
  (SETVAR "cmdecho" 0)
  (SETQ error0 *error* *error* errores)
  (SETQ ind (LOAD_DIALOG "agujeros.dcl"))
  (IF (NOT (NEW_DIALOG "agujeros" ind))
    (PROGN (PROMPT "\nNo se encuentra el archivo AGUJEROS.DCL")(QUIT))
  )
  (defecto_agujeros)
  (acciones_cuadro)
  (IF (= 1 (START_DIALOG))(insertar_agujeros))
  (UNLOAD_DIALOG ind)
  (SETVAR "cmdecho" 1)(PRIN1)
)

(DEFUN errores (mens)
  (SETQ *error* error0)
  (IF (= mens "quitar / salir abandonar")
    (PRIN1)
    (PRINC (STRCAT "\nError: " mens " ")))
  )
  (SETVAR "cmdecho" 1)(PRIN1)
)

(PROMPT "Nuevo comando DDAGUJEROS definido")(PRIN1)
```

NOTA: Apréciase la utilización de diversos códigos de estado con DONE_DIALOG para ser luego controlados en el START_DIALOG.

Letreros b-1 y b-2)

```
(DEFUN listabl_inserdir (/ arch línea)
  (IF ruta0 () (SETQ ruta0 "c:\\cad13"))
  (IF (= "" (SETQ ruta (GETSTRING (STRCAT "\nDirectorio de bloques <"
    ruta0 ">: "))))
    (SETQ ruta ruta0))
  (WHILE (NOT (FINDFILE (STRCAT ruta "\\bloques.lst"))))
    (PROMPT "\nNo se ha encontrado archivo BLOQUES.LST con listado de
    archivos de bloque"))
    (IF (SETQ ruta (GETSTRING (STRCAT "\nDirectorio de bloques <"
    ruta0 ">: ")))
      () (SETQ ruta ruta0))
  )
  (SETQ ruta0 ruta)
  (SETQ arch (OPEN (STRCAT ruta "\\bloques.lst") "r"))
  (SETQ listabl '())
  (WHILE (SETQ línea (READ-LINE arch))
    (SETQ nomb1 (SUBSTR línea 1 (- (STRLEN línea) 4)))
```

```
(IF (/= nomb1 "")(SETQ listabl (CONS nomb1 listabl)))
)
(CLOSE arch)
(SETQ listabl (REVERSE listabl))
(SETQ totbl (LENGTH listabl))
)

(DEFUN cuadro (/ nimg totimg nbl nomb1 nomtile)
  (IF (<= restbl 16)(SETQ totimg restbl)(SETQ totimg 16))
  (SETQ nimg 0)
  (REPEAT totimg
    (SETQ nbl (+ indbl nimg))
    (SETQ nomb1 (STRCAT ruta "\\\" (NTH nbl listabl)))
    (SETQ nomtile (STRCAT "img" (ITOA nimg)))
    (START_IMAGE nomtile)
    (FILL_IMAGE 0 0 (DIMX_TILE nomtile) (DIMY_TILE nomtile) 0)
    (SLIDE_IMAGE 0 0 (DIMX_TILE nomtile) (DIMY_TILE nomtile) nomb1)
    (END_IMAGE)
    (SETQ nimg (1+ nimg))
  )
  (REPEAT (- 16 totimg)
    (SETQ nbl (+ indbl nimg))
    (SETQ nomtile (STRCAT "img" (ITOA nimg)))
    (START_IMAGE nomtile)
    (FILL_IMAGE 0 0 (DIMX_TILE nomtile) (DIMY_TILE nomtile) 0)
    (END_IMAGE)
    (SETQ nimg (1+ nimg))
  )
  (IF (= indbl 0) (MODE_TILE "pre" 1) (MODE_TILE "pre" 0))
  (IF (<= restbl 16) (MODE_TILE "sig" 1) (MODE_TILE "sig" 0))
)

(DEFUN lista (/ indblst)
  (SETQ nomb1 (NTH nblast listabl))
  (IF (= ins 4) (DONE_DIALOG 1))
  (SETQ indblst (* 16 (FIX (/ nblast 16))))
  (SETQ restbl (- totbl indblst))
  (IF (/= indblst indbl) (PROGN (SETQ indbl indblst)(cuadro)) )
  (SETQ nimg (- nblast indbl))
  (MODE_TILE (STRCAT "img" (ITOA nimg)) 2)
)

(DEFUN mostrar ( )
  (NEW_DIALOG "muestra" ind)
  (START_IMAGE "muestra")
  (FILL_IMAGE 0 0 (DIMX_TILE "muestra") (DIMY_TILE "muestra") 0)
  (SLIDE_IMAGE 0 0 (DIMX_TILE "muestra") (DIMY_TILE "muestra")(STRCAT
    ruta "\\\" nomb1))
  (END_IMAGE)
  (START_DIALOG)
  (MODE_TILE (STRCAT "img" (ITOA nimg)) 2)
)

(DEFUN funciones_img (/ nfun nomfun)
  (SETQ nfun 0)
  (REPEAT 16
    (SETQ nomfun (STRCAT "img" (ITOA nfun)))
    (EVAL (READ (STRCAT "(defun " nomfun " ( ) "
      "(setq nimg " (ITOA nfun) )"
      "(setq nbl (+ " (ITOA nfun) " indbl))"
      "(setq nomb1 (nth nbl listabl))"
    ))))
  )
)
```

```
                "(if (= ins 4) (done_dialog 1))"
                "(set_tile \"lista\" (itoa nbl)))"   )))
    (SETQ nfun (1+ nfun))
  )
)

(DEFUN sig ()
  (SETQ indbl (+ indbl 16) restbl (- restbl 16))
  (cuadro)
  (SET_TILE "lista" (ITOA indbl))
  (MODE_TILE "img0" 2)
  (SETQ nombl (NTH indbl listabl))
  (SETQ nimg 0)
)

(DEFUN pre ()
  (SETQ indbl (- indbl 16) restbl (+ restbl 16))
  (cuadro)
  (SET_TILE "lista" (ITOA indbl))
  (MODE_TILE "img0" 2)
  (SETQ nombl (NTH indbl listabl))
  (SETQ nimg 0)
)

(DEFUN c:ddinserdir (/ ruta listabl totbl indbl restbl nblist ins ind
                    nimg nombl nomtile)
  (SETVAR "cmdecho" 0)(SETQ error0 *error* *error* err_ddinserdir)
  (listabl_inserdir)
  (SETQ indbl 0 restbl totbl)
  (SETQ ind (LOAD_DIALOG "inserdir"))
  (NEW_DIALOG "inserdir" ind)
  (START_LIST "lista")
  (MAPCAR 'add_list listabl)
  (END_LIST)
  (cuadro)
  (SET_TILE "lista" (ITOA indbl))
  (MODE_TILE "img0" 2)
  (SETQ nombl (NTH 0 listabl))
  (ACTION_TILE "lista" "(setq nblist (ATOI $value))(setq ins
$reason)(lista)")
  (ACTION_TILE "mostrar" "(if nombl (mostrar))")
  (SETQ nimg 0)
  (REPEAT 16
    (SETQ nomtile (STRCAT "img" (ITOA nimg)))
    (ACTION_TILE nomtile (STRCAT "(setq ins $reason)(" nomtile ")"))
    (SETQ nimg (1+ nimg))
  )
  (ACTION_TILE "sig" "(sig)")
  (ACTION_TILE "pre" "(pre)")
  (funciones_img)
  (IF (= 1 (START_DIALOG))
    (PROGN (PROMPT "Punto de inserción: ")
      (COMMAND "insert" (STRCAT ruta "\\\" nombl) pause "" ""))
  )
  (SETVAR "cmdecho" 1)(SETQ *error* error0)(PRIN1)
)

(DEFUN err_ddinserdir (mens)
  (SETQ *error* error0)
  (IF (= mens "quitar / salir abandonar")
    (PRIN1)
```

```
(PRINC (STRCAT "\nError: " mens " "))
)
(SETVAR "cmdecho" 1)(SETQ *error* error0)(PRIN1)
)

(PROMPT "Nuevo comando DDINSERTDIR definido.")(PRIN1)
```

Letrero c)

```
(DEFUN c:ddsombra (/error0 ind edge muestra)
  (SETQ error0 *error* *error* err-ddmodelar)
  (IF esfr0 () (SETQ esfr0 "1"))
  (SETQ ind (LOAD_DIALOG "modelar"))
  (IF (NOT (NEW_DIALOG "modelar" ind))
    (PROGN (PROMPT "No se encuentra el archivo MODELAR.DCL")(QUIT))
  )
  (COND
    ((= 0 (GETVAR "shadededge"))
      (SET_TILE "edge0" "1")
      (SETQ edge "edge0"))
    ((= 1 (GETVAR "shadededge"))
      (SET_TILE "edge1" "1")
      (SETQ edge "edge1"))
    ((= 2 (GETVAR "shadededge"))
      (SET_TILE "edge2" "1")
      (SETQ edge "edge2"))
    ((= 3 (GETVAR "shadededge"))
      (SET_TILE "edge3" "1")
      (SETQ edge "edge3"))
  )
  (COND
    ((= (GETVAR "shadedif") 30)(SET_TILE "dif30" "1"))
    ((= (GETVAR "shadedif") 50)(SET_TILE "dif50" "1"))
    ((= (GETVAR "shadedif") 70)(SET_TILE "dif70" "1"))
    ((= (GETVAR "shadedif") 90)(SET_TILE "dif90" "1"))
  )
  (SET_TILE "desliza" (ITOA (GETVAR "shadedif")))
  (SET_TILE "valor" (ITOA (GETVAR "shadedif")))
  (SET_TILE "esfera" esfr0)
  (SET_TILE "cubo" (ITOA (1- (ATOI esfr0))))
  (IF (= "1" esfr0) (SETQ muestra "esfera") (SETQ muestra "cubo"))
  (ACTION_TILE "radio_edge" "(SETQ edge $value)")
  (ACTION_TILE "defecto" "(defecto)")
  (ACTION_TILE "desliza" "(desliza)")
  (ACTION_TILE "valor" "(valor)")
  (ACTION_TILE "dif30" "(valorp \"30\")")
  (ACTION_TILE "dif50" "(valorp \"50\")")
  (ACTION_TILE "dif90" "(valorp \"70\")")
  (ACTION_TILE "dif90" "(valorp \"90\")")
  (ACTION_TILE "muestra" "(SETQ muestra $value)")
  (ACTION_TILE "muestra" "(mostrar)")
  (ACTION_TILE "accept" "(aceptar)")
  (IF (= 1 (START_DIALOG)) (COMMAND "sombra"))
  (SETQ *error* error0) (PRIN1)
)

(DEFUN aceptar (/val)
  (SETQ val (GET_TILE "valor"))
  (IF (AND (<= (ATOI val) 100) (>= (ATOI val) 0))
    (PROGN
```



```
(SETVAR "shadedge" (ATOI (SUBSTR edge 5)))
(SETVAR "shadeif" (ATOI val))
(SETQ esfr0 (GET_TILE "esfera"))
(DONE_DIALOG 1)
)
(MODE_TILE "valor" 2)
)
)

(DEFUN mostrar (/ foto ancho alto)
  (SETQ foto (STRCAT (SUBSTR muestra 1 3)(SUBSTR edge 5)))
  (SETQ ancho (DIMX_TILE "imagen") alto DIMY_TILE "imagen"))
  (START_IMAGE "imagen")
  (FILL_IMAGE 0 0 ancho alto -2)
  (SLIDE_IMAGE 0 0 ancho alto foto)
  (END_IMAGE)
)

(DEFUN defecto ()
  (SET_TILE "edge" "1")
  (SET_TILE "dif70" "1")
  (SET_TILE "desliza" "70")
  (SET_TILE "valor" "70")(SET_TILE "error" "")
  (SET_TILE "esfera" "1")
  (SETQ edge "edgel" muestra "esfera")
)

(DEFUN desliza ()
  (SET_TILE "valor" (GET_TILE "desliza"))
  (FOREACH p '("dif30" "dif50" "dif70" "dif90") (SET_TILE p "0"))
  (SET_TILE "error" "")
)

(DEFUN valor (/ val)
  (SETQ val (GET_TILE "valor"))
  IF (AND (<= (ATOI val) 100)(>= (ATOI val) 0))
    (PROGN
      (SET_TILE "error" "")
      (SET_TILE "desliza" val)
      (FOREACH p '("dif30" "dif50" "dif70" "dif90") (SET_TILE p "0"))
    )
    (SET_TILE "error" "El valor debe estar entre 0 y 100")
  )
)

(DEFUN valorp (p)
  (SET_TILE "valor" p) (SET_TILE "desliza" p) (SET_TILE "error" "")
)

(DEFUN err-ddmodelar (mens)
  (SETQ *error* error0)
  (IF (= mens "quitar / salir abandonar")
    (PRIN1)
    (PRINC (STRCAT "\nError: " mens " ")))
  )
  (PRIN1)
)

(PROMPT "Nueva orden DDMODELAR definida.")(PRIN1)
```

NOTA: Apréciase la manera de incluir caracteres de comillas dobles con el código de control correspondiente \"; como por ejemplo en "(valorp \"30\")". Esto se hace así por estar incluidas dentro de una cadena —que lleva ya comillas de por sí—.

NOTA: En estos ejercicios que hemos visto los archivos de fotos (.SLD) que se representen en casillas de imagen habrán de estar en un directorio de soporte. También pueden situarse en cualquier otro directorio, pero habremos de tomar la precaución de incluir la ruta en la llamada al fichero en cuestión. Lo mismo para los archivos de definición del diseño gráfico del letrero de diálogo (.DCL).

15ª fase intermedia de ejercicios

PUNTO I

```
— (X G F)
— (56 56 78 65.6)
— (RESIDO AQUÍ MISMO)
— (TIPO . 32)
— (45 32 5 2 4)
— (12 ER 45 FG DR FR 54 3.45 (12 34) (DF FR))
— 9
— YH
— (23.45 54.43)
— nil
— C5
— nil
— (RESIDO AQUÍ MISMO)
— ("01" "02" "12" "as" "aw" "h" "perra" "perro")
```

16ª fase intermedia de ejercicios

PUNTO I

```
— (GRAPHSCR)
— (TEXTSCR)
— Para asegurar la visualización en determinado momento de una de ellas.
— "AutoLISP Release 14.0 (en)"
— SET atribuye valores a símbolos literales; SETQ a símbolos no literales.
```

17ª fase intermedia de ejercicios

PUNTO I

```
(DEFUN Datos_Intercala (/ mens)
  (IF nbl0 () (SETQ nbl0 ""))
  (SETQ mens (STRCAT "Nombre de bloque unitario <" nbl0 ">: "))
  (IF (= "" (SETQ nbl (GETSTRING mens))) (SETQ nbl nbl0) )
  (WHILE (AND (NOT (TBLSEARCH "block" nbl))(NOT (FINDFILE (STRCAT nbl ".dwg"))))
    (PROMPT (STRCAT "Bloque no encontrado en dibujo ni en directorios ACAD.\n"))
    (IF (= "" (SETQ nbl (GETSTRING mens))) (SETQ nbl nbl0) )
  )
  (SETQ nbl0 nbl)
  (SETVAR "osmode" 512)
```

```
(PROMPT "\nSeñalar la línea a partir. ")
(INITGET 1)
(WHILE (NOT (SETQ lin (ENTSEL "Punto de inserción CERca a: "))))
(SETQ pti (OSNAP (CADR lin) "cer"))
(SETQ lin (SUBST pti (CADR lin) lin))
(IF escx0 () (SETQ escx0 1))
(SETQ mens (STRCAT "\nFactor de escala X <" (RTOS escx0 2 2) ">: "))
(INITGET 2)
(IF (SETQ escx (GETREAL mens)) () (SETQ escx escx0))
(SETQ escx0 escx)
(INITGET 2)
(SETQ mens (STRCAT "\nFactor de escala Y <" (RTOS escx 2 2) ">: "))
(IF (SETQ escy (GETREAL mens)) () (SETQ escy escx))
(IF ang0 () (SETQ ang0 0))
(SETQ mens (STRCAT "\nModo de referencia CERca. Angulo de rotación <"
(RTOS (/ (* 180 ang0) PI) 2 2) ">: "))
(IF (SETQ ang (GETANGLE pti mens)) () (SETQ ang ang0))
(SETQ ang0 ang)
(SETVAR "osmode" 0)
)

(DEFUN Intercala (/ pf angins)
(SETQ angins (/ (* ang 180) PI))
(COMMAND "insert" nbl pti escx escy angins)
(SETQ pf (POLAR pti ang escx))
(COMMAND "parte" lin pf)
(REDRAW)
)

(DEFUN C:Intercala (/ Refnt0 nbl lin pti escx escy ang)
(SETVAR "cmdecho" 0)
(SETQ Error0 *error* *error* Errores)
(SETQ Refnt0 (GETVAR "osmode"))
(Datos_Intercala)
(Intercala)
(SETVAR "osmode" Refnt0)(SETVAR "cmdecho" 1)(PRIN1)
)
(SETFUNHELP "C:Intercala" "alisp.ahp" "Intercala")

(DEFUN Errores (Mensaje)
(SETQ *error* Error0)
(PRINC Mensaje)
(SETVAR "cmdecho" 1)(SETVAR "osmode" Refnt0)(PRIN1)
)
)
```

18ª fase intermedia de ejercicios

PUNTO I

```
(DEFUN opcion_protext (/ num op)
(SETQ num (SSLENGTH conj))
(SETQ op T)
(WHILE op
(INITGET "MAYúsculas MINúsculas Oblicuidad Anchura desHacer")
(SETQ op (GETKEYWORD "Propiedad que desea cambiar
MAYúsculas/MINúsculas/Oblicuidad/Anchura/desHacer: "))
(TERPRI)
(COND ((= op "MAYúsculas")(cam_may_min nil))
((= op "MINúsculas")(cam_may_min T))
((= op "Oblicuidad")(cam_obl))
((= op "Anchura")(cam_anch))
((= op "desHacer")(COMMAND "_u")))
)
```

```
)
)
)

(DEFUN cam_may_min (min / n tx txact txn listxn)
  (COMMAND "_undo" "_begin")
  (SETQ n 0)
  (REPEAT num
    (SETQ tx (SSNAME conj n))
    (SETQ txact (CDR (ASSOC 1 (ENTGET tx))))
    (SETQ txn (STRCASE txact min))
    (SETQ listxn (SUBST (CONS 1 txn) (ASSOC 1 (ENTGET tx))
      (ENTGET tx)))
    (ENTMOD listxn)
    (SETQ n (1+ n))
  )
  (COMMAND "_undo" "_end")
)

(DEFUN cam_obl (/ n lent oblact nobl)
  (SETQ oblact (CDR (ASSOC 51 (ENTGET (SSNAME conj 0)))))
  (IF oblact ( ) (SETQ oblact 0))
  (SETQ oblact (/ (* oblact 180) PI))
  (IF (SETQ nobl (GETREAL (STRCAT "Nuevo valor de oblicuidad <"
    (RTOS oblact 2 2) ">: ")))
    ( )
    (SETQ nobl oblact)
  )
  (SETQ nobl (/ (* nobl PI) 180))
  (COMMAND "_undo" "_begin")
  (SETQ n 0)
  (REPEAT num
    (SETQ lent (ENTGET (SSNAME conj n)))
    (SETQ lent (SUBST (CONS 51 nobl)(ASSOC 51 lent) lent))
    (ENTMOD lent)
    (SETQ n (+ 1 n))
  )
  (COMMAND "_undo" "_end")
)

(DEFUN cam_anch (/ anact nanch n lent )
  (SETQ anact (CDR (ASSOC 41 (ENTGET (SSNAME conj 0)))))
  (IF anact ( ) (SETQ anact 1))
  (INITGET 6)
  (IF (SETQ nanch (GETREAL (STRCAT "Nueva anchura <" (RTOS anact 2 2)
    ">: ")))
    ( )
    (SETQ nanch anact)
  )
  (TERPRI)
  (SETQ n 0)
  (COMMAND "_undo" "_begin")
  (REPEAT num
    (SETQ lent (ENTGET (SSNAME conj n)))
    (SETQ lent (SUBST (CONS 41 nanch)(ASSOC 7 lent) lent))
    (ENTMOD lent)
    (SETQ n (+ 1 n))
  )
  (COMMAND "_undo" "_end")
)
```

```
(DEFUN c:protext (/ conj error0)
  (SETVAR "cmdecho" 0)
  (SETQ error0 *error* *error* err_protext)
  (IF (SETQ conj (SSGET (LIST (CONS 0 "text"))))
    (opcion_protext)
    (PROMPT "Ningún texto encontrado."))
  (SETQ *error* error0)(SETVAR "cmdecho" 0)(PRIN1)
)

(DEFUN err_protext (mens)
  (SETQ *error* error0)
  (PRIN1)
  (PRINC (STRCAT "\nError: " mens " "))
  (COMMAND "_undo" "_end")
  (SETVAR "cmdecho" 1)(PRIN1)
)
```

PUNTO II

```
(DEFUN inic_unepol ()
  (WHILE (NOT (SETQ polb (ENTSEL "\nDesignar 3dpol: "))))
  (SETQ polb (CAR polb))
  (WHILE (OR (/= (CDR (ASSOC 0 (ENTGET polb))) "POLYLINE")
    (/= 1 (FIX (/ (CDR (ASSOC 70 (ENTGET polb))) 8))))
    (PROMPT "\nEntidad designada no es 3dpol")(TERPRI)
    (WHILE (NOT (SETQ polb (ENTSEL "\nDesignar 3dpol: "))))
    (SETQ polb (CAR polb))
  )
  (WHILE (NOT (SETQ polj (ENTSEL "\nDesignar 3dpol a juntar: "))))
  (SETQ polj (CAR polj))
  (WHILE (OR (EQUAL polb polj)
    (/= (CDR (ASSOC 0 (ENTGET polj))) "POLYLINE")
    (/= 1 (FIX (/ (CDR (ASSOC 70 (ENTGET polj))) 8))))
    (IF (EQUAL polb polj)
      (PROMPT "\nSe ha designado la misma 3dpol.")
      (PROMPT "\nEntidad designada no es 3dpol."))
    (WHILE (NOT (SETQ polj (ENTSEL "\nDesignar 3dpol a juntar: "))))
    (SETQ polj (CAR polj))
  )
)

(DEFUN juntar_unepol (/ cabec listob lis e1 e2 pb1 pb2 pj1 pj2 endb
  liston)
  (SETQ cabec (ENTGET polb))
  (SETQ e1 (ENTNEXT polb))
  (SETQ lis (ENTGET e1))(SETQ listob (LIST lis))
  (SETQ pb1 (CDR (ASSOC 10 lis)))
  (WHILE (/= (CDR (ASSOC 0 lis)) "SEQEND")
    (SETQ pb2 (CDR (ASSOC 10 lis)))
    (SETQ e2 (ENTNEXT e1))
    (SETQ lis (ENTGET e2))
    (SETQ listob (CONS lis listob))
    (SETQ e1 e2)
  )
  (SETQ endb lis)
  (SETQ listob (CDR listob))
  (SETQ e1 (ENTNEXT polj))
  (SETQ lis (ENTGET e1))(SETQ liston (LIST lis))
  (SETQ pj1 (CDR (ASSOC 10 lis)))
  (WHILE (/= (CDR (ASSOC 0 lis)) "SEQEND")
```

```
(SETQ pj2 (CDR (ASSOC 10 lis)))
(SETQ e2 (ENTNEXT e1))
(SETQ lis (ENTGET e2))
(SETQ liston (CONS lis liston))
(SETQ e1 e2)
)
(SETQ liston (CDR liston))
(COND ((EQUAL pb2 pj1) (PROGN (SETQ listob (CDR listob))
                              (ENTDEL polj)(ENTDEL polb)
                              (ENTMAKE cabec)
                              (makeb_unepol)(makej_unepol)
                              (ENTMAKE endb)))
      ((EQUAL pb2 pj2) (PROGN (SETQ listob (CDR listob))
                              (SETQ liston (REVERSE liston))
                              (ENTDEL polb)(ENTDEL polj)
                              (ENTMAKE cabec)
                              (makeb_unepol)(makej_unepol)
                              (ENTMAKE endb)))
      ((EQUAL pb1 pj2) (PROGN (SETQ liston (CDR liston))
                              (ENTDEL polb)(ENTDEL polj)
                              (ENTMAKE cabec)
                              (makej_unepol)(makeb_unepol)
                              (ENTMAKE endb)))
      ((EQUAL pb1 pj1) (PROGN (SETQ liston (CDR liston))
                              (SETQ liston (REVERSE liston))
                              (ENTDEL polb)(ENTDEL polj)
                              (ENTMAKE cabec)
                              (makej_unepol)(makeb_unepol)
                              (ENTMAKE endb)))
      (T (PROMPT "Entidades no se tocan por un extremo")))
)
)

(DEFUN makeb_unepol (/ n longl lis)
  (SETQ longl (LENGTH listob))
  (SETQ n 1)
  (REPEAT longl
    (SETQ lis (NTH (- longl n) listob))
    (ENTMAKE lis)(SETQ n (+ n 1))
  )
)

(DEFUN makej_unepol (/ n long lis)
  (SETQ longl (LENGTH liston))
  (SETQ n 1)
  (REPEAT longl
    (SETQ lis (NTH (- longl n) liston))
    (ENTMAKE lis)(SETQ n (+ n 1))
  )
)

(DEFUN c:unepol ()
  (SETQ error0 *error* *error* err_unepol)
  (SETVAR "cmdecho" 0)
  (inic_unepol)
  (COMMAND "_undo" "_begin")
  (juntar_unepol)
  (COMMAND "_undo" "_end")
  (SETVAR "cmdecho" 1)(PRIN1)
)
```

```
(DEFUN err_unepol (mens)
  (SETQ *error* error0)
  (IF (= mens "quitar / salir abandonar")
    (PRIN1)
    (PRINC (STRCAT "\nError: " mens " ")))
  )
  (COMMAND "_undo" "_end")
  (SETVAR "cmdecho" 1)(PRIN1)
)
```

PUNTO III

```
(DEFUN intr_buscabl ()
  (SETQ nomb1 (GETSTRING "Nombre de bloque a buscar (Todos):
                          "))(TERPRI)
  (IF (OR (= nomb1 "") (= nomb1 "T") (= nomb1 "t"))
    (SETQ nomb1 "$T$")
  )
)

(DEFUN buscabl (/ conjbl fl numb1 pins escfl mtrans n)
  (IF (= nomb1 "$T$")
    (SETQ conjbl (SSGET "X" (LIST (CONS 0 "insert"))))
    (SETQ conjbl (SSGET "X" (LIST (CONS 0 "insert")(CONS 2 nomb1)))))
  )
  (IF conjbl (resalta_buscabl)
    (PROMPT "¡Ninguna inserción de bloque encontrada!")
  )
)

(DEFUN resalta_buscabl ()
  (SETQ fl '( 3 (0.680083 -0.734131) (0.734131 -0.680083)
               (0.734131 -0.680083) (0.277744 -0.223695)
               (0.277744 -0.223695) (0.516306 -0.194718)
               (0.516306 -0.194718) (0.0 0.0)
               (0.194718 -0.516306) (0.0 0.0)
               (0.223695 -0.277744) (0.194718 -0.516306)
               (0.680083 -0.734131) (0.223695 -0.277744)
             ))
  )
  (SETQ numb1 (SLENGTH conjbl))(SETQ n 0)
  (REPEAT numb1
    (REDRAW (SSNAME conjbl n) 3)
    (SETQ pins (CDR (ASSOC 10 (ENTGET (SSNAME conjbl n)))))
    (SETQ escfl (/ (GETVAR "viewsize") 20))
    (SETQ mtrans (LIST (LIST escfl 0 0 (CAR pins))
                       (LIST 0 escfl 0 (CADR pins))
                       (LIST 0 0 1 (CADDR pins))
                       (LIST 0 0 0 1)))
    (GRVECS fl mtrans)
    (SETQ n (+ n 1))
  )
  (PROMPT (STRCAT "Encontradas " (ITOA numb1) " inserciones de
                  bloque"))(TERPRI)
)

(DEFUN c:buscabl (/ nomb1 bl error0)
  (SETQ error0 *error* *error* err_buscabl)
  (intr_buscabl)
  (COND ((= nomb1 "$T$")
```

```
        (IF (TBLNEXT "block" T)
            (buscabl)
            (PROMPT "¡No hay bloques definidos en el dibujo!"))
    )
    ((TBLSEARCH "block" nomb1)
     (buscabl)
    )
    (T (PROMPT "¡No está definido el bloque en el dibujo!"))
)
(SETQ *error* error0)(PRIN1)
)

(DEFUN err_buscabl (mens)
  (SETQ *error* error0)
  (PRIN1)
  (PRINC (STRCAT "\nError: " mens " "))
  (PRIN1)
)

(PROMPT "Nuevo comando BUSCABL definido.")(PRIN1)
```

19ª fase intermedia de ejercicios

PUNTO I

```
(DEFUN cu (/ op path valor nop lstgru unil uni2)
  (SETQ error0 *error* *error* errores_cu)
  (IF (NOT (SETQ path (FINDFILE "acad.unt"))))
    (PROMPT "Archivo ACAD.UNT no se encuentra en los caminos de acceso ACAD.")
    (PROGN (INITGET 1) (SETQ valor (GETREAL "Valor que convertir: "))(TERPRI)
      (grupos_unidades)
      (INITGET 7)
      (WHILE (<= nop (SETQ op (GETINT "Seleccionar opción: ")))
        (INITGET 7))
      (unidades)
      (conversión)
      (SETQ *error* error0)
      (SETQ valor (CVUNIT valor unil uni2))
    )
  )
)

(DEFUN grupos_unidades (/ arch nref nlin car0 car1 car2 línea)
  (TEXTPAGE)
  (PROMPT " Tipos de unidades existentes.\n\n")
  (SETQ arch (OPEN path "r"))
  (SETQ nop 1 nref 0)
  (SETQ nlin 1 car0 "")
  (WHILE (SETQ línea (READ-LINE arch))(buscar_grupo))
  (CLOSE arch)
)

(DEFUN buscar_grupo ()
  (IF (/= ";" (SETQ car1 (SUBSTR línea 1 1)))
    (SETQ car0 car1 nlin (1+ nlin))
    (PROGN (SETQ car2 (SUBSTR (READ-LINE arch) 1 1))
      (IF (AND (= car0 "") (= car2 ""))
        (PROGN (SETQ nref (1+ nref))
          (IF (AND (> nref 3)(< nref 13)) (opcion_grupo)))
      )
    )
  )
)
```



```
)
  (SETQ nlin (+ 2 nlin) car0 car2)
)
)
)

(DEFUN opción_grupo (/ lista mens)
  (SETQ línea (SUBSTR línea 2))
  (SETQ lista (LIST nop línea nlin))
  (SETQ lstgru (CONS lista lstgru))
  (SETQ mens (STRCAT (ITOA nop) " - " línea))
  (PROMPT mens)(TERPRI)(TERPRI)
  (SETQ nop (1+ nop))
)

(DEFUN unidades (/ arch línea)
  (TEXTPAGE)
  (PROMPT (STRCAT "\n" (CADR (ASSOC op lstgru)) "\n\n"))
  (SETQ arch (OPEN path "r"))
  (REPEAT (CADDR (ASSOC op lstgru)) (READ-LINE arch))
  (READ-LINE arch)
  (WHILE (/= "" (SETQ línea (READ-LINE arch)))
    (IF (= "*" (SUBSTR línea 1 1))(impr_unidad))
  )
  (CLOSE arch)
)

(DEFUN impr_unidad ()
  (SETQ línea (SUBSTR línea 2))
  (PRINC línea)(PRINC "\t")
)

(DEFUN conversión ()
  (PROMPT (STRCAT "\n\nConvertir " (RTOS valor 2 8)))
  (INITGET 1)
  (SETQ uni1 (GETSTRING "    Unidades de origen: "))
  (INITGET 1)
  (SETQ uni2 (GETSTRING "Unidades de destino: "))
)

(DEFUN errores_cu (mensaje)
  (SETQ *error* error0)
  (PRIN1)
  (PRINC (STRCAT "\nError: " mensaje ". "))
  (PRIN1)
)

(PROMPT "Nueva función transparente (cu) creada.")(PRIN1)
```

PUNTO II

```
(DEFUN datos_lineat (/ mens)
  (IF nomtl0 () (SETQ nomtl0 ""))
  (SETQ mens (STRCAT "Nombre del tipo de línea compleja con texto <" nomtl0 ">: "))
  (IF (= "" (SETQ nomtl1 (GETSTRING mens))) (SETQ nomtl1 nomtl0) )
  (SETQ nomtl0 nomtl1)
  (IF tx0 () (SETQ tx0 ""))
  (SETQ mens (STRCAT "\nTexto incluido en el tipo de línea compleja <" tx0 ">: "))
  (IF (= "" (SETQ tx (GETSTRING mens))) (SETQ tx tx0) )
  (SETQ tx0 tx)
```

```
(SETQ estilo0 (GETVAR "textstyle"))
(SETQ mens (STRCAT "\nEstilo de texto <" estilo0 ">: "))
(IF (= "" (SETQ estil (GETSTRING mens))) (SETQ estil estilo0))
(PROMPT "\nIntroducir valores unitarios (serán multiplicados por ESCALATL).")
(IF altx0 () (SETQ altx0 1))
(SETQ mens (STRCAT "\nAltura del texto <" (RTOS altx0 2 2) ">: "))
(INITGET 6)
(IF (SETQ altx (GETREAL mens)) () (SETQ altx altx0))
(SETQ altx0 altx)
(IF rot0 () (SETQ rot0 0))
(SETQ mens (STRCAT "\nRotación del texto respecto a la línea <" (RTOS rot0 2 2)
">: "))
(IF (SETQ rot (GETREAL mens)) () (SETQ rot rot0))
(SETQ rot0 rot)
(IF esp0 () (SETQ esp0 1))
(SETQ mens (STRCAT "\nEspacio entre final de línea y comienzo de "
"texto <" (RTOS esp0 2 2) ">: "))
(INITGET 4)
(IF (SETQ esp (GETREAL mens)) () (SETQ esp esp0))
(SETQ esp0 esp)
(SETQ hueco0 (+ esp (* 0.9 (STRLEN tx) altx)))
(PROMPT (STRCAT "\nEspacio necesario para alojar el texto desde "
"su inserción hasta la reanudación de la línea "
(RRTOS hueco0 2 2) ))
(SETQ mens (STRCAT "\nEspacio desde inserción de texto hasta "
"reanudación de la línea <" (RTOS hueco0 2 2) ">: "))
(INITGET 4)
(IF (SETQ hueco (GETREAL mens)) () (SETQ hueco hueco0))
(IF dx0 () (SETQ dx0 0))
(SETQ mens (STRCAT "\nDesplazamiento del texto en el sentido "
"de la línea <" (RTOS dx0 2 2) ">: "))
(IF (SETQ dx (GETREAL mens)) () (SETQ dx dx0))
(SETQ dx0 dx)
(IF dy0 () (SETQ dy0 0))
(SETQ mens (STRCAT "\nDesplazamiento del texto en perpendicular "
"a la línea <" (RTOS dy0 2 2) ">: "))
(IF (SETQ dy (GETREAL mens)) () (SETQ dy dy0))
(SETQ dy0 dy)
(IF dis0 () (SETQ dis0 1))
(SETQ mens (STRCAT "\nLongitud de cada tramo de línea entre "
"textos <" (RTOS dis0 2 2) ">: "))
(INITGET 6)
(IF (SETQ dis (GETREAL mens)) () (SETQ dis dis0))
(SETQ dis0 dis)
)

(DEFUN lineat (/ path arch valt vcen x1 x2 y1 y2 pt1 pt2 pt3 pt4)
(SETQ lineal (STRCAT "*" nomtl ", ----- " tx " ----- " tx " ----- " tx " -----"))
(IF (= 0 hueco)
(SETQ hueco "")
(SETQ hueco (STRCAT ",-" (RTOS hueco 2 2))))
)
(SETQ linea2 (STRCAT "A, " (RTOS dis 2 2) ",-" (RTOS esp 2 2)
",[" tx "\"", " estil ",S=" (RTOS altx 2 2)
",R=" (RTOS rot 2 2) ",X=" (RTOS dx 2 2)
",Y=" (RTOS dy 2 2) "]" hueco ))
(IF (SETQ path (FINDFILE "$lineat.lin")) ()
(PROGN (SETQ path (STRCAT (GETVAR "dwgprefix") "$lineat.lin")))
)
(SETQ arch (OPEN path "w"))
(WRITE-LINE lineal arch) (WRITE-LINE linea2 arch)
```

```
(CLOSE arch)
(COMMAND "tipolin" "cargar" nomtl path "def" nomtl "")
(SETQ valt (GETVAR "viewsize") vcen (GETVAR "viewctr"))
(SETQ lescal (FIX (/ valt 50)))
(COMMAND "escalatl" lescal)
(COMMAND "circulo" vcen (/ valt 2.8))
(SETQ valt (/ valt 1.7))
(SETQ x1 (CAR (POLAR vcen PI valt)) x2 (CAR (POLAR vcen 0 valt)))
(SETQ y1 (CADR (POLAR vcen (- (/ PI 2)) (* valt 0.75))))
(SETQ y2 (CADR (POLAR vcen (/ PI 2) (* valt 0.75))))
(SETQ pt1 (LIST x1 y1) pt2 (LIST x1 y2) pt3 (LIST x2 y2) pt4 (LIST x2 y1) )
(COMMAND "linea" pt1 pt2 pt3 pt4 "c")
)

(DEFUN def_lineat (/ mens path path0 arch op)
  (SETQ path0 (FINDFILE "acad.lin"))
  (SETQ mens "\nNombre de archivo de tipos de línea (sin extensión) <acad>: ")
  (IF (= "" (SETQ path (GETSTRING mens)))
    (SETQ path path0)
    (SETQ path (STRCAT (GETVAR "dwgprefix") path ".lin")))
  )
  (SETQ arch (OPEN path "a"))
  (WRITE-LINE lineal arch) (WRITE-LINE linea2 arch)
  (CLOSE arch)
  (INITGET "Sí No")
  (SETQ mens (STRCAT "¿Definir tipo de línea " nomtl " como actual? S/N <S>: "))
  (IF (SETQ op (GETKEYWORD mens)) () (SETQ op "Sí") )
  (IF (= op "No") (COMMAND "tipolin" "def" tl0 "")) )
)

(DEFUN c:lineat (/ tl0 exp0 op nomtl tx estil altx rot esp hueco dx dy
  dis lineal linea2 lescal)
  (SETQ error0 *error* *error* err_lineat)
  (SETVAR "cmdecho" 0)
  (SETQ exp0 (GETVAR "expert"))(SETVAR "expert" 3)
  (SETQ tl0 (GETVAR "celtype"))
  (datos_lineat)
  (lineat)
  (INITGET "Sí No")
  (IF (SETQ op (GETKEYWORD "\n¿Correcto S/N <S>: "))()(SETQ op "Sí"))
  (WHILE (= op "No")
    (COMMAND "h")(COMMAND "h")
    (datos_lineat)
    (lineat)
    (INITGET "Sí No")
    (IF (SETQ op (GETKEYWORD "\n¿Correcto S/N <S>: "))()(SETQ op "Sí"))
  )
  (COMMAND "h") (COMMAND "h")
  (def_lineat)
  (PROMPT (STRCAT "\nValor actual de ESCALATL establecido en " (RTOS lescal 2 0)
    "."))
  (SETVAR "expert" exp0) (SETVAR "cmdecho" 1)
  (PRIN1)
)

(DEFUN err_lineat (mens)
  (SETQ *error* error0)
  (IF (= mens "quitar / salir abandonar")
    (PRIN1)
    (PRINC (STRCAT "\nError: " mens " ")))
)
```

```
(SETVAR "expert" exp0) (SETVAR "cmdecho" 1)
(PRIN1)
)

(PROMPT "Nuevo comando LINEAT definido.")(PRIN1)
```

EJERCICIOS RESUELTOS DEL MÓDULO ONCE

EJERCICIO I

```
(DEFUN Datos_Perfil ()
  (INITGET 7)
  (SETQ h (GETREAL "Altura total del perfil: ")) (TERPRI)
  (INITGET 7)
  (SETQ e (GETREAL "Altura del alma: "))
  (WHILE (<= h e)
    (PROMPT "La altura del alma debe ser menor que la del perfil.")
    (TERPRI)
    (INITGET 7)
    (SETQ e (GETREAL "Altura del alma: "))(TERPRI)
  )
  (INITGET 7)
  (SETQ l (GETREAL "Anchura del ala: ")) (TERPRI)
  (INITGET 7)
  (SETQ a (GETREAL "Espesor del alma: "))
  (WHILE (<= l a)
    (PROMPT "El espesor del alma debe ser menor que la anchura del
      ala.")
    (TERPRI)
    (INITGET 7)
    (SETQ a (GETREAL "Espesor del alma: "))(TERPRI)
  )
  (INITGET 1)
  (SETQ pb (GETPOINT "Punto de base para el trazado: ")) (TERPRI)
)

(DEFUN Perfil (/ x1 x2 x3 x4 y1 y2 y3 y4 p1 p2 p3 p4 p5 p6 p7 p8 p9
  p10 p11)
  (SETQ x1 (CAR pb) y1 (CADR pb))
  (SETQ x2 (+ x1 (/ (- l a)2)))
  (SETQ x3 (+ x2 a))
  (SETQ x4 (+ x1 l))
  (SETQ y2 (+ y1 (/ (- h e)2)))
  (SETQ y3 (+ y2 e) y4 (+ y1 h))
  (SETQ p1 (LIST x4 y1))
  (SETQ p2 (LIST x4 y2))
  (SETQ p3 (LIST x3 y2))
  (SETQ p4 (LIST x3 y3))
  (SETQ p5 (LIST x4 y3))
  (SETQ p6 (LIST x4 y4))
  (SETQ p7 (LIST x1 y4))
  (SETQ p8 (LIST x1 y3))
  (SETQ p9 (LIST x2 y3))
  (SETQ p10 (LIST x2 y2))
  (SETQ p11 (LIST x1 y2))
  (COMMAND "_pline" pb p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11 "_c")
)
```

```
(DEFUN C:Perfil (/ h e l a pb refnt0)
  (SETVAR "cmdecho" 0)
  (Datos_Perfil)
  (SETQ refnt0 (GETVAR "osmode"))(SETVAR "osmode" 0)
  (COMMAND "_undo" "_begin")
  (Perfil)
  (COMMAND "_undo" "_end")
  (SETVAR "osmode" refnt0)
  (SETVAR "cmdecho" 1)(PRIN1)
)

(PROMPT "Nuevo comando PERFIL definido.")(PRIN1)
```

EJERCICIO II

```
(DEFUN tododisco (/ mens ulcam listbl cam camt)
  (IF cam0 () (SETQ cam0 "c:"))
  (SETQ mens (STRCAT "Unidad y directorio para almacenar los bloques
    <" cam0 ">: "))
  (PROMPT mens)
  (IF (= "" (SETQ cam (READ-LINE)))
    (SETQ cam cam0)
  )
  (SETQ ulcam (SUBSTR cam (STRLEN cam) 1))
  (IF (OR (= "\\\" ulcam)(= "/" ulcam)(= ":" ulcam))
    ()(SETQ camt (STRCAT cam "\\\"))
  )
  (IF (SETQ listbl (TBLNEXT "block" T)) (bloque_tododisco))
  (WHILE (SETQ listbl (TBLNEXT "block"))
    (bloque_tododisco)
  )
  (SETQ cam0 cam)
)

(DEFUN bloque_tododisco (/ nom narch camrch)
  (SETQ nom (CDR (ASSOC 2 listbl)))
  (SETQ narch (SUBSTR nom 1 8))
  (SETQ camrch (STRCAT camt narch))
  (IF (FINDFILE (STRCAT camrch ".dwg"))
    (redef_tododisco)
    (PROGN (COMMAND "_wblock" camrch nom)
      (IF (FINDFILE (STRCAT camrch ".dwg")) ()
        (PROGN (PROMPT "\nNombre de directorio no válido.\n")(QUIT)))
      (SETQ totbl (CONS nom totbl))
    )
  )
)

(DEFUN redef_tododisco (/ op)
  (INITGET "Sí No")
  (IF (SETQ op (GETKEYWORD
    (STRCAT "Archivo " narch " ya existe.Desea sustituirlo <N>: ")))
    ()(SETQ op "No")
  )
  (IF (= op "Sí")
    (PROGN (COMMAND "_wblock" camrch "_y" nom)
      (SETQ totbl (CONS nom totbl))
    )
  )
)
```

```
)

(DEFUN lista_tododisco (/ n nbl)
  (SETQ nbl (LENGTH totbl))
  (IF (> nbl 0)
    (PROGN
      (TEXTPAGE)
      (PROMPT "\n\n")
      (PROMPT " LISTADO DE BLOQUES\n")
      (PROMPT " ENVIADOS A DISCO\n")
      (PROMPT "-----\n")(TERPRI)
      (SETQ n (- nbl 1))
      (REPEAT nbl
        (PROMPT (NTH n totbl))(TERPRI)
        (SETQ n (- n 1))
      )
    )
  )
)

(DEFUN C:TodoDisco (/ totbl error0)
  (SETVAR "cmdecho" 0)
  (SETQ error0 *error* *error* err_tododisco)
  (tododisco)
  (IF totbl (lista_tododisco)(PROMPT "\nNingún bloque enviado a
                                disco."))

  (SETQ *error* error0)
  (SETVAR "cmdecho" 1)(PRIN1)
)

(DEFUN err_tododisco (mens)
  (SETQ *error* error0)
  (IF (= mens "quitar / salir abandonar")
    (PRIN1)
    (PRINC (STRCAT "\nError: " mens " ")))
  )
  (SETVAR "cmdecho" 1)(PRIN1)
)
```

EJERCICIO III

```
(DEFUN C:RoscaSec (/ mens opcj oprs pticj ang diacj prfcj pt1 pt2 pt3
                  pt4 pt5 prfgj ptigj cap0 refnt0)
  (SETVAR "cmdecho" 0)
  (Datos_RoscaSec)
  (SETQ refnt0 (GETVAR "osmode")) (SETVAR "osmode" 0)
  (COMMAND "_undo" "_begin")
  (INITGET "Sí No")
  (IF (SETQ opcj (GETKEYWORD "¿Existe cajera Sí/<No>?: "))
    ()
    (SETQ opcj "No")
  )
  (TERPRI)
  (IF (= opcj "Sí") (Cajera) (NoCajera))
  (Agujero)
  (INITGET "Sí No")
  (IF (SETQ oprs (GETKEYWORD "¿Agujero roscado No/<Sí>: "))
    ()
    (SETQ oprs "Sí")
  )(TERPRI)
```

```
(IF (= oprs "Sí") (Rosca))
(Eje)
(COMMAND "_undo" "_end")
(SETVAR "osmode" refnt0)(SETVAR "cmdecho" 1)(PRIN1)
)

(DEFUN Datos_RoscaSec ( )
  (INITGET 1)
  (SETQ pticj (GETPOINT "Punto de inserción: ")) (TERPRI)
  (IF ang0 ( ) (SETQ ang0 0))
  (SETQ mens (STRCAT "Angulo para la inserción <" (ANGTOS ang0 0 2)
    ">: "))
  (IF (SETQ ang (GETANGLE pticj mens)) ( ) (SETQ ang ang0)) (TERPRI)
  (SETQ ang0 ang)
)

(DEFUN Cajera ( )
  (IF diacj0 ( ) (SETQ diacj0 1))
  (SETQ mens (STRCAT "Diámetro de cajera <" (RTOS diacj0 2 2) ">: "))
  (INITGET 6)
  (IF (SETQ diacj (GETREAL mens)) ( ) (SETQ diacj diacj0)) (TERPRI)
  (SETQ diacj0 diacj)
  (IF prfcj0 ( ) (SETQ prfcj0 1))
  (SETQ mens (STRCAT "Profundidad de cajera recta <" (RTOS prfcj0 2 2)
    ">: "))
  (INITGET 6)
  (IF (SETQ prfcj (GETDIST pticj mens)) ( ) (SETQ prfcj prfcj0))
  (TERPRI)
  (SETQ prfcj0 prfcj)
  (SETQ pt1 (POLAR pticj (- ang (/ PI 2)) (/ diacj 2)))
  (SETQ pt2 (POLAR pt1 ang prfcj))
  (SETQ pt3 (POLAR pt2 (+ ang (/ PI 2)) diacj))
  (SETQ pt4 (POLAR pt3 (+ ang PI) prfcj))
  (COMMAND "_line" pt1 pt2 pt3 pt4 "")
  (SETQ ptigj (POLAR pticj ang prfcj))
)

(DEFUN NoCajera ( )
  (SETQ ptigj pticj prfcj 0)
)

(DEFUN Agujero ( )
  (IF prfgj0 ( ) (SETQ prfgj0 1))
  (SETQ mens (STRCAT "Profundidad del agujero <" (RTOS prfgj0 2 2) ">:
    ""))
  (INITGET 6)
  (IF (SETQ prfgj (GETDIST ptigj mens)) ( ) (SETQ prfgj prfgj0))
  (TERPRI)
  (SETQ prfgj0 prfgj)
  (IF (= opcj "Sí") (Diam_Cajera) (Diam_NoCajera))
  (SETQ diagj0 diagj)
  (SETQ pt1 (POLAR ptigj (- ang (/ PI 2)) (/ diagj 2)))
  (SETQ pt2 (POLAR pt1 ang prfgj))
  (SETQ pt3 (POLAR pt2 (+ ang (/ PI 2)) (/ diagj 2)))
  (SETQ pt3 (POLAR pt3 ang (/ diagj 2)))
  (SETQ pt4 (POLAR pt2 (+ ang (/ PI 2)) diagj))
  (SETQ pt5 (POLAR pt4 (+ ang PI) prfgj))
  (COMMAND "_line" pt1 pt2 pt4 pt5 "")
  (COMMAND "_line" pt2 pt3 pt4 "")
)
```

```
(DEFUN Diam_Cajera ()
  (IF diagj0 () (SETQ diagj0 (/ diacj 2)))
  (IF (< diagj0 diacj) () (SETQ diagj0 (/ diacj 2)))
  (SETQ mens (STRCAT "Diámetro de agujero <" (RTOS diagj0 2 2) ">: "))
  (INITGET 6)
  (IF (SETQ diagj (GETREAL mens)) () (SETQ diagj diagj0)) (TERPRI)
  (WHILE (>= diagj diacj)
    (PROMPT "Diámetro de agujero debe ser menor que el de cajera.\n")
    (INITGET 6)
    (IF (SETQ diagj (GETREAL mens)) () (SETQ diagj diagj0)) (TERPRI)
  )
)

(DEFUN Diam_NoCajera ()
  (IF diagj0 () (SETQ diagj0 1))
  (SETQ mens (STRCAT "Diámetro de agujero <" (RTOS diagj0 2 2) ">: "))
  (INITGET 6)
  (IF (SETQ diagj (GETREAL mens)) () (SETQ diagj diagj0)) (TERPRI)
)

(DEFUN Rosca (/ prfrs caprs)
  (SETQ cap0 (GETVAR "clayer"))
  (SETQ mens (STRCAT "Profundidad del roscado <" (RTOS prfgj 2 2) ">: "
    ))
  (INITGET 6)
  (IF (SETQ prfrs (GETDIST ptigj mens)) () (SETQ prfrs prfgj))
  (TERPRI)
  (WHILE (< prfgj prfrs)
    (PROMPT (STRCAT "Profundidad del roscado no puede ser mayor de "
      (RTOS prfgj 2 2) ".")) (TERPRI)
    (INITGET 6)
    (IF (SETQ prfrs (GETDIST ptigj mens)) () (SETQ prfrs prfgj))
  )
  (IF (= "" (SETQ caprs (GETSTRING "Nombre de capa para el roscado
    <ROSCADOS>: ")))
    (SETQ caprs "ROSCADOS")
  )(TERPRI)
  (SETQ pt1 (POLAR ptigj (- ang (/ PI 2)) (* diagj 0.6)))
  (SETQ pt2 (POLAR pt1 ang prfrs))
  (SETQ pt3 (POLAR pt2 (+ ang (/ PI 2)) (* diagj 1.2)))
  (SETQ pt4 (POLAR pt3 (+ ang PI) prfrs))
  (COMMAND "_layer" "_make" caprs "")
  (COMMAND "_line" pt1 pt2 pt3 pt4 "")
)

(DEFUN Eje (/ lon lonj ptij capj ptfj)
  (SETQ lon (+ prfcj prfgj (/ diagj 2)))
  (SETQ lonj (* lon 1.25))
  (SETQ ptij (POLAR pticj (+ ang PI) (* lon 0.125)))
  (IF (= "" (SETQ capj (GETSTRING "Nombre de capa para el eje <EJES>: "
    ))))
    (SETQ capj "EJES")
  )(TERPRI)
  (SETQ ptfj (POLAR ptij ang lonj))
  (COMMAND "_layer" "_make" capj "")
  (COMMAND "_line" ptij ptfj "")
  (COMMAND "_layer" "_set" cap0 "")
)

(PROMPT "Nuevo comando ROSCASEC definido.")(PRIN1)
```


EJERCICIO IV

DCL

```
nivel:dialog {label="Generar curva de nivel";
:row {
:column {
:boxed_column {label="Fichero";
:row {
:edit_box {label="Entra&da:";edit_width=25;
key="TileEntrada";}
:button {label="E&xaminar..." ;fixed_width=true;
key="TileExaminar1";}
}
:row {
:edit_box {label="&Salida:";edit_width=25;key="TileSalida";}
:button {label="Exa&minar..." ;fixed_width=true;
key="TileExaminar2";}
}
spacer_1;
:toggle {label="Ver fic&hero de salida al terminar";
key="TileVerSalida";}
spacer_1;
:toggle {label="No generar &fichero de salida";
key="TileNoSalida";}
}
:column {
:row {
:boxed_column {label="Generar curva";
:radio_button {label="&Polilínea";key="TilePolilínea";}
:radio_button {label="Polilínea &adaptada";
key="TileAdaptada";}
:radio_button {label="Spl&ine";key="TileSpline";}
:radio_button {label="Lí&neas";key="TileLínea";}
spacer_0;
}
:boxed_column {label="Propiedades de la curva";
:popup_list {label="&Capa:";edit_width=13;key="TileCapa";}
:popup_list {label="C&olor: ";edit_width=13;
key="TileColor";}
:popup_list {label="&Tipo de línea: ";edit_width=13;
key="TileTipoLínea";}
spacer_0;
}
}
}
}
:column {
:boxed_column {label="Generar marcas";
:edit_box {label="Es&tilo:";edit_width=11;key="TileEstilo";}
:edit_box {label="A&ltura:";edit_width=11;key="TileAltura";}
:edit_box {label="Rotaci&on:";edit_width=11;
key="TileRotación";}
spacer_1;
:edit_box {label="Comen&zar en:";edit_width=11;
key="TileComenzar";}
spacer_1;
:toggle {label="No &generar marcas";key="TileNoMarca";}
}
```

```
:boxed_column {label="Separador";
:radio_button {label="Co&ma";key="TileComa";}
:radio_button {label="Espacio &blanco";key="TileEspacio";}
spacer_1;
:toggle {label="Ot&ro";key="TileOtro";}
:edit_box {label="&Carácter:";edit_width=5;edit_limit=1;
          key="TileCarácter";}
spacer_1;
}
}
}
spacer_1;
ok_cancel;
errrtile;
}
```

AutoLISP

```
(DEFUN Cuadro (/ Ind ElemCapa ElemTipo)
  (SETQ ListaCapa nil ListaTipo nil)
  (SETQ Ind (LOAD_DIALOG "c:/misdoc~1/autocad/autolisp/nivel.dcl"))
  (NEW_DIALOG "nivel" Ind)

  (IF Entrada () (SETQ Entrada ""))
  (SET_TILE "TileEntrada" Entrada)

  (IF Salida () (SETQ Salida ""))
  (SET_TILE "TileSalida" Salida)
  (IF ModeSalida () (SETQ ModeSalida 0))
  (MODE_TILE "TileSalida" ModeSalida)

  (IF ModeExaminar2 () (SETQ ModeExaminar2 0))
  (MODE_TILE "TileExaminar2" ModeExaminar2)

  (IF VerSalida () (SETQ VerSalida "0"))
  (SET_TILE "TileVerSalida" VerSalida)
  (IF ModeVerSalida () (SETQ ModeVerSalida 0))
  (MODE_TILE "TileVerSalida" ModeVerSalida)

  (IF NoSalida () (SETQ NoSalida "0"))
  (SET_TILE "TileNoSalida" NoSalida)

  (IF GenerarCurva () (SETQ GenerarCurva 2))
  (COND
    ((= GenerarCurva 1) (SET_TILE "TilePolilínea" "1"))
    ((= GenerarCurva 2) (SET_TILE "TileAdaptada" "1"))
    ((= GenerarCurva 3) (SET_TILE "TileSpline" "1"))
    ((= GenerarCurva 4) (SET_TILE "TileLínea" "1"))
  )

  (START_LIST "TileCapa")
  (SETQ ElemCapa (CDR (ASSOC 2 (TBLNEXT "layer" T))))
  (WHILE ElemCapa
    (ADD_LIST ElemCapa)
    (SETQ ListaCapa (CONS ElemCapa ListaCapa))
    (SETQ ElemCapa (CDR (ASSOC 2 (TBLNEXT "layer"))))
  )
  (SETQ ListaCapa (REVERSE ListaCapa))
  (END_LIST)
  (IF Capa () (SETQ Capa "0")))
```

```
(SET_TILE "TileCapa" Capa)

(START_LIST "TileColor")
  (ADD_LIST "PorCapa") (ADD_LIST "PorBloque")
  (SETQ Contador 0)
  (REPEAT 255
    (SETQ Contador (1+ Contador))
    (SETQ ElemColor Contador)
    (ADD_LIST (ITOA ElemColor))
  )
(END_LIST)
(IF Color () (SETQ Color "0"))
(SET_TILE "TileColor" Color)

(START_LIST "TileTipoLínea")
  (SETQ ElemTipo (CDR (ASSOC 2 (TBLNEXT "ltype" T))))
  (WHILE ElemTipo
    (ADD_LIST ElemTipo)
    (SETQ ListaTipo (CONS ElemTipo ListaTipo))
    (SETQ ElemTipo (CDR (ASSOC 2 (TBLNEXT "ltype"))))
  )
  (SETQ ListaTipo (REVERSE ListaTipo))
(END_LIST)
(IF TipoLínea () (SETQ TipoLínea "0"))
(SET_TILE "TileTipoLínea" TipoLínea)

(IF Estilo () (SETQ Estilo "STANDARD"))
(SET_TILE "TileEstilo" Estilo)
(IF ModeEstilo () (SETQ ModeEstilo 0))
(MODE_TILE "TileEstilo" ModeEstilo)

(IF Altura () (SETQ Altura "1"))
(SET_TILE "TileAltura" Altura)
(IF ModeAltura () (SETQ ModeAltura 0))
(MODE_TILE "TileAltura" ModeAltura)

(IF Rotación () (SETQ Rotación "0"))
(SET_TILE "TileRotación" Rotación)
(IF ModeRotación () (SETQ ModeRotación 0))
(MODE_TILE "TileRotación" ModeRotación)

(IF Comenzar () (SETQ Comenzar "1"))
(SET_TILE "TileComenzar" Comenzar)
(IF ModeComenzar () (SETQ ModeComenzar 0))
(MODE_TILE "TileComenzar" ModeComenzar)

(IF NoMarca () (SETQ NoMarca "0"))
(SET_TILE "TileNoMarca" NoMarca)

(IF Coma () (SETQ Coma "1"))
(SET_TILE "TileComa" Coma)
(IF ModeComa () (SETQ ModeComa 0))
(MODE_TILE "TileComa" ModeComa)

(IF Espacio () (SETQ Espacio "0"))
(SET_TILE "TileEspacio" Espacio)
(IF ModeEspacio () (SETQ ModeEspacio 0))
(MODE_TILE "TileEspacio" ModeEspacio)

(IF Otro () (SETQ Otro "0"))
(SET_TILE "TileOtro" Otro)
```

```
(IF Carácter () (SETQ Carácter ""))
(SET_TILE "TileCarácter" Carácter)
(IF ModeCarácter () (SETQ ModeCarácter 1))
(MODE_TILE "TileCarácter" ModeCarácter)

(ACTION_TILE "TileNoSalida" "(CambiarModes1)")
(ACTION_TILE "TileNoMarca" "(CambiarModes2)")
(ACTION_TILE "TileOtro" "(CambiarModes3)")

(ACTION_TILE "TileExaminar1" "(SETQ Examinar 1) (BotónExaminar)")
(ACTION_TILE "TileExaminar2" "(SETQ Examinar 2) (BotónExaminar)")

(ACTION_TILE "accept" "(GuardaVariables) (ControlDCL)
                      (IF ErrorDCL () (DONE_DIALOG 1))"
)
(ACTION_TILE "cancel" "(DONE_DIALOG 0)")

(COND
  ((= (START_DIALOG) 1) (Aceptar))
  ((= (START_DIALOG) 0) ())
)
)

(DEFUN CambiarModes1 ()
  (IF (= (GET_TILE "TileNoSalida") "1")
    (PROGN
      (MODE_TILE "TileSalida" 1)
      (MODE_TILE "TileExaminar2" 1)
      (MODE_TILE "TileVerSalida" 1)
    )
    (PROGN
      (MODE_TILE "TileSalida" 0)
      (MODE_TILE "TileExaminar2" 0)
      (MODE_TILE "TileVerSalida" 0)
    )
  )
)

(DEFUN CambiarModes2 ()
  (IF (= (GET_TILE "TileNoMarca") "1")
    (PROGN
      (MODE_TILE "TileEstilo" 1)
      (MODE_TILE "TileAltura" 1)
      (MODE_TILE "TileRotación" 1)
      (MODE_TILE "TileComenzar" 1)
    )
    (PROGN
      (MODE_TILE "TileEstilo" 0)
      (MODE_TILE "TileAltura" 0)
      (MODE_TILE "TileRotación" 0)
      (MODE_TILE "TileComenzar" 0)
    )
  )
)

(DEFUN CambiarModes3 ()
  (IF (= (GET_TILE "TileOtro") "1")
    (PROGN
      (MODE_TILE "TileComa" 1)
      (MODE_TILE "TileEspacio" 1)
    )
  )
)
```

```
(MODE_TILE "TileCarácter" 0)

)
(PROGN
  (MODE_TILE "TileComa" 0)
  (MODE_TILE "TileEspacio" 0)
  (MODE_TILE "TileCarácter" 1)
)
)
)

(DEFUN GuardaVariables ()
  (IF (= (GET_TILE "TileNoSalida") "1")
    (PROGN
      (SETQ ModeSalida 1)
      (SETQ ModeExaminar2 1)
      (SETQ ModeVerSalida 1)
    )
    (PROGN
      (SETQ ModeSalida 0)
      (SETQ ModeExaminar2 0)
      (SETQ ModeVerSalida 0)
    )
  )
  (IF (= (GET_TILE "TileNoMarca") "1")
    (PROGN
      (SETQ ModeEstilo 1)
      (SETQ ModeAltura 1)
      (SETQ ModeRotación 1)
      (SETQ ModeComenzar 1)
    )
    (PROGN
      (SETQ ModeEstilo 0)
      (SETQ ModeAltura 0)
      (SETQ ModeRotación 0)
      (SETQ ModeComenzar 0)
    )
  )
  (IF (= (GET_TILE "TileOtro") "1")
    (PROGN
      (SETQ ModeComa 1)
      (SETQ ModeEspacio 1)
      (SETQ ModeCarácter 0)
    )
    (PROGN
      (SETQ ModeComa 0)
      (SETQ ModeEspacio 0)
      (SETQ ModeCarácter 1)
    )
  )
  (SETQ Entrada (GET_TILE "TileEntrada"))
  (SETQ Salida (GET_TILE "TileSalida"))
  (IF (NOT Archiv1) (SETQ Archiv1 Entrada))
  (IF (NOT Archiv2) (SETQ Archiv2 Salida))
  (SETQ NoSalida (GET_TILE "TileNoSalida"))
  (SETQ VerSalida (GET_TILE "TileVerSalida"))
  (IF (= (GET_TILE "TilePolilínea") "1") (SETQ GenerarCurva 1))
  (IF (= (GET_TILE "TileAdaptada") "1") (SETQ GenerarCurva 2))
  (IF (= (GET_TILE "TileSpline") "1") (SETQ GenerarCurva 3))
  (IF (= (GET_TILE "TileLínea") "1") (SETQ GenerarCurva 4))
  (SETQ Capa (GET_TILE "TileCapa"))
```

```
(SETQ Color (GET_TILE "TileColor"))
(SETQ TipoLínea (GET_TILE "TileTipoLínea"))
(SETQ Estilo (GET_TILE "TileEstilo"))
(SETQ Altura (GET_TILE "TileAltura"))
(SETQ Rotación (GET_TILE "TileRotación"))
(SETQ Comenzar (GET_TILE "TileComenzar"))
(SETQ NoMarca (GET_TILE "TileNoMarca"))
(IF (= (SETQ Coma (GET_TILE "TileComa")) "1") (SETQ Separador ","))
(IF (= (SETQ Espacio (GET_TILE "TileEspacio")) "1")
  (SETQ Separador " "))
)
(SETQ Carácter (GET_TILE "TileCarácter"))
(IF (= (SETQ Otro (GET_TILE "TileOtro")) "1")
  (SETQ Separador Carácter)
)
)

(DEFUN BotónExaminar (/ Camino)
  (IF Camino () (SETQ Camino "c:\\"))
  (IF (= Examinar 1)
    (PROGN
      (SETQ Archivo1 (GETFILED "Abrir fichero de entrada"
        Camino "txt;dat;* " 0)
      )
      (IF (NOT Archivo1) () (SET_TILE "TileEntrada" Archivo1))
    )
    (PROGN
      (SETQ Archivo2 (GETFILED "Abrir o crear fichero de salida"
        Camino "txt;dat;* " 1)
      )
      (IF (NOT Archivo2) () (SET_TILE "TileSalida" Archivo2))
    )
  )
)

(DEFUN Aceptar (/ Cadena Coordinada Arch1 LongitudCadena Arch2
  CarácterASCII)
  (PROMPT "\nCalculando..." (PRIN1))
  (SETQ ComenzarG Comenzar)
  (SETQ Cadena T)
  (SETQ Coordinada "")
  (SETQ Conjunto (SSADD))
  (SETQ Arch1 (OPEN Archivo1 "r"))
  (IF (= NoSalida "0") (SETQ Arch2 (OPEN Archivo2 "w")))
  (WHILE (SETQ Cadena (READ-LINE Arch1))
    (SETQ Contador 1)
    (SETQ LongitudCadena (STRLEN Cadena))
    (REPEAT (1+ LongitudCadena)
      (IF (/= (SETQ CarácterASCII (SUBSTR Cadena Contador 1))
        Separador)
        (PROGN
          (IF (= CarácterASCII " ")
            (PROGN
              (COMMAND "_point" Coordinada)
              (SETQ Conjunto (SSADD (ENTLAST) Conjunto))
              (IF (= NoSalida "0")
                (PROGN
                  (SETQ AñadirCadena (STRCAT Comenzar ".- " Coordinada))
                  (WRITE-LINE AñadirCadena Arch2)
                  (SETQ Comenzar (ITOA (1+ (ATOI Comenzar)))))
                )
            )
          )
        )
      )
    )
  )
)
```

```
        )
        (SETQ Coordenada "")
    )
    (SETQ Coordenada (STRCAT Coordenada CarácterASCII))
)
)
)
(SETQ Coordenada (STRCAT Coordenada ","))
)
(SETQ Contador (1+ Contador))
)
)
(CLOSE Arch1)
(IF (= NoSalida "0") (CLOSE Arch2))
(Marcas)
(Dibujo)
(Curva)
)

(DEFUN Marcas (/ CoordenadaMarca CoordenadaXMarca CoordenadaYMarca)
  (SETQ Comenzar ComenzarG)
  (IF (= NoMarca "0")
    (PROGN
      (SETQ Contador 0)
      (SETQ NúmeroMarcas (SLENGTH Conjunto))
      (REPEAT NúmeroMarcas
        (SETQ CoordenadaMarca (ASSOC 10 (ENTGET (SSNAME Conjunto
          Contador))))
        (SETQ CoordenadaXMarca (CADR CoordenadaMarca))
        (SETQ CoordenadaXMarca (- CoordenadaXMarca 10))
        (SETQ CoordenadaYMarca (CADDR CoordenadaMarca))
        (SETQ CoordenadaYMarca (- CoordenadaYMarca 10))
        (SETQ CoordenadaMarca (LIST CoordenadaXMarca CoordenadaYMarca
          (CADDR CoordenadaMarca)))
        (COMMAND "_.text" "_s" Estilo CoordenadaMarca Altura Rotación
          Comenzar)
        (SETQ Comenzar (ITOA (1+ (ATOI Comenzar))))
        (SETQ Contador (1+ Contador))
      )
    )
  )
)

(DEFUN Dibujo (/ CapaDibujo TipoDibujo ColorDibujo CoordenadaMarca1
  PuntoDibujo1 CoordenadaMarca2 PuntoDibujo2 ListaLínea)
  (SETQ Conjunto2 (SSADD))
  (SETQ CapaDibujo (NTH (ATOI Capa) ListaCapa))
  (SETQ TipoDibujo (NTH (ATOI TipoLínea) ListaTipo))
  (IF (> (ATOI Color) 1)
    (SETQ ColorDibujo (1- (ATOI Color)))
    (PROGN
      (IF (= Color "0") (SETQ ColorDibujo 256))
      (IF (= Color "1") (SETQ ColorDibujo 0))
    )
  )
  (SETQ Contador 0)
  (SETQ NúmeroMarcas (SLENGTH Conjunto))
  (SETQ CoordenadaMarca1 (ASSOC 10 (ENTGET (SSNAME Conjunto
    Contador))))
  (SETQ PuntoDibujo1 (LIST (CADR CoordenadaMarca1) (CADDR
    CoordenadaMarca1) (CADDR CoordenadaMarca1)))
  (REPEAT (1- NúmeroMarcas)
```

```
(SETQ Contador (1+ Contador))
(SETQ CoordenadaMarca2 (ASSOC 10 (ENTGET (SSNAME Conjunto
    Contador))))
(SETQ PuntoDibujo2 (LIST (CADR CoordenadaMarca2) (CADDR
    CoordenadaMarca2) (CADDRD CoordenadaMarca2)))
(SETQ ListaLínea (LIST
    (CONS 0 "LINE")
    (CONS 8 CapaDibujo)
    (CONS 6 TipoDibujo)
    (CONS 62 ColorDibujo)
    (CONS 10 PuntoDibujo1)
    (CONS 11 PuntoDibujo2)
    )
)
(ENTMAKE ListaLínea)
(SETQ Conjunto2 (SSADD (ENTLAST) Conjunto2))
(SETQ PuntoDibujo1 PuntoDibujo2)
)
)

(DEFUN Curva (/ NúmeroLíneas Línea1 Línea2)
(SETQ Contador 0)
(IF (OR (= GenerarCurva 1) (= GenerarCurva 2) (= GenerarCurva 3))
    (PROGN
        (SETQ NúmeroLíneas (SSLENGTH Conjunto2))
        (SETQ Línea1 (SSNAME Conjunto2 Contador))
        (COMMAND "_.pedit" Línea1 "_y" "_x")
        (SETQ Línea1 (ENTLAST))
        (REPEAT (1- NúmeroLíneas)
            (SETQ Contador (1+ Contador))
            (SETQ Línea2 (SSNAME Conjunto2 Contador))
            (COMMAND "_.pedit" Línea1 "_j" Línea2 "" "_x")
        )
    )
)
(IF (= GenerarCurva 2)
    (COMMAND "_.pedit" (ENTLAST) "_f" "_x")
)
(IF (= GenerarCurva 3)
    (COMMAND "_.pedit" (ENTLAST) "_s" "_x")
)
(IF (= NoMarca "1")
    (PROGN
        (SETQ Contador 0)
        (REPEAT NúmeroMarcas
            (ENTDEL (SSNAME Conjunto Contador))
            (SETQ Contador (1+ Contador))
        )
    )
)
(IF (AND (= NoSalida "0") (= VerSalida "1"))
    (STARTAPP "NotePad" Archivo2)
)
)

(DEFUN ControlDCL ()
(SETQ ErrorDCL nil)
(IF (NOT (FINDFILE Entrada))
    (PROGN
        (SETQ ErrorDCL T)
        (SET_TILE "error" "El fichero de entrada no existe o no ha sido
```



```
        especificado.")
    (MODE_TILE "TileEntrada" 2)
)
)
(IF (AND (= NoSalida "0") (= Salida ""))
    (PROGN
        (SETQ ErrorDCL T)
        (SET_TILE "error" "El fichero de salida no ha sido
            especificado.")
        (MODE_TILE "TileSalida" 2)
    )
)
(IF (AND (= NoMarca "0") (NOT (TBLSEARCH "style" Estilo)))
    (PROGN
        (SETQ ErrorDCL T)
        (SET_TILE "error" "El estilo de texto no existe o no ha sido
            especificado.")
        (MODE_TILE "TileEstilo" 2)
    )
)
(IF (AND (= NoMarca "0") (= Altura ""))
    (PROGN
        (SETQ ErrorDCL T)
        (SET_TILE "error" "La altura para la marca de texto no ha sido
            especificada.")
        (MODE_TILE "TileAltura" 2)
    )
)
(IF (AND (= NoMarca "0") (= Rotación ""))
    (PROGN
        (SETQ ErrorDCL T)
        (SET_TILE "error" "La rotación para la marca de texto no ha sido
            especificada.")
        (MODE_TILE "TileRotación" 2)
    )
)
(IF (AND (= NoMarca "0") (= Comenzar ""))
    (PROGN
        (SETQ ErrorDCL T)
        (SET_TILE "error" "El número para la primera marca de texto no
            ha sido especificado.")
        (MODE_TILE "TileComenzar" 2)
    )
)
(IF (AND (= Otro "1") (= Carácter ""))
    (PROGN
        (SETQ ErrorDCL T)
        (SET_TILE "error" "El carácter separador no ha sido
            especificado.")
        (MODE_TILE "TileCarácter" 2)
    )
)
)

(DEFUN ErrorLSP (Mensaje)
    (SETQ *error* Errores)
    (PRINC (STRCAT "Error: " Mensaje "."))
    (SETVAR "pdmode" TipoPunto)
    (SETVAR "cmdecho" Eco)
    (PRIN1)
)
```

```
(DEFUN C:Nivel (/ Examinar Archivol Contador NúmeroMarcas Conjunto
                  Conjunto2 ErrorDCL Eco TipoPunto Errores)
  (SETQ Eco (GETVAR "cmdecho")) (SETVAR "cmdecho" 0)
  (SETQ TipoPunto (GETVAR "pdmode")) (SETVAR "pdmode" 3)
  (SETQ Errores *error* *error* ErrorLSP)
  (Cuadro)
  (SETQ *error* Errores)
  (SETVAR "pdmode" TipoPunto)
  (SETVAR "cmdecho" Eco)
  (PRIN1)
)
```

EJERCICIO V

DCL

```
caracol:dialog {label="Escalera de caracol";
:row {
  :boxed_column {label="Escalera";
    :edit_box {label="&Altura total:";edit_width=10;
      key="TAlturaTotal";}
    :edit_box {label="Diámetro &Exterior:";edit_width=10;
      key="TDiámetroExterior";}
    :edit_box {label="Diámetro &Interior:";edit_width=10;
      key="TDiámetroInterior";}
    :edit_box {label="Número de &vueltas:";edit_width=10;
      key="TNúmeroVueltas";}
    spacer_1;
    :toggle {label="&Hueco interior";key="THueco";}
    spacer_1;
    :popup_list {label="&Generación:";edit_width=13;
      list="Horaria\nAntihoraria";key="TGeneración";}
    spacer_1;
  }
  :boxed_column {label="Punto de inserción";
    :edit_box {label="&X:";key="TX";}
    :edit_box {label="&Y:";key="TY";}
    :edit_box {label="&Z:";key="TZ";}
    spacer_1;
    :button {label="De&signar <";fixed_width=true;
      alignment=centered;key="TDesignar";}
    spacer_1;
  }
}
:row {
  :boxed_column {label="Peldaños";
    :edit_box {label="Hue&lla máxima peldaño:";edit_width=10;
      key="THuella";}
    :edit_box {label="Altura &peldaño:";edit_width=10;
      key="TAlturaPeldaño";}
    :edit_box {label="&Distancia entre peldaños:";edit_width=10;
      key="TDistanciaPeldaños";}
    spacer_1;
  }
}
spacer_1;
ok_cancel;
errtile;
}
```

AutoLISP

```
(DEFUN Cuadro (/ Ind SD)
  (SETQ SD nil)

  (SETQ Ind (LOAD_DIALOG "c:/misdoc~1/autocad/autolisp/caracol.dcl"))
  (NEW_DIALOG "caracol" Ind)

  (IF AlturaTotal () (SETQ AlturaTotal 500))
  (SET_TILE "TAlturaTotal" (RTOS AlturaTotal))

  (IF DiámetroExterior () (SETQ DiámetroExterior 250))
  (SET_TILE "TDiámetroExterior" (RTOS DiámetroExterior))

  (IF DiámetroInterior () (SETQ DiámetroInterior 18))
  (SET_TILE "TDiámetroInterior" (RTOS DiámetroInterior))

  (IF NúmeroVueltas () (SETQ NúmeroVueltas 2))
  (SET_TILE "TNúmeroVueltas" (ITOA NúmeroVueltas))

  (IF Hueco () (SETQ Hueco "0"))
  (SET_TILE "THueco" Hueco)

  (IF Generación () (SETQ Generación "0"))
  (SET_TILE "TGeneración" Generación)

  (IF X () (SETQ X 0))
  (SET_TILE "TX" (RTOS X))

  (IF Y () (SETQ Y 0))
  (SET_TILE "TY" (RTOS Y))

  (IF Z () (SETQ Z 0))
  (SET_TILE "TZ" (RTOS Z))

  (IF Huella () (SETQ Huella 50))
  (SET_TILE "THuella" (RTOS Huella))

  (IF AlturaPeldaño () (SETQ AlturaPeldaño 4))
  (SET_TILE "TAlturaPeldaño" (RTOS AlturaPeldaño))

  (IF DistanciaPeldaños () (SETQ DistanciaPeldaños 16))
  (SET_TILE "TDistanciaPeldaños" (RTOS DistanciaPeldaños))

  (ACTION_TILE "TDesignar" "(GuardaVariables) (DONE_DIALOG 2)")
  (ACTION_TILE "accept" "(GuardaVariables) (IF ErrorDCL ()
    (DONE_DIALOG 1))")
  (ACTION_TILE "cancel" "(DONE_DIALOG 0)")

  (SETQ SD (START_DIALOG))
  (COND
    ((= SD 2) (Designar))
    ((= SD 1) (Aceptar))
    ((= SD 3) ())
  )
)

(DEFUN Designar ()
```

```
(SETQ PuntoInserción (GETPOINT "\nPunto centro inferior de
    inserción: "))
(SETQ X (CAR PuntoInserción))
(SETQ Y (CADR PuntoInserción))
(SETQ Z (CADDR PuntoInserción))
(Cuadro)
)

(DEFUN GuardaVariables ()
  (ControlDCL)
  (SETQ      AlturaTotal      (ATOF (GET_TILE "TAlturaTotal"))
           DiámetroExterior (ATOF (GET_TILE "TDiámetroExterior"))
           DiámetroInterior (ATOF (GET_TILE "TDiámetroInterior"))
           NúmeroVueltas    (ATOI (GET_TILE "TNúmeroVueltas"))
           X                 (ATOF (GET_TILE "TX"))
           Y                 (ATOF (GET_TILE "TY"))
           Z                 (ATOF (GET_TILE "TZ"))
           PuntoInserción   (LIST X Y Z)
           Huella            (ATOF (GET_TILE "THuella"))
           AlturaPeldaño     (ATOF (GET_TILE "TAlturaPeldaño"))
           DistanciaPeldaños (ATOF (GET_TILE "TDistanciaPeldaños"))
           Generación        (GET_TILE "TGeneración")
           Hueco             (GET_TILE "THueco"))
  )
)

(DEFUN Aceptar (/ PuntoInsBloque RadioExterior PuntoBloq1 PuntoBloq2
  PuntoBloq3 PuntoBloq4 PuntoBloq5 ConjuntoHueco
  ConjuntoBloque PuntoCorte1 PuntoCorte2 ListaArco
  Ángulo1 Ángulo2 PuntoCorte3 PuntoCorte4 PuntoCorte5
  PuntoCorte6 PuntoCorte7 ConjuntoBloque2
  ConjuntoBloque3 PuntoBloq6 NombreBloque
  DiámetroInteriorG ConjuntoCurva2 CentroCírculo)
  (DibujoCurva)

  (IF (= Hueco "0") (COMMAND "_cylinder" PuntoInserción
    RadioInterior AlturaTotal))

  (SETQ ConjuntoBloque (SSADD))
  (SETQ PuntoInsBloque '(0 0 0))
  (SETQ RadioExterior (/ DiámetroExterior 2))
  (COMMAND "_circle" PuntoInsBloque RadioInterior)
  (COMMAND "_circle" PuntoInsBloque RadioExterior)
  (SETQ PuntoBloq1 (POLAR PuntoInsBloque PI RadioExterior))
  (SETQ PuntoBloq2 (POLAR PuntoBloq1 (/ PI 2) (/ Huella 2)))
  (SETQ PuntoBloq3 (POLAR PuntoBloq1 (/ (* 3 PI) 4) (/ Huella 2)))
  (COMMAND "_circle" PuntoBloq1 (DISTANCE PuntoBloq1 PuntoBloq2))
  (SETQ PuntoCorte1 (POLAR PuntoInsBloque 0 RadioExterior))
  (SETQ PuntoCorte2 (POLAR PuntoBloq1 PI (DISTANCE PuntoBloq1
    PuntoBloq2)))
  (COMMAND "_trim" PuntoCorte2 "" PuntoCorte1 "")
  (COMMAND "_erase" "_l" "")
  (SETQ ConjuntoBloque (SSADD (ENTLAST) ConjuntoBloque))
  (SETQ ListaArco (ENTGET (SSNAME ConjuntoBloque 0)))
  (SETQ Ángulo1 (CDR (ASSOC 50 ListaArco)))
  (SETQ Ángulo2 (CDR (ASSOC 51 ListaArco)))
  (SETQ PuntoBloq4 (POLAR PuntoInsBloque Ángulo1 RadioExterior))
  (SETQ PuntoBloq5 (POLAR PuntoInsBloque Ángulo2 RadioExterior))
  (COMMAND "_line" PuntoInsBloque PuntoBloq4 "")
  (SETQ ConjuntoBloque (SSADD (ENTLAST) ConjuntoBloque))
  (COMMAND "_line" PuntoInsBloque PuntoBloq5 ""))
```

```
(SETQ ConjuntoBloque (SSADD (ENTLAST) ConjuntoBloque))
(SETQ PuntoCorte3 (POLAR PuntoInsBloque Ángulo1 (/ (DISTANCE
  PuntoInsBloque PuntoBloq4) 2)))
(SETQ PuntoCorte4 (POLAR PuntoInsBloque Ángulo2 (/ (DISTANCE
  PuntoInsBloque PuntoBloq5) 2)))
(SETQ PuntoCorte5 (POLAR PuntoInsBloque 0 RadioInterior))
(SETQ PuntoCorte6 (POLAR PuntoInsBloque Ángulo1 (/ RadioInterior 2)))
(SETQ PuntoCorte7 (POLAR PuntoInsBloque Ángulo2 (/ RadioInterior
  2)))
(COMMAND "_pan" PuntoInsBloque (GETVAR "viewctr"))
(COMMAND "_zoom" 3)
(COMMAND "_trim" PuntoCorte3 PuntoCorte4 PuntoCorte5 " "
  PuntoCorte6 PuntoCorte7 PuntoCorte5 " ")
(SETQ ConjuntoBloque2 (SSGET "_p"))
(SETQ ConjuntoBloque2 (SSADD (SSNAME ConjuntoBloque 0)
  ConjuntoBloque2))
(SETQ PuntoInsBloque (POLAR PuntoInsBloque PI RadioInterior))
(COMMAND "_zoom" "_p")
(COMMAND "_region" ConjuntoBloque2 " ")
(COMMAND "_extrude" "_l" " " AlturaPeldaño "0")
(SETQ ConjuntoBloque3 (SSGET "_l"))
(SETQ PuntoBloq6 (POLAR PuntoBloq1 0 6))
(COMMAND "_cylinder" PuntoBloq6 4 (+ 100 AlturaPeldaño))
(SETQ ConjuntoBloque3 (SSADD (ENTLAST) ConjuntoBloque3))
(COMMAND "_rotate" ConjuntoBloque3 " " PuntoInsBloque "-90")
(SETQ NombreBloque (SUBSTR (RTOS (GETVAR "date")) 9 4))
(SETQ NombreBloque (STRCAT "$BloqueCaracol" NombreBloque "$"))
(COMMAND "_block" NombreBloque PuntoInsBloque ConjuntoBloque3 " ")
(COMMAND "_measure" ConjuntoCurva "_b" Nombrebloque "_y"
  DistanciaPeldaños)
(COMMAND "_erase" ConjuntoCurva " ")
(SETQ DiámetroInteriorG DiámetroInterior DiámetroInterior (* (+
  (DISTANCE PuntoBloq6 PuntoInsBloque) RadioInterior) 2))
(SETQ PuntoInserción (LIST (CAR PuntoInserción) (CADR
  PuntoInserción) (+ CoordZ0 100 AlturaPeldaño)))
(DibujoCurva)
(SETQ ConjuntoCurva2 (SSGET "_l"))
(SETQ DiámetroInterior DiámetroInteriorG)
(IF (/= "Generación" 0)
  (SETQ CentroCírculo (LIST (- (CAR PuntoInserción) RadioInterior)
    (CADR PuntoInserción) (CADDR PuntoInserción)))
  (SETQ CentroCírculo (LIST (+ (CAR PuntoInserción) RadioInterior)
    (CADR PuntoInserción) (CADDR PuntoInserción))))
)
(COMMAND "_ucs" "_o" CentroCírculo)
(COMMAND "_ucs" "_x" "90")
(COMMAND "_circle" "0,0,0" 6)
(COMMAND "_extrude" "_l" " " "_p" ConjuntoCurva2)
)

(DEFUN DibujoCurva (/ Contador)
  (SETQ Paso (/ AlturaTotal NúmeroVueltas))
  (SETQ Ángulo (/ (* 2 PI) 32))
  (IF (= Generación "0") (SETQ Ángulo (* Ángulo -1)))
  (SETQ IncrementoZ (/ Paso 32))
  (SETQ RadioInterior (/ DiámetroInterior 2))
  (IF (= Generación "0")
    (SETQ Punto0 (POLAR PuntoInserción PI RadioInterior))
    (SETQ Punto0 (POLAR PuntoInserción 0 RadioInterior)))
  )
  (SETQ CoordZ0 (CADDR Punto0))
```

```
(COMMAND "_3dpoly" Punto0)
(SETQ Contador 1)
(REPEAT (* NúmeroVueltas 32)
  (SETQ CoordZ (+ CoordZ0 (* IncrementoZ Contador)))
  (IF (= Generación "0")
    (SETQ CoordXY (POLAR PuntoInserción (+ (* Ángulo Contador) PI)
RadioInterior))
    (SETQ CoordXY (POLAR PuntoInserción (* Ángulo Contador)
RadioInterior))
  )
  (SETQ Punto (LIST (CAR CoordXY) (CADR CoordXY) CoordZ))
  (COMMAND Punto)
  (SETQ Contador (1+ Contador))
)
(COMMAND)
(SETQ ConjuntoCurva (SSADD))
(SETQ ConjuntoCurva (SSADD (ENTLAST) ConjuntoCurva))
)

(DEFUN ControlDCL ()
  (SETQ ErrorDCL nil)
  (IF (<= (ATOF (GET_TILE "TAlturaTotal"))) 0)
  (PROGN
    (SETQ ErrorDCL T)
    (SET_TILE "error" "Falta la altura de la escalera, es cero o
negativa.")
    (MODE_TILE "TAlturaTotal" 2)
  )
)
(
  (IF (<= (ATOF (GET_TILE "TDiámetroExterior"))) 0)
  (PROGN
    (SETQ ErrorDCL T)
    (SET_TILE "error" "Falta el diámetro exterior de la escalera,
es cero o negativo.")
    (MODE_TILE "TDiámetroExterior" 2)
  )
)
(
  (IF (<= (ATOF (GET_TILE "TDiámetroInterior"))) 0)
  (PROGN
    (SETQ ErrorDCL T)
    (SET_TILE "error" "Falta el diámetro interior de la escalera,
es cero o negativo.")
    (MODE_TILE "TDiámetroInterior" 2)
  )
)
(
  (IF (<= (ATOF (GET_TILE "TNúmeroVueltas"))) 0)
  (PROGN
    (SETQ ErrorDCL T)
    (SET_TILE "error" "Falta el número de vueltas de la escalera
o es cero.")
    (MODE_TILE "TNúmeroVueltas" 2)
  )
)
(
  (IF (WCMATCH (GET_TILE "TNúmeroVueltas") ".*"))
  (PROGN
    (SETQ ErrorDCL T)
    (SET_TILE "error" "El número de vueltas ha de ser un valor
entero y positivo.")
    (MODE_TILE "TNúmeroVueltas" 2)
  )
)
)
```

```
(IF (= (GET_TILE "TX") "" )
  (PROGN
    (SETQ ErrorDCL T)
    (SET_TILE "error" "Falta la coordenada X del punto de
      inserción.")
    (MODE_TILE "TX" 2)
  )
)
(IF (= (GET_TILE "TY") "" )
  (PROGN
    (SETQ ErrorDCL T)
    (SET_TILE "error" "Falta la coordenada Y del punto de
      inserción.")
    (MODE_TILE "TY" 2)
  )
)
(IF (= (GET_TILE "TZ") "" )
  (PROGN
    (SETQ ErrorDCL T)
    (SET_TILE "error" "Falta la coordenada Z del punto de
      inserción.")
    (MODE_TILE "TZ" 2)
  )
)
(IF (<= (ATOF (GET_TILE "THuella"))) 0)
  (PROGN
    (SETQ ErrorDCL T)
    (SET_TILE "error" "Falta el valor de la huella máxima, es
      cero o negativo.")
    (MODE_TILE "THuella" 2)
  )
)
(IF (<= (ATOF (GET_TILE "TDistanciaPeldaños"))) 0)
  (PROGN
    (SETQ ErrorDCL T)
    (SET_TILE "error" "Falta la distancia entre peldaños, es cero
      o negativa.")
    (MODE_TILE "TDistanciaPeldaños" 2)
  )
)
(IF (<= (ATOF (GET_TILE "TAlturaPeldaño"))) 0)
  (PROGN
    (SETQ ErrorDCL T)
    (SET_TILE "error" "Falta la altura de peldaño, es cero o
      negativa.")
    (MODE_TILE "TAlturaPeldaño" 2)
  )
)
(IF (> (ATOF (GET_TILE "TDiámetroInterior"))) (ATOF (GET_TILE
  "TDiámetroExterior")))
  (PROGN
    (SETQ ErrorDCL T)
    (SET_TILE "error" "El diámetro interior de la escalera es
      mayor que el diámetro exterior.")
    (MODE_TILE "TDiámetroInterior" 2)
  )
)
(IF (> (ATOF (GET_TILE "TAlturaPeldaño"))) (ATOF (GET_TILE
  "TAlturaTotal")))
  (PROGN
    (SETQ ErrorDCL T)
```

```
(SET_TILE "error" "La altura de peldaño es mayor que la
    altura total de la escalera.")
(MODE_TILE "TalturaPeldaño" 2)
)
)
(IF (> (ATOF (GET_TILE "TDistanciaPeldaños")) (ATOF (GET_TILE
    "TalturaTotal"))))
    (PROGN
        (SETQ ErrorDCL T)
        (SET_TILE "error" "La distancia entre peldaños es mayor que
            la altura total de la escalera.")
        (MODE_TILE "TDistanciaPeldaños" 2)
    )
)
)

(DEFUN ControllSP (Mensaje)
    (COMMAND "_undo" "_e")
    (SETQ *error* ErroresG)
    (SETVAR "isolines" IsoLin)
    (SETVAR "pickbox" Pick)
    (SETVAR "cmdecho" Eco)
    (PRINC (STRCAT "\nError: " Mensaje "."))
    (PRIN1)
)

(DEFUN C:Caracol (/ ErrorDCL X Y Z Paso Ángulo IncrementoZ
    RadioInterior Punto0 CoordZ CoordZ0
    PuntoInserción NúmeroVueltas CoordXY Ángulo Punto
    ConjuntoCurva)
    (SETQ Eco (GETVAR "cmdecho")) (SETVAR "cmdecho" 0)
    (SETQ ErroresG *error* *error* ControllSP)
    (SETQ IsoLin (GETVAR "isolines")) (SETVAR "isolines" 10)
    (SETQ Pick (GETVAR "pickbox")) (SETVAR "pickbox" 0)
    (COMMAND "_undo" "_be")
    (Cuadro)
    (COMMAND "_undo" "_e")
    (SETVAR "isolines" IsoLin)
    (SETVAR "pickbox" Pick)
    (SETQ *error* ErroresG)
    (SETVAR "cmdecho" Eco)
    (PRIN1)
)

(DEFUN C:CL ()
    (C:Caracol)
)

(PROMPT "\nNuevo comando CARACOL (abreviatura CL) definido.") (PRIN1)
```

EJERCICIO VI

DCL

```
acumular:dialog {label="Acumular distancias";
:row {
    :boxed_column {label="Actuar en capa";
        :popup_list {label="&Capa:";edit_width=16;key="TileCapa";}
        spacer_1;
        :radio_button {label="Capa de la lis&ta";key="TileLista";}
```



```

        :radio_button {label="Capa &actual";key="TileActual";}
        :radio_button {label="Desi&gnar por objeto";key="TileObjeto";}
    }
    :boxed_column {label="Acumular";
        :toggle {label="&Líneas";key="TileLínea";}
        :toggle {label="P&olilíneas";key="TilePolilínea";}
        :toggle {label="A&rcos";key="TileArco";}
        spacer_1;
        :toggle {label="&Todo";key="TileTodo";}
    }
}
:row {
    :boxed_column {label="Mostrar texto";
        :edit_box {label="&Estilo:";edit_width=15;key="TileEstilo";}
        :edit_box {label="A&ltura:";edit_width=15;key="TileAltura";}
        :edit_box {label="&Rotación:";edit_width=15;key="TileRotación";}
        spacer_1;
        :edit_box {label="Texto &inicial:";edit_width=15;
            key="TileTextoIni";}
        :edit_box {label="Texto &final:";edit_width=15;
            key="TileTextoFin";}
        spacer_1;
        :button {label="Prefere&ncias...";fixed_width=true;
            alignment=centered;key="TilePreferencias";}
        spacer_1;
    }
    :boxed_column {label="Inserción del texto";
        :popup_list {label="Ca&pa:";edit_width=9;list="Actuación";
            key="TileCapa2";}
        spacer_1;
        :edit_box {label="&X:";key="TileX";}
        :edit_box {label="&Y:";key="TileY";}
        :edit_box {label="&Z:";key="TileZ";}
        spacer_1;
        :button {label="&Designar <";fixed_width=true;
            alignment=centered;key="TileDesignar";}
        spacer_1;
    }
}
spacer_1;
ok_cancel;
errtile;
}

```

```

preferencias:dialog {label="Preferencias de texto";
:row{
    :boxed_column {label="Mostrar subtotales";
        :toggle {label="Subtotal de tramos &rectos";
            key="TileSubtotalesRectos";}
        spacer_1;
        :edit_box {label="Texto &inicial:";edit_width=15;
            key="TileRectosTextoIni";}
        :edit_box {label="Texto &final:";edit_width=15;
            key="TileRectosTextoFin";}
        spacer_1;
        :toggle {label="Subtotal de tramos cur&vos";
            key="TileSubtotalesCurvos";}
        spacer_1;
        :edit_box {label="Texto ini&cial:";edit_width=15;
            key="TileCurvosTextoIni";}
    }
}

```

```
:edit_box {label="Te&xto final:";edit_width=15;
           key="TileCurvosTextoFin";}
spacer_1;
}
:column {
:boxed_row {label="Mostrar fecha y hora";
            :toggle {label="Fec&ha";key="TileFecha";}
            :toggle {label="H&ora";key="TileHora";}
            }
:boxed_row {label="Mostrar autor y empresa";
            :column {
                :toggle {label="Autor &y empresa";key="TileAutorEmpresa";}
                :edit_box {label="&Autor:";edit_width=20;key="TileAutor";}
                :edit_box {label="&Empresa:";edit_width=20;
                           key="TileEmpresa";}
                :edit_box {label="&Ubicación:";edit_width=20;
                           key="TileUbicación";}
                spacer_1;
            }
            }
}
spacer_1;
ok_cancel;
errtile;
}
```

AutoLISP

```
(DEFUN Cuadro (/ ElemCapa)
  (SETQ ListaCapa nil ListaCapa2 nil)
  (SETQ SD nil)
  (IF PuntoIns ()
    (SETQ Ind (LOAD_DIALOG "c:/misdoc~1/autocad/
                          autolisp/acumular.dcl"))
  )
  (NEW_DIALOG "acumular" Ind)
  (START_LIST "TileCapa")
  (SETQ ElemCapa (CDR (ASSOC 2 (TBLNEXT "layer" T))))
  (WHILE ElemCapa
    (ADD_LIST ElemCapa)
    (SETQ ListaCapa (CONS ElemCapa ListaCapa))
    (SETQ ElemCapa (CDR (ASSOC 2 (TBLNEXT "layer" T))))
  )
  (SETQ ListaCapa (REVERSE ListaCapa))
  (END_LIST)
  (START_LIST "TileCapa2" 2)
  (SETQ ElemCapa (CDR (ASSOC 2 (TBLNEXT "layer" T))))
  (WHILE ElemCapa
    (ADD_LIST ElemCapa)
    (SETQ ElemCapa (CDR (ASSOC 2 (TBLNEXT "layer" T))))
  )
  (SETQ ListaCapa2 (CONS "Actuación" ListaCapa))
  (END_LIST)

  (IF ÍndiceCapa () (SETQ ÍndiceCapa "0"))
  (SET_TILE "TileCapa" ÍndiceCapa)
  (IF ModeCapa () (SETQ ModeCapa 0))
  (MODE_TILE "TileCapa" ModeCapa)
```

```
(IF Lista () (SETQ Lista "1"))
(SET_TILE "TileLista" Lista)
(IF Actual () (SETQ Actual "0"))
(SET_TILE "TileActual" Actual)

(IF Objeto () (SETQ Objeto "0"))
(SET_TILE "TileObjeto" Objeto)

(IF Estilo () (SETQ Estilo "STANDARD"))
(SET_TILE "TileEstilo" Estilo)

(IF Altura () (SETQ Altura "1"))
(SET_TILE "TileAltura" Altura)

(IF Rotación () (SETQ Rotación "0"))
(SET_TILE "TileRotación" Rotación)

(IF TextoIni () (SETQ TextoIni "Distancia total:"))
(SET_TILE "TileTextoIni" TextoIni)

(IF TextoFin () (SETQ TextoFin "milímetros.))
(SET_TILE "TileTextoFin" TextoFin)

(IF Línea () (SETQ Línea "1"))
(SET_TILE "TileLínea" Línea)
(IF ModeLínea () (SETQ ModeLínea 0))
(MODE_TILE "TileLínea" ModeLínea)

(IF Polilínea () (SETQ Polilínea "0"))
(SET_TILE "TilePolilínea" Polilínea)
(IF ModePolilínea () (SETQ ModePolilínea 0))
(MODE_TILE "TilePolilínea" ModePolilínea)

(IF Arco () (SETQ Arco "1"))
(SET_TILE "TileArco" Arco)
(IF ModeArco () (SETQ ModeArco 0))
(MODE_TILE "TileArco" ModeArco)

(IF Todo () (SETQ Todo "0"))
(SET_TILE "TileTodo" Todo)

(IF ÍndiceCapa2 () (SETQ ÍndiceCapa2 "0"))
(SET_TILE "TileCapa2" ÍndiceCapa2)

(IF X () (SETQ X "0"))
(SET_TILE "TileX" X)

(IF Y () (SETQ Y "0"))
(SET_TILE "TileY" Y)

(IF Z () (SETQ Z "0"))
(SET_TILE "TileZ" Z)

(ACTION_TILE "TileActual" "(MODE_TILE \"TileCapa\" 1) (SETQ
    ModeCapa 1)")
(ACTION_TILE "TileObjeto" "(MODE_TILE \"TileCapa\" 1) (SETQ
    ModeCapa 1)")
(ACTION_TILE "TileLista" "(MODE_TILE \"TileCapa\" 0) (SETQ ModeCapa
    0)")
(ACTION_TILE "TileTodo" "(SETQ Cambio 1) (CambiarCasillas)")
(ACTION_TILE "TileDesignar" "(GuardaVariables) (DONE_DIALOG 2)"))
```

```
(ACTION_TILE "TilePreferencias" "(PreferenciasTexto)")
(ACTION_TILE "accept" "(GuardaVariables) (ControlDCL1) (IF
    ErrorDCL1 () (DONE_DIALOG 1))")
(ACTION_TILE "cancel" "(DONE_DIALOG 0)")

(SETQ SD (START_DIALOG))
(COND
    ((= SD 0) ())
    ((= SD 1) (Aceptar))
    ((= SD 2) (Designar))
)
)

(DEFUN CambiarCasillas ()
    (IF (= Cambio 1) (PROGN
        (IF (= (GET_TILE "TileTodo") "0")
            (PROGN
                (MODE_TILE "TileLínea" 0)
                (SETQ ModeLínea 0)
                (MODE_TILE "TilePolilínea" 0)
                (SETQ ModePolilínea 0)
                (MODE_TILE "TileArco" 0)
                (SETQ ModeArco 0)
            )
            (PROGN
                (SET_TILE "TileLínea" "1")
                (MODE_TILE "TileLínea" 1)
                (SETQ ModeLínea 1)
                (SET_TILE "TilePolilínea" "1")
                (MODE_TILE "TilePolilínea" 1)
                (SETQ ModePolilínea 1)
                (SET_TILE "TileArco" "1")
                (SETQ ModeArco 1)
                (MODE_TILE "TileArco" 1)
            )
        )
    ))
)

(IF (= Cambio 2) (PROGN
    (IF (= (GET_TILE "TileSubtotalesRectos") "0")
        (PROGN
            (MODE_TILE "TileRectosTextoIni" 1)
            (SETQ ModeRectosTextoIni 1)
            (MODE_TILE "TileRectosTextoFin" 1)
            (SETQ ModeRectosTextoFin 1)
        )
        (PROGN
            (MODE_TILE "TileRectosTextoIni" 0)
            (SETQ ModeRectosTextoIni 0)
            (MODE_TILE "TileRectosTextoFin" 0)
            (SETQ ModeRectosTextoFin 0)
        )
    ))
)

(IF (= Cambio 3) (PROGN
    (IF (= (GET_TILE "TileSubtotalesCurvos") "0")
        (PROGN
            (MODE_TILE "TileCurvosTextoIni" 1)
            (SETQ ModeCurvosTextoIni 1)
            (MODE_TILE "TileCurvosTextoFin" 1)
        )
    ))
)
```

```
(SETQ ModeCurvosTextoFin 1)
)
(PROGN
  (MODE_TILE "TileCurvosTextoIni" 0)
  (SETQ ModeCurvosTextoIni 0)
  (MODE_TILE "TileCurvosTextoFin" 0)
  (SETQ ModeCurvosTextoFin 0)
)
))
)

(IF (= Cambio 4) (PROGN
  (IF (= (GET_TILE "TileAutorEmpresa") "0")
    (PROGN
      (MODE_TILE "TileAutor" 1)
      (SETQ ModeAutor 1)
      (MODE_TILE "TileEmpresa" 1)
      (SETQ ModeEmpresa 1)
      (MODE_TILE "TileUbicación" 1)
      (SETQ ModeUbicación 1)
    )
    (PROGN
      (MODE_TILE "TileAutor" 0)
      (SETQ ModeAutor 0)
      (MODE_TILE "TileEmpresa" 0)
      (SETQ ModeEmpresa 0)
      (MODE_TILE "TileUbicación" 0)
      (SETQ ModeUbicación 0)
    )
  )
))
)

(SETQ Cambio nil)
)

(DEFUN GuardaVariables ()
  (SETQ ÍndiceCapa (GET_TILE "TileCapa"))
  (IF (= (GET_TILE "TileLista") "0") () (SETQ Capa (NTH (ATOI
    (GET_TILE "TileCapa")) ListaCapa)))
  (IF (= (GET_TILE "TileActual") "0") () (SETQ Capa (GETVAR
    "clayer"))))
  (IF (= (GET_TILE "TileObjeto") "0") () (SETQ Capa nil))
  (SETQ Lista (GET_TILE "TileLista"))
  (SETQ Actual (GET_TILE "TileActual"))
  (SETQ Objeto (GET_TILE "TileObjeto"))
  (SETQ Estilo (GET_TILE "TileEstilo"))
  (SETQ Altura (GET_TILE "TileAltura"))
  (SETQ Rotación (GET_TILE "TileRotación"))
  (SETQ TextoIni (GET_TILE "TileTextoIni"))
  (SETQ TextoFin (GET_TILE "TileTextoFin"))
  (SETQ Línea (GET_TILE "TileLínea"))
  (SETQ Polilínea (GET_TILE "TilePolilínea"))
  (SETQ Arco (GET_TILE "TileArco"))
  (SETQ Todo (GET_TILE "TileTodo"))
  (SETQ ÍndiceCapa2 (GET_TILE "TileCapa2"))
  (SETQ Capa2 (NTH (ATOI (GET_TILE "TileCapa2")) ListaCapa2))
  (IF (= Capa2 "Actuación") (SETQ Capa2 Capa))
  (SETQ X (GET_TILE "TileX"))
  (SETQ Y (GET_TILE "TileY"))
  (SETQ Z (GET_TILE "TileZ"))
```

```
)

(DEFUN Designar ()
  (SETQ PuntoIns (GETPOINT "\nPunto de inserción para el texto: "))
  (SETQ X (RTOS (CAR PuntoIns)))
  (SETQ Y (RTOS (CADR PuntoIns)))
  (SETQ Z (RTOS (CADDR PuntoIns)))
  (Cuadro)
)

(DEFUN PreferenciasTexto ()
  (NEW_DIALOG "preferencias" Ind)

  (IF SubtotalesRectos () (SETQ SubtotalesRectos "0"))
  (SET_TILE "TileSubtotalesRectos" SubtotalesRectos)

  (IF RectosTextoIni () (SETQ RectosTextoIni "Total tramos rectos:"))
  (SET_TILE "TileRectosTextoIni" RectosTextoIni)
  (IF ModeRectosTextoIni () (SETQ ModeRectosTextoIni 1))
  (MODE_TILE "TileRectosTextoIni" ModeRectosTextoIni)

  (IF RectosTextoFin () (SETQ RectosTextoFin "milímetros.))
  (SET_TILE "TileRectosTextoFin" RectosTextoFin)
  (IF ModeRectosTextoFin () (SETQ ModeRectosTextoFin 1))
  (MODE_TILE "TileRectosTextoFin" ModeRectosTextoFin)

  (IF SubtotalesCurvos () (SETQ SubtotalesCurvos "0"))
  (SET_TILE "TileSubtotalesCurvos" SubtotalesCurvos)

  (IF CurvosTextoIni () (SETQ CurvosTextoIni "Total tramos curvos:"))
  (SET_TILE "TileCurvosTextoIni" CurvosTextoIni)
  (IF ModeCurvosTextoIni () (SETQ ModeCurvosTextoIni 1))
  (MODE_TILE "TileCurvosTextoIni" ModeCurvosTextoIni)

  (IF CurvosTextoFin () (SETQ CurvosTextoFin "milímetros.))
  (SET_TILE "TileCurvosTextoFin" CurvosTextoFin)
  (IF ModeCurvosTextoFin () (SETQ ModeCurvosTextoFin 1))
  (MODE_TILE "TileCurvosTextoFin" ModeCurvosTextoFin)

  (IF Fecha () (SETQ Fecha "1"))
  (SET_TILE "TileFecha" Fecha)

  (IF Hora () (SETQ Hora "0"))
  (SET_TILE "TileHora" Hora)

  (IF AutorEmpresa () (SETQ AutorEmpresa "0"))
  (SET_TILE "TileAutorEmpresa" AutorEmpresa)

  (IF Autor () (SETQ Autor ""))
  (SET_TILE "TileAutor" Autor)
  (IF ModeAutor () (SETQ ModeAutor 1))
  (MODE_TILE "TileAutor" ModeAutor)

  (IF Empresa () (SETQ Empresa ""))
  (SET_TILE "TileEmpresa" Empresa)
  (IF ModeEmpresa () (SETQ ModeEmpresa 1))
  (MODE_TILE "TileEmpresa" ModeEmpresa)

  (IF Ubicación () (SETQ Ubicación ""))
  (SET_TILE "TileUbicación" Ubicación)
  (IF ModeUbicación () (SETQ ModeUbicación 1))
```

```
(MODE_TILE "TileUbicación" ModeUbicación)

(ACTION_TILE "cancel" "(DONE_DIALOG 3)")
(ACTION_TILE "accept" "(GuardaVariables2) (ControlDCL2) (IF
    ErrorDCL2 () (DONE_DIALOG 4))")
(ACTION_TILE "TileSubtotalesRectos" "(SETQ Cambio 2)
    (CambiarCasillas)")
(ACTION_TILE "TileSubtotalesCurvos" "(SETQ Cambio 3)
    (CambiarCasillas)")
(ACTION_TILE "TileAutorEmpresa" "(SETQ Cambio 4) (CambiarCasillas)")

(SETQ SD (START_DIALOG))
(COND
    ((= SD 3) ())
    ((= SD 4) ())
)
)

(DEFUN GuardaVariables2 ()
    (SETQ SubtotalesRectos (GET_TILE "TileSubtotalesRectos"))
    (SETQ RectosTextoIni (GET_TILE "TileRectosTextoIni"))
    (SETQ RectosTextoFin (GET_TILE "TileRectosTextoFin"))
    (SETQ SubtotalesCurvos (GET_TILE "TileSubtotalesCurvos"))
    (SETQ CurvosTextoIni (GET_TILE "TileCurvosTextoIni"))
    (SETQ CurvosTextoFin (GET_TILE "TileCurvosTextoFin"))
    (SETQ Fecha (GET_TILE "TileFecha"))
    (SETQ Hora (GET_TILE "TileHora"))
    (SETQ AutorEmpresa (GET_TILE "TileAutorEmpresa"))
    (SETQ Autor (GET_TILE "TileAutor"))
    (SETQ Empresa (GET_TILE "TileEmpresa"))
    (SETQ Ubicación (GET_TILE "TileUbicación"))
)

(DEFUN Aceptar (/ CapaObjeto)
    (IF (NOT Capa)
        (PROGN
            (WHILE (NOT (SETQ CapaObjeto (ENTSEL "\nObjeto en cuya capa
                se actuará: "))))
            (SETQ Capa (CDR (ASSOC 8 (ENTGET (CAR CapaObjeto)))))
            (IF (NOT Capa2) (SETQ Capa2 Capa))
        )
    )
    (SETQ Altura (ATOF Altura))
    (SETQ Rotación (ATOF Rotación))
    (SETQ X (ATOF X))
    (SETQ Y (ATOF Y))
    (SETQ Z (ATOF Z))

    (PROMPT "\nCalculando...")(PRIN1)
    (IF (= Línea "1") (CálculoLíneas))
    (IF (= Arco "1") (CálculoArcos))
    (IF (= Polilínea "1") (CálculoPolilíneas))

    (InsertarTexto)
)

(DEFUN CálculoLíneas (/ ConjuntoLíneas NúmeroLíneas ListaLínea
    XPuntoIniLínea YPuntoIniLínea ZPuntoIniLínea
    XPuntoFinLínea YPuntoFinLínea ZPuntoFinLínea
    ListaPuntoIniLínea ListaPuntoFinLínea
    ParcialLíneas)
```

```
(SETQ Contador 0)
(SETQ ConjuntoLíneas (SSGET "X" (LIST (CONS 0 "line") (CONS 8
    Capa))))
(IF (NOT ConjuntoLíneas) ()
    (PROGN
        (SETQ NúmeroLíneas (SLENGTH ConjuntoLíneas))
        (REPEAT NúmeroLíneas
            (SETQ ListaLínea (ENTGET (SSNAME ConjuntoLíneas Contador)))
            (SETQ XPuntoIniLínea (CADR (ASSOC 10 ListaLínea)))
            (SETQ YPuntoIniLínea (CADDR (ASSOC 10 ListaLínea)))
            (SETQ ZPuntoIniLínea (CADDR (ASSOC 10 ListaLínea)))
            (SETQ XPuntoFinLínea (CADR (ASSOC 11 ListaLínea)))
            (SETQ YPuntoFinLínea (CADDR (ASSOC 11 ListaLínea)))
            (SETQ ZPuntoFinLínea (CADDR (ASSOC 11 ListaLínea)))
            (SETQ ListaPuntoIniLínea (LIST XPuntoIniLínea YPuntoIniLínea
                ZPuntoIniLínea))
            (SETQ ListaPuntoFinLínea (LIST XPuntoFinLínea YPuntoFinLínea
                ZPuntoFinLínea))
            (SETQ ParcialLíneas (DISTANCE ListaPuntoIniLínea
                ListaPuntoFinLínea))
            (SETQ TotalLíneas (+ TotalLíneas ParcialLíneas))
            (SETQ Contador (1+ Contador))
        )
    )
)

(DEFUN CálculoArcos (/ ConjuntoArcos NúmeroArcos ListaArco
    ÁnguloIniArco ÁnguloFinArco RadioArco
    DiferenciaÁngulos ParcialArcos)
    (SETQ Contador 0)
    (SETQ ConjuntoArcos (SSGET "X" (LIST (CONS 0 "arc") (CONS 8 Capa))))
    (IF (NOT ConjuntoArcos) ()
        (PROGN
            (SETQ NúmeroArcos (SLENGTH ConjuntoArcos))
            (REPEAT NúmeroArcos
                (SETQ ListaArco (ENTGET (SSNAME ConjuntoArcos Contador)))
                (SETQ ÁnguloIniArco (CDR (ASSOC 50 ListaArco)))
                (SETQ ÁnguloFinArco (CDR (ASSOC 51 ListaArco)))
                (SETQ RadioArco (CDR (ASSOC 40 ListaArco)))
                (SETQ DiferenciaÁngulos (- ÁnguloFinArco ÁnguloIniArco))
                (WHILE (< DiferenciaÁngulos 0)
                    (SETQ DiferenciaÁngulos (+ DiferenciaÁngulos (* 2 PI)))
                )
                (SETQ ParcialArcos ((* 2 PI RadioArco) (/ DiferenciaÁngulos
                    (* 2 PI))))
                (SETQ TotalArcos (+ TotalArcos ParcialArcos))
                (SETQ Contador (1+ Contador))
            )
        )
    )
)

(DEFUN CálculoPolilíneas (/ Contador2 ConjuntoPolilíneas
    NúmeroPolilíneas
    NombrePolilínea ListaPolilínea
    ConjuntoDescomponer
    NombreDescomponer GuardaCapa)
    (SETQ Contador 0)
    (SETQ Contador2 0)
    (SETQ ConjuntoPolilíneas (SSGET "X" (LIST (CONS 0 "lwpolyline")
```



```
(CONS 8 Capa)))
(IF (NOT ConjuntoPolilíneas) ()
  (PROGN
    (SETQ NúmeroPolilíneas (SLENGTH ConjuntoPolilíneas))
    (REPEAT NúmeroPolilíneas
      (SETQ NombrePolilínea (SSNAME ConjuntoPolilíneas Contador))
      (SETQ ListaPolilínea (ENTGET NombrePolilínea))
      (COMMAND "_.copy" NombrePolilínea "" "0,0,0" "0,0,0")
      (SETQ ListaPolilínea (SUBST (CONS 8 "$Acu$Capa$Pol$") (CONS 8
        Capa) ListaPolilínea))
      (ENTMOD ListaPolilínea)
      (SETQ Contador (1+ Contador))
    )
    (SETQ ConjuntoDescomponer (SSGET "X" (LIST (CONS 8
      "$Acu$Capa$Pol$"))))
    (SETQ NúmeroDescomponer (SLENGTH ConjuntoDescomponer))
    (REPEAT NúmeroDescomponer
      (SETQ NombreDescomponer (SSNAME ConjuntoDescomponer Contador2))
      (COMMAND "_.explode" NombreDescomponer "")
      (SETQ Contador2 (1+ Contador2))
    )
    (SETQ GuardaCapa Capa Capa "$Acu$Capa$Pol$")
    (CálculoLíneas)
    (CálculoArcos)
    (SETQ Capa GuardaCapa)
    (COMMAND "_.-layer" "_lo" "*" "")
    (COMMAND "_.-layer" "_u" "$Acu$Capa$Pol$" "")
    (COMMAND "_.erase" "_all" "")
    (COMMAND "_.-layer" "_u" "*" "")
    (COMMAND "_.purge" "_la" "$Acu$Capa$Pol$" "_n")
  )
)
)

(DEFUN InsertarTexto (/ TextoPrincipal Coordenadas
  TextoRectos TextoCurvos TextoAutor TextoEmpresa
  TextoUbicación TextoFecha TextoHora)
  (SETVAR "clayer" Capa2)
  (GRAPHSCR)
  (SETQ TextoPrincipal (STRCAT TextoIni " " (RTOS (+ TotalLíneas
    TotalArcos)) " " TextoFin))
  (SETQ Coordenadas (LIST X Y Z))
  (COMMAND "_.text" "_s" Estilo Coordenadas Altura Rotación
    TextoPrincipal)
  (IF (= SubtotalesRectos "1")
    (PROGN
      (SETQ TextoRectos (STRCAT RectosTextoIni " " (RTOS TotalLíneas)
        " " RectosTextoFin))
      (COMMAND "_.text" "" TextoRectos)
    )
  )
  (IF (= SubtotalesCurvos "1")
    (PROGN
      (SETQ TextoCurvos (STRCAT CurvosTextoIni " " (RTOS TotalArcos) "
        " CurvosTextoFin))
      (COMMAND "_.text" "" TextoCurvos)
    )
  )
  (IF (= AutorEmpresa "1")
    (PROGN
      (COMMAND "_.text" "" " " " ")
    )
  )
)
```

```
(SETQ TextoAutor (STRCAT "Autor: " Autor))
(COMMAND "_text" "" TextoAutor)
(SETQ TextoEmpresa (STRCAT "Empresa: " Empresa))
(COMMAND "_text" "" TextoEmpresa)
(COMMAND "_text" "" (STRCAT "Ubicación: " Ubicación))
)
)
(IF (= Fecha "1")
  (PROGN
    (COMMAND "_text" "" " ")
    (SETQ TextoFecha (MENUCMD "M=$(edtime,$(getvar,date),DDDD\\",\\ " D
      MON YY"))
    (COMMAND "_text" "" (STRCAT "Fecha: " TextoFecha))
  )
)

(IF (= Hora "1")
  (PROGN
    (IF (= Fecha "1") () (COMMAND "_text" "" " "))
    (SETQ TextoHora (MENUCMD "M=$(edtime,$(getvar,date),HH:MM
      am/pm"))
    (COMMAND "_text" "" (STRCAT "Hora: " TextoHora))
  )
)

(SETQ Altura (RTOS Altura))
(SETQ Rotación (RTOS Rotación))
(SETQ X (RTOS X))
(SETQ Y (RTOS Y))
(SETQ Z (RTOS Z))

)

(DEFUN C:Acumular (/ ListaCapa ListaCapa2 ÍndiceCapa ÍndiceCapa2 Capa
  Capa2 Clay Contador Cambio SD Eco ErrorDCL1)
  (SETQ Eco (GETVAR "cmdecho"))
  (SETVAR "cmdecho" 0)
  (ComprobarVersión)
  (COMMAND "_undo" "_be")
  (SETQ ErrorLSP *error* *error* ControllSP)
  (SETQ CLay (GETVAR "clayer"))
  (SETQ TotalLíneas 0 TotalArcos 0 TotalPolilíneas 0)
  (IF Fecha () (SETQ Fecha "1"))
  (Cuadro)
  (SETVAR "clayer" CLay)
  (SETVAR "cmdecho" Eco)
  (PRIN1)
)

(DEFUN C:Acu ()
  (C:Acumular)
)

(DEFUN ComprobarVersión ()
  (SETQ Versión (VER))
  (IF (NOT (OR (WCMATCH Versión "*14*") (WCMATCH Versión "*2000*"))))
    (PROGN
      (ALERT "Versión incorrecta de AutoCAD. \\"Acumular
        distancias\\\"\\núnicamente corre bajo AutoCAD y/o AutoCAD
        2000.")
      (EXIT)
    )
  )
)
```

```
(PRIN1)
)
)
)

(DEFUN ControllSP (Mensaje)
  (SETQ *error* ErrorLSP)
  (PRINC Mensaje)(TERPRI)
  (COMMAND "_.undo" "_e")
  (SETVAR "clayer" CLay)
  (SETQ Capa nil Capa2 nil ListaCapa nil ListaCapa2 nil Lista nil
    Actual nil Objeto nil Estilo nil Altura nil Rotación nil
    TextoIni nil TextoFin nil Línea nil Polilínea nil Arco nil
    Todo nil ÍndiceCapa nil ÍndiceCapa2 nil X nil Y nil
    Z nil SubtotalesRectos nil RectosTextoIni nil RectosTextoFin
    nil SubtotalesCurvos nil CurvosTextoIni nil
    CurvosTextoFin nil Fecha nil Hora nil AutorEmpresa nil Autor
    nil Empresa nil Ubicación nil ModeLínea nil
    ModePolilínea nil ModeArco nil ModeRectosTextoIni nil
    ModeRectosTextoFin nil ModeCurvosTextoIni nil
    ModeCurvosTextoFin nil ModeAutor nil ModeEmpresa ModeUbicación
    nil ModeCapa nil)
  (SETVAR "cmdecho" Eco)
  (PRIN1)
)

(DEFUN ControlDCL1 ()
  (SETQ ErrorDCL1 nil)
  (IF (= (GET_TILE "TileEstilo") "")
    (PROGN
      (SETQ ErrorDCL1 T)
      (SET_TILE "error" "Introduzca un estilo para el texto.")
      (MODE_TILE "TileEstilo" 2)
    )
    (PROGN
      (IF (NOT (TBLSEARCH "style" Estilo))
        (PROGN
          (SETQ ErrorDCL1 T)
          (SET_TILE "error" "Estilo de texto no existente en el dibujo
            actual.")
          (MODE_TILE "TileEstilo" 2)
        )
      )
    )
  )
)
)
)
(DEFUN ControlDCL2 ()
  (IF (= (GET_TILE "TileAltura") "")
    (PROGN
      (SETQ ErrorDCL1 T)
      (SET_TILE "error" "Introduzca una altura para el texto.")
      (MODE_TILE "TileAltura" 2)
    )
    (PROGN
      (IF (OR (< (ATOF (GET_TILE "TileAltura")) 0) (= (ATOF (GET_TILE
        "TileAltura")) 0))
        (PROGN
          (SETQ ErrorDCL1 T)
          (SET_TILE "error" "La altura del texto debe ser mayor de
            cero.")
          (MODE_TILE "TileAltura" 2)
        )
      )
    )
  )
)
```

```
)
)
(IF (= (GET_TILE "TileRotación") "")
  (PROGN
    (SETQ ErrorDCL1 T)
    (SET_TILE "error" "Introduzca un ángulo de rotación para el
      texto.")
    (MODE_TILE "TileRotación" 2)
  )
)
)
(IF (= (GET_TILE "TileTextoIni") "")
  (PROGN
    (SETQ ErrorDCL1 T)
    (SET_TILE "error" "Falta el texto de inicio, introduzca un texto
      o un espacio blanco.")
    (MODE_TILE "TileTextoIni" 2)
  )
)
)
(IF (= (GET_TILE "TileTextoFin") "")
  (PROGN
    (SETQ ErrorDCL1 T)
    (SET_TILE "error" "Falta el texto de final, introduzca un texto
      o un espacio blanco.")
    (MODE_TILE "TileTextoFin" 2)
  )
)
)
(IF (AND (= (GET_TILE "TileLínea") "0") (= (GET_TILE
  "TilePolilínea") "0") (= (GET_TILE "TileArco") "0")))
  (PROGN
    (SETQ ErrorDCL1 T)
    (SET_TILE "error" "Introduzca al menos un objeto con el que
      acumular distancias.")
    (MODE_TILE "TileTodo" 2)
  )
)
)
(IF (= (GET_TILE "TileX") "")
  (PROGN
    (SETQ ErrorDCL1 T)
    (SET_TILE "error" "Introduzca la coordenada X de inserción o
      pulse el botón \"Designar <\".")
    (MODE_TILE "TileX" 2)
  )
)
)
(IF (= (GET_TILE "TileY") "")
  (PROGN
    (SETQ ErrorDCL1 T)
    (SET_TILE "error" "Introduzca la coordenada Y de inserción o
      pulse el botón \"Designar <\".")
    (MODE_TILE "TileY" 2)
  )
)
)
(IF (= (GET_TILE "TileZ") "")
  (PROGN
    (SETQ ErrorDCL1 T)
    (SET_TILE "error" "Introduzca la coordenada Z de inserción o
      pulse el botón \"Designar <\".")
    (MODE_TILE "TileZ" 2)
  )
)
)
)
```

```
(DEFUN ControlDCL2 ()
  (SETQ ErrorDCL2 nil)
  (IF (= (GET_TILE "TileSubtotalesRectos") "1")
    (PROGN
      (IF (= (GET_TILE "TileRectosTextoIni") "")
        (PROGN
          (SETQ ErrorDCL2 T)
          (SET_TILE "error" "Falta el texto de inicio, introduzca un
            texto o un espacio blanco.")
          (MODE_TILE "TileRectosTextoIni" 2)
        )
      )
      (IF (= (GET_TILE "TileRectosTextoFin") "")
        (PROGN
          (SETQ ErrorDCL2 T)
          (SET_TILE "error" "Falta el texto de final, introduzca un
            texto o un espacio blanco.")
          (MODE_TILE "TileRectosTextoFin" 2)
        )
      )
    )
  )
  (IF (= (GET_TILE "TileSubtotalesCurvos") "1")
    (PROGN
      (IF (= (GET_TILE "TileCurvosTextoIni") "")
        (PROGN
          (SETQ ErrorDCL2 T)
          (SET_TILE "error" "Falta el texto de inicio, introduzca un
            texto o un espacio blanco.")
          (MODE_TILE "TileCurvosTextoIni" 2)
        )
      )
      (IF (= (GET_TILE "TileCurvosTextoFin") "")
        (PROGN
          (SETQ ErrorDCL2 T)
          (SET_TILE "error" "Falta el texto de final, introduzca un
            texto o un espacio blanco.")
          (MODE_TILE "TileCurvosTextoFin" 2)
        )
      )
    )
  )
  (IF (= (GET_TILE "TileAutorEmpresa") "1")
    (PROGN
      (IF (= (GET_TILE "TileAutor") "")
        (PROGN
          (SETQ ErrorDCL2 T)
          (SET_TILE "error" "Falta el nombre del autor, introduzca un
            nombre o un espacio blanco.")
          (MODE_TILE "TileAutor" 2)
        )
      )
      (IF (= (GET_TILE "TileEmpresa") "")
        (PROGN
          (SETQ ErrorDCL2 T)
          (SET_TILE "error" "Falta el nombre de la empresa, introduzca
            un nombre o un espacio blanco.")
          (MODE_TILE "TileEmpresa" 2)
        )
      )
      (IF (= (GET_TILE "TileUbicación") "")
```

```
(PROGN
  (SETQ ErrorDCL2 T)
  (SET_TILE "error" "Falta la ubicación de la empresa,
    introduzca una ubicación o un espacio blanco.")
  (MODE_TILE "TileUbicación" 2)
)
)
)
)
```


MÓDULO TRECE

Entorno de programación Visual Lisp

TRECE.1. Visual Lisp ES...

Visual Lisp es un entorno de desarrollo diseñado para proporcionar herramientas de ayuda en la creación y modificación de programas fuente de AutoLISP, y herramientas de chequeo y depuración durante la prueba y simulación de los programas. Además, permite controlar objetos ARX desde AutoLISP. En este **MÓDULO TRECE** se estudia su entorno.

TRECE.2. PROCESO DE CREACIÓN DE UN PROGRAMA

El proceso de creación de un programa en AutoLISP se puede desglosar en una serie de etapas características. El entorno de Visual Lisp interviene en cada una de la manera explicada a continuación:

- **Generación de archivos de código fuente.** Visual Lisp proporciona:
 - Editor de texto de AutoLISP y DCL, con códigos de colores y herramientas para evidenciar la sintaxis de las expresiones.
 - Formateador de AutoLISP para proporcionar al texto un formato y anidación adecuados.
 - Ayuda sensible al contexto para recordar sintaxis de funciones de AutoLISP y detección de símbolos de funciones o variables.
- **Ensayo y depuración de programas.** Visual Lisp proporciona:
 - Corrector de sintaxis que reconoce construcciones de expresiones erróneas y argumentos impropios.
 - Simulación paso a paso de ejecución del programa, mostrando simultáneamente las líneas del código fuente y su efecto en **AutoCAD**.
 - Acceso a valores de variables y expresiones para examinar la estructura de datos del programa. También se incluye el acceso a datos de los objetos de dibujo de **AutoCAD**.
- **Utilización de programas.** Visual Lisp proporciona:
 - Compilador de archivos para aumentar la eficiencia y seguridad de utilización de los programas.
 - Gestión de proyectos para trabajar con varios programas relacionados a la vez.
 - Empaquetamiento de varios archivos compilados en un único módulo ADS o ARX.

TRECE.3. INSTALACIÓN E INICIACIÓN

Visual Lisp no funciona de manera independiente sino que necesita que haya una sesión de **AutoCAD** ejecutándose, como ocurriría con el módulo VBA. El módulo de Visual Lisp se instala desde un CD-ROM, creando su propio grupo de programas y una carpeta específica Visual Lisp dentro del directorio de **AutoCAD**. El módulo se ha diseñado como una

aplicación ARX, por lo que se debe cargar desde **AutoCAD**, desde el *menú Herr.>Cargar aplicación*, archivo `VLIDE.ARX`. Nada más cargarse, se muestra la ventana de aplicación de Visual Lisp.

Dicha ventana contiene una serie de áreas de trabajo que son las siguientes:

- **Barra de menús.** Serie de menús desplegables con los comandos y utilidades de Visual Lisp. Las opciones de los menús pueden cambiar según el proceso de trabajo. Los menús son:

Menú	Función
<i>File</i>	Operaciones con archivos de programas.
<i>Edit</i>	Operaciones de edición de textos, cambios, comandos Visual Lisp, etc.
<i>Search</i>	Búsqueda de texto, marcas, saltos a línea, etc.
<i>View</i>	Activación de ventanas de trabajo, barras de herramientas, etc.
<i>Project</i>	Gestión de proyectos y programas compilados.
<i>Debug</i>	Herramientas de depuración de programas.
<i>Tools</i>	Herramientas generales.
<i>Window</i>	Gestión de ventanas de trabajo.
<i>Help</i>	Ayuda en línea.

- **Barras de herramientas.** Hay un total de cinco, con los comandos y utilidades más empleados. Son *Debug*, *Edit*, *Find*, *Inspect* y *Run*.

- **Escritorio.** Superficie donde se despliegan las diferentes ventanas de trabajo de Visual Lisp. La de *Consola* se muestra desde el principio, la del *Editor de texto* cada vez que se carga un archivo fuente (al principio se carga `VLIDE.LSP`), y la de *Trace* aparece minimizada hasta que se activa el rastreo. El resto se activan durante el trabajo o expresamente desde el menú *View*.

- **Ventana de Consola.** En ella se introducen los comandos de Visual Lisp y las funciones de AutoLISP. Equivale al historial de comandos de **AutoCAD**. Ofrece un *prompt* o señal con el aspecto `_$.`

- **Editor de texto:** se abre una ventana por cada archivo fuente cargado. Allí se muestran las expresiones de AutoLISP mediante colores, para distinguir visualmente los diferentes tipos de símbolos.

- **Línea de estado:** línea en la parte inferior de la pantalla, con información permanente.

TRECE.3.1. Carga y ejecución de programas

Los archivos con el código fuente se pueden abrir o cargar. En el primer caso (menú *File>Open File*) se muestra el listado en una ventana de Editor de texto. En el segundo caso (menú *File>Load File*), se cargan además en memoria según se abren, tal como se haría desde **AutoCAD** mediante *Herr.>Cargar aplicación*. Si existen errores en el programa, la Consola muestra mensajes de advertencia. Se puede cargar el programa abierto actualmente, desde un icono de la barra *Tools*.

Si todo es correcto, se puede utilizar el programa llamando a la nueva función o comando de **AutoCAD** directamente desde la Consola de Visual Lisp. Ésta pasa automáticamente a **AutoCAD** y allí se especifican los datos o puntos solicitados por el programa. Cuando éste termina, se regresa a Visual Lisp. Si **AutoCAD** se encuentra

minimizado o no está en ejecución, Visual Lisp muestra en el puntero del ratón un cursor con un par de paréntesis. Esto significa que no puede comunicarse con **AutoCAD**. Se debe abrir una sesión de dibujo o maximizar **AutoCAD** para seguir trabajando.

Durante el trabajo, resulta muy habitual alternar entre Visual Lisp y **AutoCAD**. Esto se puede hacer mediante los procedimientos estándar de Windows pero, para facilitar la tarea, Visual Lisp dispone de un botón específico para pasar a **AutoCAD**. A su vez, el comando `VLIDE` de **AutoCAD** permite pasar directamente a Visual Lisp.

Se pueden ejecutar expresiones individuales y trozos del archivo fuente, para examinar su funcionamiento. Para ello se seleccionan en el Editor de texto y se elige la opción de menú *Tools>Load selection* o el icono correspondiente.

Para salir de Visual Lisp, se elige el menú *File>Exit*. Si ha habido modificaciones en los archivos abiertos, se solicita si se desea guardar los cambios. La próxima vez que se entre en Visual Lisp, los últimos archivos utilizados serán automáticamente abiertos.

TRECE.4. ESCRITURA DEL CÓDIGO FUENTE

Se examinan en esta sección las dos ventanas básicas de trabajo en Visual Lisp: Consola y Editor de texto. Además, se estudian también las herramientas que inciden en el proceso de escritura y corrección del código fuente.

TRECE.4.1. Ventana de Consola

Como ya se ha adelantado, contiene el *prompt* de Visual Lisp `_ $` y un historial de comandos. La última línea es la línea de comando, donde se introducen las instrucciones o expresiones que evaluar. Las características de la Consola son:

- Se introducen directamente expresiones y variables de AutoLISP para evaluar sus resultados (no es necesario el carácter `!` como en **AutoCAD** para evaluar variables). Si una expresión no cabe en una línea, mediante `CTRL+INTRO` se establece una continuidad con la siguiente. El código de colores es el mismo que en el Editor de texto.
- Se puede copiar e intercambiar texto entre la Consola y el Editor de texto. Si se selecciona un texto del historial en la Consola (arrastrando el cursor hasta iluminarlo) y después se pulsa `INTRO`, ese texto se copia a la línea de comando.
- Se pueden recuperar comandos del historial de Visual Lisp, mediante las teclas de `TAB` (hacia arriba) y `SHIFT+TAB` (hacia abajo).
- Si se utilizan las teclas anteriores después de haber empezado una expresión en la Consola, por ejemplo `(SETQ`, se recuperan —hacia arriba y hacia abajo— todas las expresiones que comienzan igual.
- La tecla `ESC` elimina el texto actual en la línea de comando de la Consola. La combinación `SHIFT+ESC` empieza una nueva línea de comando, pero mantiene el texto de la anterior sin evaluar, por si se desea recuperar más tarde.
- Existe un menú flotante contextual desde el botón derecho del ratón (también desde `SHIFT+F10`), con las opciones indicadas en la tabla siguiente:

Opción de menú	Descripción
<i>Cut</i>	Corta el texto seleccionado hacia el portapapeles.
<i>Copy</i>	Copia el texto seleccionado al portapapeles.
<i>Paste</i>	Pega el texto del portapapeles en la posición del cursor.

Opción de menú	Descripción
<i>Clear Console Window</i>	Vacía de texto la ventana de Consola.
<i>Find</i>	Busca un texto especificado en la ventana de Consola
<i>Inspect</i>	Abre el cuadro de diálogo de <i>Inspección</i> .
<i>Add Watch</i>	Abre la ventana de seguimiento de expresiones.
<i>Apropos Window</i>	Abre la ventana de localización de símbolos y funciones.
<i>Symbol service</i>	Abre el cuadro de diálogo para establecer rastreos de símbolos.
<i>Undo</i>	Deshace la última operación.
<i>Redo</i>	Rehace los efectos de la última operación deshecha.
<i>AutoCAD Mode</i>	Activa la transferencia automática a línea de comando de AutoCAD .
<i>Toggle Console Log</i>	Abre/cierra un archivo de registro para escribir el historial de Consola.

Existe una ayuda sensible al contexto, a la que se accede desde el botón correspondiente en la barra de herramientas de *Tools*. Se puede invocar tras seleccionar un símbolo o expresión.

Si se ha activado *AutoCAD Mode*, el *prompt* pasa a ser *Command:.* Ahora todas las expresiones se transfieren automáticamente a **AutoCAD**. Además, es posible escribir comandos de **AutoCAD** en Visual Lisp.

TRECE.4.2. Editor de texto

Es una ventana que contiene el texto de un archivo abierto. Si se abren varios archivos a la vez, habrá una ventana por archivo. Los archivos que se pueden abrir son de cuatro tipos: LSP, DCL, SQL y C/C++. En este apartado se consideran fundamentalmente los archivos de AutoLISP. El Editor de texto va sangrando las expresiones conforme se escriben, en función del número de paréntesis que se van abriendo. Para ello, se puede utilizar la tecla *INTRO* de tres maneras:

1) Pulsando *INTRO* sin más, el cursor cambia de línea y sangra su posición de acuerdo con los paréntesis que van quedando abiertos.

2) Pulsando *SHIFT+INTRO*, el cursor cambia de línea sin sangrar su posición, quedándose al mismo nivel que la línea de texto anterior.

3) Pulsando *CTRL+INTRO*, el cursor cambia de línea retrocediendo hasta la posición inicial sin sangrado, y no tiene en cuenta el nivel de anidación actual.

La tecla *TAB* inserta tabuladores y la combinación *SHIFT+TAB* (se puede usar incluso en mitad de una línea) elimina las tabulaciones al principio de la línea. El tamaño de las tabulaciones se puede modificar desde el menú *Tools>Window Attributes>Configure Current*.

Es posible deshacer y rehacer sucesivas veces las últimas modificaciones en el texto, pero haciéndolo de forma continua. Si se escriben, por ejemplo, tres líneas de texto, se pueden deshacer las tres seguidas y recuperar de nuevo las tres seguidas. Pero si se deshace una y después se escribe una nueva, ya no es posible rehacer la eliminada.

Conviene ir guardando el texto de cuando en cuando para evitar que un suceso imprevisto ocasione su pérdida. Existe un mecanismo de copia de seguridad similar al de **AutoCAD**, de manera que se genera un archivo de copia con el contenido del archivo sobrescrito. La extensión de la copia de seguridad se forma con el carácter *_* y los dos

primeros de la extensión original. Así, para los archivos de AutoLISP .LSP, la extensión es ._LS. Si se desea desechar las últimas modificaciones, la opción *File>Revert* recupera el contenido del archivo .LSP tal como se había guardado la última vez.

El código de colores empleado en el Editor de texto se ofrece en la siguiente tabla:

Color	Símbolo de AutoLISP
Rojo	Paréntesis
Azul	Funciones propias de AutoLISP y símbolos protegidos
Magenta	Cadenas de texto
Verde	Valores numéricos enteros
Verde azulado	Valores numéricos reales
Negro	Símbolos no reconocidos (como las funciones y variables de usuario)
Magenta en fondo gris	Comentarios

Estos colores se pueden modificar desde el *menú Tools>Window Attributes>Configure Current*. Aparece un cuadro de diálogo donde es posible seleccionar el elemento que vamos a modificar en una lista desplegable, y después especificar colores de fondo y primer plano. También se puede indicar el intervalo del tabulador (ya comentado) y un margen izquierdo.

Igual que en la Consola, se puede utilizar el mismo botón de ayuda sensible al contexto. También existe un menú flotante con algunas opciones idénticas a la Consola, añadiéndose dos específicas:

Opción de menú	Descripción
<i>Go to Last Edited</i>	Mueve el cursor a la posición del último texto editado.
<i>Toggle Breakpoint</i>	Sitúa un punto de ruptura en la posición del cursor o lo elimina si existía.

Existen dos herramientas muy útiles a la hora de ir escribiendo los símbolos y funciones, para completar una palabra:

- Coincidencia (*Match*): Se utiliza para funciones y variables existentes en el archivo. Consiste en escribir los primeros caracteres de la palabra y pulsar sucesivas veces las teclas ALT+/ en el teclado inglés o ALT+ç en el teclado español. Visual Lisp va ofreciendo todas aquellas funciones y variables existentes en el archivo cuyos primeros caracteres coinciden, para que el usuario elija.
- Tabla de símbolos de AutoLISP (*Apropos*): Se utiliza para funciones de AutoLISP. Tras escribir los primeros caracteres de la palabra, se pulsa CTRL+SHIFT+/ en el teclado inglés o CTRL+SHIFT+ç en el teclado español. Existe un botón de herramienta que realiza esta misma función. Visual Lisp escribe la función o símbolo de AutoLISP más próximo que empieza por esos caracteres. Si se vuelve a pulsar la combinación y hay más funciones posibles, se ofrece una lista para seleccionar la deseada. Si hay más de 15 funciones posibles, o no se ha empezado ninguna palabra, se abre automáticamente la ventana de *Apropos*, explicada en el siguiente apartado.

TRECE.4.2.1. La herramienta *Apropos*

Esta herramienta, cuyo empleo se ha explicado para completar una función de AutoLISP empezada en el Editor de texto, presenta una utilidad más general. Consta de un cuadro de diálogo para establecer la búsqueda, y una ventana con las funciones encontradas.

Al cuadro de diálogo se accede desde el menú *View>Apropos Window*, desde la opción homónima del menú flotante, desde el botón de herramienta correspondiente, o pulsando CTRL+SHIFT+/ (teclado inglés) o CTRL+SHIFT+Ç (teclado español), siempre que no haya texto empezado en la posición del cursor o texto seleccionado. Las opciones del cuadro son:

Opción	Descripción
<i>Caracteres de búsqueda</i>	Uno o más caracteres para realizar la búsqueda. Por defecto, se ofrecen los de la última búsqueda.
<i>Match by Prefix</i>	Si se activa, sólo se localizan las funciones de AutoLISP que empiezan por los caracteres de búsqueda. Si no se activa, se localizan todas las funciones que presentan esos caracteres aunque no sea al comienzo.
<i>Use WCMATCH</i>	Permite utilizar caracteres comodín como el asterisco, para localizar nombres que cumplan un patrón.
<i>Downcase symbols</i>	Si se activa, la función seleccionada por el usuario, entre las que han sido localizadas en la búsqueda, se copiará en minúsculas. En caso contrario, se copiará en mayúsculas tal como aparecen en la lista de localizadas.
<i>Filter Value</i>	Abre un subcuadro para establecer un filtro que condicione la búsqueda: <i>All</i> : No hay filtro; se consideran por defecto sólo las funciones de AutoLISP. <i>Null value</i> : Sólo se consideran los símbolos cuyo valor es <i>nil</i> . <i>Nonull value</i> : Sólo se consideran los símbolos cuyo valor no es <i>nil</i> . <i>Functions</i> : Se consideran todas las funciones (también las de usuario). <i>User function</i> : Sólo se consideran las funciones de usuario (no AutoLISP). <i>Built-in function</i> : Sólo las funciones construidas en Visual Lisp. <i>EXSUBR</i> : Sólo las subrutinas externas.
<i>Filter Flags</i>	Abre un subcuadro para establecer condiciones de búsqueda a partir del tipo de atributo o señal de la función.
<i>OK</i>	Hace efectiva la búsqueda. Si existen funciones que cumplen las condiciones se abre una ventana con las encontradas.

Para seleccionar una de las funciones localizadas, se ilumina en la ventana de *Apropos* y, mediante el botón derecho del ratón, se elige *Copy to clipboard*. Posteriormente, se pega mediante *Paste* en la posición del cursor en el archivo de texto. La ventana de *Apropos* presenta tres botones de herramientas: el primero invoca el cuadro de diálogo por si se quiere repetir la búsqueda o establecer nuevas condiciones, el segundo traslada la función seleccionada a la ventana de rastreo *Trace* y el tercero proporciona ayuda en línea. Por último, el menú flotante de la ventana *Apropos* (botón derecho del ratón) contiene una serie de opciones:

Opción	Descripción
<i>Copy to clipboard</i>	Copia la función seleccionada al portapapeles.
<i>Inspect</i>	Abre una ventana de inspección con la función seleccionada.
<i>Print</i>	Pasa la función a la Consola para su evaluación.
<i>Symbol</i>	Llama al cuadro de diálogo de <i>Symbol service</i> .
<i>Copy</i>	Copia la función en la variable especial *obj* para su uso posterior.
<i>Add to Watch</i>	Añade la función a la lista en la ventana de <i>Watch</i> .
<i>Help</i>	Proporciona ayuda en línea sobre la función.

TRECE.4.2.2. Utilidades de gestión de texto

El contenido del archivo abierto en el Editor de texto se puede modificar mediante una serie de opciones y herramientas. La primera es un menú flotante especial que se invoca mediante CTRL+E. Sus opciones son:

Opción de menú	Descripción
<i>Indent Block</i>	Sangra el bloque de texto seleccionado una posición de tabulador.
<i>Unindent</i>	Elimina una posición de tabulador del sangrado del bloque de texto.
<i>Prefix With</i>	Añade un texto al comienzo de todas las líneas del bloque de texto.
<i>Append With</i>	Añade un texto al final de todas las líneas del bloque de texto.
<i>Comment Block</i>	Convierte el bloque de texto en comentarios.
<i>Uncomment Block</i>	Deshace la conversión del bloque de texto en comentarios.
<i>Save Block As</i>	Copia el bloque de texto a un archivo (por defecto extensión .LSP)
<i>Uppcase</i>	Convierte a mayúsculas todo el texto del bloque seleccionado.
<i>Downcase</i>	Convierte a minúsculas todo el bloque de texto.
<i>Capitalize</i>	Pone en mayúsculas la primera letra de cada palabra.
<i>Insert Date</i>	Inserta la fecha actual (por defecto en el formato MM/DD/AA).
<i>Insert Time</i>	Inserta la hora actual (por defecto en el formato HH/MM/SS).
<i>Format Date/Time</i>	Permite cambiar el formato de fecha y hora.
<i>Sort Block</i>	Ordena alfabéticamente las líneas del bloque de texto.
<i>Insert File</i>	Inserta en la posición del cursor el contenido de un archivo de texto.
<i>Delete to EOL</i>	Borra el texto desde la posición del cursor hasta el final de la línea.
<i>Delete Blanks</i>	Borra tabuladores y espacios en blanco sobrantes en la línea actual.

En la siguiente tabla, se enumeran las diferentes teclas aceleradoras para gestionar el texto en la ventana del Editor:

Desplazamientos del cursor

Teclas	Efecto
CTRL+← y CTRL+→ INICIO y FIN CTRL+FIN y CTRL+INICIO	Una palabra hacia la izquierda o hacia la derecha. Comienzo o final de línea. Final del archivo de texto o inicio del archivo de texto.
CTRL+↑	Desplaza una línea hacia arriba el contenido de la ventana.
CTRL+↓	Desplaza una línea hacia abajo el contenido de la ventana.
RE PÁG y AV PÁG	Desplaza una página hacia arriba o abajo el contenido de la ventana.
CTRL+[(teclado inglés) CTRL+` (teclado español)	Salta al paréntesis izquierdo abierto al mismo nivel que la posición del cursor. Después, a los sucesivos, desde los interiores a los exteriores.
CTRL+] (teclado inglés) CTRL+; (teclado español)	Salta al paréntesis derecho cerrado al mismo nivel que la posición del cursor. Después, a los sucesivos, desde los interiores a los exteriores.

Selección de texto

Teclas	Efecto
SHIFT+← y SHIFT+→	Añade o elimina de la selección un carácter a la izquierda o a la derecha.
CTRL+SHIFT+← y CTRL+SHIFT+→	Añade o elimina de la selección una palabra a la izquierda o a la derecha.
SHIFT+↓ y SHIFT+↑	Añade o elimina de la selección una línea hacia abajo o hacia arriba.
CTRL+SHIFT+↓ y CTRL+SHIFT+↑	Añade o elimina el trozo de línea de abajo o arriba hasta el cursor.
SHIFT+FIN y SHIFT+INICIO	Amplía la selección hasta final o inicio de línea actual y de las sucesivas.
SHIFT+RE PÁG y SHIFT+AV PÁG	Añade o elimina de la selección una página arriba o abajo de la ventana.
CTRL+SHIFT+[(teclado inglés) CTRL+SHIFT+` (teclado español)	Amplía la selección hasta el paréntesis izquierdo abierto al mismo nivel del cursor. Después a los sucesivos desde los interiores a los exteriores.
CTRL+SHIFT+] (teclado inglés) CTRL+SHIFT+; (teclado español)	Amplía la selección hasta el paréntesis derecho cerrado al mismo nivel del cursor. Después a los sucesivos desde los interiores a los exteriores.
ALT+INTRO	Salta el cursor al principio o final del texto seleccionado.

Corrección de texto

Teclas	Efecto
CTRL+RETROCESO	Borra la palabra a la izquierda.
SHIFT+RETROCESO	Borra la palabra a la derecha.
CTRL+SUPR	Borra la palabra a la derecha.

Teclas	Efecto
CTRL+E	Con su opción <i>Delete to EOL</i> borra desde cursor hasta final de línea.

Un texto seleccionado en el Editor se puede arrastrar mediante el ratón hasta otra posición del archivo. Si se mantiene pulsada la tecla CTRL se copiará, en caso contrario se desplazará. Para desplazar o copiar a otras ventanas, se utiliza el portapapeles de Windows.

Para buscar o reemplazar texto existen utilidades en el menú *Search* o en la barra de herramientas del mismo nombre. Desde *Search>Find*, o el botón correspondiente, se llama a un cuadro de diálogo con opciones de búsqueda. Son las siguientes:

Opción	Descripción
<i>Find What</i>	Cadena de texto que se localizará.
<i>Search Current selection</i>	Busca sólo en el texto seleccionado.
<i>Search Current file</i>	Busca en todo el archivo de la ventana actual.
<i>Search Find in project</i>	Busca en los archivos del proyecto especificado.
<i>Search Find in files</i>	Busca en todos los archivos de la carpeta especificada. Se puede indicar que incluya también los subdirectorios.
<i>Direction</i>	Se especifica la dirección de búsqueda (hacia arriba o hacia abajo).
<i>Match whole Word only</i>	Sólo se buscan palabras completas.
<i>Match case</i>	Se tienen en cuenta las mayúsculas y minúsculas.
<i>Mark instances</i>	Se añade una marca al texto localizado, para poder volver a él.
<i>Find</i>	Ejecuta la búsqueda, deteniéndose en el primer texto localizado. Mediante F3 se pueden ir localizando los siguientes textos que coincidan.

El texto localizado en cada búsqueda, se añade a una lista desplegable en la barra de herramienta de *Find*. Cualquiera de los textos de la lista se puede seleccionar y efectuar una nueva búsqueda desde un botón específico a la derecha.

Para buscar y reemplazar un texto por otro, desde el menú *Search>Replace*, o el botón correspondiente, se llama a un cuadro de diálogo similar al de búsqueda, que añade una casilla para indicar el nuevo texto y dos botones para reemplazar los textos uno a uno o todos de una vez.

Una última herramienta para el desplazamiento por el texto es la posibilidad de introducir hasta 32 marcas (*Bookmarks*) en diferentes posiciones, para después localizarlas cómodamente. Cada ventana tiene su propio gestor de marcas. Las opciones disponibles son:

Opción	Descripción
<i>Crear marca</i>	Para situar una marca en la posición del cursor se pulsa el botón de herramienta correspondiente o las teclas ALT+. (punto). Pulsando otra vez, se elimina la marca.
<i>Mover el cursor hasta la marca previa</i>	Se pulsa el botón de herramienta correspondiente o las teclas CTRL+, (coma). El cursor salta hasta la marca previa, en el orden en que se han creado, no el de posición en el archivo.

Opción	Descripción
<i>Mover el cursor hasta la marca siguiente</i>	Se pulsa el botón de herramienta correspondiente o las teclas <code>CTRL+.</code> (punto). El cursor salta hasta la marca siguiente, en el orden en que se han creado, no el de posición en el archivo.
<i>Seleccionar texto hasta la marca previa</i>	Se pulsan las teclas <code>CTRL+SHIFT+.</code> (coma). El texto entre posición del cursor y marca previa queda seleccionado.
<i>Seleccionar texto hasta la marca siguiente</i>	Se pulsan las teclas <code>CTRL+SHIFT+.</code> (punto). El texto entre posición del cursor y marca siguiente queda seleccionado.
<i>Eliminar marca</i>	Se sitúa el cursor en la posición de la marca y se pulsa el mismo botón de herramienta de creación o las teclas <code>ALT+.</code> (punto). Para eliminar todas las marcas de una vez, existe un botón de herramienta específico.

TRECE.4.2.3. Formateado del código fuente

Formatear un texto consiste en distribuirlo de una manera adecuada para su correcta legibilidad. En Visual Lisp se puede actuar de dos maneras:

- **Formatear todo el texto del archivo.** Para ello se elige la opción de menú *Tools>Format AutoLISP in Editor*, o el botón de herramienta correspondiente.

- **Formatear el texto seleccionado.** Una vez iluminado el texto, se elige la opción de menú *Tools>Format AutoLISP in Selection*, o el botón de herramienta correspondiente.

Durante el formateo se comprueba el balance de paréntesis, mostrándose un mensaje de advertencia si se detecta algún error. Sin embargo, los errores no se corrigen; esto se hace mediante la herramienta de chequeo (*Check*) explicada en la siguiente sección. Visual Lisp aplica automáticamente cuatro estilos de formateo, según el tipo de expresiones que encuentre:

- Formateo sencillo (*Plane*). Todos los elementos se sitúan en la misma línea de texto, separados por simples espacios. Visual Lisp lo aplica cuando la expresión cabe en una línea (de acuerdo con el valor de *Right text margin* del cuadro de diálogo explicado más adelante), cuando no sobrepasa el margen de impresión (de acuerdo con el valor de *Maximun length for plane expression* del cuadro de diálogo) y cuando no tiene comentarios que se extienden más de una línea.
- Formateo ancho (*Wide*). El primer argumento de cada función se mantiene en línea, pero el resto de argumentos se sitúan en nuevas líneas, debajo del primero. Visual Lisp lo aplica cuando no se cumplen las condiciones del formateo sencillo y además el primer elemento es una función y su longitud no sobrepasa el valor de *Maximun wide-style car length* del cuadro de diálogo.
- Formateo estrecho (*Narrow*). Cada argumento, incluido el primero, se sitúa en nueva línea debajo de la función, sangrados de acuerdo con el valor de *Narrow style indentation* del cuadro de diálogo. Visual Lisp lo aplica a determinadas funciones como `PROGN`.
- Formateo en columna (*Column*). Todos los elementos se sitúan en columna. Visual Lisp lo aplica a funciones como `COND`.

El estilo de formateo se controla desde el menú *Tools>Environment Options>AutoLISP Format options*. Aparece un cuadro de diálogo con las siguientes opciones:

Opción	Descripción
<i>Right text margin</i>	Número máximo de caracteres en una línea.
<i>Narrow style indentation</i>	Caracteres de sangrado para el formateo estrecho.
<i>Maximun wide-style car length</i>	Máximo sangrado para el formateo ancho.
<i>Single semicolon comment indentation</i>	Posición para los comentarios.
<i>Closing paren style</i>	Controla la situación de los paréntesis de cierre.
<i>Insert tabs</i>	Inserta tabuladores en los sangrados.
<i>Save formatting options in source file</i>	Inserta un comentario al final del archivo, con los formateos efectuados.
<i>Insert form-closing comment</i>	Inserta un comentario al final de cada función que contiene otras anidadas. Por ejemplo <code>;_end of DEFUN.</code>
<i>Form-closing comment prefix</i>	Prefijo para los comentarios de la casilla anterior. Por defecto es <code>end of.</code>
<i>More options</i>	Despliega nuevas opciones ampliando el cuadro de diálogo.
<i>Preserve existing line breaks</i>	Si alguna expresión se encuentra partida al final de la línea, se conserva esta partición al formatear.
<i>Split comments</i>	Si hay comentarios al final de una línea, se subdividen para que queden ajustados al margen derecho, distribuyéndose en varias líneas.
<i>Setting Case for symbols</i>	Las funciones y símbolos de AutoLISP (<i>Protected</i>) y las de usuario (<i>UnProtected</i>) pueden ser convertidas a minúsculas (<i>downcase</i>) o mayúsculas (<i>UPCASE</i>) durante el formateo.
<i>Longlist format style</i>	Controla el formateo de las expresiones con más de cinco argumentos.

Para que las opciones de formateo modificadas se mantengan en futuras sesiones de Visual Lisp, se pueden guardar desde el menú *Tools>Save Settings*.

TRECE.4.2.4. Chequeo de errores de sintaxis

Antes de proceder a la depuración mediante la ejecución del programa, desde el Editor de texto se puede realizar un primer chequeo para detectar si el balance de paréntesis es correcto. Ya se ha visto en el apartado anterior que al formatear un texto, se detectan los paréntesis no balanceados. El formateador ofrece la posibilidad de añadir esos paréntesis, pero lo hace generalmente al final de línea o de expresión, sin permitir un control sobre la situación adecuada de los mismos.

Incluso antes de realizar ningún chequeo, el código de colores del Editor de texto facilita una primera detección de errores evidentes. Por ejemplo, si el color magenta se extiende más de lo previsto, es porque hay una cadena de texto sin comillas de cierre. Si un nombre de función aparece en negro y no en azul, es porque se ha escrito mal y no se reconoce como función de AutoLISP. También después de formatear, si aparecen una serie de expresiones en columna, puede denotar la falta de algún paréntesis de cierre y por ello se han considerado todas las expresiones como argumentos de una función inicial no cerrada.

Se recuerda además que existen teclas aceleradoras de desplazamiento de cursor, que ayudan a evidenciar los anidamientos de paréntesis.

Una vez detectados estos errores más evidentes, el usuario puede utilizar la herramienta de chequeo (*Check*) de Visual Lisp. Existen dos opciones:

- **Chequear todo el texto del archivo.** Para ello se elige la opción de menú *Tools>Check Text in Editor*, o el botón de herramienta correspondiente.
- **Chequear el texto seleccionado.** Una vez iluminado el texto, se elige la opción de menú *Tools>Check Selection*, o el botón de herramienta correspondiente.

Durante el chequeo, Visual Lisp examina todas las expresiones y detecta los errores de sintaxis (no los de funcionamiento), escribiéndolos en una ventana denominada *Build Output* que abre automáticamente. Los errores se resaltan sobre un fondo de color ciano. Si se hace doble clic sobre el mensaje de error, Visual Lisp vuelve al Editor y sitúa el cursor al comienzo de la expresión que ha producido el error, iluminándola entera. El usuario corrige la expresión y vuelve a ejecutar el chequeo hasta que dejan de detectarse errores.

La última operación, una vez corregidos todos los errores de sintaxis, sería cargar y ejecutar el programa para comprobar su correcto funcionamiento. También aquí existen dos posibilidades:

- **Cargar todo el texto del archivo.** Para ello se elige la opción de menú *Tools>Load Text in Editor*, o el botón de herramienta correspondiente.
- **Cargar el texto seleccionado.** Una vez iluminado el texto, se elige la opción de menú *Tools>Load Selection*, o el botón de herramienta correspondiente.

El resultado es que se cargan en memoria las expresiones indicadas. Si contienen instrucciones de ejecución directa, éstas se realizan. Si contienen definiciones de función, debe llamarse a éstas desde la Consola para ejecutarlas. Los errores de funcionamiento que se puedan detectar, requieren de las herramientas de depuración para su corrección. Esto se explica en la sección siguiente.

TRECE.5. DEPURACIÓN DE PROGRAMAS

Un procedimiento de depuración sencillo de un programa se podría resumir en una serie de etapas características:

- 1) Situar en el texto fuente, uno o más puntos de ruptura (*Breakpoints*) en los sitios donde se desea examinar en detalle la rutina.
- 2) Ejecutar todo o parte del programa. Este se detendrá en el primer punto de ruptura.
- 3) Seleccionar las funciones o variables que se desea examinar y añadirlas a la lista de la ventana de seguimiento *Watch*.
- 4) Ejecutar las expresiones paso a paso mediante las herramientas *Step*.
- 5) En la ventana *Watch* ir comprobando los valores que adoptan las funciones y variables seleccionadas.
- 6) Una vez comprobado el funcionamiento en detalle de esas determinadas zonas del programa, continuar con el resto o abortarlo mediante las herramientas de *Continue*, *Quit* y *Reset*.

El proceso de depuración más elemental, consiste pues en situar puntos de ruptura. El trabajo con dichos puntos dispone de una serie de opciones:

- **Creación y eliminación.** Cada punto se crea (o elimina si ya existiera) desde el menú *Debug>Toggle Breakpoint* o desde el botón correspondiente. Existe una opción *Debug>Clear All Breakpoints* para eliminar todos los puntos de ruptura. Al ejecutar un programa, se detiene en cada punto de ruptura y entra en la modalidad de *Break Loop* explicada más adelante.

- **Desactivación.** Si un punto no va a ser usado en este momento, pero se desea mantenerlo para usarlo más adelante, se puede desactivar. Para ello, se sitúa el cursor sobre él, se pulsa el botón derecho del ratón y, en el menú flotante, se elige la opción *Breakpoint service*. Aparece un cuadro de diálogo con botones para desactivar, eliminar o mostrar el punto de ruptura.

- **Control de colores.** El punto de ruptura se muestra con un rectángulo rojo para su fácil identificación. Si está desactivado, se muestra con un fondo gris. Estos colores se puede modificar desde el menú *Tools>Windows Attributes>Configure Current* escogiendo *:BPT-ACTIVE* o *:BPT-DISABLE* en la lista desplegable.

- **Gestión de todos los puntos.** Todos los puntos de ruptura de todos los archivos abiertos se pueden gestionar desde el menú *View>Breakpoints Window*. Se muestra un cuadro de diálogo con una lista de todos los puntos, ofreciendo el archivo fuente y la posición (con signo más los activados y signo menos los desactivados). El botón *Edit* muestra el cuadro de *Breakpoint service* ya explicado. El botón *Show* muestra la posición del punto seleccionado en la lista, desplazando el cursor hasta él. El botón *Delete* elimina el punto seleccionado, y *Delete All* elimina todos los puntos de la lista.

Los puntos de ruptura se pierden si se elimina el trozo de texto donde están, si se edita el archivo fuente con otra aplicación diferente de Visual Lisp, y también si se formatea el texto con alguna de las herramientas de Visual Lisp. Si se modifica el texto fuente y se ejecuta después sin haber cargado de nuevo en memoria el archivo, aparece un mensaje de advertencia y no se permite continuar con la depuración. Es preciso cargar de nuevo el programa.

Para otros procesos de depuración más completos o específicos, Visual Lisp proporciona dos modos de depuración (*Break Loop* y *Trace*) que incluyen las siguientes herramientas:

- **Modo Break Loop.** Detiene la ejecución del programa en un punto de ruptura y permite examinar paso a paso el valor de las funciones y variables.

- **Facilidad Trace Stack.** Permite examinar las últimas acciones del programa antes de detenerse en un punto de ruptura o abortarse al surgir un error.

- **Modo Trace.** Corresponde a la función de AutoLISP *TRACE*, que marca con atributos de rastreo las funciones definidas en el programa.

- **Ventana Watch.** Proporciona en tiempo real los valores de funciones y variables.

- **Servicio de Símbolos.** Proporciona asistencia para la depuración de símbolos.

- **Inspector.** Proporciona información detallada de un elemento, descendiendo al máximo nivel de anidación si así se solicita.

TRECE.5.1. Modo de depuración *Break Loop*

La estructura básica en los programas de AutoLISP son las expresiones. La ejecución normal de un programa consiste en leer, evaluar y escribir resultados a partir de funciones y variables. Esto es lo que se conoce como nivel primario de ejecución o *Top Level*. Cuando desde Visual Lisp se interrumpe la ejecución de un programa mediante alguna de las herramientas disponibles, se entra en un ciclo de ruptura o *break loop*. El control pasa a la Consola y el *prompt* muestra el nivel de anidación del ciclo de ruptura, por ejemplo:

1\$

Desde un nivel de ruptura se puede iniciar otro anidado y así sucesivamente. Mientras se permanece en esta modalidad, el usuario puede examinar y modificar los valores de variables dentro de la porción de programa involucrada en la ruptura, para analizar su comportamiento (por ejemplo mediante funciones *SETQ* introducidas desde la Consola). Pero no se puede modificar el texto fuente ni pasar a **AutoCAD**. Cuando finaliza el examen, una serie de herramientas permiten salir al nivel superior de ruptura o directamente al nivel primario, o bien continuar la ejecución del resto del programa. Sólo cuando la ruptura se debe a un error, no se permite continuar la ejecución del mismo y se debe salir forzosamente del ciclo de ruptura.

Las opciones disponibles para especificar el tipo de ruptura son:

Opción	Descripción
<i>Toggle Breakpoint</i>	Consiste en situar manualmente puntos de ruptura en las expresiones deseadas del programa, como ya se ha explicado.
<i>Stop Once</i>	Se activa desde el menú <i>Debug>Stop Once</i> y hace que Visual Lisp rompa incondicionalmente la ejecución de un programa al encontrar la primera expresión depurable.
<i>Break on Function Entry</i>	Consiste en marcar expresamente funciones para su depuración. Visual Lisp provocará una ruptura cada vez que se encuentre con esas funciones durante la ejecución de un programa. Las marcas de depuración se colocan, tras seleccionar una función o variable en el Editor, desde el menú <i>View>Symbol Service</i> activando la casilla <i>Trace</i> .
<i>Modo de depuración Top Level</i>	Hace que las rupturas para depuración se produzcan durante la propia carga del archivo de programa, antes de evaluar las funciones de anidamiento más exterior (como <i>DEFUN</i>). Se activa desde el menú <i>Tools>Environment Options>General Options</i> , desde la pestaña <i>Diagnostic</i> , desactivando la casilla <i>Do not debug top-level</i> .
<i>Modo Animate</i>	Ejecuta el programa paso a paso, ofreciendo en la ventana Watch los resultados de todas las funciones y variables según se producen. Se activa desde el menú <i>Debug>Animate</i> . La velocidad de ejecución paso a paso se controla desde el menú <i>Tools>Environment Options>General Options</i> , desde la pestaña <i>Diagnostic</i> , indicando el valor en milisegundos en la casilla <i>Animation delay</i> .
<i>Break on Error</i>	Se activa desde el menú <i>Debug>Break On Error</i> y

Opción	Descripción
	hace que Visual Lisp entre automáticamente en el modo <i>Break Loop</i> en cuanto se produce un error, sin necesidad de haber creado puntos de ruptura.

Mientras se permanece dentro de un ciclo de ruptura, la ejecución del programa se controla mediante una serie de opciones o botones. Son:

Opción	Descripción
<i>Continue</i>	Continúa la ejecución del programa hasta el siguiente punto de ruptura, o hasta el final si no hay más.
<i>Quit</i>	Sale del actual nivel de ruptura hasta el inmediato superior.
<i>Reset</i>	Sale directamente al nivel primario (<i>Top Level</i>).
<i>Step into</i>	Evalúa la siguiente expresión anidada en el punto de ruptura y, sucesivamente, las siguientes expresiones incluyendo las anidadas, una a una.
<i>Step over</i>	Evalúa las siguientes expresiones del mismo nivel de anidamiento que el punto de ruptura, sin pasar por las interiores que pudieran tener anidadas.
<i>Step out</i>	Evalúa de golpe todo el bloque desde el punto de ruptura hasta el final de la función en cuyo interior se ha producido.
<i>Last Break</i>	Ilumina el último trozo de texto fuente evaluado durante el modo de ruptura.
<i>Stopped</i>	Informa, mediante una línea roja en el propio botón, si la última evaluación ha finalizado en un paréntesis de apertura o de cierre.

TRECE.5.2. Modo de depuración *Trace*

Visual Lisp dispone de una herramienta de depuración denominada *Trace Stack*. Consiste en una pila donde va almacenando todas las evaluaciones efectuadas cuando se ejecuta una función o una expresión desde la Consola. De esta manera, cuando ocurre una interrupción debido a un error o por haberse encontrado un punto de ruptura, el historial de lo ocurrido en el programa antes de la interrupción se encuentra a disposición del usuario en una serie de registros en el *Stack*. Estos pueden ser de cinco tipos:

- **Llamada de función.** Se muestra el nombre y los valores de cada argumento.
- **Origen de los registros.** Palabra clave que indica el origen de los registros de *Trace Stack*. Se sitúa al principio o final de la lista de registros y puede ser :*TOP-COMMAND*, :*USER-INPUT*, :*BREAK-POINT*, :*ERROR-BREAK*, etcétera.
- **Registro primario (*Top*).** Indica una acción iniciada expresamente en la Consola en el nivel *Top* o durante la carga de un programa o texto seleccionado.
- **Registro Lambda.** Cuando se trata de la llamada a una función *LAMBDA*.
- **Funciones especiales.** Las llamadas a *FOREACH* y *REPEAT* no muestran los valores de los argumentos.

Para examinar la ventana de *Trace Stack* se elige el menú *View>Trace Stack* o el botón correspondiente. Esta ventana tiene dos botones de herramienta propios: actualizar el contenido y copiar la lista de elementos a la ventana de *Trace* (que ofrece así un historial de todas las veces que se ha utilizado el *Trace Stack*). Además, una vez seleccionado un elemento, existe un menú flotante accesible desde el botón derecho del ratón con las siguientes opciones:

Opción de menú	Descripción
<i>Inspect</i>	Invoca el cuadro de diálogo de inspección (<i>Inspector</i>).
<i>Print</i>	Escribe el elemento en la Consola.
<i>Function Symbol</i>	Si el elemento es una llamada de función, muestra el cuadro de diálogo de <i>Symbol Service</i> .
<i>Copy</i>	Copia el valor del elemento a la variable de sistema <i>*obj*</i> .
<i>Local Variables</i>	Si el elemento es una llamada de función, muestra el cuadro de diálogo de <i>Frame Binding</i> con los valores de variables en la llamada a la función. Esta ventana tiene su propio menú flotante, con opciones similares.
<i>Source Position</i>	Sitúa el cursor en la posición del elemento que ha originado la depuración (por ejemplo, donde está el punto de ruptura o donde se ha producido el error) en el texto fuente, iluminando éste.
<i>Call point Source</i>	Sitúa el cursor en la posición del elemento en el texto fuente.

Además de la ventana de *Trace Stack*, existe una ventana de *Trace* donde se muestran todas las llamadas a las funciones que han sido marcadas con un atributo de rastreo.

TRECE.5.3. Ventana de seguimiento *Watch*

Consiste en una ventana a la que se añaden las funciones o variables que se desea someter a seguimiento. Esto se hace situando el cursor sobre el nombre de la función o variable (en la ventana del Editor, en la Consola, etc.) y escogiendo la opción *Add Watch* del menú desplegable *Debug* o del menú flotante (botón derecho del ratón), y también mediante el botón correspondiente de barra de herramientas.

En la ventana *Watch* se ofrece en tiempo real los valores que van adoptando todas las funciones y variables, durante la ejecución o depuración del programa. Esta ventana tiene cuatro botones de herramienta propios: añadir otro elemento a la lista, eliminar todos los elementos para empezar una lista nueva, ordenar alfabéticamente la lista y copiar la lista a la ventana de *Trace*. Además, cuando hay un elemento seleccionado en la lista, el botón derecho del ratón despliega un menú flotante específico, con las opciones:

Opción	Descripción
<i>Inspect value</i>	Invoca el cuadro de diálogo de inspección (<i>Inspector</i>).
<i>Copy value</i>	Copia el valor del elemento a la variable de sistema <i>*obj*</i> .
<i>Print value</i>	Escribe el valor del elemento en la Consola, precedido de <code>'</code> (QUOTE).
<i>Symbol...</i>	Llama al cuadro de diálogo de <i>Symbol Service</i> .
<i>Apropos...</i>	Llama al cuadro de diálogo de <i>Apropos</i> .

Remove from Watch

Elimina el elemento de la lista.

TRECE.5.4. Cuadro de diálogo de servicio de símbolos *Symbol Service*

Proporciona una herramienta para facilitar la depuración de símbolos como nombres de funciones y variables. Se accede situando el cursor sobre un nombre o iluminando éste en el Editor, y escogiendo el menú *View>Symbol Service* o el botón correspondiente. También se accede desde opciones del menú flotante contextual en las ventanas de *Trace Stack* o de *Watch*, como ya se ha explicado. Este cuadro tiene cuatro botones de herramienta propios: añadir el símbolo a la ventana *Watch*, llamar al cuadro de diálogo de *Inspect*, mostrar la definición del símbolo y una ayuda en línea.

Sus opciones son:

Opción	Descripción
<i>Name</i>	Nombre del símbolo que depurar.
<i>Value</i>	Valor del símbolo o su cadena de definición interna.
<i>Trace</i>	Marca una función de usuario con un atributo de rastreo (<i>Tr</i>). Durante la ejecución del programa, todas las llamadas a esta función se mostrarán en la ventana de <i>Trace</i> .
<i>Debug on Entry</i>	Origina un punto de ruptura en cada llamada a la función (atributo <i>De</i>).
<i>Protect Assign</i>	Asigna una protección al símbolo para que no pueda ser sobrescrito (mediante <i>SETQ</i> o <i>DEFUN</i> , por ejemplo). Todas las funciones y símbolos propios de AutoLISP están protegidos (atributo <i>Pa</i>).
<i>Export to ACAD</i>	Exporta el símbolo a AutoCAD , que lo tratará como una subrutina externa (atributo <i>Ea</i>).

TRECE.5.5. Ventana de inspección de objetos *Inspect*

Permite examinar y modificar los objetos de AutoLISP (listas, números, cadenas, variables...) y **AutoCAD** (entidades de dibujo, conjuntos de selección...). Se accede situando el cursor sobre un nombre, o iluminando éste, y escogiendo el menú *View>Inspect* o el botón correspondiente. También se accede desde opciones del menú flotante contextual en las ventanas de *Trace Stack* o de *Watch*, o desde un botón de otros cuadros como *Simbol Service*, cosa que ya se ha explicado. Si no se selecciona ningún símbolo, se solicita el nombre de éste ofreciendo una lista desplegable con los 15 últimos símbolos inspeccionados.

Cada objeto inspeccionado abre una nueva ventana de inspección. Desde el menú *Window>Close Windows>Inspectors* se cierran de una vez todas las ventanas de inspección. La ventana de inspección de objetos ofrece tres elementos:

- **Caption.** Es el título del propio cuadro, donde se ofrece el tipo del objeto que se está inspeccionando.

- **Object line.** Muestra una representación del objeto en caracteres imprimibles. Existe un menú contextual específico (botón derecho del ratón) con opciones estándar.

- **Element list.** Lista de elementos componentes del objeto. Cada componente presenta el nombre (entre corchetes cuando puede ser modificado y entre llaves cuando no) y el valor. Cada elemento de la lista puede ser a su vez inspeccionado, desde un menú contextual (botón derecho del ratón) que contiene una serie de opciones estándar.

Los tipos de objetos y los valores se ofrecen en la siguiente tabla:

Tipo de objeto	Descripción
INT	Número entero. La lista de componentes muestra representaciones del número en binario, octal, hexadecimal, etc.
REAL	Número real. La lista de componentes se muestra vacía.
STRING	Cadena de texto. La lista de componentes muestra los caracteres. Cada uno puede examinarse como un entero a partir de su código ASCII.
SYMBOL	Símbolo. La lista muestra el valor, el nombre imprimible y los atributos (<i>Tr</i> , <i>Pa</i> , <i>De</i> o <i>Ea</i>), explicados en <i>Symbol Service</i> .
LIST	Lista de elementos. Se muestran todos los elementos. Si la lista no es propia de AutoLISP, se muestra el resultado de CAR y CDR aplicados a ella.
FILE	Descriptor de archivo. Se muestra el nombre, atributos de apertura, ID del archivo, posición en el archivo y señal de final EOF.
SUBR, EXSUBR y USUBR	Funciones de AutoLISP externas y de usuario. Se muestra el nombre y los argumentos.
ENAME	Nombre de entidad de dibujo. Se muestra la lista de datos completa.
PICKSET	Conjunto de selección. Se muestra la lista de nombres de todos los objetos del conjunto.

Para examinar la lista de datos de un objeto de dibujo de **AutoCAD**, se puede llamar a la ventana de Inspección para una variable que contenga un nombre de entidad. La lista de componentes de *Inspect* mostrará todos los datos de la entidad, siempre que desde el menú *Tools>Environment Options>General Options*, en la pestaña *Diagnostic*, esté activada la casilla *Inspect drawing objects verbosely*.

Para abrir una ventana de inspección de todos los objetos del dibujo actual de **AutoCAD**, se puede hacer desde el menú *View>Browse Drawing Database*. Existen opciones para examinar todas las entidades de dibujo, tablas de símbolos, definiciones de bloques, objetos designados o datos extendidos. En cada caso se mostrará una lista con todos los objetos encontrados. A partir de cada elemento de la lista, mediante el menú contextual (botón derecho del ratón) se pueden inspeccionar sus componentes (por ejemplo los vértices de una polilínea o entidades de un bloque) y así sucesivamente.

TRECE.6. CONSTRUCCIÓN Y GESTIÓN DE APLICACIONES

Los programas e instrucciones de AutoLISP son sometidos a una compilación inmediata cuando se ejecutan desde Visual Lisp. Se trata de un mecanismo semejante al intérprete de expresiones que tiene **AutoCAD**. Sin embargo, Visual Lisp proporciona una importante herramienta de compilación de archivos, que permite encriptar y proteger el programa, y además ejecutarlo desde fuera del entorno de Visual Lisp. Los archivos de programas compilados tienen la extensión *.FAS*.

Cuando el usuario diseña una aplicación compleja, con varios archivos diferentes de programas que funcionan en paralelo, Visual Lisp introduce el concepto de proyecto, que se estudiará un poco más adelante.

TRECE.6.1. Compilación de archivos de programa

La compilación de un archivo de AutoLISP se hace mediante el comando `vlisp-compile`, desde la Consola. La sintaxis es:

```
(vlisp-compile 'modo "Archivo_Fuente" ["Archivo_Salida"])
```

Donde *modo* se debe indicar precedido de apóstrofo (QUOTE). El archivo fuente es el de AutoLISP y el de salida el compilado que obtener. Si no se indica éste último, su nombre será el mismo que el del fuente. Los modos son los de la siguiente tabla:

Modo	Descripción
<code>'st</code>	Modo estándar. Produce el archivo de salida más pequeño. Es el más indicado para programas que ocupan un solo archivo.
<code>'lsm</code>	Modo optimizado. Con compactación de nombres de funciones y variables, pero sin vínculos.
<code>'lsa</code>	Modo optimizado. Con vínculos entre funciones.

Durante la compilación la ventana de salida de aplicaciones *Build Output*, ofrece las incidencias y errores encontrados. Si se produce alguno, un doble clic sobre el mensaje sitúa el cursor automáticamente en el Editor, sobre el punto del código fuente que ha originado el error.

Como ya se ha dicho, un archivo individual compilado tiene la extensión `.FAS`. Estos archivos se cargan y ejecutan desde Visual Lisp como los no compilados, desde el menú *File>Load File* o mediante `LOAD`. Es posible compilar varios archivos `.FAS` y `.DCL` que formen parte de una misma aplicación, en un archivo ejecutable `.VLX`. Todos estos archivos necesitan el *Run Time System (RTS)* de Visual Lisp para su ejecución. Por lo tanto, para ser utilizados desde **AutoCAD**, deberá cargarse la aplicación `VLRTS.ARX` o `VLARTS.ARX` (ésta última con metodología *ActiveX*). Por último, es posible incluir todos los archivos de una aplicación en un módulo `ARX`. Este se carga desde **AutoCAD** como cualquier otra aplicación `ARX`, y por lo tanto no necesita del *RTS* de Visual Lisp para su ejecución.

Cuando un programa en AutoLISP llama a otras subrutinas externas definidas mediante aplicaciones `ADS` o `ARX`, debe crearse un archivo de definición de funciones externas `.XDF` (generalmente con el mismo nombre que la aplicación y en el mismo directorio, aunque también pueden definirse archivos genéricos `XLOAD.XDF` y `ARXLOAD.XDF` en la carpeta de Visual Lisp). Esto se puede hacer proporcionando el nombre de la aplicación externa, con el comando externo `MAKE-XDF` que debe ejecutarse desde **AutoCAD**, tras cargar el archivo `MAKE-XDF.LSP`. El archivo `.XDF` creado contiene el número de versión de **AutoCAD**, el nombre de la aplicación externa y los nombres de subrutinas externas que contiene de dicha aplicación.

Los pasos para ejecutar desde **AutoCAD** un archivo compilado son:

1) Cargar el *RTS* de Visual Lisp. Desde **AutoCAD** se carga la aplicación `VLRTS.ARX` o `VLARTS.ARX`.

2) Inicializar el *RTS*. Se ejecuta la función `(VLRTS-INIT)` o `(VLARTS-INIT)`.

3) Cargar el archivo compilado. Se hace mediante el comando externo `VL-LOAD`. Permite cargar archivos `.LSP`, `.FAS` y `.VLX`. Todas las funciones de los archivos cargados, que

hayan sido exportadas a **AutoCAD**, podrán ser utilizadas a partir de ese momento como nativas de **AutoCAD**. Si no han sido exportadas hay que utilizar el comando externo VL-EVAL.

TRECE.6.2. Creación de módulos de aplicación

Un módulo ejecutable ARX contiene todos los archivos con programas y además el *RTS* de Visual Lisp. Esto permite ejecutarlo desde **AutoCAD** sin necesidad de cargar e inicializar el *RTS*. Para crear módulos, Visual Lisp dispone de un completo asistente en el menú *File>Make Application*. La opción *New Application Wizard* llama a un asistente para crear un archivo de descripción .MKP que después se usará al construir la aplicación (tanto ARX como VLX). Este asistente guía paso a paso en la creación del archivo de descripción. Posteriormente, la opción *Build Application from Make File* creará la aplicación (ARX o VLX) utilizando el archivo de descripción. Hay una tercera opción *Existing Application Wizard* para modificar un archivo de descripción existente y después volver a construir la aplicación modificada.

TRECE.6.3. Gestión de proyectos

El concepto de proyecto en Visual Lisp engloba básicamente una serie de archivos fuente y una serie de archivos con instrucciones sobre su compilación. Todos los tipos de archivos se resumen en la siguiente tabla:

Tipo de archivo (extensión)	Descripción
.DSK	Datos del entorno de Visual Lisp y especificaciones de sus ventanas.
.FAS	Archivos de AutoLISP compilados en Visual Lisp.
.LSP	Archivos fuente de AutoLISP.
.DCL	Archivos fuente de cuadros de diálogo en DCL.
.OB	Códigos de compilación, usado internamente por Visual Lisp.
.PDB	Base de datos de proyectos, usado internamente por Visual Lisp.
.PRJ	Definición de proyecto, con todos los archivos que incluye y ciertos parámetros y reglas para la compilación final.
._XX	Copias de seguridad de los archivos fuente, como ._LS (para archivos .LSP) o ._DC (para archivos .DCL), por ejemplo.
.ARX y .VLX	Archivos resultantes de la compilación.
.C , .CPP, .CCH, .HPP, .HH	Archivos fuente de códigos de los lenguajes.
.MKP	Describe el contenido de la aplicación.
.SQL	Instrucciones SQL; son reconocidas desde el Editor de Visual Lisp.
.XDF	Lista de funciones externas para ser llamadas desde aplicaciones.
.XDV	Lista de funciones de inicialización generadas durante la compilación.

Las opciones para crear y gestionar proyectos (que son archivos .PRJ) se encuentran en el menú *Project*. Al crear un nuevo proyecto, se muestra el cuadro de diálogo de propiedades para especificar los archivos que se quieren incluir y el orden de los mismos, además de una serie de parámetros de creación.

Al cargar un proyecto existente, se abre una ventana con los nombres de todos los archivos incluidos y una serie de botones de herramienta para mostrar el cuadro de diálogo de propiedades, cargar los archivos compilados .FAS del proyecto, cargar los archivos fuente no compilados, compilar de nuevo los archivos que han sido modificados y compilar de nuevo todos los archivos del proyecto (modificados o no). Además, existe un menú contextual (botón derecho del ratón) con opciones para actuar sobre cada elemento de la lista. Es posible seleccionar varios elementos a la vez.

TRECE.7. UTILIZACIÓN DE OBJETOS *ActiveX*

ActiveX Automation es un mecanismo por el que las aplicaciones de Windows que lo integran, entre ellas **AutoCAD**, pueden ser gestionadas desde lenguajes de programación orientados a objeto como C++ o Visual Basic. En el **MÓDULO DOCE** de este curso se dan más detalles sobre la estructuración *ActiveX* de los objetos de **AutoCAD**, las propiedades de cada uno y los métodos o eventos que pueden realizar. Aquí simplemente se muestran los pasos para Visual Lisp a la hora de establecer la conexión del entorno con los objetos por medio de *ActiveX Automation*. Estos pasos nos sonarán a los que utilizábamos con el VBA de **AutoCAD**:

Primero hemos de establecer una conexión con la aplicación **AutoCAD** para que sus objetos sean utilizables desde *ActiveX*. Esto se hace desde Visual Lisp mediante la función `VLAX-GET-ACAD-OBJECT`. Por ejemplo:

```
(SETQ AcadObj (VLAX-GET-ACAD-OBJECT))
```

La variable `AcadObj` contiene ahora la conexión *ActiveX* de la aplicación **AutoCAD**. Este tipo de datos se denomina `VLA-OBJECT`.

Segundo tenemos que establecer la siguiente conexión en la jerarquía de objetos de **AutoCAD**, que es el documento de dibujo activo:

```
(SETQ AcadDoc (VLA-GET-ACTIVEDOCUMENT AcadObj))
```

Tercero establecer la conexión al Espacio (Modelo o Papel) de **AutoCAD** (en el ejemplo Espacio Modelo):

```
(SETQ AcadEsp (VLA-GET-MODELSPACE AcadDoc))
```

4 – Establecer la última conexión al objeto de dibujo deseado (en este caso un círculo):

```
(SETQ ObjCir (VLA-ADDCIRCLE AcadEsp cen rad))
```

TRECE.7.1. Funciones Visual Lisp

Visual Lisp proporciona una serie de funciones propias (empiezan todas por `VL`) para trabajar con objetos *ActiveX*. Muchas de ellas tienen un nombre y sintaxis similar a las funciones homónimas de Visual Basic explicadas a lo largo del **MÓDULO DOCE**. Aquí se ofrece una tabla resumen de dichas funciones:

Sintaxis de función	Descripción
<code>(vl-acad-defun 'función)</code>	Define una función de Visual Lisp como función nativa de AutoLISP.
<code>(vl-acad-undefun 'función)</code>	Deshace la definición de una función de Visual Lisp como función nativa de AutoLISP.

Curso Práctico de Personalización y Programación bajo AutoCAD
Entorno de programación Visual Lisp

<code>(vl-consp lista)</code>	Determina si una lista existe o es nil.
<code>(vl-directory-files [dir patrón subdir])</code>	Devuelve los archivos del directorio indicado o del

Sintaxis de función

Descripción

	actual. Admite caracteres comodín en <i>patrón</i> . El modo <i>subdir</i> puede ser: -1 = sólo subdirectorios; 0 = archivos y subdirectorios; 1 = sólo archivos.
<code>(vl-every condición lista1 [lista2...])</code>	Examina si la condición se cumple para cada lista de argumentos.
<code>(vl-exe-filename)</code>	Devuelve el camino completo del archivo ejecutable actual en Visual Lisp.
<code>(vl-file-copy fuente destino [add])</code>	Copia un archivo sobre otro, sobrescribiendo o añadiendo (si <i>add</i> es diferente de nil).
<code>(vl-file-delete archivo)</code>	Borra un archivo.
<code>(vl-file-directory-p nombre)</code>	Determina si <i>nombre</i> es un directorio.
<code>(vl-file-rename antiguo nuevo)</code>	Renombra un archivo de nombre <i>antiguo</i> cambiándolo a <i>nuevo</i> .
<code>(vl-file-size archivo)</code>	Devuelve el tamaño en bytes de un archivo.
<code>(vl-file-systime archivo)</code>	Devuelve la hora de la última modificación de <i>archivo</i> .
<code>(vl-filename-base archivo)</code>	Devuelve el nombre del archivo indicado, eliminando el camino.
<code>(vl-filename-directory archivo)</code>	Devuelve el camino completo del archivo indicado, eliminando nombre y extensión.
<code>(vl-filename-extension archivo)</code>	Devuelve la extensión del archivo indicado, eliminando camino y nombre.
<code>(vl-filename-mktemp [patrón directorio ext])</code>	Calcula nombre de archivo temporal indicando patrón, directorio y extensión (si no, es \$VL...). Equivale a inicializar AutoLISP cuando se abre o se inicia un dibujo en AutoCAD .
<code>(vl-init)</code>	Construye una lista con los objetos indicados.
<code>(vl-list* objeto1 [objeto2...])</code>	Construye una cadena de texto a partir de una lista de enteros, tomándolos como valores ASCII.
<code>(vl-list->string lista_ascii)</code>	Devuelve la longitud de una lista.
<code>(vl-list-length lista)</code>	Determina si la condición se cumple para uno de los elementos de la lista.
<code>(vl-member-if condición lista)</code>	Determina si la condición no se cumple para uno de los elementos de la lista.
<code>(vl-member-if-not condición lista)</code>	Devuelve el índice de posición del elemento símbolo en la lista.
<code>(vl-position símbolo lista)</code>	Devuelve la representación impresa de un objeto AutoLISP tal como haría PRIN1.
<code>(vl-prin1-to-string objeto)</code>	Devuelve la representación impresa de un objeto AutoLISP tal como haría PRINC.
<code>(vl-princ-to-string objeto)</code>	Borra la clave o nombre de valor indicados del registro de Windows.
<code>(vl-registry-delete clave [nombre])</code>	Devuelve una lista de descendientes de la clave o valor indicados en el registro de Windows.
<code>(vl-registry-descendents clave [nombre])</code>	Devuelve los datos almacenados en el registro de Windows para la clave o valor indicados.
<code>(vl-registry-read clave [nombre])</code>	

<code>(vl-registry-write clave [nombre datos])</code>	Crea una clave en el registro de Windows con el nombre y datos indicados.
<code>(vl-remove elemento lista)</code>	Elimina el elemento indicado de la lista.
Sintaxis de función	Descripción
<code>(vl-remove-if condición lista)</code>	Devuelve una lista con todos los elementos de la lista indicada que fallan la condición.
<code>(vl-remove-if-not condición lista)</code>	Devuelve una lista con todos los elementos de la lista indicada que no fallan la condición.
<code>(vl-some función lista1 [lista2...])</code>	Examina si una de las listas de argumentos no hace nil la función.
<code>(vl-sort lista función_orden)</code>	Devuelve una lista con los elementos de la lista indicada ordenados según la función de orden.
<code>(vl-sort-i lista función_orden)</code>	Devuelve una lista con los índices de posición de los elementos para que estuvieran ordenados.
<code>(vl-string->list cadena)</code>	Convierte la cadena de texto en una lista de números con los códigos ASCII de cada carácter.
<code>(vl-string-elt cadena posición)</code>	Devuelve el código ASCII del carácter que ocupa la posición indicada en la cadena de texto.
<code>(vl-string-left-trim lista_car cadena)</code>	Elimina de la cadena de texto los caracteres especificados en la lista.
<code>(vl-string-mismatch cad1 cad2 [pos1 pos2 may])</code>	Devuelve la longitud del mayor trozo común entre dos textos a partir de las posiciones indicadas, pudiendo especificar si se tienen en cuenta las mayúsculas.
<code>(vl-string-position ascii cad [inicio final])</code>	Examina si el carácter cuyo ASCII se indica está en la cadena entre dos posiciones dadas.
<code>(vl-string-right-trim lista_car cadena)</code>	Elimina de la cadena los caracteres indicados en la lista desde el final de la cadena.
<code>(vl-string-search patrón cadena [inicio])</code>	Localiza el patrón en la cadena indicada a partir de una posición de inicio.
<code>(vl-string-subst nuevo antiguo cad [inicio])</code>	Sustituye el texto antiguo por el nuevo en la cadena indicada a partir de una posición dada.
<code>(vl-string-translate ls_fuente ls_dest cad)</code>	Reemplaza cada carácter de una lista fuente por los de una lista destino en la cadena.
<code>(vl-string-trim lista_car cadena)</code>	Elimina de la cadena los caracteres indicados en la lista, desde el principio y final de la cadena.
<code>(vl-symbol-name símbolo)</code>	Devuelve una cadena con el nombre del símbolo.
<code>(vl-symbol-value símbolo)</code>	Devuelve el valor actual de un símbolo.
<code>(vl-symbolp objeto)</code>	Examina si el objeto es un símbolo.
<code>(vlax-3D-point lista_pto / x y [z])</code>	Crea un punto 3D de AutoCAD pudiendo especificarse una lista de punto o las coordenadas.
<code>(vlax-add-cmd comando función [local att])</code>	Crea un comando a partir de una función AutoLISP, pudiendo tener un nombre local y atributos.
<code>(vlax-curve-getArea obj_curva)</code>	Calcula el área interior del objeto de curva.

(vlax-curve-getDistAtParam
obj_curva pto)

Calcula la longitud de la curva a partir del punto indicado hasta el final.

Sintaxis de función

Descripción

(vlax-curve-getDistAtPoint
obj_curva pto)

Calcula la longitud de la curva desde el inicio hasta el punto indicado.

(vlax-curve-getEndParam *obj_curva*)
(vlax-curve-getEndPoint *obj_curva*)

Devuelve el parámetro de punto final de la curva.
Devuelve el punto final de la curva en coordenadas de SCU.

(vlax-curve-getParamAtDist *obj_curva*
par)

Calcula la longitud de la curva entre el inicio y el punto con el parámetro indicado.

(vlax-curve-getParamAtPoint *obj_curva*
pto)

Devuelve el parámetro de la curva en el punto indicado.

(vlax-curve-getPointAtDist *obj_curva*
dis)

Devuelve el punto situado a la distancia indicada, medida a lo largo de la curva desde el inicio.

(vlax-curve-getPointAtParam *obj_curva*
par)

Devuelve el punto de la curva que tiene el parámetro indicado.

(vlax-curve-getStartParam *obj_curva*)
(vlax-curve-getStartPoint *obj_curva*)

Devuelve el primer parámetro de la curva.
Devuelve el punto inicial de la curva en coordenadas de SCU.

(vlax-curve-isClosed *obj_curva*)
(vlax-curve-isPeriodic *obj_curva*)

Determina si la curva es cerrada.
Determina si la curva es periódica, o bien presenta un rango infinito en ambas direcciones.
Determina si la curva es plana.

(vlax-curve-isPlanar *obj_curva*)
(vlax-curve-getClosestPointTo
obj_curva pto [prolongar])

Devuelve el punto más cercano de la curva al punto especificado, pudiéndose indicar que se tenga en cuenta la prolongación de la curva por sus extremos.

(vlax-curve-getClosestPointToProjection
obj_curva pto vector [prolongar])

Devuelve el punto más cercano de la curva al punto especificado, proyectando éste en la dirección del vector indicado, pudiéndose tener en cuenta la prolongación por los extremos de la curva.

(vlax-curve-getFirstDeriv
obj_curva par)

Calcula la primera derivada de la curva, en coordenadas del SCU, en el punto de parámetro indicado

(vlax-curve-getSecondDeriv
obj_curva par)

Calcula la segunda derivada de la curva en coordenadas del SCU, en el punto de parámetro indicado.

(vlax-dump-object *objeto*)

Lista los métodos y propiedades existentes para el objeto indicado.

(vlax-ename->vla-object *nombre_ent*)

Transforma el objeto de nombre de entidad indicado en un objeto VLA accesible vía *ActiveX*.
Determina si un objeto ha sido borrado.

(vlax-erased-p *objeto*)
(vlax-for *símbolo colección*
expr1 [expr2...])

Asigna el símbolo a cada objeto de la colección y

(vlax-get <i>objeto propiedad</i>)	evalúa todas las expresiones de AutoLISP.
(vlax-get-acad-object)	Devuelve el valor de la propiedad indicada al objeto.
	Devuelve el más alto nivel (<i>Top Level</i>) de la

Sintaxis de función

Descripción

(vlax-invoke <i>objeto método lista</i>)	aplicación AutoCAD en la actual sesión. Invoca el método indicado para un objeto, proporcionando una lista de argumentos.
(vlax-ldata-delete <i>diccionario clave</i>)	Borra la entrada cuya clave se indica —en un diccionario—.
(vlax-ldata-get <i>diccionario clave [defecto]</i>)	Devuelve los datos de un diccionario para la clave indicada. Si ésta no se halla, devuelve <i>defecto</i> .
(vlax-ldata-list <i>diccionario</i>)	Devuelve todos los datos de un diccionario.
(vlax-ldata-put <i>diccionario clave datos</i>)	Almacena los datos especificados en la clave indicada de un diccionario.
(vlax-ldata-test <i>datos</i>)	Determina si los datos indicados pueden ser guardados en la sesión actual.
(vlax-map-collection <i>colección función</i>)	Aplica la función indicada a todos los objetos de una colección.
(vlax-method-applicable-p <i>objeto método</i>)	Determina si un objeto soporta el método indicado.
(vlax-object-released-p <i>objeto</i>)	Determina si un objeto ha sido liberado de cualquier enlace con un objeto del dibujo.
(vlax-product-key)	Devuelve los datos de registro de producto de AutoCAD .
(vlax-property-available-p <i>objeto prop [T]</i>)	Determina si un objeto presenta una propiedad. Si se indica T, examina si es modificable.
(vlax-put <i>objeto prop valor</i>)	Asigna el valor indicado a la propiedad especificada del objeto.
(vlax-read-enabled-p <i>objeto</i>)	Determina si un objeto puede ser leído.
(vlax-reg-app <i>registro coms carga [nom [err]]</i>)	Registra una aplicación: datos de producto, lista de comandos, número de carga, nombre, error.
(vlax-release-object <i>objeto</i>)	Libera un objeto de dibujo de la variable indicada.
(vlax-remove-cmd <i>nombre_global</i>)	Elimina el nombre de comando del grupo de la sesión actual. Si se indica T elimina el grupo.
(vlax-tmatrix <i>lista_de_cuatro</i>)	Toma una lista de cuatro listas con cuatro números y devuelve una matriz de transformación de 4x4.
(vlax-typeinfo-available-p <i>objeto</i>)	Determina si existe librería de información en Visual Lisp con métodos y propiedades del objeto.
(vlax-vla-object->ename <i>objeto</i>)	Transforma el objeto VLA indicado en un nombre de entidad de AutoLISP.
(vlax-write-enabled-p <i>objeto</i>)	Determina si un objeto de dibujo puede ser modificado.
(vlisp-export-symbol <i>símbolo</i>)	Asigna una variable nativa de AutoLISP al valor del símbolo de Visual Lisp indicado.
(vlisp-import-exsubrs <i>'(nom fun1 fun2...)</i>)	Registra las funciones indicadas del nombre de aplicación ADS o ARX dentro de Visual Lisp.
(vlisp-import-symbol <i>'símbolo</i>)	Asigna un símbolo de Visual Lisp al homónimo de

(vlr-acdb-reactor <i>datos llamadas</i>)	AutoLISP con el valor de éste. Construye un objeto reactor con los datos y llamadas indicados. Cada llamada es un par (<i>evento . función</i>).
Sintaxis de función	Descripción
(vlr-add <i>objeto</i>)	Habilita un objeto reactor inhabilitado previamente.
(vlr-added-p <i>objeto</i>)	Examina si un objeto reactor se encuentra habilitado.
(vlr-beep-reaction [<i>argumentos</i>])	Produce sonidos de computadora.
(vlr-current-reaction-name)	Devuelve el nombre del evento actual.
(vlr-data <i>objeto</i>)	Devuelve los datos específicos de aplicación asociados a un objeto reactor.
(vlr-data-set <i>objeto datos</i>)	Sobreescribe los datos específicos de aplicación de un objeto reactor con los datos indicados.
(vlr-editor-reactor <i>datos llamadas</i>)	Construye un reactor <i>Editor</i> con los datos y llamadas indicados.
(vlr-linker-reactor <i>datos llamadas</i>)	Construye un reactor <i>Linker</i> con los datos y llamadas indicados.
(vlr-object-reactor <i>objetos datos llamadas</i>)	Construye un reactor <i>Object</i> con los objetos, datos y llamadas indicados.
(vlr-owner-add reactor <i>objeto</i>)	Añade el objeto indicado al reactor.
(vlr-owner-remove reactor <i>objeto</i>)	Elimina el objeto indicado del reactor.
(vlr-owners reactor)	Devuelve la lista de objetos del reactor.
(vlr-pers reactor)	Convierte el reactor en persistente.
(vlr-pers-p reactor)	Determina si el reactor es o no persistente.
(vlr-pers-release reactor)	Elimina la persistencia de un reactor.
(vlr-reaction-names <i>tipo_reactor</i>)	Devuelve la lista de eventos del tipo de reactor indicado.
(vlr-reaction-set reactor <i>evento función</i>)	Añade o reemplaza un evento de una función en un reactor.
(vlr-reactions reactor)	Devuelve la lista de llamadas de un reactor. Cada llamada es un par (<i>evento . función</i>).
(vlr-reactors <i>tipo_reactor</i>)	Devuelve una lista con todos los reactor del tipo indicado.
(vlr-remove reactor)	Inhabilita el reactor indicado.
(vlr-remove-all reactor)	Inhabilita todos los reactores del tipo especificado.
(vlr-trace-reaction <i>argumentos</i>)	Argumentos para que sean sometidos a rastreo en la ventana <i>Trace</i> de Visual Lisp.
(vlr-type reactor)	Devuelve el tipo de reactor.
(vlr-types)	Devuelve una lista con todos los tipos de reactor.

EJERCICIOS RESUELTOS DEL MÓDULO DOCE

1ª fase intermedia de ejercicios

PUNTO I

Option Explicit
Dim AcadPapel As Object

Curso Práctico de Personalización y Programación bajo AutoCAD
Entorno de programación Visual Lisp

```
Dim MatrizSombra(0) As Object
Dim Sombreado As Object
Dim Elipse As Object

Sub Dibujo()
    Set AcadPapel = GetObject(, "AutoCAD.Application").ActiveDocument
                                                                .PaperSpace

    Dim PtoLin1(1 To 3) As Double
    Dim PtoLin2(1 To 3) As Double
    Dim PtoCir(1 To 3) As Double
    Dim RadioCir As Double
    Dim PtoElip1(1 To 3) As Double
    Dim PtoElip2(1 To 3) As Double
    Dim Ratio As Double

    PtoLin1(1) = 10.45: PtoLin1(2) = 7.012: PtoLin1(3) = 0
    PtoLin1(1) = 20: PtoLin1(2) = 10: PtoLin2(3) = 0
    PtoCir(1) = 10.45: PtoCir(2) = 23.66: PtoCir(3) = 0
    RadioCir = 34.567
    PtoElip1(1) = 14: PtoElip1(2) = 30.023: PtoElip1(3) = 0
    PtoElip2(1) = 5: PtoElip2(2) = 30.023: PtoElip2(3) = 0
    Ratio = 0.65

    Call AcadPapel.AddLine(PtoLin1, PtoLin2)
    Call AcadPapel.AddCircle(PtoCir, RadioCir)
    Set Elipse = AcadPapel.AddEllipse(PtoElip1, PtoElip2, Ratio)

    Set MatrizSombra(0) = Elipse
    Set Sombreado = AcadPapel.AddHatch(1, "ansi35", True)
    Call Sombreado.AppendOuterLoop(MatrizSombra)
End Sub
```

PUNTO II

```
Option Explicit
Dim AcadApp As Object
```

```
Private Sub Aplicar_Click()
    Dim AcadModel As Object
    Dim Elem As Object
    Dim Max As Integer
    Dim Min As Integer
    Dim i As Integer
    Dim NumInd As Integer
    Dim Width As Double
    Dim StartWidth As Double
    Dim EndWidth As Double

    Set AcadApp = GetObject(, "AutoCAD.Application")
    Set AcadModel = AcadApp.ActiveDocument.ModelSpace
    Width = CDbl(Text1.Text)
    StartWidth = Width
    EndWidth = Width
    If Width < 0# Then
        MsgBox "Introduzca un valor positivo para el espesor."
    Else
        For Each Elem In AcadModel
            With Elem
                If StrComp(.EntityName, "AcDbPolyline", 1) = 0 Then
                    Max = UBound(.Coordinates)
```

```
        Min = LBound(.Coordinates)
        NumInd = ((Max - Min) / 3) + 1
        For i = 0 To NumInd
            .SetWidth i, StartWidth, EndWidth
        Next i
        .Update
    End If
End With
Next Elem
End If
End Sub
```

```
Private Sub Salir_Click()
    Unload Me
End Sub
```

2ª fase intermedia de ejercicios

PUNTO I

Option Explicit

Dim AcadApp As Object

```
Private Sub UserForm_Initialize()
    Set AcadApp = GetObject(, "AutoCAD.Application")
    Dim ListaARX As Variant
    Dim NombreApp As String
    Dim Índice As Integer

    ListaARX = AcadApp.ListArx
    On Error Resume Next
    Índice = 0
    Do
        NombreApp = ListaARX(Índice)
        If Err.Number <> 0 Then Err.Clear: Exit Do
        Lista.AddItem NombreApp, Índice
        Índice = Índice + 1
    Loop
    Lista.Text = Lista.List(0)
End Sub
```

```
Private Sub Descargar_Click()
    Dim DescargaARX As String

    DescargaARX = Lista.Text
    Call AcadApp.UnloadArx(DescargaARX)
End
End Sub
```

3ª fase intermedia de ejercicios

PUNTO I

Option Explicit
Dim AcadDoc As Object

Curso Práctico de Personalización y Programación bajo AutoCAD
Entorno de programación Visual Lisp

```
Dim Objeto As Object
Dim Conjunto As Object
Dim NombreObjeto As String
```

```
Private Sub Aplicar_Click()
    Dim Identificador As String, Posición1 As Integer, Posición2 As Integer
    If comboObjeto.Text = "" Or comboTipoLin.Text = "" Then Exit Sub
    Posición1 = InStr(1, comboObjeto.Text, "(")
    Posición2 = InStr(1, comboObjeto.Text, ")")
    Dim Longitud As Integer
    Identificador = Val(Mid(comboObjeto.Text, Posición1 + 1,
        (Posición2 - Posición1) - 1))
    For Each Objeto In AcadDoc.ModelSpace
        If Objeto.ObjectID = Identificador Then
            Objeto.Linetype = comboTipoLin.Text
            Objeto.Update
        End If
    Next Objeto
End Sub
```

```
Private Sub Salir_Click()
    End
End Sub
```

```
Private Sub Designar_Click()
    Dim Ind As Integer
    Dim NumObjetos As Integer
    Ind = -1

    Me.Hide
    Set Conjunto = AcadDoc.SelectionSets.Add("Conjunto1")
    Call Conjunto.SelectOnScreen

    NumObjetos = Conjunto.Count

    For Each Objeto In Conjunto
        Ind = Ind + 1
        Cotejar (Objeto.EntityType)
        comboObjeto.AddItem NombreObjeto & " (" & Objeto.ObjectID & ")", Ind
    Next Objeto
    comboObjeto.Text = comboObjeto.List(0)

    Me.Show
End Sub
```

```
Private Sub UserForm_Initialize()
    Dim NumLin As Integer
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument

    NumLin = AcadDoc.Linetypes.Count
    Dim i As Integer
    For i = 0 To NumLin - 1
        comboTipoLin.AddItem AcadDoc.Linetypes.Item(i).Name, i
    Next i
    comboTipoLin.Text = comboTipoLin.List(0)
End Sub
```

```
Function Cotejar(NumObjeto As Integer)
    Select Case NumObjeto
        Case 1
```

```
NombreObjeto = "Cara 3D"
Case 2
  NombreObjeto = "Polilínea 3D"
Case 3
  NombreObjeto = "Sólido 3D"
Case 4
  NombreObjeto = "Arco"
Case 5
  NombreObjeto = "Atributo"
Case 6
  NombreObjeto = "Referencia de atributo"
Case 7
  NombreObjeto = "Referencia de bloque"
Case 8
  NombreObjeto = "Círculo"
Case 9
  NombreObjeto = "Cota alineada"
Case 10
  NombreObjeto = "Cota angular"
Case 11
  NombreObjeto = "Cota diamétrica"
Case 12
  NombreObjeto = "Cota por coordenadas"
Case 13
  NombreObjeto = "Cota radial"
Case 14
  NombreObjeto = "Cota girada"
Case 15
  NombreObjeto = "Elipse"
Case 16
  NombreObjeto = "Grupo"
Case 17
  NombreObjeto = "Sombreado"
Case 18
  NombreObjeto = "Directriz"
Case 19
  NombreObjeto = "Línea"
Case 20
  NombreObjeto = "Texto múltiple"
Case 21
  NombreObjeto = "Punto"
Case 22
  NombreObjeto = "Polilínea 2D antigua"
Case 23
  NombreObjeto = "Polilínea 3D nueva"
Case 24
  NombreObjeto = "Malla poligonal"
Case 25
  NombreObjeto = "Ventana de Espacio Papel"
Case 26
  NombreObjeto = "Imagen de trama"
Case 27
  NombreObjeto = "Rayo"
Case 28
  NombreObjeto = "Región"
Case 29
  NombreObjeto = "Forma"
Case 30
  NombreObjeto = "Sólido 2D"
Case 31
  NombreObjeto = "Spline"
```

```
Case 32
  NombreObjeto = "Texto"
Case 33
  NombreObjeto = "Tolerancia"
Case 34
  NombreObjeto = "Trazo"
Case 35
  NombreObjeto = "Línea auxiliar"
End Select
End Function
```

4ª fase intermedia de ejercicios

PUNTO I

```
Option Explicit
Dim AcadDoc As Object
Dim AcadUtil As Object
Dim AcadModel
```

```
Dim PtoInserciónVar As Variant, PtoInserción(1 To 3) As Double
Dim Dist As Double, Alt As Double
```

```
Private Sub buttonDibujar_Click()
  If Distancia.Text = "" Or Distancia.Text = "0" Or
    Val(Distancia.Text) < 0 Then
    MsgBox "La distancia ha de ser positiva y mayor de cero.",
      vbOKOnly + vbCritical, "Error en entrada de datos"
  Exit Sub
End If

If Altura.Text = "" Or Altura.Text = "0" Or Val(Altura.Text) < 0 Then
  MsgBox "La altura ha de ser positiva y mayor de cero.", vbOKOnly +
    vbCritical, "Error en entrada de datos"
Exit Sub
End If

Dim Pto1(1 To 3) As Double
Dim Pto2(1 To 3) As Double
Dim Pto3(1 To 3) As Double
Dim Pto4(1 To 3) As Double
Dim Pc1(1 To 3) As Double
Dim Pc2(1 To 3) As Double
Dim Pc3(1 To 3) As Double
Dim Pb1(1 To 3) As Double
Dim Pb2(1 To 3) As Double
Dim Pb3(1 To 3) As Double
Dim Pb4(1 To 3) As Double

Dist = Val(Distancia.Text)
Alt = Val(Altura.Text)
Pto1(1) = PtoInserción(1) - (Dist / 2)
Pto1(2) = PtoInserción(2) + (Alt - (0.0428932 * Dist))
Pto2(1) = PtoInserción(1) - (Dist / 4)
Pto2(2) = PtoInserción(2) + (Alt - (0.0428932 * Dist))
Pto3(1) = PtoInserción(1) + (Dist / 4)
Pto3(2) = PtoInserción(2) + (Alt - (0.0428932 * Dist))
Pto4(1) = PtoInserción(1) + (Dist / 2)
Pto4(2) = PtoInserción(2) + (Alt - (0.0428932 * Dist))
```

```
Pc1(1) = PtoInserción(1) - (0.375 * Dist)
Pc1(2) = PtoInserción(2) + Alt
Pc2(1) = PtoInserción(1)
Pc2(2) = PtoInserción(2) + Alt
Pc3(1) = PtoInserción(1) + (0.375 * Dist)
Pc3(2) = PtoInserción(2) + Alt

Pb1(1) = PtoInserción(1) - (Dist / 2)
Pb1(2) = PtoInserción(2) + (Alt / 2)
Pb2(1) = PtoInserción(1) - (Dist / 4)
Pb2(2) = PtoInserción(2) + (Alt / 2)
Pb3(1) = PtoInserción(1) + (Dist / 4)
Pb3(2) = PtoInserción(2) + (Alt / 2)
Pb4(1) = PtoInserción(1) + (Dist / 2)
Pb4(2) = PtoInserción(2) + (Alt / 2)

AcadUtil.StartUndoMark

Dim Pol1 As Object
Dim Pol2 As Object
Dim Pol3 As Object
Dim Lin1 As Object, Lin2 As Object, Lin3 As Object
Dim Lin4 As Object, Lin5 As Object
Dim MatrizVértices(0 To 3 * 2 - 1) As Double
MatrizVértices(0) = Pto1(1): MatrizVértices(1) = Pto1(2)
MatrizVértices(2) = Pc1(1): MatrizVértices(3) = Pc1(2)
MatrizVértices(4) = Pto2(1): MatrizVértices(5) = Pto2(2)
Set Pol1 = AcadModel.AddLightWeightPolyline(MatrizVértices)
Call Pol1.SetBulge(0, -0.2)
Call Pol1.SetBulge(1, -0.2)

MatrizVértices(0) = Pto2(1): MatrizVértices(1) = Pto2(2)
MatrizVértices(2) = Pc2(1): MatrizVértices(3) = Pc2(2)
MatrizVértices(4) = Pto3(1): MatrizVértices(5) = Pto3(2)

Set Pol2 = AcadModel.AddLightWeightPolyline(MatrizVértices)
Call Pol2.SetBulge(0, -0.1)
Call Pol2.SetBulge(1, -0.1)

MatrizVértices(0) = Pto3(1): MatrizVértices(1) = Pto3(2)
MatrizVértices(2) = Pc3(1): MatrizVértices(3) = Pc3(2)
MatrizVértices(4) = Pto4(1): MatrizVértices(5) = Pto4(2)

Set Pol3 = AcadModel.AddLightWeightPolyline(MatrizVértices)
Call Pol3.SetBulge(0, -0.2)
Call Pol3.SetBulge(1, -0.2)

Set Lin1 = AcadModel.AddLine(Pc1, Pc3)
Set Lin2 = AcadModel.AddLine(Pb1, Pto1)
Set Lin3 = AcadModel.AddLine(Pb2, Pto2)
Set Lin4 = AcadModel.AddLine(Pb3, Pto3)
Set Lin5 = AcadModel.AddLine(Pb4, Pto4)

Call Pol1.Mirror(Pb1, Pb4)
Call Pol2.Mirror(Pb1, Pb4)
Call Pol3.Mirror(Pb1, Pb4)
Call Lin1.Mirror(Pb1, Pb4)
Call Lin2.Mirror(Pb1, Pb4)
Call Lin3.Mirror(Pb1, Pb4)
Call Lin4.Mirror(Pb1, Pb4)
```

```
Call Lin5.Mirror(Pb1, Pb4)

AcadUtil.EndUndoMark

Unload Me
End Sub
```

```
Private Sub buttonInserción_Click()
    Me.Hide
    PtoInserciónVar = AcadUtil.GetPoint(, "Punto de inserción: ")
    PtoInserción(1) = PtoInserciónVar(0): PtoInserción(2) =
        PtoInserciónVar(1): PtoInserción(3) = PtoInserciónVar(2)

    X.Text = Left(LTrim(Str(PtoInserción(1))), 11)
    Y.Text = Left(LTrim(Str(PtoInserción(2))), 11)
    Z.Text = Left(LTrim(Str(PtoInserción(3))), 11)
    Me.Show
End Sub
```

```
Private Sub buttonCancelar_Click()
    Unload Me
End Sub
```

```
Private Sub UserForm_Initialize()
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument
    Set AcadUtil = AcadDoc.Utility
    Set AcadModel = AcadDoc.ModelSpace
End Sub
```

EJERCICIOS RESUELTOS DEL MÓDULO DOCE

EJERCICIO I

```
Option Explicit

Dim AcadDoc As Object
Dim Blk As Object
Dim i As Integer
```

```
Private Sub Cambiar_Click()
    For Each Blk In AcadDoc.Blocks
        If Left(Blk.Name, 1) <> "*" Then
            For i = 0 To Blk.Count - 1
                Blk(i).Color = Val(ListaColor.Text)
            Next i
        End If
    Next
    Call AcadDoc.Regen(0)
End Sub
```

```
Private Sub Salir_Click()
    End
End Sub
```

```
Private Sub UserForm_Initialize()
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument
```



```
For i = 0 To 256
    ListaColor.AddItem i
Next i
ListaColor.Text = ListaColor.List(0)
End Sub
```

EJERCICIO II

Option Explicit

Const Pi = 3.141592654

```
Function Distancia(p1() As Double, p2() As Double) As Double
    Dim xDist As Double
    Dim yDist As Double
    Dim zDist As Double

    xDist = p1(0) - p2(0)
    yDist = p1(1) - p2(1)
    zDist = p1(2) - p2(2)
    Distancia = Sqr(xDist ^ 2 + yDist ^ 2 + zDist ^ 2)
End Function
```

```
Sub Matriz(Origen As Variant, Destino As Variant)
    Dim nIdx As Long

    If (UBound(Origen) - LBound(Origen)) = (UBound(Destino) -
                                                LBound(Destino)) Then
        nIdx = LBound(Origen)
        While nIdx <= UBound(Origen)
            Destino(nIdx) = Origen(nIdx)
            nIdx = nIdx + 1
        Wend
    End If
End Sub
```

```
Function ExisteCapa(ByVal NombreCapa As String) As Boolean
    Dim Elemento As Object

    ExisteCapa = False
    For Each Elemento In ThisDrawing.Layers
        If Elemento.Name = UCase(NombreCapa) Then
            ExisteCapa = True
            Exit For
        End If
    Next
End Function
```

```
Function ExisteTipoLin(ByVal NombreTipoLin As String) As Boolean
    Dim Elemento As Object

    ExisteTipoLin = False
    For Each Elemento In ThisDrawing.Linetypes
        If Elemento.Name = UCase(NombreTipoLin) Then
            ExisteTipoLin = True
            Exit For
        End If
    Next
End Function
```

```
Private Sub InicializarCapa(NombreCapa As String, LayerLType As
                             String, LayerColor As Long)
```

```
Dim ValorV As Long
Dim NuevaCapa As IAcadLayer

If ExisteCapa(NombreCapa) = False Then
  If ExisteTipoLin(LayerLType) = False Then
    ValorV = ThisDrawing.Linetypes.Load(LayerLType, "acadiso.lin")
  End If
  Set NuevaCapa = ThisDrawing.Layers.Add(NombreCapa)
  NuevaCapa.Color = LayerColor
  NuevaCapa.Linetype = LayerLType
End If
End Sub
```

```
Sub EjesLin()
  Const LineExtension = 0.25
  Const LineMultiplier = 1.75
  Const NombreCapa = "Ejes"
  Const LayerColor = 1
  Const LayerLType = "Trazo_y_punto"

  Dim SSet As Object
  Dim Espacio As Long
  Dim Elemento As Object
  Dim PuntoCentro(0 To 2) As Double
  Dim PtoFinal1(0 To 2) As Double
  Dim PtoFinal2(0 To 2) As Double
  Dim Cline As Object
  Dim MediaLong As Double
  Dim CodGrupo(0 To 4) As Integer
  Dim ValorDato(0 To 4) As Variant

  MsgBox "Seleccione arcos, círculos y/o elipses."

  Set SSet = ThisDrawing.SelectionSets.Add("ArcsCirclesEllipses")
  CodGrupo(0) = -4
  ValorDato(0) = "<or"
  CodGrupo(1) = 0
  ValorDato(1) = "Arc"
  CodGrupo(2) = 0
  ValorDato(2) = "Circle"
  CodGrupo(3) = 0
  ValorDato(3) = "Ellipse"
  CodGrupo(4) = -4
  ValorDato(4) = ">or"
  SSet.SelectOnScreen CodGrupo, ValorDato
  For Each Elemento In SSet
    Espacio = ThisDrawing.ActiveSpace
    InicializarCapa NombreCapa, LayerLType, LayerColor

    If Elemento.EntityType = acCircle Or Elemento.EntityType = acArc
      Then
        Dim CurRadius As Double

        CurRadius = Elemento.Radius
        If CurRadius <= LineExtension Then
          MediaLong = LineMultiplier * CurRadius
        Else
          MediaLong = LineExtension + CurRadius
        End If
        Matriz Elemento.center, PuntoCentro
        PtoFinal1(0) = PuntoCentro(0)
```

```
PtoFinal1(1) = PuntoCentro(1) - MediaLong
PtoFinal1(2) = PuntoCentro(2)
PtoFinal2(0) = PuntoCentro(0)
PtoFinal2(1) = PuntoCentro(1) + MediaLong
PtoFinal2(2) = PuntoCentro(2)
If Espacio = acModelSpace Then
    Set Cline = ThisDrawing.ModelSpace.AddLine(PtoFinal1,
                                                PtoFinal2)
Else
    Set Cline = ThisDrawing.PaperSpace.AddLine(PtoFinal1,
                                                PtoFinal2)
End If
Cline.Layer = NombreCapa
PtoFinal1(0) = PuntoCentro(0) - MediaLong
PtoFinal1(1) = PuntoCentro(1)
PtoFinal1(2) = PuntoCentro(2)
PtoFinal2(0) = PuntoCentro(0) + MediaLong
PtoFinal2(1) = PuntoCentro(1)
PtoFinal2(2) = PuntoCentro(2)
If Espacio = acModelSpace Then
    Set Cline = ThisDrawing.ModelSpace.AddLine(PtoFinal1,
                                                PtoFinal2)
Else
    Set Cline = ThisDrawing.PaperSpace.AddLine(PtoFinal1,
                                                PtoFinal2)
End If
Cline.Layer = NombreCapa
ElseIf Elemento.EntityType = acEllipse Then
    Dim EjeMayor(0 To 2) As Double
    Dim EjeMenor(0 To 2) As Double
    Dim PtoCuadrante(0 To 2) As Double
    Dim AngRadMayor As Double
    Dim RadioMenor As Double
    Dim RadioMayor As Double
    Dim MediaLongMenor As Double

    Matriz Elemento.center, PuntoCentro
    Matriz Elemento.EjeMayor, EjeMayor
    Matriz Elemento.EjeMenor, EjeMenor
    PtoCuadrante(0) = EjeMayor(0) + PuntoCentro(0)
    PtoCuadrante(1) = EjeMayor(1) + PuntoCentro(1)
    PtoCuadrante(2) = EjeMayor(2) + PuntoCentro(2)
    RadioMayor = Distancia(PtoCuadrante, PuntoCentro)
    RadioMenor = RadioMayor * Elemento.radiusRatio
    AngRadMayor = ThisDrawing.Utility.AngleFromXAxis(PuntoCentro,
                                                    PtoCuadrante)

    If RadioMenor <= LineExtension Then
        MediaLongMenor = LineMultiplier * RadioMenor
    Else
        MediaLongMenor = LineExtension + RadioMenor
    End If
    If RadioMayor <= LineExtension Then
        MediaLong = LineMultiplier * RadioMayor
    Else
        MediaLong = LineExtension + RadioMayor
    End If

    With ThisDrawing.Utility
        Matriz .PolarPoint(PuntoCentro, AngRadMayor, MediaLong),
                PtoFinal1
        Matriz .PolarPoint(PuntoCentro, AngRadMayor + Pi, MediaLong),
```

```

                                                                    PtoFinal2
End With

If Espacio = acModelSpace Then
    Set Cline = ThisDrawing.ModelSpace.AddLine(PtoFinal1,
                                                                    PtoFinal2)
Else
    Set Cline = ThisDrawing.PaperSpace.AddLine(PtoFinal1,
                                                                    PtoFinal2)
End If
Cline.Layer = NombreCapa

With ThisDrawing.Utility
    Matriz .PolarPoint(PuntoCentro, AngRadMayor - (Pi / 2),
                        MediaLongMenor), PtoFinal1
    Matriz .PolarPoint(PuntoCentro, AngRadMayor + (Pi / 2),
                        MediaLongMenor), PtoFinal2
End With

If Espacio = acModelSpace Then
    Set Cline = ThisDrawing.ModelSpace.AddLine(PtoFinal1,
                                                                    PtoFinal2)
Else
    Set Cline = ThisDrawing.PaperSpace.AddLine(PtoFinal1,
                                                                    PtoFinal2)
End If
Cline.Layer = NombreCapa
End If
Next
End Sub
```

EJERCICIO III

```
Option Explicit
Dim AcadDoc As Object
Dim AcadModel As Object
```

```
Const EPS = 0.0000001
```

```
Private Sub Aplicar_Click()
    Dim Elem As Object
    Dim AlturaTexto As Double
    Set AcadDoc = GetObject(, "AutoCAD.Application").ActiveDocument
    Set AcadModel = AcadDoc.ModelSpace
    AlturaTexto = CDBl(Text1.Text)

    If AlturaTexto < CDBl(EPS) Then
        MsgBox "Especifique una altura mayor que 0.0"
    Else
        For Each Elem In mspace
            With Elem
                If StrComp(.EntityName, "AcDbText", 1) = 0 Then
                    .Height = AlturaTexto
                    .Update
                End If
            End With
            Set Elem = Nothing
        Next Elem
    End If
End Sub
```

```
Private Sub Salir_Click()  
    Unload Me  
End  
End Sub
```

EJERCICIO IV

(Ejercicio completo para resolver por técnicos y/o especialistas).

APÉNDICES

APÉNDICE A

Comandos y abreviaturas de AutoCAD

A.1. COMANDOS DE AutoCAD CON SU CORRESPONDENCIA EN INGLÉS

Castellano	Inglés
3D	3D
3DARRAY	3DARRAY
3DCARA	3DFACE
3DMALLA	3DMESH
3DPOL	3DPOLY
ABRE	OPEN
ACERCA	ABOUT
ACISIN	ACISIN
ACISOUT	ACISOUT
ACOALINEADA	DIMALIGNED
ACOANGULO	DIMANGULAR
ACOCENTRO	DIMCENTER
ACOCONTINUA	DIMCONTINUE
ACOCOORDENADA	DIMORDINATE
ACODIAMETRO	DIMDIAMETER
ACOEDIC	DIMEDIT
ACOESTIL	DIMSTYLE
ACOLINEABASE	DIMBASELINE
ACOLINEAL	DIMLINEAR
ACORADIO	DIMRADIUS

Curso Práctico de Personalización y Programación bajo AutoCAD
Comandos y abreviaturas de AutoCAD

Castellano	Inglés
ACOREEMPLAZAR	DIMOVERRIDE
ACOTA	DIM
ACOTEDIC	DIMTEDIT
AJUSTARIMG	IMAGEADJUST
ALARGA	ENTEND
ALINEAR	ALIGN
ANULADEF	UNDEFINE
APERTURA	APERTURE
APpload	APpload
ARANDELA	DONUT
ARCO	ARC
AREA	AREA
ARRASTRAPs	PSDRAG
ARRASTRE	DRAGMODE
ARX	ARX
ASEADMIN	ASEADMIN
ASEEXPORT	ASEEXPORT
ASELINKS	ASELINKS
ASEROWS	ASEROWS
ASESELECT	ASESELECT
ASESQLED	ASESQLED
ATRDEF	ATTDEF
ATREDIT	ATTEDIT
ATREXT	ATTTEXT
ATRVIS	ATTDISP
ATTREDEF	ATTDEREF
AYUDA	HELP
BARRAHERR	TOOLBAR
BASE	BASE
BIBLIOMAT	MATLIB
BIBPAISAJE	LSLIB
BLOQUE	BLOCK
BLOQUEDISC	WBLOCK
BMAKE	BMAKE
BOCETO	SKETCH
BORRA	ERASE
CAL	CAL
CALIDADIMG	IMAGEQUALITY
CAMBIA	CHANGE
CAMBPROP	CHPROP
CAPA	LAYER
CARGA	LOAD
CARGADXB	DXBIN
CARGADXF	DXFIN
CARGAPS	PSIN
CARGAR3DS	3DSIN
CARGARMENU	MENULOAD
CARGAWMF	WMFIN
CHAFLAN	CHAMFER
CILINDRO	CYLINDER
CIRCULO	CIRCLE
COLOR	COLOR
COMPILA	COMPILE
CONFIG	CONFIG
CONO	CONE
CONTORNO	BOUNDARY
CONVERTAME	AMECONVERT
CONVERTIR	CONVERT
COPIA	COPY

Curso Práctico de Personalización y Programación bajo AutoCAD
Comandos y abreviaturas de AutoCAD

Castellano	Inglés
COPIAENLACE	COPYLINK
COPIAHIST	COPYHIST
COPIAPP	COPYCLIP
CORTAPP	CUTCLIP
CORTE	SLICE
CUÑA	WEDGE
DCOTA	DDIM
DDAMODOS	DDRMODES
DDATTDEF	DDATTDEF
DDATTE	DDATTE
DDATTEXT	DDATTEXT
DDCHPROP	DDCHPROP
DDCOLOR	DDCOLOR
DDEDIC	DDEDIT
DDGRIPS	DDGRIPS
DDINSERT	DDINSERT
DDMODIFY	DDMODIFY
DDOSNAP	DDOSNAP
DDPTYPE	DDPTYPE
DDRENAME	DDRENAME
DDSCP	DDUCS
DDSELECT	DDSELECT
DDUCSP	DDUCSP
DDUNITS	DDUNITS
DDVIEW	DDVIEW
DDVPOINT	DDVPOINT
DELIMITARIMG	IMAGECLIP
DELIMITARX	XCLIP
DESCARGAMENU	MENUUNLOAD
DESCOMP	EXPLODE
DESHACER	UNDO
DESIGNA	SELECT
DESPLAZA	MOVE
DIFERENCIA	SUBTRACT
DIRECTRIZ	LEADER
DIST	DIST
DIVIDE	DIVIDE
EDGE	EDGE
EDITARLM	MLEDIT
EDITPOL	PEDIT
EDITSOMB	HATCHEDIT
EDITSPLINE	SPLINEDIT
EDPAISAJE	LSEDIT
ELEV	ELEV
ELIPSE	ELLIPSE
EMPALME	FILLET
ENCUADRE	PAN
ENLAZARIMG	IMAGEATTACH
EMLAZARX	XATTACH
EQDIST	OFFSET
ESCALA	SCALE
ESCALATL	LTSCALE
ESCENA	SCENE
ESFERA	SPHERE
ESPACIOM	MSPACE
ESPACIOP	PSPACE
ESTADIST	STATS
ESTADO	STATUS
ESTADOARB	TREESTAT

Curso Práctico de Personalización y Programación bajo AutoCAD
Comandos y abreviaturas de AutoCAD

Castellano	Inglés
ESTILO	STYLE
ESTILOLM	MLSTYLE
ESTIRA	STRETCH
EXAMINAR	BROWSER
EXPORTAR	EXPORT
EXTRUSION	EXTRUDE
FILTER	FILTER
FORMA	SHAPE
FORZCURSOR	SNAP
GIRA	ROTATE
GIRA3D	ROTATE3D
GRADUA	MEASURE
GRUPO	GROUP
GUARDAR	SAVE
GUARDARCOMO	SAVEAS
GUARDARIMG	SAVEIMG
GUARDARR	QSAVE
H	U
ID	ID
IGUALARPROP	MATCHPROP
IMAGEN	IMAGE
IMPORTAR	IMPORT
INSERT	INSERT
INSERTM	MININSERT
INSERTOBJ	INSERTOBJ
INTERF	INTERFERE
INTERSEC	INTERSECT
ISOPLANO	ISOPLANE
LIMITES	LIMITS
LIMPIA	PURGE
LINEA	LINE
LINEAM	MLINE
LINEAX	XLINE
LIST	LIST
LISTDB	DBLIST
LOCTEXTO	QTEXT
LOGFILEOFF	LOGFILEOFF
LOGFILEON	LOGFILEON
LONGITUD	LENGTHEN
LUZ	LIGHT
MAPEADO	SETUV
MARCAAUX	BLIPMODE
MARCOIMG	IMAGEFRAME
MATERIALR	RMAT
MATRIZ	ARRAY
MENU	MENU
MIRAFOTO	VSLIDE
MODIVAR	SETVAR
MOSTRMAT	SHOWMAT
MULTIPLE	MULTIPLE
MVSETUP	MVSETUP
NIEBLA	FOG
NUEVO	NEW
NVPAISAJE	LSNEW
OCULTA	HIDE
ORDENAOBJETOS	DRAWORDER
ORTO	ORTHO
ORTOGRAFIA	SPELL
PAISAJE	BACKGROUND

Curso Práctico de Personalización y Programación bajo AutoCAD
Comandos y abreviaturas de AutoCAD

Castellano	Inglés
PANTGRAF	GRAPHSCR
PANTTEXT	TEXTSCR
PARTE	BREAK
PCARA	PFACE
PEGAESP	PASTESPEC
PEGAPP	PASTECLIP
PLANTA	PLAN
POL	PLINE
POLIGONO	POLYGON
PREFERENCIAS	PREFERENCES
PREFR	RPREF
PREVISUALIZAR	PREVIEW
PRISMARECT	BOX
PROPFIS	MASSPROP
PTOVISTA	VPOINT
PUNTO	POINT
QUITA	QUIT
RAYO	RAY
REANUDA	RESUME
RECORTA	TRIM
RECTANG	RECTANG
RECUPERAR	RECOVER
REDEFINE	REDEFINE
REDIBT	REDRAWALL
REDIBUJA	REDRAW
REFENT	OSNAP
REFX	XREF
REGEN	REGEN
REGENAUTO	REGENAUTO
REGENT	REGENALL
REGION	REGION
REHACER	REDO
REINICIA	REINIT
REJILLA	GRID
RELLENAPS	PSFILL
RELLENAR	FILL
RENDER	RENDER
RENOMBRA	RENAME
REPRODUCIR	REPLAY
RESVISTA	VIEWRES
RETARDA	DELAY
REVISION	AUDIT
REVOLUCION	REVOLVE
RFILEOPT	RFILEOPT
RSCRIPT	RSCRIPT
SACAFOTO	MSLIDE
SALTRAZ	PLOT
SALVABMP	BMPOUT
SALVADWF	DWFOUT
SALVADXF	DXFOUT
SALVAPS	PSOUT
SALVAR3DS	3DSOUT
SALVASTL	STLOUT
SALVAWMF	WMFOUT
SCP	UCS
SCRIPT	SCRIPT
SECCION	SECTION
SHELL	SHELL
SIMBSCP	UCSICON

Castellano	Inglés
SIMETRIA	MIRROR
SIMETRIA3D	3DMIRROR
SISWINDOWS	SYSWINDOWS
SOLDRAW	SOLDRAW
SOLIDO	SOLID
SOLPERFIL	SOLPROF
SOMBCONT	BHATCH
SOMBRA	SHADE
SOMBREA	HATCH
SPLINE	SPLINE
SUPLADOS	EDGESURF
SUPREGLA	RULESURF
SUPTAB	TABSURF
TABLERO	TABLET
TEXTO	TEXT
TEXTODIN	DTEXT
TEXTOM	MTEXT
TIEMPO	TIME
TIPOLIN	LINETYPE
TOLERANCIA	TOLERANCE
TOROIDE	TORUS
TRANSPARENCIA	TRANSPARENCY
TRAZO	TRACE
UNIDADES	UNITS
UNION	UNION
UNIRX	XBIND
UY	OOPS
VENTANAS	VPORTS
VGCAPA	VPLAYER
VINCOLE	OLELINKS
VISTA	VIEW
VISTADIN	DVIEW
VISTAAREA	DSVIEWER
VMULT	MVIEW
WMFOPS	WMFOPTS
XPLODE	XPLODE
ZOOM	ZOOM

A.2. ABREVIATURAS MÁS TÍPICAS PREDEFINIDAS DE AutoCAD CON SU CORRESPONDENCIA EN INGLÉS

Comando	Abreviatura castellana	Abreviatura inglesa
ALARGA	AL	EX
ARANDELA	AR	DO
ARCO	A	A
BMAKE	BM	B
BORRA	B	E
CAPA	CA	LA
CHAFLAN	CH	CHA
CIRCULO	C	C
COPIA	CP y DUP	CP y CO
DCOTA	DCO	D
DDAMODOS	MO	RM
DDEDIC	DD	ED

Curso Práctico de Personalización y Programación bajo AutoCAD
Comandos y abreviaturas de AutoCAD

Comando	Abreviatura castellana	Abreviatura inglesa
DDINSERT	IN	I
DDMODIFY	M	MO
DDVIEW	DV	V
DDVPOINT	PV	VP
DESCOMP	DP	X
DESPLAZA	D	M
DIST	DI	DI
DIVIDE	DIV	DIV
EDITPOL	PE	PE
ELIPSE	EL	EL
EMPALME	MP	F
ENCUADRE	E	P
EQDIST	EQ	O
ESCALA	ES	SC
ESCALATL	EC	LTS
ESPACIOM	EM	
ESPACIOP	EP	
ESTILO	EST	ST
ESTIRA	EI	S
FORZCURSOR	FC	SN
GIRA	GI	RO
GRADUA	GD	ME
IGUALARPROP	IP	MA
IMAGEN	IMG	IM
LINEA	L	L
LIST	LT y LS	LI y LS
MATRIZ	MA	AR
OCULTA	OC	HI
ORDENAOBJETOS	OB	DR
PARTE	P	BR
POLIGONO	PG	POL
POL	PL	PL
PREFERENCIAS	PF	PR
PREVISUALIZAR	PRE	PRE
PTOVISTA	PV	VP
PUNTO	PU	PO
RECORTA	RR	TR
RECTANG	REC	REC
REDIBUJA	RE	R
REFENT	RF y OS	OS
REFX	RX	XR
REGEN	RG	RE
RENDER	R	RR
SALTRAZ	ST	PRINT
SIMETRIA	SI	MI
SOMBCONT	SB y SBC	BH y H
SOMBRA	SM	SHA
SOMBREA	SMB	H
SPLINE	SPL	SPL
TEXTODIN	T	DT
TEXTOM	TXM	MT
TILEMODE	TM	TI
TIPOLIN	TL	LT
UNIRX	UX	XB
VISTADIN	VD	DV
VISTA	VI	V
VMULT	VM	MV
ZOOM	Z	Z

APÉNDICE B

Variables de sistema y acotación

B.1. VARIABLES DE SISTEMA Y ACOTACIÓN

Nombre	Tipo	En	Significado
ACADPREFIX	Cadena	No guardada (de sólo lectura)	Almacena el camino de directorios determinado por la variable de entorno ACAD.
ACADVER	Cadena	No guardada (de sólo lectura)	Almacena el número de versión de AutoCAD , que consta de valores del tipo 14 ó 14a
ACISOUTVER	Entero	Dibujo	Controla la versión ACIS de los archivos .SAT guardados por ACISOUT.
AFLAGS	Entero	No guardada	Establece los indicadores de atributo para el código binario de ATRDEF. Es la suma de: 0 Ningún modo de atributo seleccionado. 1 Invisible. 2 Constante. 4 Verificable. 8 Predefinido.
ANGBASE	Real	Dibujo	Establece el ángulo base 0 con respecto al SCP actual.
ANGDIR	Entero	Dibujo	Establece la dirección positiva de ángulos desde el ángulo 0 con respecto al SCP actual: 0 En sentido contrario a las agujas del reloj. 1 En el sentido de las agujas del reloj.
APBOX	Entero	Registro	Activa o desactiva la mira de señalamiento al utilizar <i>AutoSnap</i> .
APERTURE	Entero	Registro	Establece la altura de mira de referencia a objetos, en píxeles.
AREA	Real	No guardada (de sólo lectura)	Almacena la última área calculada por AREA, LIST O LISTDB.
ATTDIA	Entero	Dibujo	Controla si INSERT utiliza un cuadro de diálogo para la entrada de valores de atributo: 0 Presenta solicitudes en la línea de comando. 1 Utiliza un cuadro de diálogo.
ATTMODE	Entero	Dibujo	Controla el modo de visualización de atributos: 0 Desactivado: no se visualiza ningún atributo. 1 Normal: sólo se visualizan los atributos visibles. 2 Activado: se visualizan todos los atributos.
ATTREQ	Entero	Dibujo	Determina si INSERT utiliza parámetros de atributos por defecto durante la inserción de bloques: 0 Asume los valores por defecto de todos los atributos. 1 Activa solicitudes o cuadros de diálogo para los valores de atributo,

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	Significado
AUDITCTL	Entero	Registro	según la selección realizada por ATTDIA. Controla si AutoCAD crea un archivo .ADT (informe de revisión): 0 Desactiva o evita que se escriba en los archivos .ADT. 1 Activa la capacidad de escritura en los archivos .ADT por parte de REVISION.
AUNITS	Entero	Dibujo	Establece el modo de unidades de ángulos: 0 Grados decimales. 1 Grados/min/seg. 2 Grados. 3 Radianes. 4 Geodésicas.
AUPREC	Entero	Dibujo	Establece el número de posiciones decimales de las unidades angulares.
AUTOSNAP			Controla la visualización del marcador, atracción e información de <i>AutoSnap</i> . Es la suma de: 0 Descativa marcador, atracción e información. 1 Activa el marcador. 2 Activa la información de la referencia. 3 Activa la atracción.
BACKZ	Real	Dibujo (de sólo lectura)	Almacena la variación de los planos delimitadores opuestos en relación con el plano de mira de la ventana gráfica actual, en unidades de dibujo. La distancia del plano delimitador opuesto en relación con el punto de cámara puede calcularse restando BACKZ de la distancia del punto de cámara al punto de mira.
BLIPMODE	Entero	Dibujo	Controla si las marcas auxiliares están visibles o no.
CDATE	Real	No guardada (de sólo lectura)	Establece la fecha y hora del calendario.
CECOLOR	Cadena	Dibujo	Establece el color en el dibujo para los nuevos objetos.
CELTSCALE	Real	Dibujo	Establece la escala global de tipo de línea para los objetos.
CELTYPE	Cadena	Dibujo	Establece el tipo de línea en el dibujo para los nuevos objetos.
CHAMFERA	Real	Dibujo	Establece la primera distancia de chaflán.
CHAMFERB	Real	Dibujo	Establece la segunda distancia de chaflán.
CHAMFERC	Real	Dibujo	Establece la longitud del chaflán.
CHAMFERD	Real	Dibujo	Establece el ángulo del chaflán.
CHAMMODE	Entero	No guardada	Establece el método de entrada mediante el que AutoCAD crea chaflanes: 0 Requiere dos distancias de chaflán. 1 Requiere una longitud de chaflán y un ángulo.
CIRCLERAD	Real	No guardada	Establece el radio del círculo por defecto. Si es cero no se establece ningún valor por defecto.
CLAYER	Cadena	Dibujo	Establece la capa actual.
CMDACTIVE	Entero	No guardada (de sólo lectura)	Establece el código binario que indica si está activo un comando normal, transparente, un guión o un cuadro de

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	Significado
			<p>diálogo. Es la suma de los siguientes valores:</p> <p>1 Comando normal activo.</p> <p>2 Comando normal y comando transparente activos.</p> <p>4 Guión activo.</p> <p>8 Cuadro de diálogo activo.</p> <p>16 Comando de AutoLISP o DDE activo</p>
CMDDIA	Entero	Registro	<p>Controla la activación de cuadros de diálogo para <i>SALTRAZ</i> y bases de datos externas:</p> <p>0 Desactiva los cuadros de diálogo.</p>
CMDECHO	Entero	No guardada	<p>1 Activa los cuadros de diálogo.</p> <p>Controla si se visualizan solicitudes y datos durante la función <i>COMMAND</i> de AutoLISP:</p> <p>0 Desactiva la visualización.</p>
CMDNAMES	Cadena	No guardada (de sólo lectura)	<p>1 Activa la visualización.</p> <p>Visualiza el nombre (en inglés) del comando y del comando transparente actualmente activos. Por ejemplo, <i>LINE</i> ' <i>ZOOM</i> indica que se está usando el comando <i>ZOOM</i> de forma transparente mientras se ejecuta el comando <i>LINEA</i>.</p>
CMLJUST	Entero	Dibujo	<p>Determina la justificación de las líneas múltiples:</p> <p>0 Superior.</p> <p>1 Central.</p>
CMLSCALE	Real	Dibujo	<p>2 Inferior.</p> <p>Controla la anchura total de una multilínea. Un factor de escala de 2.0 produce una multilínea con el doble de grosor que la definición de estilo. Un valor negativo invierte el orden de líneas.</p>
CMLSTYLE	Cadena	Dibujo	<p>Establece el nombre del estilo de multilínea utilizado en el dibujo.</p>
COORDS	Entero	Dibujo	<p>Controla visualización de coordenadas del cursor en la línea de estado:</p> <p>0 Las coordenadas sólo se actualizan al designar puntos con el cursor.</p> <p>1 Las coordenadas absolutas se actualizan continuamente.</p> <p>2 Se visualizan la distancia y ángulo del último punto al mover el cursor a uno nuevo.</p>
CURSORSIZE	Entero	Registro	<p>Determina el tamaño del cursor en cruz como un porcentaje del tamaño de la pantalla</p>
CVPORT	Entero	Dibujo	<p>Establece el número de identificación de la ventana gráfica actual. Se puede modificar este valor y, por tanto, la ventana gráfica actual, si se satisfacen las siguientes condiciones:</p> <ul style="list-style-type: none"> • El número de identificación designado es el de una ventana gráfica activa. • Un comando en curso no ha bloqueado el movimiento del cursor en esa ventana gráfica. • El modo Tablero está desactivado.
DATE	Real	No guardada (de sólo lectura)	<p>Almacena la fecha y hora actuales representadas como fecha juliana con las fracciones en números reales: <i><fecha juliana>.<Fracción></i>. Por ejemplo, el 24 de febrero de 1998, a</p>

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	Significado																
DBMOD	Entero	No guardada (de sólo lectura)	las 13:18:34 de la tarde, la variable DATE contendría 2450869.55458912. Indica el estado de modificación del dibujo mediante un código binario. Es la suma de los valores: 1 Base de datos de objetos modificada. 4 Variable de base de datos modificada. 8 Ventana modificada. 16 Vista modificada.																
DCTCUST	Cadena	Registro	Visualiza el camino y nombre de archivo del diccionario de ortografía personalizado actual.																
DCTMAIN	Cadena	Registro	Visualiza el nombre de archivo del diccionario de ortografía principal actual. No se muestra todo el camino, ya que se supone que este archivo se encuentra en el directorio \SUPPORT\.																
			Algunos valores pueden ser:																
			<table><tr><th>Palabra clave</th><th>Idioma</th></tr><tr><td>ca</td><td>catalán</td></tr><tr><td>enu</td><td>inglés americano</td></tr><tr><td>ena</td><td>inglés australiano</td></tr><tr><td>ens</td><td>inglés británico (ise)</td></tr><tr><td>enz</td><td>inglés británico (ize)</td></tr><tr><td>esa</td><td>español (mayúsculas acentuadas)</td></tr><tr><td>es</td><td>español (mayúsculas sin acentuar)</td></tr></table>	Palabra clave	Idioma	ca	catalán	enu	inglés americano	ena	inglés australiano	ens	inglés británico (ise)	enz	inglés británico (ize)	esa	español (mayúsculas acentuadas)	es	español (mayúsculas sin acentuar)
Palabra clave	Idioma																		
ca	catalán																		
enu	inglés americano																		
ena	inglés australiano																		
ens	inglés británico (ise)																		
enz	inglés británico (ize)																		
esa	español (mayúsculas acentuadas)																		
es	español (mayúsculas sin acentuar)																		
DELOBJ	Entero	Dibujo	Controla si los objetos utilizados para crear otros objetos se mantienen o suprimen del dibujo: 0 Los objetos se mantienen. 1 Los objetos se suprimen.																
DEMANDLOAD	Entero	Registro	Controla la carga bajo demanda de aplicaciones desarrolladas por terceros. Los valores son: 0 Desactiva la carga bajo demanda 1 Carga la aplicación externa al abrir un dibujo con objetos realizados mediante ella. 2 Carga la aplicación externa al invocar un comando de la misma 3 Carga la aplicación externa en cualquiera de los dos casos anteriores.																
DIASSTAT	Entero	No guardada (de sólo lectura)	Almacena el método de salida del cuadro de diálogo utilizado por última vez: 0 Cancelar. 1 Aceptar.																
DIMADEC	Entero	Dibujo	Controla el número de decimales de precisión para las cotas angulares. Si vale -1, se utiliza el mismo valor que DIMDEC.																
DIMALT	Conmutador	Dibujo	Activa el uso de unidades alternativas en acotación.																
DIMALTD	Entero	Dibujo	Controla el número de decimales de precisión para las unidades alternativas.																
DIMALTF	Real	Dibujo	Controla el factor de escala de las unidades alternativas. Si DIMALT está activada, DIMALTF multiplica los valores de cotas lineales en unidades alternativas por un factor.																
DIMALTTD	Entero	Dibujo	Establece el número de posiciones																

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	Significado
DIMALTTZ	Entero	Dibujo	<p>decimales para los valores de tolerancia de una cota en unidades alternativas.</p> <p>Activa y desactiva la supresión de ceros para los valores de tolerancia. Sus valores son:</p> <p>0 Suprime cero pies y cero pulgadas.</p> <p>1 Mantiene cero pies y cero pulgadas.</p> <p>2 Mantiene cero pies y suprime cero . pulgadas.</p> <p>3 Suprime cero pies y mantiene cero pulgadas.</p> <p>4 Suprime cero pies, cero pulgadas y los ceros a la izquierda para valores inferiores a 1.</p> <p>8 Suprime cero pies, cero pulgadas y los ceros a la derecha.</p> <p>12 Suma de los dos anteriores, suprime todos los tipos de ceros.</p>
DIMALTU	Entero	Dibujo	<p>Establece el formato de las unidades alternativas de todas las familias de estilos de acotación a excepción de la angular:</p> <p>1 Científicas.</p> <p>2 Decimales.</p> <p>3 Pies/PI.</p> <p>4 Pies/PII (apiladas).</p> <p>5 Fraccionarias (apiladas).</p> <p>6 Pies y pulgadas II (arquitectura).</p> <p>7 Fraccionarias .</p> <p>8 Unidades de Windows.</p>
DIMALTZ	Entero	Dibujo	<p>Controla la supresión de ceros para los valores de cota en unidades alternativas. Sus valores posibles son los mismos que para DIMALTTZ.</p>
DIMAPOST	Cadena	Dibujo	<p>Determina un prefijo o sufijo de texto (o ambos) para las unidades alternativas de cota en todos los tipos de acotación, a excepción de la cota angular. Para separar el prefijo del sufijo, se indica el texto de cota mediante <>. Por ejemplo: Valor <> cm. Para desactivar los prefijos o sufijos, se introduce punto (.).</p>
DIMASO	Conmutador	Dibujo	<p>Activa la creación de objetos de cota asociativa. Su valor no se almacena en un estilo de cota.</p>
DIMASZ	Real	Dibujo	<p>Controla el tamaño de la línea de cota y de los extremos de flecha de las líneas directrices. Controla también el tamaño de las líneas de conexión. Los múltiplos del tamaño de los extremos de flecha determinan si las líneas de cota y el texto encajan entre las líneas de referencia.</p>
DIMAUNIT	Entero	Dibujo	<p>Establece el formato de unidades de ángulo para las cotas angulares:</p> <p>0 Grados en fracción decimal.</p> <p>1 Grados/minutos/segundos</p> <p>2 Grados centesimales.</p> <p>3 Radianes.</p> <p>4 Unidades geodésicas.</p>
DIMBLK	Cadena	Dibujo	<p>Establece el nombre del bloque que se va a situar en lugar de la cabeza de flecha. Para desactivar un nombre de bloque establecido, debe introducirse un punto (.).</p>

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	Significado
DIMBLK1	Cadena	Dibujo	Si DIMSAH está activada, DIMBLK1 determina un nombre de bloque para situar en lugar del extremo de flecha en la primera línea de referencia de la cota. Para desactivar un nombre de bloque establecido, se introduce un punto (.).
DIMBLK2	Cadena	Dibujo	Si DIMSAH está activada, DIMBLK2 determina un nombre de bloque para situar en lugar del extremo de flecha en la segunda línea de referencia. Para desactivar un nombre de bloque establecido, se introduce un punto (.).
DIMCEN	Real	Dibujo	Controla el dibujo de las marcas de centro y líneas de centro del círculo o arco acotado mediante ACOCENTRO, ACODIAMETRO y ACORADIO. Sus valores son: =0 No se dibuja ninguna marca ni línea de centro. <0 Se dibujan líneas de centro con el tamaño de marca indicado. >0 Se dibujan marcas de centro con el tamaño indicado.
DIMCLRD	Entero	Dibujo	Asigna colores a las líneas de cota, extremos de flecha y líneas directrices de cota. Los enteros equivalentes para PORBLOQUE y PORCAPA son 0 y 256 respectivamente.
DIMCLRE	Entero	Dibujo	Asigna colores a las líneas de referencia de cota. Los valores válidos son de 0 a 256.
DIMCLRT	Entero	Dibujo	Asigna colores a los textos de cota. Los valores válidos son de 0 a 256.
DIMDEC	Entero	Dibujo	Establece el número de posiciones decimales para los valores de tolerancia de una cota en unidades principales.
DIMDLE	Real	Dibujo	Distancia de extensión de la línea de cota a los lados de la línea de referencia cuando se dibujan trazos oblicuos en lugar de extremos de flecha.
DIMDLI	Real	Dibujo	Controla el espaciado de las líneas de cota cuando se acota con línea de base.
DIMEXE	Real	Dibujo	Determina cuánto se extiende la línea de referencia por encima de la línea de cota.
DIMEXO	Real	Dibujo	Determina cuánto se separan las líneas de referencia de sus puntos de origen.
DIMFIT	Entero	Dibujo	Controla la inserción de texto y extremos de flecha dentro o fuera de las líneas de referencia en función del espacio disponible entre ellas. Sus valores son: 0 Ambos, texto y extremos de flecha, se sitúan o dentro o fuera. 1 Si no hay espacio disponible, se inserta preferentemente el texto dentro. 2 Si no hay espacio disponible, se insertan preferentemente los extremos de flecha dentro. 3 Se inserta el mejor ajuste, texto o extremos de flecha, en función del espacio disponible. 4 Se crean líneas directrices cuando no

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	Significado
DIMGAP	Real	Dibujo	<p>hay espacio suficiente para el texto. 5 No se crean en ningún caso líneas directrices. Establece la distancia de salvaguarda alrededor del texto de cota, al partir la línea de cota, el espacio al situar el texto encima de la línea de cota y el espacio entre texto y extremo de línea directriz. Un valor negativo de DIMGAP genera una acotación básica: texto de cota rodeado en todo su perímetro por un rectángulo.</p>
DIMJUST	Entero	Dibujo	<p>Controla la posición horizontal del texto de cota: 0 Coloca el texto centrado en la línea de cota. 1 Coloca el texto junto a la primera línea de referencia. 2 Coloca el texto junto a la segunda línea de referencia. 3 Coloca el texto encima y alineado con la primera línea de referencia. 4 Coloca el texto encima y alineado con la segunda línea de referencia.</p>
DIMLFAC	Real	Dibujo	<p>Establece un factor de escala global para la medición de cotas lineales. Todas las distancias lineales medidas durante la acotación (incluidos radios, diámetros y coordenadas) se multiplican por el parámetro de DIMLFAC antes de convertirse en texto de cota. En el Espacio Papel existe una opción Ventana que permite especificar factores de escala diferentes en ventanas múltiples.</p>
DIMLIM	Conmutador	Dibujo	<p>Al activarse, genera límites de cota como en lugar de valores de tolerancia.</p>
DIMPOST	Cadena	Dibujo	<p>Determina un prefijo o sufijo de texto (o ambos) para los textos de cota en unidades principales. Para separar el prefijo del sufijo se indica el texto de cota mediante <>. Por ejemplo: Valor <> cm. Para prescindir, se introduce un punto (.).</p>
DIMRND	Real	Dibujo	<p>Redondea todas las dimensiones de acotación al valor determinado. Por ejemplo, si se establece en 0.25, todas las distancias se redondean a la unidad en 0,25 más cercana. Si se establece en 1.0, todas las distancias se redondean al entero más próximo.</p>
DIMSAH	Conmutador	Dibujo	<p>Controla la generación de bloques de usuario en lugar de extremos de flecha. Si esta desactivado, se utilizan los extremos de flecha proporcionados por AutoCAD, o un mismo bloque en ambos extremos, especificado por DIMBLK. Si está activada, se generan bloques diferentes en ambos extremos, especificados por DIMBLK1 y DIMBLK2.</p>
DIMSCALE	Real	Dibujo	<p>Establece el factor de escala global aplicado a todas las demás variables de acotación que determinan tamaños de elementos de cota: =0 AutoCAD calcula el factor de escala respecto al Espacio Papel. >0 AutoCAD aplica el factor de escala especificado.</p>

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	Significado
DIMSD1	Conmutador	Dibujo	Al activarse se suprime la primera línea de cota.
DIMSD2	Conmutador	Dibujo	Al activarse se suprime la segunda línea de cota.
DIMSE1	Conmutador	Dibujo	Al activarse se suprime la primera línea de referencia.
DIMSE2	Conmutador	Dibujo	Al activarse se suprime la segunda línea de referencia.
DIMSHO	Conmutador	Dibujo	Al activarse las cotas asociativas se recalculan de forma dinámica conforme se arrastran.
DIMSOXD	Conmutador	Dibujo	Al activarse, suprime el dibujo de líneas de cota que está fuera de las líneas de referencia. En caso de que las líneas de cota estén fuera de las líneas de referencia y DIMTIX está activada, al activar DIMSOXD, se suprime la línea de cota. Si DIMTIX está desactivada, DIMSOXD no tiene ningún efecto.
DIMSTYLE	Conmutador	Dibujo (de sólo lectura)	Almacena el nombre del estilo de acotación actual.
DIMTAD	Entero	Dibujo	Controla la posición vertical del texto en relación con la línea de cota: 0 Centra el texto de cota entre las líneas de referencia. 1 Coloca el texto de cota sobre la línea de cota, excepto cuando la línea de cota no es horizontal y se fuerza a que el texto que se encuentra dentro de la línea de referencia sea horizontal. 2 Coloca el texto de cota a un lado de la línea de cota, en el lugar más alejado de los puntos definidores. 3 Coloca el texto de cota de forma que se adapte a las normas JIS.
DIMTDEC	Entero	Dibujo	Establece el número de posiciones decimales para visualizar los valores de tolerancia de una cota.
DIMTFAC	Real	Dibujo	Especifica la altura de texto de los valores de tolerancia en relación con la altura del texto de cota, según el valor establecido por DIMTXT. Por ejemplo, si se establece en 1, la altura de texto de las tolerancias es igual al texto de cota. Si se establece en 0.75, la altura de texto de las tolerancias es tres cuartas partes el tamaño del texto de cota.
DIMTIH	Conmutador	Dibujo	Controla la orientación del texto de cota cuando cabe dentro de las líneas de referencia, para todos los tipos de acotaciones excepto las de coordenadas: 0 Alinea el texto con la línea de cota. 1 Dibuja el texto horizontalmente.
DIMTIX	Conmutador	Dibujo	Fuerza al texto a dibujarse entre las líneas de referencia: 0 El resultado varía según el tipo de cota. En cotas lineales y angulares, AutoCAD inserta el texto dentro de las líneas de referencia si hay suficiente espacio. En las cotas de radio y de diámetro, al desactivar DIMTIX, el texto queda fuera del círculo o arco. 1 Dibuja el texto de cota entre las líneas de referencia incluso aunque AutoCAD lo coloque normalmente fuera de estas líneas.

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	Significado
DIMTM	Real	Dibujo	Cuando DIMTOL o DIMLIM están activadas, establece el límite de tolerancia mínimo o inferior para el texto de cota. AutoCAD acepta valores con signo. Si DIMTOL está activada, y DIMTP y DIMTM tienen el mismo valor, AutoCAD dibuja un símbolo ± seguido del valor de tolerancia. Si los valores de DIMTM y DIMTP son diferentes, se dibuja la tolerancia superior sobre la inferior y se añade un signo más al valor de DIMTP si es positivo.
DIMTOFL	Conmutador	Dibujo	Fuerza a dibujarse una línea de cota entre las líneas de referencia incluso cuando el texto quede fuera de estas líneas. En las cotas de radio y de diámetro (si DIMTIX está desactivada), dibuja una línea de cota y los extremos de flecha dentro del círculo o arco, y deja fuera el texto y la directriz.
DIMTOH	Conmutador	Dibujo	Al activarse, controla la orientación del texto de cota cuando no cabe entre las líneas de referencia: 0 Alinea el texto con la línea de cota. 1 Dibuja el texto horizontalmente.
DIMTOL	Conmutador	Dibujo	Al activarse, añade tolerancias de acotación al texto de cota. Activar DIMTOL fuerza a que se desactive DIMLIM.
DIMTOLJ	Entero	Dibujo	Establece la justificación vertical para los valores de tolerancia en relación con el texto de cota: 0 Inferior. 1 Central. 2 Superior.
DIMTP	Real	Dibujo	Cuando DIMTOL o DIMLIM están activadas, establece el límite de tolerancia máximo o superior para el texto de cota. AutoCAD acepta valores con signo para DIMTP. Si DIMTOL está activada, y DIMTP y DIMTM tienen el mismo valor, AutoCAD dibuja un símbolo ± seguido del valor de tolerancia. Si los valores de DIMTM y DIMTP son diferentes, la tolerancia superior se dibuja sobre la inferior y se añade un signo más al valor de DIMTP si es positivo.
DIMTSZ	Real	Dibujo	Determina el tamaño de los trazos oblicuos dibujados en lugar de los extremos de flecha para las cotas lineales, de radio y de diámetro: = 0 Dibuja extremos de flecha. > 0 Dibuja trazos oblicuos en lugar de flechas. El tamaño de los trazos oblicuos viene determinado por este valor multiplicado por el valor de DIMSCALE.
DIMTVP	Real	Dibujo	Ajusta la posición vertical del texto de cota sobre o debajo de la línea de cota. AutoCAD utiliza el valor de DIMTVP cuando DIMTAD está desactivada. La magnitud de la variación vertical del texto es el producto de la altura del texto y el valor de DIMTVP. Establecer DIMTVP en 1.0 es equivalente a activar

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	Significado
			DIMTAD. AutoCAD corta la línea de
			cota para ajustar el texto sólo si el valor absoluto de DIMTVP es inferior a 0.7.
DIMTXSTY	Cadena	Dibujo	Especifica el estilo del texto de la cota.
DIMTXT	Real	Dibujo	Determina la altura del texto de cota, salvo si el estilo de texto actual tiene una altura fija.
DIMTZIN	Entero	Dibujo	Controla la supresión de ceros para los valores de tolerancia. Sus valores son los mismos que para DIMALTTZ.
DIMUNIT	Entero	Dibujo	Establece el formato de las unidades para todas las familias de estilos de cota salvo el angular. Sus valores son los mismos que para DIMALTU.
DIMUPT	Conmutador	Dibujo	Controla las funciones del cursor al generarse las cotas por el usuario: 0 El cursor sólo controla la ubicación de la línea de cota. 1 El cursor controla también la posición del texto de cota.
DIMZIN	Entero	Dibujo	Controla la supresión de ceros en las unidades principales. Sus valores son los mismos que para DIMALTTZ.
DISPSILH	Entero	Dibujo	Controla la visualización de siluetas en sólidos al ocultar.
DISTANCE	Real	No guardada (de sólo lectura)	Almacena la distancia calculada por el comando DIST.
DONUTID	Real	No guardada	Establece el valor por defecto del diámetro interior de una arandela.
DONUTOD	Real	No guardada	Establece el valor por defecto del diámetro exterior de una arandela. Debe ser distinto de cero.
DRAGMODE	Entero	Dibujo	Establece el modo de arrastre de objetos durante las operaciones de edición: 0 Sin arrastre. 1 Se activa bajo solicitud expresa mediante 'ARRASTRE'. 2 Automático. Se activa en aquellos comandos que lo incorporan.
DRAGP1	Entero	Registro	Establece la velocidad de muestreo de regeneración durante el arrastre.
DRAGP2	Entero	Registro	Establece la velocidad de muestreo durante el arrastre rápido.
DWGCODEPAGE	Cadena	Dibujo (de sólo lectura)	Almacena la página de códigos del dibujo. Es el mismo valor que SYSCODEPAGE; se mantiene esta variable por razones de compatibilidad.
DWGNAME	Cadena	No guardada (de sólo lectura)	Almacena el nombre del dibujo tal como lo escribió el usuario. Si todavía no se ha asignado un nombre al dibujo, DWGNAME almacena S-nombre. Si el usuario define un prefijo con la unidad y los directorios, éste se incluye en DWGPREFIX.
DWGPREFIX	Cadena	No guardada (de sólo lectura)	Almacena el prefijo de unidad/directorio del dibujo.
DWGTITLED	Entero	No guardada (de sólo lectura)	Indica si se le ha asignado un nombre al dibujo actual: 0 El dibujo no tiene nombre. 1 El dibujo tiene nombre.
EDGEMODE	Entero	Registro	Controla la forma en la que los

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	Significado
ELEVATION	Real	Dibujo	<p>comandos RECORTA y ALARGA determinan las aristas cortantes y los contornos:</p> <p>0 Utiliza la arista designada sin alargarla.</p> <p>1 Alarga la arista designada hasta su máxima longitud imaginaria.</p> <p>Almacena la elevación en 3D actual en relación con el SCP del espacio actual.</p> <p>Controla la visualización de determinadas solicitudes de los comandos:</p> <p>0 Presenta todas las solicitudes normalmente.</p> <p>1 Elimina "Regeneración necesaria - ¿continuar?" y "¿Realmente desea desactivar la capa actual?".</p> <p>2 Elimina las solicitudes anteriores y "Bloque ya existe ¿Redefinirlo?" (BLOQUE) y "Ya existe un dibujo con este nombre. ¿Desea reemplazarlo?" (GUARDAR o BLOQUEDISC).</p> <p>3 Elimina las solicitudes anteriores y las enviadas por TIPOLIN, si trata de cargar un tipo de línea ya cargado o crear un nuevo tipo de línea en un archivo en el que ya está definida.</p> <p>4 Elimina las solicitudes anteriores y las enviadas por SCP Guardar y VENTANAS Guardar, si el nombre proporcionado ya existe.</p> <p>5 Elimina las solicitudes anteriores y las enviadas por la opción ACOESTIL Guardar y ACOEMPLAZAR, si el nombre del estilo de acotación proporcionado ya existe (las entradas se vuelven a definir).</p> <p>Cuando EXPERT elimina una solicitud, la operación en cuestión se realiza aunque se escriba S en la solicitud. El parámetro de EXPERT puede afectar a guiones, macros de menús, AutoLISP y a las funciones de los comandos.</p>
EXPERT	Entero	No guardada	
EXPLMODE	Entero	No guardado	
EXTMAX	Punto	Dibujo (de sólo lectura)	
EXTMIN	Punto	Dibujo (de sólo lectura)	<p>Controla si el comando DESCOMP admite bloques a escala no uniforme (ENU):</p> <p>0 No descompone bloques ENU.</p> <p>1 Descompone bloques ENU.</p> <p>Almacena el punto superior derecho de la extensión del dibujo. Este se expande hacia fuera conforme se dibujan nuevos objetos. Está expresada en coordenadas universales del espacio actual.</p>
FACETRES	Real	Dibujo	<p>Almacena el punto inferior izquierdo de la extensión del dibujo, en coord. universales</p> <p>Controla el ajuste del suavizado de los sólidos al ocultar o modelizar. Los valores válidos comprenden del 0.01 al 10.0.</p>
FILEDIA	Entero	Registro	<p>Suprime la visualización de los cuadros de diálogo de gestión de archivos.</p> <p>0 Desactiva los cuadros de diálogo. Si desea aún solicitar que aparezcan, se puede teclear una tilde (~) como</p>

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	respuesta a la solicitud del comando: Significado
FILLETRAD	Real	Dibujo	1 Activa los cuadros de diálogo. Sin embargo, si hay un guión o programa de AutoLISP/ADS activo, la solicitud se hace en línea de comando. Almacena el radio de empalme actual.
FILLMODE	Entero	Dibujo	
			Determina si se muestra el relleno de los objetos con áreas rellenas: 0 Los objetos no se rellenan. 1 Los objetos se rellenan.
FONTALT	Cadena	Registro	Determina el tipo de letra alternativo utilizado cuando no pueda encontrarse el archivo de tipos de letra designado. Si no se define un tipo de letra alternativo, AutoCAD visualiza una advertencia.
FONTMAP	Cadena	Registro	Determina el archivo de correspondencia de tipos de letra utilizado cuando no pueda encontrarse la fuente designada. Este archivo contiene en cada línea una correspondencia de tipo de letra, con el tipo de letra original y el sustituto separados por un punto y coma (;). Por ejemplo: romans;c:\windows\system\times.ttf
FRONTZ	Real	Dibujo (de sólo lectura)	Almacena la distancia del plano delimitador frontal en relación con el plano de mira de la ventana gráfica actual, en unidades de dibujo. La distancia entre el plano delimitador frontal y el punto de cámara se calcula al restar FRONTZ de la distancia del punto de cámara al punto de mira.
GRIDMODE	Entero	Dibujo	Determina si la rejilla está activada o desactivada.
GRIDUNIT	Punto	Dibujo	Determina el intervalo de la rejilla (X e Y) de la ventana gráfica actual.
GRIPBLOCK	Entero	Registro	Controla la asignación de pinzamientos en bloques: 0 Sólo asigna pinzamientos al punto de inserción del bloque. 1 Asigna pinzamientos a todos los objetos del bloque.
GRIPCOLOR	Entero	Registro	Controla el color de pinzamientos no seleccionados. El rango válido comprende del 1 al 255.
GRIPHOT	Entero	Registro	Controla el color de pinzamientos seleccionados. El rango válido comprende del 1 al 255.
GRIPS	Entero	Registro	Controla la activación de pinzamientos.
GRIPSIZE	Entero	Registro	Establece el tamaño de la casilla dibujada para visualizar el pinzamiento en píxeles. El rango válido comprende del 1 al 255.
HANDLES	Entero	Dibujo (de sólo lectura)	Informa que los identificadores de objetos están activados y las aplicaciones pueden acceder a ellos.
HIGHLIGHT	Entero	No guardada	Controla el resaltado de objetos; no afecta a los objetos designados con pinzamientos: 0 Desactiva el resaltado de designación de objetos. 1 Activa el resaltado de designación de objetos.
HPANG	Real	No guardada	Determina el ángulo de los patrones de sombreado.

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	Significado
HPBOUND	Entero	Registro	Controla el tipo de objeto creado por los comandos SOMBCONT y CONTORNO: 0 Crea una región. 1 Crea una polilínea.
HPDOUBLE	Entero	No guardada	Controla el sombreado de doble rayado para los patrones "Usuario": 0 Desactiva el sombreado de doble rayado. 1 Activa el sombreado de doble rayado.
HPNAME	Cadena	No guardada	Establece el nombre del patrón de sombreado por defecto con un máximo de 34 caracteres y sin espacios permitidos. Devuelve " " si no existe valor por defecto. Introducir un punto (.) para no establecer ningún valor por defecto.
HPSCALE	Real	No guardada	Determina el factor de escala de los patrones de sombreado; debe ser distinto de cero.
HPSPACE	Real	No guardada	Determina el espaciamiento de líneas de los patrones de sombreado de "Usuario"; debe ser distinto de cero.
INDEXCTL	Entero	Dibujo	Controla la creación de índices espacial y de capa: 0 No se crean índices. 1 Se crea índice de capa. 2 Se crea índice espacial. 3 Se crean índices espacial y de capa.
INETLOCATION	Real	Registro	Almacena la localización de Internet utilizada por el nadegador.
INSBASE	Punto	Dibujo	Almacena el punto base para la inserción establecido por el comando BASE, expresado en coordenadas de SCP del espacio actual.
INSNAME	Cadena	No guardada	Establece el nombre de bloque por defecto para DDINSERT o INSERT. Este nombre debe satisfacer las convenciones de denominación de símbolos. Devuelve " " si no existe ningún valor por defecto. Introducir un punto (.) para no establecer ningún valor por defecto.
ISAVEBAK	Entero	Registro	Controla si se crean o no archivos de copia de seguridad .BAK.
ISAVEPERCENT	Entero	Registro	Determina el tanto por ciento de tamaño añadido al archivo de dibujo durante el guardado progresivo antes de efectuar un guardado completo.
ISOLINES	Entero	Dibujo	Determina el número de isolíneas de representación alámbrica de superficies de sólidos. Los valores enteros válidos comprenden del 0 al 2047.
LASTANGLE	Real	No guardada (de sólo lectura)	Almacena el ángulo final del último arco dibujado, en relación con el plano XY del SCP actual.
LASTPOINT	Punto	Dibujo	Almacena el último punto introducido, expresado en coordenadas de SCP actual. La utilización de arroba @ en coordenadas relativas se refiere a este punto.
LASTPROMPT	Cadena	No guardada (de sólo lectura)	Almacena la última cadena de texto reflejada en la línea de comando
LENSLENGTH	Real	Dibujo (de sólo lectura)	Almacena la longitud de la lente (en milímetros) utilizada en la vista en perspectiva de la ventana gráfica

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	actual. Significado
LIMCHECK	Entero	Dibujo	Controla si se permite la creación de objetos fuera de los límites del dibujo: 0 Los objetos pueden ser creados fuera de los límites. 1 No se permite crear objetos fuera de los límites.
LIMMAX	Punto	Dibujo	Almacena el límite superior derecho del dibujo en el espacio actual en coordenadas universales.
LIMMIN	Punto	Dibujo	Almacena el límite inferior izquierdo del dibujo en el espacio actual en coordenadas universales.
LISPINIT	Entero	Registro	Especifica si las variables y funciones de AutoLISP se mantienen cuando se sale del dibujo actual y se entra en otro: 0 Las funciones y variables de AutoLISP se mantienen (AutoLISP persistente). 1 Las funciones y variables de AutoLISP no se mantienen (como en la versión 13).
LOCALE	Cadena	No guardada (de sólo lectura)	Visualiza el código del idioma ISO de la versión de AutoCAD actual en uso. Valor inicial: "en" .
LOGFILEMODE	Entero	Registro	Controla si el contenido de la ventana de texto se escribe en un archivo de registro.
LOGFILENAME	Cadena	Registro	Especifica el nombre y camino del archivo de registro para el contenido de la ventana de texto.
LOGINNAME	Cadena	No guardada (de sólo lectura)	Visualiza el nombre del usuario tal como se configuró o introdujo al cargar AutoCAD .
LTSCALE	Real	Dibujo	Establece el factor de escala global para los tipos de línea.
LUNITS	Entero	Dibujo	Establece el modo de las unidades lineales: 1 Científicas. 2 Decimales. 3 Pies/PI. 4 Pies/PII (arquitectura). 5 Fraccionarias.
LUPREC	Entero	Dibujo	Establece el número de posiciones decimales para las unidades lineales.
MAXACTVP	Entero	Dibujo	Establece el número máximo de ventanas gráficas que pueden estar activas al mismo tiempo.
MAXOBJMEM	Entero	No guardada	Controla la cantidad de memoria virtual que AutoCAD utiliza para su paginación.
MAXSORT	Entero	Registro	Establece el número máximo de nombres de símbolos o de nombres de bloques que pueden ser clasificados por comandos de enumeración. Si el número total de elementos supera este número no se clasifica ninguno.
MEASUREMENT	Entero	Dibujo	Especifica el sistema de unidades del dibujo: 0 Sistema de unidades inglesas. 1 Sistema métrico.
MENUCTL	Entero	Registro	Controla la conmutación de páginas del menú de pantalla: 0 El menú de pantalla no conmuta páginas ante la entrada de comandos por teclado. 1 El menú de pantalla conmuta páginas

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	Significado
MENUECHO	Entero	No guardada	<p>ante la entrada de comandos por teclado.</p> <p>Controla el eco de opciones de menú y solicitudes. Es la suma de los siguientes valores:</p> <p>1 Elimina la visualización de las opciones de menú (^P en una opción de menú activa y desactiva la visualización en pantalla).</p> <p>2 Elimina la visualización de las solicitudes del sistema durante el menú.</p> <p>4 Desactiva el conmutador ^P de la visualización de menús en pantalla.</p> <p>8 Visualiza las cadenas de entrada/salida como ayuda para la depuración de macros DIESEL.</p>
MENUNAME	Cadena	Encabezamiento (de sólo lectura)	<p>Almacena el nombre de MENUGROUP.</p> <p>Si el menú principal actual no tiene nombre de MENUGROUP, el archivo del menú incluye el camino cuando el emplazamiento del archivo no esté especificado en el parámetro de entorno de AutoCAD.</p>
MIRRTEXT	Entero	Dibujo	<p>Controla la forma en que SIMETRIA refleja texto:</p> <p>0 Conserva el contenido del texto sin reflejar.</p> <p>1 Refleja el contenido del texto en simetría.</p>
MODEMACRO	Cadena	No guardada	<p>Visualiza en la línea de estado la cadena de texto que el usuario especifique. Esta cadena puede ser fija o se pueden utilizar cadenas de texto especiales escritas en el lenguaje de macros DIESEL, de forma que AutoCAD evalúe la macro de forma continua y actualice la línea de estado.</p>
MTEXTED	Cadena	Registro	<p>Establece el nombre del programa utilizado como editor de textos múltiples de TEXTOM.</p>
OFFSETDIST	Real	No guardada	<p>Establece la distancia de EQDIST por defecto:</p> <p><0 Equidista los objetos mediante Punto a atravesar.</p> <p>>0 Establece el valor de equidistancia por defecto.</p>
OLEHIDE	Entero	Registro	<p>Controla la visualización de objetos OLE en AutoCAD:</p> <p>0 Todos los objetos OLE son visibles</p> <p>1 Los objetos OLE son visibles sólo en el Espacio Papel.</p> <p>2 Los objetos OLE son visibles solo en el Espacio Modelo.</p> <p>3 Ningún objeto OLE es visible.</p>
ORTHOMODE	Entero	Dibujo	<p>Controla la activación del modo Orto.</p>
OSMODE	Entero	Dibujo	<p>Establece los modos de referencia a objetos en ejecución a través de los siguientes códigos binarios. Para designar varias referencias a objetos se escribe la suma de sus valores:</p> <p>0 NINGuno.</p> <p>1 PuntoFINal.</p> <p>2 PuntoMEDio.</p> <p>4 CENTro.</p> <p>8 PunTO.</p> <p>16 CUAdrante.</p>

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	Significado
			32 INTERsección.
			64 INSerción.
			128 PERpendicular.
			256 TANgente.
			512 CERcano.
			1024 RAPido.
			2048 InterFICticia.
OSNAPCOORD	Entero	Registro	Controla la prioridad de coordenadas introducidas desde el teclado: 0 Los modos de referencia establecidos tienen prioridad sobre las coordenadas por teclado. 1 Las coordenadas por teclado tienen prioridad sobre los modos de referencia establecidos. 2 Las coordenadas por teclado tienen preferencia, excepto en los archivos de guión.
PDMODE	Entero	Dibujo	Establece el modo de visualización de objetos de punto. Los valores posibles, en el mismo orden en que aparecen en el cuadro de diálogo de DDPTYPE son: 0, 1, 2, 3, 4, 32, 33, 34, 35, 36, 64, 65, 66, 67, 68, 96, 97, 98, 99, 100.
PDSIZE	Real	Dibujo	Establece el tamaño de visualización de los objetos de punto: =0 Crea un punto al 5% de la altura del área gráfica. >0 Especifica un tamaño absoluto. <0 Especifica un porcentaje del tamaño de la ventana gráfica.
PELLIPSE	Entero	Dibujo	Controla el tipo de elipse creado con ELIPSE: 0 Crea un objeto de elipse verdadero. 1 Crea una representación polilíneal de una elipse.
PERIMETER	Real	No guardada (de sólo lectura)	Almacena el último valor de perímetro calculado por AREA, LIST o LISTDB.
PFACEVMAX	Entero	No guardada (de sólo lectura)	Establece el número máximo de vértices por cara.
PICKADD	Entero	Registro	Controla la designación aditiva de objetos: 0 Desactiva PICKADD. Los últimos objetos designados, ya sea mediante designación individual o con ventanas de designación, se convierten en el conjunto de selección. Los objetos anteriormente designados se suprimen del conjunto de selección. Para añadir más objetos al conjunto de selección se debe mantener pulsado SHIFT al designar. 1 Activa PICKADD. Cada objeto designado, ya sea individualmente o por ventanas de designación, se añade al conjunto de selección actual. Para suprimir objetos del conjunto se debe mantener pulsado SHIFT durante la designación.
PICKAUTO	Entero	Registro	Controla la creación automática de una ventana de designación al señalar un punto vacío: 0 Desactiva PICKAUTO. 1 Dibuja automáticamente una ventana de designación (ventana o captura).
PICKBOX	Entero	Registro	Establece la altura de la mira de

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	designación de objetos, en píxeles. Significado
PICKDRAG	Entero	Registro	Controla el método de generación de las ventanas de designación: 0 Para dibujar la ventana de designación se pulsa del ratón o digitalizador en una esquina y, a continuación, en la otra esquina. 1 Para dibujar la ventana de designación se pulsa en una esquina y se arrastra el cursor manteniendo pulsado el botón, hasta la otra esquina, momento en que se suelta.
PICKFIRST	Entero	Registro	Controla el método de designación de objetos, de forma que se pueda designar en primer lugar los objetos y después utilizar un comando de edición o consulta (designación Nombre-verbo): 0 Desactiva PICKFIRST. 1 Activa PICKFIRST.
PICKSTYLE	Entero	Dibujo	Controla la designación de grupo y de sombreado asociativo: 0 Ninguna designación de grupo ni de sombreado asociativo 1 Designación de grupo. 2 Designación de sombreado asociativo. 3 Designación de grupo y de sombreado asociativo.
PLATFORM	Cadena	No guardada (de sólo lectura)	Indica qué plataforma de AutoCAD está en uso. Puede aparecer una de las siguientes cadenas: "Microsoft Windows NT Version 3.51 (x86)" "Microsoft Windows NT Version 4.00 (x86)" "Microsoft Windows Version 4.00 (x86)"
PLINEGEN	Entero	Dibujo	Establece la generación del patrón de tipo de línea alrededor de los vértices de una polilínea bidimensional. No se aplica a polilíneas con segmentos cónicos: 0 El tipo de línea se genera siempre con un trazo en cada vértice. 1 Genera el tipo de línea en un patrón continuo, pudiendo no haber trazo en los vértices.
PLINETYPE	Entero	Registro	Especifica cuándo utiliza AutoCAD polilíneas optimizadas de la versión 14: 0 Las polilíneas de dibujos antiguos no se convierten y POL crea polilíneas sin optimizar. 1 Las polilíneas de dibujos antiguos no se convierten al abrir estos, pero POL crea polilíneas optimizadas. 2 Las polilíneas de dibujos antiguos son convertidas automáticamente al abrirse estos y POL crea polilíneas optimizadas.
PLINEWID	Real	Dibujo	Establece el gorsor de polilínea por defecto.
PLOTID	Cadena	Registro	Modifica el trazador por defecto en función de la descripción asignada, y conserva la cadena de texto de la descripción del trazador actual. Para cambiar a otro trazador configurado, se escribe su descripción total o parcial.
PLOTROTMODE	Entero	Registro	Controla la orientación de los trazados:

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	Significado
			0 Gira el área de trazado efectiva de
			forma que la esquina con el icono <i>Girar</i> se alinee con el papel en la parte inferior izquierda en 0, en la parte superior izquierda en 90, en la superior derecha en 180 y en la inferior derecha en 270:
PLOTTER	Entero	Registro	1 Alinea la esquina inferior izquierda del área de trazado efectiva con la esquina inferior izquierda del papel. Cambia el trazador por defecto, en función del entero asignado, y conserva el número entero que AutoCAD asigna a cada trazador configurado. Este número puede estar comprendido entre 0 y el número de trazadores configurados menos 1. Se puede configurar un máximo de 29 trazadores por defecto.
POLYSIDES	Entero	No guardada	Establece el número de lados por defecto para POLIGONO. El rango válido es 3 a 1024.
POPUPS	Entero	No guardada (de sólo lectura)	Visualiza el estado del gestor de pantalla actualmente configurado: 0 No admite cuadros de diálogo, barra de menús, menús desplegables ni menús de iconos.
PROJECTNAME	Cadena	Dibujo	1 Admite estas características. Almacena el nombre del proyecto actualmente seleccionado. Un proyecto consiste en uno o varios caminos de búsqueda para la localización de referencias externas e imágenes de trama.
PROJMODE	Entero	Registro	Establece el modo de proyección actual para las operaciones de recortar o alargar: 0 Modo tridimensional verdadero (sin proyección).
PROXYGRAPHICS	Entero	Dibujo	1 Proyecta al plano XY del SCP actual. 2 Proyecta al plano de vista actual. Especifica si las imágenes de objetos Proxy son guardadas en el dibujo: 0 Las imágenes no se guardan en el dibujo; sólo se muestra un rectángulo de borde.
PROXYNOTICE	Entero	Registro	1 Las imágenes se guardan en el dibujo. Controla si se muestra un mensaje de advertencia cuando se crean objetos Proxy en el dibujo: 0 No se muestra mensaje de advertencia.
PROXYSHOW	Entero	Registro	1 Se muestra mensaje de advertencia. Controla la visualización de objetos Proxy en el dibujo: 0 No se visualizan los objetos Proxy
PSLTSCALE	Entero	Dibujo	1 Se visualizan imágenes gráficas para todos los objetos Proxy 2 Sólo se muestran rectángulos de borde para todos los objetos Proxy. Controla la escala de tipos de línea relativa al Espacio Papel: 0 La escala de los tipos de línea, determinada por ESCALATL, se toma en unidades del Espacio (Modelo o Papel) en el que se crearon los objetos.

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	Significado
			1 La escala de la ventana gráfica
PSPROLOG	Cadena	Registro	controla la escala de tipo de línea. Si TILEMODE está establecida en 0, la escala se toma relativa al Espacio Papel. Asigna un nombre a una sección de prólogo leída desde el archivo ACAD.PSF al usar SALVAPS.
PSQUALITY	Entero	Dibujo	Controla la calidad de modelizado de las imágenes <i>PostScript</i> y si se dibujan como objetos rellenos o como contornos: =0 Desactiva la generación de imágenes <i>PostScript</i> . <0 Establece el número de píxeles por unidad de dibujo de AutoCAD para la resolución <i>PostScript</i> . >0 Establece el número de píxeles por unidad de dibujo, pero utiliza el valor absoluto, lo que hace que AutoCAD muestre los caminos <i>PostScript</i> como contornos sin rellenar.
QTEXTMODE	Entero	Dibujo	Controla la visualización de textos (comando LOCTEXTO): 0 Los textos se muestran con todo su contenido. 1 Los textos se muestran sólo como rectángulos.
RASTERPREVIEW	Entero	Registro	Controla si se guarda una imagen de previsualización BMP junto con el dibujo: 0 No se crea imagen de previsualización. 1 Se crea imagen de previsualización.
REGENMODE	Entero	Dibujo	Especifica si está desactivada o activada la regeneración automática del dibujo.
RE-INIT	Entero	No guardada	Reinicializa las puertas de E/S, el digitalizador, y el archivo ACAD.PGP mediante los siguientes códigos binarios: 1 Reinicialización de la puerta E/S del digitalizador. 4 Reinicialización del digitalizador. 16 Reinicialización del archivo PGP (se vuelve a cargar).
RTDISPLAY	Entero	Registro	Controla la visualización de imágenes <i>raster</i> durante el zoom y encuadre en tiempo real: 0 Se muestra el contenido de la imagen <i>raster</i> . 1 No se muestra el contenido de la imagen <i>raster</i> .
SAVEFILE	Cadena	Registro (de sólo lectura)	Almacena el nombre del archivo de guardado automático actual.
SAVENAME	Cadena	No guardada (de sólo lectura)	Almacena el nombre del archivo en el que se guarda el dibujo actual.
SAVETIME	Entero	Registro	Establece el intervalo de guardado automático en minutos: =0 Desactiva el guardado automático. >0 Guarda automáticamente el dibujo en intervalos determinados por un entero distinto de cero. El cronómetro de SAVETIME se pone en marcha en cuanto se realiza un cambio en un dibujo. Se restablece y reinicializa mediante la ejecución

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	Significado
			manual del comando GUARDAR, GUARDARCOMO o GUARDARR. El dibujo 0
SCREENBOXES	Entero	Registro (de sólo lectura)	actual se guarda en AUTO.SV\$, AUTO1.SV\$, etc. Almacena el número de casillas en el área del menú de pantalla del área gráfica. Si el menú de pantalla está desactivado, SCREENBOXES vale cero. En plataformas que permitan ajustar el tamaño de la ventana gráfica de AutoCAD o volver a configurar el menú de pantalla durante una sesión de edición, es posible que el valor de esta variable cambie durante la sesión de edición.
SCREENMODE	Entero	Registro (de sólo lectura)	Almacena un código binario que indica el estado de los gráficos o texto de la ventana de AutoCAD . Es la suma de los siguientes valores de bit: 0 Se visualiza la pantalla de texto. 1 Se visualiza el modo gráfico. 2 Se configura una pantalla dual.
SCREENSIZE	Punto	No guardada (de sólo lectura)	Almacena el tamaño de la ventana gráfica actual en píxeles (X e Y).
SHADEEDGE	Entero	Dibujo	Controla el matizado de lados durante el modelizado: 0 Caras sombreadas (256 colores), lados no resaltados. 1 Caras sombreadas (256 colores), lados dibujados en el color de fondo. 2 Caras no rellenas, lados en el color del objeto. Semejante a OCULTA. 3 Caras en el color del objeto (16 colores), lados en el color de fondo.
SHADEDIF	Entero	Dibujo	Establece la relación entre la luz reflectante difusa y la luz ambiental (en porcentaje de luz reflectante difusa).
SHPNAME	Cadena	No guardada	Establece el nombre de la forma por defecto. Debe satisfacer las convenciones de denominación de símbolos. Si no se establece ningún valor por defecto, devuelve "". Se introduce un punto (.) para no establecer ningún valor por defecto.
SKETCHINC	Real	Dibujo	Establece el incremento de generación de BOCETO.
SKPOLY	Entero	Registro	Determina si BOCETO genera líneas o polilíneas: 0 Genera líneas. 1 Genera polilíneas.
SNAPANG	Real	Dibujo	Establece el ángulo de rotación de Forzcursor/Rejilla (en relación con el SCP) de la ventana gráfica actual.
SNAPBASE	Punto	Dibujo	Establece el punto de origen de Forzcursor/Rejilla de la ventana gráfica bidimens. actual (en coordenadas X e Y del SCP).
SNAPISOPAIR	Entero	Dibujo	Controla el plano isométrico en curso de la ventana gráfica actual: 0 Izquierda. 1 Superior. 2 Derecha.
SNAPMODE	Entero	Dibujo	Controla el modo Forzcursor:

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	Significado
			0 Desactiva el modo Forzcursor. 1 Activa el modo Forzcursor para la
SNAPSTYL	Entero	Dibujo	ventana gráfica actual. Establece el estilo de resolución de la ventana gráfica actual: 0 Estándar. 1 Isométrico.
SNAPUNIT	Punto	Dibujo	Establece el intervalo de la malla de resolución (X e Y) de la ventana gráfica actual.
SORTENTS	Entero	Dibujo	Controla la visualización de las operaciones de ordenación de objetos mediante la suma de uno o varios de los siguientes códigos: 0 Desactiva SORTENTS. 1 Ordena para la designación de objetos. 2 Ordena para la referencia a objetos. 4 Ordena para el redibujado. 8 Ordena para la creación de fotos de SACA FOTO. 16 Ordena para REGEN. 32 Ordena para el trazado. 64 Ordena para la salida <i>PostScript</i> .
SPLFRAME	Entero	Dibujo	Controla la visualización de curvas spline y polilíneas adaptadas a curva-B: 0 No se visualiza el polígono de control para las splines y polilíneas adaptadas. Se visualiza la superficie de ajuste de una malla poligonal, y no las mallas de definición. No se muestran los lados invisibles de las caras 3D ni de las mallas policara. 1 Se visualizan el polígono de control, la malla de definición y los lados invisibles en los casos anteriores.
SPLINESEGS	Entero	Dibujo	Establece el número de segmentos de línea generado para cada segmento de polilínea adaptada en curva-B.
SPLINETYPE	Entero	Dibujo	Establece el tipo de curva spline generado por EDITPOL CurvaB: 5 Curva-B cuadrática. 6 Curva-B cúbica.
SURFTAB1	Entero	Dibujo	Establece el número de caras generadas por SUPREGA y SUPTAB. Define también la resolución de malla en la dirección M para SUPREV y SUPLADOS.
SURFTAB2	Entero	Dibujo	Establece la resolución de malla en la dirección N para SUPREV y SUPLADOS.
SURFTYPE	Entero	Dibujo	Controla el tipo de ajuste de superficie realizada por EDITPOL Amoldar en mallas poligonales: 5 Superficie B-spline cuadrática. 6 Superficie B-spline cúbica. 8 Superficie Bézier.
SURFU	Entero	Dibujo	Establece la resolución de superficie amoldada en la dirección M.
SURFV	Entero	Dibujo	Establece la resolución de superficie amoldada en la dirección N.
SYSCODEPAGE	Cadena	Dibujo (de sólo lectura)	Indica la página de códigos del sistema determinada en ACAD.XMF. Los códigos son los siguientes: ascii, dos860, dos932, iso8859-7, big5, dos861, gb2312, iso8859-8, dos437, dos863, iso8859-1, iso88599,

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	Significado
			dos850, dos864, iso8859-2, johab, dos852, dos865, iso8859-3, ksc5601,
			dos855, dos866, iso8859-4, mac-roman, dos857, dos869, iso8859-6.
TABMODE	Entero	No guardada	Controla el uso del modo Tablero: 0 Desactiva el modo Tablero. 1 Activa el modo Tablero.
TARGET	Punto	Dibujo (de sólo lectura)	Almacena la ubicación (en coordenadas del SCP) del punto de mira de la ventana gráfica actual.
TDCREATE	Real	Dibujo (de sólo lectura)	Almacena la fecha y hora de creación del dibujo.
TDINDWG	Real	Dibujo (de sólo lectura)	Almacena el tiempo total de edición.
TDUPDATE	Real	Dibujo (de sólo lectura)	Almacena la fecha y hora de la última actualización o guardado.
TDUSRTIMER	Real	Dibujo (de sólo lectura)	Almacena el tiempo transcurrido en el cronómetro del usuario.
TEMPPREFIX	Cadena	No guardada (de sólo lectura)	Contiene el nombre del directorio (si existe) configurado para ubicar los archivos temporales, con un separador de camino incluido.
TEXTEVAL	Entero	No guardada	Controla el método de evaluación de cadenas de texto: 0 Todas las respuestas a las solicitudes de cadenas de texto y valores de atributo se interpretan literalmente. 1 Si el texto que comienza con (o ! se evalúa como una expresión de AutoLISP.
TEXTFILL	Entero	Registro	Controla el relleno de los tipos de letra <i>TrueType</i> durante el trazado, exportación con SALVAPS y modelizado: 0 Visualiza texto como contornos. 1 Visualiza texto como imágenes rellenas.
TEXTQLTY	Real	Dibujo	Establece la resolución de los tipos de letra <i>TrueType</i> . Los valores representan puntos por pulgada. Con valores bajos se reduce la resolución y aumenta la velocidad de visualización y de trazado. Con valores altos aumenta la resolución y disminuye la velocidad de visualización y de trazado. Los valores válidos comprenden del 0 al 100.0.
TEXTSIZE	Real	Dibujo	Establece la altura por defecto para los textos (independientemente de que el estilo tenga una altura fija).
TEXTSTYLE	Cadena	Dibujo	Contiene el nombre del estilo de texto actual.
THICKNESS	Real	Dibujo	Establece la altura de objeto tridimensional actual.
TILEMODE	Entero	Dibujo	Controla el acceso al Espacio Papel, así como el comportamiento de las ventanas gráficas de AutoCAD : 0 Activa los objetos de Espacio Papel y de ventana gráfica (utiliza VMULT). AutoCAD borra el área gráfica y pide al usuario que cree una o varias ventanas gráficas. 1 Activa el modo de compatibilidad con la versión 10 (utiliza VENTANAS). AutoCAD vuelve al modo de ventanas en mosaico, restableciendo la última configuración activa de ventanas gráficas en mosaico. No se visualizan

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	Significado
			los objetos de espacio papel, ni siquiera los objetos de ventana gráfica, y se
TOOLTIPS	Entero	Registro	desactivan los comandos VMULT, ESPACIOM, ESPACIOP y VGCAPA. Controla si se activa o no la visualización de pistas al posar el cursor sobre los botones de herramientas.
TRACEWID	Real	Dibujo	Establece la anchura de trazo por defecto.
TREEDEPTH	Entero	Dibujo	Determina la profundidad máxima, es decir, el número de veces que el índice espacial de árbol octal puede dividirse en ramas: =0 Elimina totalmente el índice espacial, suprimiendo la mejora de rendimiento que proporciona cuando se trabaja con dibujos grandes. Con este parámetro, se asegura que los objetos sean procesados siempre en el orden de la base de datos, haciendo innecesario establecer la variable de sistema SORTENTS. >0 Activa TREEDEPTH. Es válido cualquier entero de cuatro dígitos como máximo. Los primeros dos dígitos hacen referencia al Espacio Modelo y los últimos dos dígitos al Espacio Papel. <0 Los objetos de Espacio Modelo se tratan como bidimensionales (se ignoran las coordenadas Z), como sucede siempre con los objetos de Espacio Papel. Este parámetro es apropiado para los dibujos bidimensionales y proporciona un uso más eficaz de la memoria, sin pérdida de rendimiento.
TREEMAX	Entero	Registro	Limita el consumo de memoria durante la regeneración del dibujo al limitar el número máximo de puntos del índice espacial (árbol octal). Al imponer un límite fijo con TREEMAX, se pueden cargar dibujos creados en sistemas con más memoria que el sistema del usuario y con un valor de TREEDEPTH superior al que el sistema pueda manejar.
TRIMMODE	Entero	Registro	Controla si AutoCAD recorta los lados designados para chaflanes y empalmes: 0 Deja intactos los lados designados. 1 Recorta los lados designados hasta los puntos finales de las líneas de chaflán y arcos de empalme.
UCSFOLLOW	Entero	Dibujo	Genera una vista en planta siempre que se cambia de SCP. Se puede establecer UCSFOLLOW para cada ventana gráfica por separado. Si UCSFOLLOW está activada para una determinada ventana gráfica, AutoCAD genera una vista en planta en esa ventana gráfica al cambiar los sistemas de coordenadas: 0 El SCP no afecta a la vista. 1 Cualquier cambio de SCP genera una

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

UCSICON Nombre	Entero Tipo	Dibujo En	vista en planta del nuevo SCP en la ventana gráfica actual. Visualiza el icono del sistema de Significado
			<p>coordenadas mediante un código binario para la ventana gráfica actual. Es la suma de los siguientes valores:</p> <p>1 Activado; se activa la visualización del icono.</p> <p>2 Origen; si está activada la visualización del icono, el icono se desplaza hasta el origen del SCP, si es posible.</p>
UCSNAME	Cadena	Dibujo (de sólo lectura)	<p>Almacena el nombre del sistema de coordenadas del espacio actual. Devuelve una cadena vacía si el SCP actual no tiene asignado un nombre.</p>
UCSORG	Punto	Dibujo (de sólo lectura)	<p>Almacena el punto de origen del sistema de coordenadas para el espacio tridimensional actual. El valor se devuelve siempre en coordenadas universales.</p>
UCSXDIR	Punto	Dibujo (de sólo lectura)	<p>Almacena la dirección X del SCP para el espacio actual tridimensional.</p>
UCSYDIR	Punto	Dibujo (de sólo lectura)	<p>Almacena la dirección Y del SCP para el espacio actual tridimensional.</p>
UNDOCTL	Entero	No guardada (de sólo lectura)	<p>Almacena un código binario que indica el estado de la función DESHACER. Es la suma de los siguientes valores:</p> <p>0 Se desactiva DESHACER.</p> <p>1 Se activa DESHACER.</p> <p>2 Sólo puede deshacerse un comando.</p> <p>4 Se activa la opción Auto.</p> <p>8 Un grupo está actualmente activo.</p>
UNDOMARKS	Entero	No guardada (de sólo lectura)	<p>Almacena el número de marcas colocadas en la secuencia de control DESHACER por la opción Marca. Las opciones Marca y Retorno no están disponibles si un grupo está actualmente activo.</p>
UNITMODE	Entero	Dibujo	<p>Controla el formato de visualización de las unidades:</p> <p>0 Visualiza ángulos fraccionarios, en pies y pulgadas, y geodésicos, según se hayan definido previamente.</p> <p>1 Visualiza ángulos fraccionarios, en pies y pulgadas, y geodésicos en formato de entrada.</p>
USERI1-5	Entero	Dibujo	<p>Disponibles para el usuario para almacenar y recuperar valores enteros.</p>
USERR1-5	Real	Dibujo	<p>Disponibles para el usuario para almacenar y recuperar valores reales.</p>
USERS1-5	Cadena	No guardadas	<p>Disponibles para el usuario para almacenar y recuperar valores de cadenas de texto.</p>
VIEWCTR	Punto	Dibujo (de sólo lectura)	<p>Almacena el centro de la vista de la ventana gráfica actual, expresado en coordenadas de SCP.</p>
VIEWDIR	Vector	Dibujo (de sólo lectura)	<p>Almacena la línea de mira de la ventana gráfica actual, expresada en coordenadas de SCP, que describe el punto de cámara mediante su distancia del punto de mira.</p>
VIEWMODE	Entero	Dibujo (de sólo lectura)	<p>Controla el modo de visualización de la ventana gráfica actual mediante códigos binarios. El valor es la suma de los siguientes valores de bit:</p>

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

Nombre	Tipo	En	Significado
			0 Desactivada. 1 Vista en perspectiva activa. 2 Plano delimitador frontal activo. 4 Plano delimitador trasero activo. 8 Modo de seguimiento de SCP activo (UCSFOLLOW). 16 Plano delimitador frontal no ubicado en la mira. Si está activado, la distancia del plano delimitador frontal (FRONTZ) determina el plano delimitador frontal. Si está desactivado se ignora FRONTZ y se establece el plano delimitador frontal para que atraviese el punto de cámara. Almacena la altura de la vista de la ventana gráfica actual, expresada en unidades de dibujo.
VIEWSIZE	Real	Dibujo (de sólo lectura)	Almacena el ángulo de ladeo de la vista en la ventana gráfica actual
VIEWTWIST	Real	Dibujo (de sólo lectura)	Controla la visibilidad de las capas en los archivos de referencias externas:
VISRETAIN	Entero	Dibujo	0 La definición de capa de RefX en el dibujo actual tiene prioridad sobre estos parámetros: Act/Des, Inutilizar/Reutilizar, color y los parámetros de tipo de línea para las capas dependientes de RefX. 1 Act/Des, Inutilizar/Reutilizar, color y los parámetros de tipo de línea para las capas dependientes de RefX tienen prioridad sobre la definición de RefX en el dibujo actual.
VSMAX	Punto	Dibujo (de sólo lectura)	Almacena la esquina superior derecha de la pantalla virtual de la ventana tridimensional gráfica actual, expresada en coordenadas de SCP.
VSMIN	Punto	Dibujo (de sólo lectura)	Almacena la esquina inferior izquierda de la pantalla virtual de la ventana gráfica tridimensional actual, expresada en coordenadas de SCP.
WORLDUCS	Entero	No guardada (de sólo lectura)	Indica si el SCP actual es el Sistema de Coordenadas Universales:
WORLDVIEW	Entero	Dibujo	0 El SCP actual es diferente al Sistema de Coordenadas Universales. 1 El SCP actual es el Sistema de Coordenadas Universales.
XCLIPFRAME	Entero	Dibujo	Controla si el SCP cambia a SCU durante VISTADIN o PTOVISTA:
XLOADCTL	Entero	Registro	0 El SCP actual permanece inalterable. 1 El SCP cambia a SCU mientras se ejecutan los comando VISTADIN o PTOVISTA. La entrada de estos comandos está en relación con el SCP actual.
XLOADPATH	Cadena	Registro	Controla la visibilidad de los bordes de delimitación de RefX. Controla la activación de la carga bajo demanda de RefX: 0 Desactiva la carga bajo demanda. Se carga el dibujo externo entero. 1 Activa la carga bajo demanda. El dibujo externo es abierto. 2 Activa la carga bajo demanda. Se abre una copia del dibujo externo.
			Crea un camino para el almacenamiento temporal de las copias de archivos externos durante la carga bajo demanda.

Curso Práctico de Personalización y Programación bajo AutoCAD
Variables de sistema y acotación

XREFCTL	Entero	Registro	Controla si AutoCAD escribe archivos de registro .XLG de incidencias de referencias externas:
Nombre	Tipo	En	Significado

0 No se escriben archivos de revisión de RefX.
1 Se escriben archivos de revisión de RefX.

APÉNDICE C

Bibliotecas suministradas

C.1. TIPOS DE LÍNEA ESTÁNDAR

Nombre	Representación ASCII aproximada
Trazos	-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
Líneas_ocultas	- - - - - - - - - - - - - - - - - - - - - -
Centro	--- --- --- --- --- --- --- --- --- --- --- ---
Vals	--- - - - - - - - - - - - - - - - - - - - - -
Puntos
Trazo_y_punto	-- . -- . -- . -- . -- . -- . -- . -- . -- . -- .
Morse_G	-- -- . --- . --- . --- . --- . --- . --- . ---
Morse_D	-- . . -- . . -- . . -- . . -- . . -- . . -- . .
Acad_iso02w100	-----
Acad_iso03w100	-----
Acad_iso04w100	----- . ----- . ----- . ----- . -----
Acad_iso05w100	----- . . ----- . . ----- . . ----- . . -----
Acad_iso06w100	----- . . . ----- . . . ----- . . . ----- . .
Acad_iso07w100
Acad_iso08w100	-----
Acad_iso09w100	-----
Acad_iso10w100	----- . ----- . ----- . ----- . ----- . -----
Acad_iso11w100	----- . ----- . ----- . ----- . ----- . -----
Acad_iso12w100	----- . . ----- . . ----- . . ----- . . -----
Acad_iso13w100	----- . . ----- . . ----- . . ----- . . -----
Acad_iso14w100	----- . . . ----- . . . ----- . . . ----- . . . -----
Acad_iso15w100	----- . . . ----- . . . ----- . . . ----- . . . -----

C.2. TIPOS DE LÍNEA COMPLEJOS

[illegible]

C.3. PATRONES DE SOMBREADO

Nombre	Descripción
ANGLE	Ángulos rectos
ANSI31	Línea continuas inclinadas 45°; misma separación
ANSI32	Líneas continuas inclinadas 45° dos a dos
ANSI33	Líneas continuas y a trazos cortas inclinadas 45°

Nombre	Descripción
ANSI34	Líneas continuas inclinadas 45°; grupos de cuatro
ANSI35	Líneas continuas y a trazos largas inclinadas 45°
ANSI36	Líneas a trazos inclinadas 45° y desfasadas
ANSI37	Líneas continuas inclinadas 45° y 135°; cruzadas
ANSI38	Líneas continuas inclinadas 45° y tramos cortos a 135°
AR-B816	Tramado de ladrillos rectangulares
AR-B816C	Tramado de ladrillos rectangulares doble
AR-B88	Tramado de ladrillos cuadrados
AR-BRELM	Tramado de ladrillos desiguales doble
AR-BRSTD	Tramado de ladrillos rectangulares pequeños
AR-CONC	Hormigón
AR-HBONE	Maderas cruzadas
AR-PARQ1	Parqué
AR-PROOF	Múltiples líneas
AR-RSHKE	Tejas
AR-SAND	Arena
BOX	Cuadrados y líneas verticales
BRASS	Líneas horizontales continuas y a trazos
BRICK	Ladrillos
BRSTONE	Ladrillos de piedra
CLAY	Líneas horizontales continuas en grupo y a trazos
CORK	Sembrados
CROSS	Cruces
DASH	Línea horizontales a trazos y desfasadas
DOLNIT	Líneas horizontales y tramos cortos a 45°
DOS	Puntos
EARTH	Tierra sembrada
ESCHER	Figura de Escher
FLEX	Símbolo de integración tumbado
GRASS	Hierba
GRATE	Tramado
HEX	Hexágonos
HONEY	Panal de abejas
HOUND	Tramado de tramos líneas cruzadas a 90°
INSUL	Líneas continuas y a trazos horizontales
LINE	Líneas horizontales continuas
MUDST	Líneas de ejes de doble punto horizontales
NET	Líneas continuas cruzadas a 90°
NET3	Líneas continuas de tres tipos de ángulos
PLAST	Líneas continuas horizontales en grupos de tres
PLASTI	Líneas continuas horizontales en grupos de tres y uno
SANCNR	Líneas continuas a 45° y puntos
SQUARE	Cuadrados
STARS	Estrellas de seis puntas
STEEL	Líneas continuas a 45° en grupos de dos
SWAP	Pequeños símbolos de matorral
TRANS	Líneas continuas horizontales y a trazos sin desfasar
TRIANG	Triángulos invertidos
ZIGZAG	Tramos horizontales y verticales haciendo zig-zag
ISO02W100	Líneas horizontales a trazos poco separadas

ISO03W100	Líneas horizontales a trazos muy separadas
-----------	--------------------------------------------

Nombre	Descripción
ISO04W100	Líneas horizontales a trazo grande y punto
ISO05W100	Líneas horizontales a trazo grande y dos puntos
ISO06W100	Líneas horizontales a trazo grande y tres puntos
ISO07W100	Puntos
ISO08W100	Líneas horizontales a trazo grande y trazo pequeño
ISO08W100	Líneas horizontales a trazo grande y dos trazos pequeños
ISO10W100	Líneas horizontales a trazo grande y punto
ISO11W100	Líneas horizontales a dos trazos grandes y punto
ISO12W100	Líneas horizontales a trazo grande y dos puntos
ISO13W100	Líneas horizontales a dos trazos grandes y dos puntos
ISO14W100	Líneas horizontales a trazo grande y tres puntos
ISO15W100	Líneas horizontales a dos trazos grandes y tres puntos

C.4. PATRONES DE RELLENO *PostScript*

Nombre	Descripción
GrayScale	Escala de grises
RGBColor	Color RGB
AILogo	Logotipo de Autodesk
LinearGray	Escala de grises lineal
RadialGray	Escala de grises radial
Square	Cuadrados
Waffle	Figuras 3D
ZigZag	Figura de zig-zag
Stars	Estrelas de cinco puntas
Brick	Ladrillos
Specs	Cuadrados sin orden aparente

C.5. Tipos de letra basados en definición de formas

Nombre	Descripción
Txt	Fuente estándar proporcional
Monotxt	Fuente estándar con monoespaciado
Iso	Fuentes normales ISO en vertical de 8 bits
Simplex	Antigua de trazo simple (rudimentaria)
Romans	Antigua de trazo simple (rudimentaria)
Italic	Cursiva simple
Scripts	Escritura corrida con enlaces (trazo simple)
Greeks	Griega de trazo simple
Romand	Antigua de trazo doble (rudimentaria)
Complex	Antigua compleja (rudimentaria)
Romanc	Antigua compleja (rudimentaria)

Italicc	Cursiva compleja
Scriptc	Escritura corrida con enlaces (compleja)
Nombre	Descripción
Greekc	Griega de compleja
Cyrillic	Cirílica alfabética
Cyriltilc	Cirílica de transliteración
Romant	Antigua de trazo simple (palo cruzado)
Italict	Cursiva triple
Ghotice	Gótica inglesa
Ghoticg	Gótica alemana
Ghotici	Gótica italiana
Syastro	Símbolos astronómicos
Symap	Símbolos cartográficos
Symath	Símbolos matemáticos
Symeteo	Símbolos meteorológicos
Symusic	Símbolos musicales
GDT	Símbolos de tolerancias geométricas

C.6. Fuentes *True Type*

Familia	Tipo	Nombre	Descripción
Swiss 721 Regular	Sans Serif regular	swiss swissl swissli swissi swisslb swissbi swissk swisski	redonda fina cursiva fina cursiva negrita cursiva negrita gruesa cursiva gruesa
Swiss 721 Condensed	Sans Serif condensada	swissc swisscl swisscli swissci swisscb swisscbi swissck swisscki	condensada condensada fina condensada fina cursiva condensada cursiva condensada negrita condensada negrita cursiva condensada gruesa condensada gruesa cursiva
Swiss 721 Expanded	Sans Serif extendida	swisse swissei swisseb swissek	extendida extendida cursiva extendida negrita extendida gruesa
Swiss 721 Outline	Sans Serif hueca	swissbo swissko swisscbo	negrita hueca gruesa hueca condensada negrita hueca
Monospace 821	Sans Serif	monos	redonda

Familia	Tipo	Nombre	Descripción
	espaciado simple	monosi monosb	cursiva negrita
Dutch 801 Regular	Serif	monosbi dutch dutchi dutchb dutchbi dutchbi	negrita cursiva redonda cursiva negrita negrita cursiva negrita cursiva
Dutch 801 Expanded	Serif extendida	dutcheb	extendida negrita
Bank Gothic	Versal	bgothl bgothm	fina media
Commercial Script	Script	comsc	normal
Vineta	Sombreada	vinet	normal
Commercial Pi	Símbolo	comp	normal
Universal Math 1	Matemático	umath	normal

C.7. Símbolos de tolerancias geométricas

Símbolo	Descripción
Posición	Círculo con diámetros
Concéntrico/coaxial	Dos círculos concéntricos
Simétrico	Tres líneas paralelas; la del medio mayor
Par	Dos líneas paralelas inclinadas
Perpendicular	Dos líneas perpendiculares
Angular	Dos líneas en ángulo agudo
Cilíndrico	Círculo con líneas laterales inclinadas
Plano	Romboide
Circular o redondo	Círculo
Recto	Línea horizontal
Perfil de superficie	Semicírculo
Perfil de líneas	Semicircunferencia
Salida circular	Flecha
Salida total	Doble flecha
Diámetro	Círculo cruzado por línea oblicua
Condición máxima de material (CMM)	M encerrada en círculo
Condición mínima de material (CIM)	L encerrada en círculo
Independiente del tamaño (RFS)	S encerrada en círculo
Tolerancia proyectada	P encerrada en círculo