6 Programación en AutoLisp

6.1. introducción

A parte de todo lo visto en cuestión de personalización, **AutoCAD 14** 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 14 proporciona diversas interfaces de programación de aplicaciones que vamos a comentar ahora de forma somera.

6.1.1. AutoLISP

6.1.1.1. 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.

6.1.1.2. 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.

6.2. 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.

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.

6.2.1. 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	
Lista	Objeto que se compone de elementos encerrados entre paréntesis.
Elemento	Cualquiera de los componentes de una lista.
Símbolo	Cualquier elemento de una lista que no sea una constante.
Enteros	Valores numéricos enteros, sin punto decimal.
Reales	Valores numéricos con coma flotante de doble precisión.
Cadenas	Valores textuales que contienen cadenas de caracteres en código ASCII.
Descriptores de archivo	Valores que representan un archivo abierto para lectura o escritura.
Nombres de objetos de dibujo	Valores que representan el "nombre" hexadecimal de un objeto de la Base de Datos.
*Conjuntos designados de **AutoCAD***	Valores que representan un conjunto de elección de objetos de dibujo.
Funciones de usuario	Símbolo que representa el nombre de una función definida por el usuario.
Función inherente o subrutina	Símbolo con el nombre de una función predefinida de AutoLISP.

6.2.2. 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 14**.

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.

6.2.3. 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.

6.2.4. 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.

6. 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.

6.3.1. Convenciones de sintaxis

Las convenciones utilizadas para las sintaxis en estecapítulo van a ser las siguientes:

- 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.

6.4. 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 estás técnicas nos será tremendamente útil a la hora de lanzarnos a la programación pura.

6.4.1. 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.

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

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))
```

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*.

6.4.2. Matemática avanzada

(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:

(ABS –25.78) devuelve 25.78
(ABS 45) devuelve 45
(ABS 0) devuelve 0
(ABS –13) devuelve 13
(ABS (/ 2 3.0)) devuelve 0.666667
(ABS (/ 2 -3.0)) 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:

(FIX 32.79) devuelve 32
(FIX –12.45) devuelve –12
(FIX (/ 10 3.0)) devuelve 3
(FIX (/ 10 –3.0)) 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 * arctag 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 en 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	**tación**
Secante (sec x)	cos (x)
Cosecante (cosec x)	sen (x)
Arco-seno (arcsen x)	arctag (x / Ö (1 – x ^ 2))
Arco-coseno (arccos x)	1.5707633 – arctag (x / Ö (1 – x ^ 2))
Arco-secante (arcsec x)	arctag (Ö (x ^ 2 – 1)) + *signo* (x) –1) * 1.5707633
Arco-cosecante (arccos x)	arctag (1/ Ö (x ^ 2 – 1)) + *signo* (x) –1) * 1.570763
Arco-cotang. (arccotag x)	1.5707633 – arctag (x)

NOTA: El símbolo ^ significa exponenciación. Ö 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 un 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.

6.4.3. 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:

(= 5 5) devuelve T
(= 65 65.0) devuelve T
(= 7 54) devuelve nil
(= 87.6 87.6 87.6) devuelve T
(= 34 34 –34 34) devuelve nil

Veamos ahora algún ejemplo con cadenas:

(= "hola" "hola") devuelve T
(= "casa" "cAsa") devuelve nil
(= "H" "H" "H" "H") devuelve T
(= "hola ahora" "hola ahora") 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:

(< 2 3) devuelve T
(< 3 4 5 89 100) devuelve T
(< 3 –4 5 6) devuelve nil
(< (* 2 2) (/ 5 3)) devuelve nil

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:

(< "a" "b") devuelve T
(< "z" "h") devuelve nil
(< "A" "a" "b") devuelve T
(< "f" "S") devuelve nil

Si las cadenas tienen más caracteres se comparan de la misma forma:

(< "abc" "abd") devuelve T
(< "abc" "ab") devuelve nil

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

6.4.4. 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 en 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

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.

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.

A partir de la siguiente sección comenzaremos a ver para qué sirve todo esto y cómo utilizarlo prácticamente en programas propios.

6.5. 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.

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)

6.5.1. 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:

(QUOTE *expresión*)

o también:

(’*expresión*)

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:

(QUOTE (SETQ x 22.5)) devuelve (SETQ x 22.5)
(QUOTE hola) devuelve HOLA
(QUOTE (+ 3 3 3)) devuelve (+ 3 3 3)

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:

(QUOTE (50 50 –23)) devuelve (50 50 –23)

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.

6.6. 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 unas 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 14** 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 14**.

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:

(DEFUN func (x)... variable global: x
(DEFUN func (x y)... variables globales: x y
(DEFUN func (x / u z)... variable global: x variables locales: u z
(DEFUN func (/ x s)... 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)
)
```

NOTA IMPORTANTE DE SINTAXIS: EN LOS PROGRAMAS INCLUIREMOS LOS SANGRANDOS EN FORMA DE GUIONES, PERO ESTOS NO DEBEN SER INCLUIDOS REALMENTE EN EL CÓDIGO, SINO QUE SERÁN SUSTITUIDOS POR ESPACIOS BLANCOS.

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.

El botón *Descargar* descarga de memoria la aplicación designada y, el botón *Suprimir*, elimina una entrada de la lista.

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 ce 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**.

6.7. Captura Y Manejo Básico De Datos

6.7.1. 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. vamos a aprender la forma en 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 las 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: Apréciese 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: 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.

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.

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 capítulo 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 "linea" 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 línea seguidas en el archivo de texto, a no ser que nos resulte incómoda su extremada longitud para la edición.

6.7.2. 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 14** 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 número 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.

6.7.3. 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 de 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 de 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.

6.7.4. 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.

6.8. Acceso A Variables De Autocad 14

Vamos a explicar ahora, en esta sección, el control que podemos tener desde AutoLISP con respecto a las variables de sistema de **AutoCAD 14**.

Para lo que se refiere a este control tenemos a nuestra disposición dos funciones muy importantes y utilizadas en la programación en AutoLISP. Esta funciones son GETVAR y SETVAR. 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 14** indicada en *nombre_variable*, o sea, de cualquier variable del programa.

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 14** 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.

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 trasparencia 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 trasparente, 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:CircEjes) 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).

6.9. 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. Esta 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

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:

(PROMPT "Hola") devuelve Holanil
(PROMPT "Hola, soy yo") devuelve Hola, soy yonil
(PROMPT "1 + 2") devuelve 1 + 2nil
(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 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 (GETKWORD "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)
```

Expliquemos el ejemplo. El programa dibuja aros, huecos o rellenos, solicitando el centro del mismo, su radio intermedio y su grosor.

Se crea un 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 GETKWORD 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 GETKWORD 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 PROGN. Si no, se pasará a evaluar lo contenido en el segundo PROGN (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 GETKWORD. 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 PROGN, 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 establecen 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 GETKWORD. Por ejemplo:

```
(DEFUN Proceso ()
--(INITGET 1 "Constante Gradual Proporcional Ninguno")
--(SETQ Op (GETKWORD "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 GETKWORD, 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 mientas 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.

6.10. 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 elemento 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:

(CAR ’((/ 1 2.2) -80.2 -23.002 (* 2 3.3))) devuelve (/ 1 2.2)
(CAR ’(34.45 décimo -12)) devuelve 34.45
(CAR ’(x y z)) devuelve X
(CAR ’(3)) 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:

(CDR ’(8 80.01 -23.4 23 34.67 12)) devuelve (80.01 -23.4 23 34.67 12)
(CDR ’(x y z)) devuelve (Y Z)
(CDR (CAR ’((1 2 4) (3 5 7) (8 1 2)))) devuelve (2 4)

Si se indica un 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:

(CDR ’(30 20)) devuelve (20)
(CAR (CDR ’(30 20))) 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:

(CAR (CDR ’(20 12.4 -3))) devuelve 12.4
(CAR (CDR ’(34 -23.012 12.33))) devuelve -23.012
(CAR (CDR ’(23 12))) devuelve 12
(CAR (CDR ’(10 20 30 40 50 60))) 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:

(CADR '(10 20 34)) devuelve 20
(CADR '(23 -2 1 34 56.0 (+ 2 2))) devuelve -2
(CADR '(19 21)) devuelve 21
(CADR '(21)) devuelve nil
(CADR '()) 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 = CDDDAR, 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 esta 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).

6.11. 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.

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 14** LUNITS. Es decir, solamente puede ser un número entero entre 1 y 5 cuyo formato es el que se indica:

Modo	**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:

(RTOS (/ 1 3) 2 0) devuelve "0"
(RTOS (/ 1 3) 2 1) devuelve "0.3"
(RTOS (/ 1 3) 2 4) devuelve "0.3333"
(RTOS (/ 1 3) 2 13) devuelve "0.3333333333333"
(RTOS (/ 1 3) 2 16) 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:

(RTOS 2.567 1 2) devuelve "2.57E+00"
(RTOS -0.5679 5 3) devuelve "-5/8"
(RTOS 12 3 12) 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 14**. Sus valores están en el intervalo de 0 a 4 según la siguiente tabla:

Modo	**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:

(ANGTOS PI 0 2) devuelve "180"
(ANGTOS 1.2 3 3) devuelve "1.2r"
(ANGTOS (/ PI 2.0) 0 4) devuelve "90"
(ANGTOS 0.34 2 10) devuelve "21.6450722605g"
(ANGTOS -0.34 2 10) devuelve "378.3549277395g"

El ángulo indicado puede ser negativo, pero el valor es siempre convertido a positivo entre 0 y 2p.

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 ANGTOS. Si se omite *modo* se toma el valor actual de la variable AUNITS. Se pueden probar los ejemplos inversos a ANGTOS, son complementarios.

6.12. 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 14**.

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 de 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.

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.

(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: Apréciese que CHR devuelve cadenas de texto entrecomilladas.

(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 la 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 desharía 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 la 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 yb 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 le valor de ORTHOMODE.

Por último en cuestión se 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 dibujado 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)
```

6.13. 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 ocurría 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:

(INTERS '(10 10) '(20 20) '(15 10) '(20 0)) devuelve nil
(INTERS '(10 10) '(20 20) '(15 10) '(20 0) nil) devuelve (13.3333 13.3333)
(INTERS '(10 10) '(20 20) '(15 10) '(20 0) T) devuelve nil
(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.

6.14 Visual LISP

LISP fué inicialmente fue desarrollado como un lenguaje interpretado, aunque las modernas versiones cuentan siempre con un compilador que transforma el código fuente en lenguaje de máquina optimizado. Esta compilación puede ejecutarse de manera inmediata al cargar en el entorno de desarrollo el código fuente del programa, lo que facilita el desarrollo al disponer de una evaluación de manera inmediata. Para acceder a este entorno, en el caso del Visual LISP, se teclea desde la línea de comando de AutoCAD las instrucciones VLISP ó VLIDE (esta última para compatibilidad con el Visual LISP de

la versión 14). Las instrucciones LISP se introducen para su evaluación en una ventana especial conocida como la *Consola Visual LISP*. Si no está a la vista, se puede abrir esta ventana pulsando el botón de la barra de herramientas.

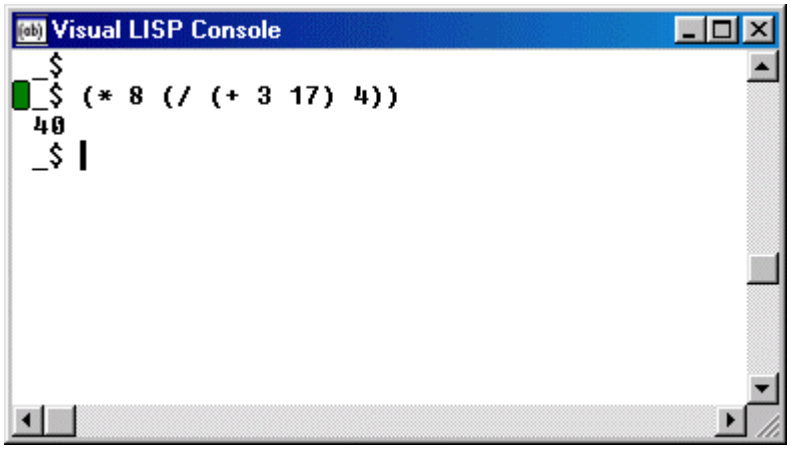

Figura 6-1 Consola de Visual AutoLisp

El cursor que aparece junto al símbolo _$ indica que le sistema está listo para recibir las expresiones LISP del usuario. La imagen anterior muestra el resultado de evaluar una expresión usando funciones aritméticas. La evaluación se produce al pulsar <INTRO>. Una vez impreso el resultado aparece de nuevo el símbolo _$ indicando que el sistema está listo para recibir una nueva expresión.

Este ciclo que se desarrolla en el intérprete se conoce como bucle de *lectura-evaluación-impresión* (*read-eval-print* loop). Esto significa que el intérprete *lee* lo que se ha tecleado, lo *evalúa* y entonces *imprime* el resultado antes de quedar listo para la nueva expresión. Al uso de la consola dedicaremos una sección específica de este curso.

El entorno de desarrollo (IDE) Visual LISP cuenta además con un Editor especializado y una serie de medios para la depuración de los programas muy superiores a los que estaban disponibles en el viejo AutoLISP.

No obstante, estas lecciones podrán ser seguidas utilizando cualquier versión de AutoLISP. Se ha tratado de señalar cuando se está hablando de funciones o modos de operación propios del Visual LISP que noi están disponibles en el entorno AutoLISP.

6.14.1 El Entorno De Desarrollo Visual Lisp

Visual LISP (VLISP) representa una renovación de LISP para AutoCAD, actualizándolo para incluir prestaciones que ya son normales en los modernos dialectos de LISP que se ajustan a la normativa COMMON LISP. Aún sin llegar a ser totalmente compatible con esta normativa, es significativo el incremento de su potencia como lenguaje de programación.

Es particularmente útil la posibilidad que se incorpora para la interacción con la jerarquía de objetos de la aplicación mediante la interfaz ActiveX ™ Automation de Microsoft, y la posibilidad de responder a eventos mediante la implementación de funciones diseñadas como reactores.

Como herramienta de desarrollo se aporta un Entorno de Desarrollo Integrado (IDE) que incluye un compilador y varias utilidades para la depuración.

El IDE Visual LISP incluye:

- Comprobador de Sintaxis que reconoce secuencias AutoLISP erróneas y el uso incorrecto de los argumentos en llamadas a las funciones primitivas del lenguaje.
- Compilador de Ficheros que incrementa la velocidad de ejecución y constituye una plataforma de distribución que brinda seguridad al código fuente.
- Depurador de Fuentes, diseñado específicamente para AutoLISP, que permite la ejecución paso a paso del código fuente en una ventana mientras se observan simultáneamente los resultados obtenidos en la pantalla gráfica de AutoCAD.
- Editor de Programación que emplea la codificación por color para LISP y DCL, así como otras características de apoyo sintáctico.
- Formateo LISP automático que redistribuye las líneas de código y las identa para facilitar la lectura de los programas.
- Amplias características de Inspección y Vigilancia (Watch) que permiten el acceso en tiempo real a los valores de las expresiones y las variables, y que pueden ser empleadas tanto para datos LISP como para objetos gráficos de AutoCAD.
- Ayuda sensible al contexto sobre las funciones AutoLISP y una ventana Apropos para búsqueda de nombres de símbolos.
- Sistema de Administración de Proyectos que facilitan el mantenimiento de aplicaciones con múltiples ficheros fuente.
- Empaquetado de los ficheros AutoLISP compilados en un único módulo de programa.
- Capacidad para guardar y recuperar la configuración del Escritorio para reutilizar la distribución de ventanas de cualquier sesión anterior de VLISP.
- Consola Visual LISP Inteligente que permite un nuevo nivel de interacción del usuario, con funciones que amplían las de la ventana de texto habitual de AutoCAD.

6.14.2 El Trabajo con Visual LISP y AutoCAD

VLISP posee su propia ventana de aplicación distinta de la de AutoCAD, pero no puede ejecutarse de manera independiente. Para acceder al IDE Visiual LISP, antes deberá haberse iniciado una sesión de AutoCAD.

Iniciar Visual LISP

Como se dijo antes, Debe haberse iniciado una sesión de AutoCAD. Esta sesión puede contener sólo un dibujo vacío, o pueden estar abiertos dibujos cuyo contenido se desee procesar.

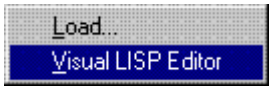

Figura 6-2 Como cargar el editor de Visual AutoLisp

Para activar el IDE VLISP tenemos tres opciones:

- Seleccionar del menú Herramientas>AutoLISP>Editor Visual LISP
- Teclear en la línea de comandos: VLISP

Nota: Hemos encontrado al menos en una versión localizada española que el comando VLIDE **no es reconocido** por el sistema. La Ayuda de esa misma versión aeñala como alternativa el comando VISUALLISPIDE, que tampoco es reconocido. En estos casos siempre se puede recurrir al comando VLIDE, descrito en el punto siguiente.

La versión anterior de Visual LISP utilizaba con los mismos fines el comando VLIDE, que sigue siendo reconocido por la versión 2000. De hecho, internamente la llamada de AutoCAD al IDE Visual LISP se realiza mediante este comando, que veremos aparecer en la línea de comandos cada vez que se cambie a ese entorno.

La Ventana de la Aplicación

Al iniciarse Visual LISP pasa a primer plano la siguiente ventana de aplicación:

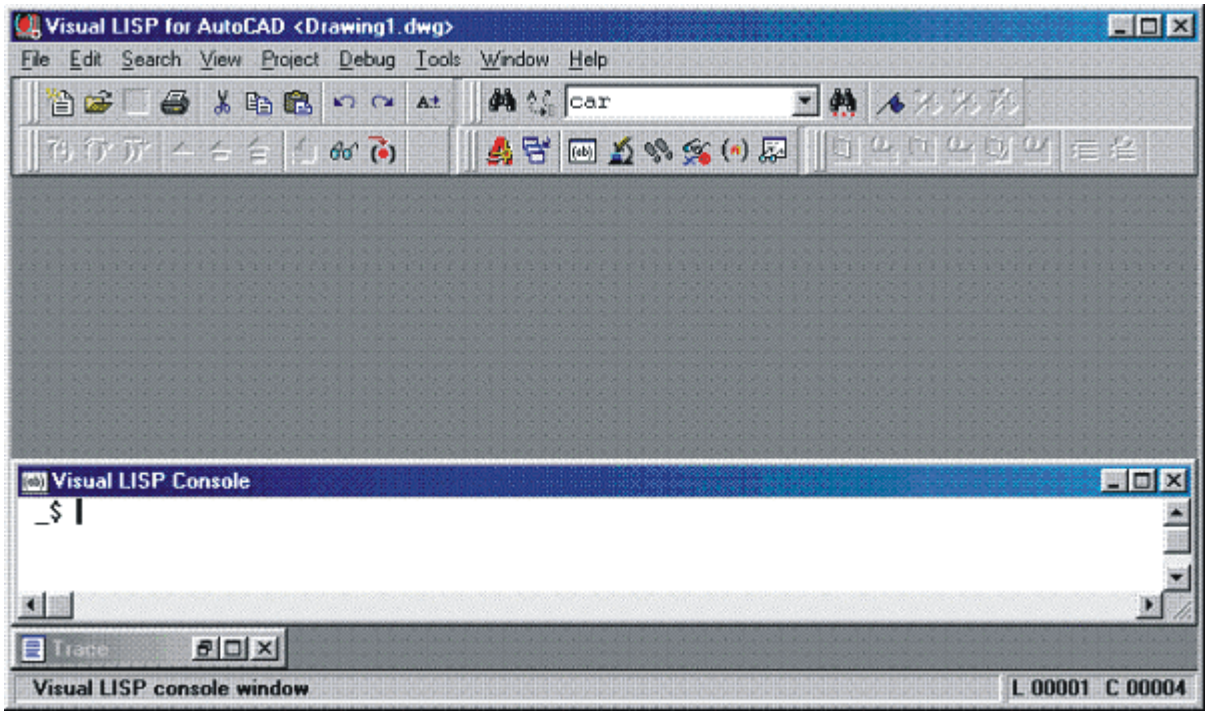

Figura 6-3 Entorno de Visual AutoLisp

6.14.3 Barra de Menús

Asumimos que el lector está familiarizado con el uso de menús desplegables en AutoCAD u otras aplicaciones. Sólo cabría destacar que éstos menús son sensibles al contexto en que se utilizan. Es decir, que algunas opciones que pueden estar activas si se abre la ventana del Editor pueden no estarlas si el foco se encuentra en la Consola o en la ventana de TRACE.

Las funciones de la aplicación se distribuyen entre los menús desplegables de la siguiente manera

FILE

Creación de nuevos ficheros de programas abiertos para su edición

Apertura de ficheros existentes

Guardar los cambios efectuados

Compilar aplicaciones Visual LISP

Imprimir los ficheros de programas

EDIT

Copiar y Pegar texto

Deshacer el último cambio en el Editor o la última función ejecutada desde la

Consola

Seleccionar texto en el Editor o la Consola

Comprobar el cierre de paréntesis

Recuperar funciones ejecutadas desde la Consola

SEARCH

Buscar y Reemplazar texto

Colocar marcadores de referencia (bookmarks) y navegar en el texto utilizando estos marcadores.

VIEW

Buscar y mostrar el valor de variables y símbolos en su código AutoLISP.

PROJECT

Trabaja con los proyectos y la compilación de programas

DEBUG

Establecer y quitar puntos de ruptura en los programas

Ejecutar programas paso a paso comprobando el estado de las variables y el valor devuelto por las expresiones.

TOOLS

Establecer las opciones para el formateado del texto y varias otras opciones del entorno, tales como la ubicación de ventanas y barras de herramientas.

WINDOW

Organiza, abre y cierra ventanas en el IDE.

HELP

Pues eso, la Ayuda en línea.

6.14.4 Las Barras de Herramientas

Visual LISP dispone de cinco Barras de Herramientas que pueden activarse/desactivarse desde el menú View>Toolbars... que abre el siguiente diálogo:

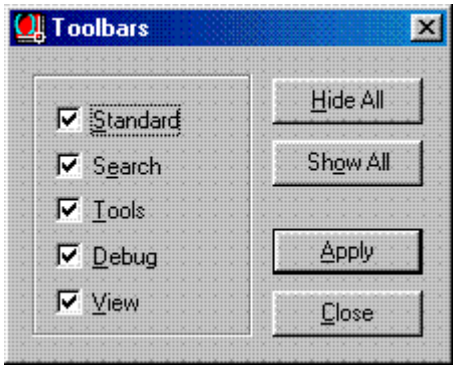

Figura 6-4 Propiedades de visualización de los ToolBars de Visual AutoLisp

Estas cinco Barras de Herramientas contienen las opciones u comandos esenciales del IDE, facilitando el acceso a los mismos.

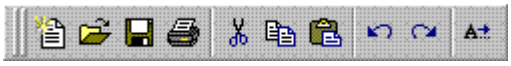

Figura 6-5 Barra de herramientas Estándar de Visual AutoLisp

Contiene las herramientas usuales de Crear Nuevo, Abrir, Guardar, Imprimir, Cortar, Copiar, Pegar, Deshacer, Rehacer y por último un acceso a la fución Apropos que sirve para completar el texto buscando una correlación de una subcadena con nombres de funciones, etc.

Figura 6-6 Barra de ejecución y búsqueda

Incluye las funciones de Buscar, Buscar y Reemplazar. Además una casilla de lista desplegable donde se guardan los términos anteriormente buscados durante la sesión de trabajo, lo que permite repetir una busqueda con más facilidad, cosa que se hace con la herramienta situada a la derecha de la casilla. Por último incluye una serie de herramientas para navegar dentro del texto mediante marcadores, que se introducen con la herramienta de la izquierda, se avanza o retrocede con las dos siguientes y se eliminan con la última.

Figura 6-7 Creación de etiquetas y carga de programas de Visual AutoLisp

La barra Tools (Herramientas) opera sólo con la ventana del Editor activa. Sus funciones son, de izquierda a derecha: cargar el código del Editor para su ejecución desde la Consola, cargar sólo el código seleccionado, comprobar la sintaxis de todo el contenido de la ventana del Editor, o con la siguiente, comprobar sólo lo seleccionado. Para formatear el texto se utilizan los próximos dos botones, el primero para todo el editor y el segundo sólo para el textoseleccionado. Los dos siguientes botones sirven para marcar como coimentario el texto seleccionado o para desmarcarlo. Y por supuesto, el último se trata de la Ayuda en línea.

Figura 6-8 Entrono ejecución de programas de Visual AutoLisp

Los tres primeros botones de esta barra determinan la acción al encontrar un punto de ruptura durante la evaluación. El primero entra en las expresiones anidadas posteriores al punto de ruptura, evaluándolas desde la más interior. El segundo evalúa esta expresión y se detiene antes de la siguiente para de nuevo decidir si se quiere entrar a evaluar las expresiones anidadas. El tercer botón continúa hasta el final de la función en curso y entonces cuando se detiene de nuevo la evaluación.

El segundo trío de botones tene que ver con las acciones a tomar cuando se produce una ruptura del flujo de ejecución a causa de un error, o de que se alcance un punto de ruptura prefijado dentro del programa. Aunque es un tema que se explicará más adelante, cabe decir ahora que estos estados de suspensión en la ejecución del programa se utilizan para examinar los valores asumidos por las variables, cambiarlos si es preciso, etc.

El primer botón *(Continue)* permite terminar con esta pausa y continuar la ejecución normal del programa. El segundo botón *(Quit)* permite abandonar el nivel de evaluación actual (pueden superponerse varios ciclos de evaluación si se producen varios errores durante la depuración) y pasar al nivel de más arriba. Y el tercer botón *(Reset)* pasa el control de la ejecución directamente al nivel superior *(Top Level)*.

El tercer grupo de botones incluye otras útiles herramientas de depuración. El botón *Toggle Breakpoint* permite añadir un nuevo punto de ruptura en el programa, situado en la posición actual del cursor. El segundo botón *(Add Watch)* da acceso al diálogo que permite seleccionar un nombre de variable para observar sus resultados durante la ejecución. Estos resultados se exhiben en una ventana especial, la ventana *Watch*.

El botón *Last Break* resalta en la ventana del editor la expresión que dio origen a la última ruptura. En caso de error, de esta manera se detecta de inmediato dónde se produjo éste.

El último botón no es realmente un botón de comando. Sirve simplemente para indicar si la interrupción actual se encuentra antes o después de la expresión.

Figura 6-9 Trace y Debug de Visual AutoLisp

El primer botón sirve para poner en primer plano la ventana de aplicación de AutoCAD. El segundo botón abre un menú donde podemos seleccionar la ventana del IDE Visual LISP que deseamos poner en primer plano. Esto se hace necesario pues podemos tener abiertas de manera simultánea un gran número de programas y puede no der fácil localizar aquélo que queremos.

El tercer botón traslada el foco a la Consola de Visual LISP. El siguiente permite activar la característica de Inspección *(Inspect)*. *Inspect* permite examinar y modificar objetos AutoLISP así como AutoCAD. La herramienta *Inspect* crea una ventana separada para cada objeto sometido a inspección.

La siguiente herramienta *(Trace Stack)* necesita explicaciones que se salen del marco de esta introducción. Baste decir que nos permite acceder a la memoria de pila donde se guardan las llamadas a función. Puede invocarse en un momento de suspensión en la ejecución de un programa y permite mediante un menú contextual acceder a datos relacionados con la operación del programa.

El botón que le sigue *(Symbol Service)* está diseñado para simplificar el acceso a las distintas utilidades de depuración relacionadas con símbolos. Podemos resaltar cualquier nombre de símbolo en el Editor y al pulsar sobre este botón se abrirá la ventana *Symbol Service* donde se muestra el valor vinculado. Este valor se puede cambiar directamente en esta ventana. Además tiene una barra de herramientas que permiten otros procesos con el símbolo seleccionado.

El siguiente botón permite abrir la ventana *Apropos* que describimos en detalle más adelante

Y por último un botón que permite acceder a la ventana *Watch*.

6.14.5 La Consola Visual LISP

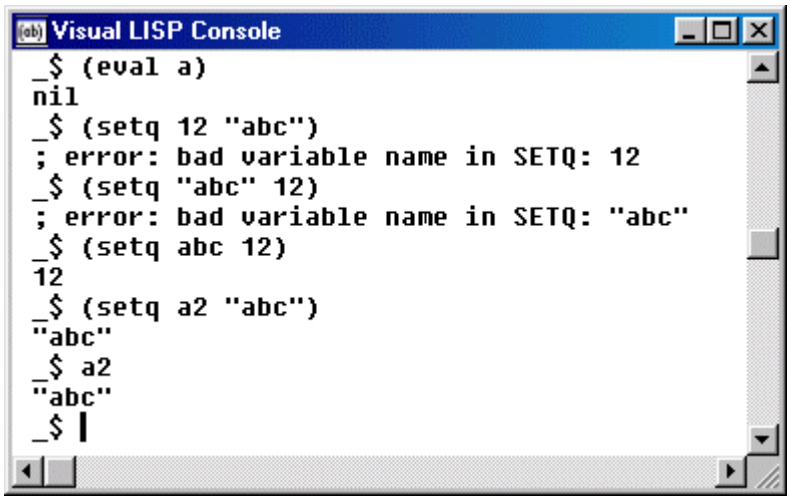

Figura 6-10 Consola de Visual AutoLisp

Aunque parecida a la ventana de texto de AutoCAD en el hecho de que se puden introducir funciones para evaluarlas y obtener el resultado en pantalla, la Consola es una herramienta de mucha mayor potencia y existen varias diferencias en su manera de operar que es importante tener en cuenta.

Por ejemplo, para conocer el valor asociado a un símbolo no es necesario, como en AutoCAD, precederlo de un signo de admiración <!>. Basta teclear el nombre de la variable y pulsar <INTRO>. En la Consola se emiten una serie de mensajes de diagnóstico durante la ejecución de las funciones y si se encuentra un error interrumpe la

ejecución y habilita un nuevo nivel de evaluación donde podemos ensayar cambios en valores de variables y realizar pruebas para detectar los problemas existentes. De producirse un nuevo error, se habilita un nuevo nivel y así sucesivamente, hasta que decidamos regresar al nivel superior. Las baras de desplazamiento de la consola nos permiten revisar los resultados anteriores.

Las prestaciones más importantes de la Consola se resumen a continuación:

- Evaluar expresiones AutoLISP y mostrar el resultado devuelto por dichas expresiones.
- Introducir expresiones AutoLISP en líneas múltiples pulsando para el cambio de líne la combinación <CTRL> + <INTRO>. Pulsar sólo <INTRO> provocaría la evaluación de la expresión tecleada hasta entonces.
- Evaluar múltiples expresiones a la misma vez.
- Copiar y transferir texto entre las ventanas de la Consola y el Editor. La mayor parte de los comandos de texto están disponibles también en la Consola.
- Recuperar expresiones tecleadas anteriormente en la Consola, pulsando la tecla <TAB>. Pulsando esta tecla varias veces se van recuperando las expresiones anteriores. Para realizar el ciclo en sentido inverso puede utilizarse la combinación <SHIFT> + <TAB>.
- La tecla <TAB> también permite realizar una búsqueda de carácter asociativo a través del conjunto de expresiones anteriores. Si se quiere buscar anteriores expresiones de creación de variables bastaría teclear (SETQ y entonces pulsar <TAB>, con lo que se iría directamente a la última expresión tecleada que comenzaba así. Para realizar el ciclo en sentido inverso aquí también puede utilizarse la combinación <SHIFT> + <TAB>.
- Pulsando <ESC> se elimina cualquier texto a continuación del símbolo (prompt) del evaluador.
- Pulsando <SHIFT> + <ESC> abre una nueva Consola, dejando el texto escrito en la ventana de Consola anterior sin evaluar.
- Al pulsar el botón derecho del ratón en cualquier punto de la Consola o tecleando <SHIFT> + <F10> abre un menú contextual de funciones y opviones VLISP. Esto facilita, por ejemplo, copiar y pegar texto en la línea de comandos de la consola, buscar texto e iniciar las utilidades de depuración VLISP.

6.14.6 El Editor Visual LISP

Es más que un simple editor de texto. Será, en realidad, nuestro lugar de trabajo habitual para la programación LISP dentro de AutoCAD. Cada fichero de programa abierto tendrá su propia ventana del Editor
Un programa en el editor tendrá más o menos este aspecto:

Caras.lsp

```
;;;CARAS.LSP Simplificación de la entrada de datos para la
;;;herramienta PCARA. Esta herramienta requiere, después de
;;;introducidos los vértices, indicar la manera que los mismos
;;;se conectan. Cuando se desee construir una cara poligonal y
;;;los vértices se introducen de manera secuencial, este programa
;;;simplifica la entrada de datos aportando de manera automática
;;;la información sobre la conectividad.
;;;(c) 1999, Reinaldo Togores

;;;Función VerticesCara:
;;;Devuelve una lista con los vértices entrados por el usuario
(defun VerticesCara (/ PosVert)
  (defun PosVert ()
    (if (null pos)(setq pos 1)(setq pos (1+ pos)))
  ) ;_ fin de defun
  (while (setq
           vertice
            (getpoint (strcat "\n\tVértice " (itoa (PosVert)) ": ")
         ) ;_ fin de setq
    (setq ListaVertices (cons vertice ListaVertices))
  ) ;_ fin de while
) ;_ fin de defun

;;;Función Principal CARA
;;;Ejecuta el comando PCARA (_pface) usando la lista
;;;de vértices generada por la función ListaVertices
(defun C:CARA (/ pos listavertices)
  (prompt "\nEntrada de Vértices para PCARA:")
  (command "_pface")
  (foreach term (VerticesCara) (command term))
  (command "")
  (repeat (1- pos) (command (setq pos (1- pos))))
  (command "" "")
) ;_ fin de defun
```

Figura 6-11 EntronoIntegrado de Visual AutoLisp

- Codificación sintáctica por color. Lo que primero llama la atención la ventana del editor es el color que adoptan los diferentes componentes del programa. El editor identifica las distintas partes de un programa LISP y le asigna distintos colores. Esto permite detectar a primera vista elementos tales como nombres de función, números enteros o reales y cadenas, distinguiéndolos de los nombres de funciones y variables asignados por el usuario. Los errores mecanográficos saltan así a la vista de manera inmediata. Estos colores pueden personalizarse a gusto del usuario.

- Controles de la Ventana del Editor. La ventana del Editor no posee Barras de Menús ni de Herramientas. Al estar el foco situado en una ventana del Editor se activarán las opciones de Menú y las Herramientas de la ventana de la Aplicación

que pueden usarse para operaciones del Editor. Muchas de las opciones pueden también ser ejecutadas desde el menú contextual que se abre al pusar el botón derecho del ratón. Existe la posibilidad de utilizar también las combinaciones de teclas rápidas usuales en los editores de texto y además algunas combinaciones que se utilizan para funciones exclusivas de este editor. En términos generales, además de la codificación por color el editor ofrece otras ayudas que facilitan grandemente el desarrollo de programas. Algunas de estas utildades son:

- Comprobación del cierre de paréntesis
- Formateo del Texto
- Comentarios automáticos en los cierres de Expresiones
- Comentado y Descomentado automático de las líneas seleccionadas
- Búsqueda y Sustitución de Texto
- Comprobación de la Sintaxis
- Carga de expresiones LISP para ser probadas.

En los próximos apartados pasaremos a exponer estas funcionalidades:

- Barras de Herramientas
- Menú Contextual
- Teclas Rápidas

Figura 6-12 Menú Archivo de Visual AutoLisp

- Copias de Seguridad. El Editor VLISP viene configurado para hacer siempre copias de seguridad de los ficheros. Estas copias de seguridad se sitúan en el mismo directorio de la fuente del programa usando el mismo nombre y la extensión _ls. De

esta manera la copia de seguridad del fichero **caras.lsp** se llamará **caras._ls**. Para recuperar la última versión guardada de un fichero se emplea la opción Revert del menú Files.

6.14.7 Barras de Herramientas

Casi todas las funcionalidades del programa pueden accederse desde iconos de las Barras de Herramientas. Los iconos que aparecen agrisados cuando se activa la ventana del Editor no son utilizables en este contexto. Las Herramientas utilizables para operaciones en el Editor son:

Herramientas de Edición

Búsqueda y Marcadores

Revisión y Formateo

Acceso a otras Ventanas de la Aplicación

Utilidades de Depuración

Figura 6-13 Barras de herramienta de Visual AutoLisp

6.15 Ejemplo Conversión Entre Binarios Y Decimales (I)

Aplicando lo visto hasta ahora podemos crear una función que convierta cualquier número decimal, positivo o negativo, en su representación binaria.

El manual de Personalización de AutoCAD * propone una solución, pero que es aplicable sólo a enteros positivos:

```
;;;Del manual de Personalización:
;;;convierte un entero POSITIVO a una cadena
;;;en la base dada:
(defun base (bas int / ret yyy zot)
  (defun zot (i1 i2 / xxx)
    (if       (> (setq xxx (rem i2 i1)) 9)
      (chr (+ 55 xxx))
      (itoa xxx)
    ) ;_ fin de if
  ) ;_ fin de defun
  (setq     ret (zot bas int)
```

```
            yyy (/ int bas)
  ) ;_ fin de setq
  (while (>= yyy bas)
    (setq ret (strcat (zot bas yyy) ret))
    (setq yyy (/ yyy bas))
  ) ;_ fin de while
  (strcat (zot bas yyy) ret)
) ;_ fin de defun
```

Para números enteros positivos opera correctamente pero no así para los negativos, como puede verse en los siguientes ejemplos:

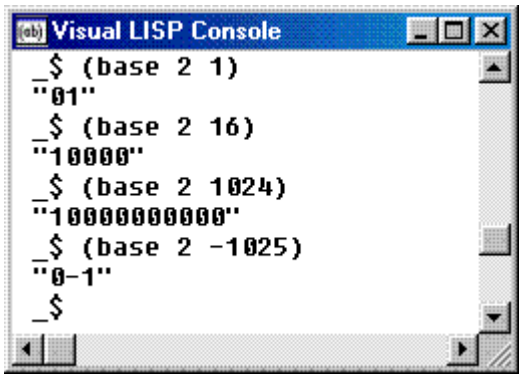

Figura 6-14 Ejecución del programa

Nuestra función BINARIO tiene en cuenta la longitud de palabra del sistema operativo y devuelve la secuencia correcta de ceros y unos incluso cuando se trate de valores negativos. La función espera recibir un entero, pero prevé el caso de un número real, truncándolo al entero menor más cercano.

Se definen las funciones utilitarias BIT y POSBIT, esta última utilizada para guardar el valor de la posición del bit que se analiza. La conversión en sí misma la realiza la función ConvierteBinario que recibe como argumentos el número a convertir y una función predicado a plicar según el número sea positivo o negativo.

Esta función de conversión es llamada desde la función principal BINARIO que analiza si el número recibido es positivo o negativo. En caso de ser negativo pasa a la función de conversión su NOT lógico **(~ numdec)** y el predicado **'=** en lugar del número recibido y el predicado **'/=** que pasaría en caso de ser el número positivo.

```
;;;Binario.lsp
;;;El ciclo continuará hasta que LSH devuelva un valor negativo
;;;significando que se ha llegado al final de la palabra (posición
;;;extrema izquierda). El valor binario es devuelto en formato de lista.
;;;
;;; Función auxiliar Bit: devuelve el valor decimal del bit en la posición indicada.
```

```
;;;
(defun Bit (pos) (lsh 1 (1- pos)))
;;;
;;;
;;Función utilizada como acumulador del valor de la posición
;;;
(defun PosBit ()
  (if (null posicion)
    (setq posicion 1)
    (setq posicion (1+ posicion))
  ) ;_ fin de if
) ;_ fin de defun
;;;
;;;
;;;Función para procesamiento del número decimal
;;;Recibe como argumento el predicado a aplicar
;;;según sea el argumento numérico positivo o negativo
;;;
(defun ConvierteBinario (numdec predicado / posicion numbin)
  (while (not (minusp (bit (1- (PosBit)))))
    (setq numbin
              (cons
                (if
                  (apply
                          predicado
                          (list (logand (bit posicion) (fix numdec)) 0)
                  ) ;_ fin de apply
                          1
                          0
                ) ;_ fin de if
                numbin
              ) ;_ fin de cons
    ) ;_ fin de setq
  ) ;_ fin de while
) ;_ fin de defun
;;;
;;;Función principal
;;;Tiene en cuenta si se trata de un número positivo o negativo:
;;;
(defun Binario (numdec /)
  (if (minusp numdec)
    (ConvierteBinario (~ numdec) '=)
    (ConvierteBinario numdec '/=)
  ) ;_ fin de if
) ;_ fin de defun
```

A diferencia de la función BASE, se obtiene una respuesta correcta tanto de enteros positivos como negativos:

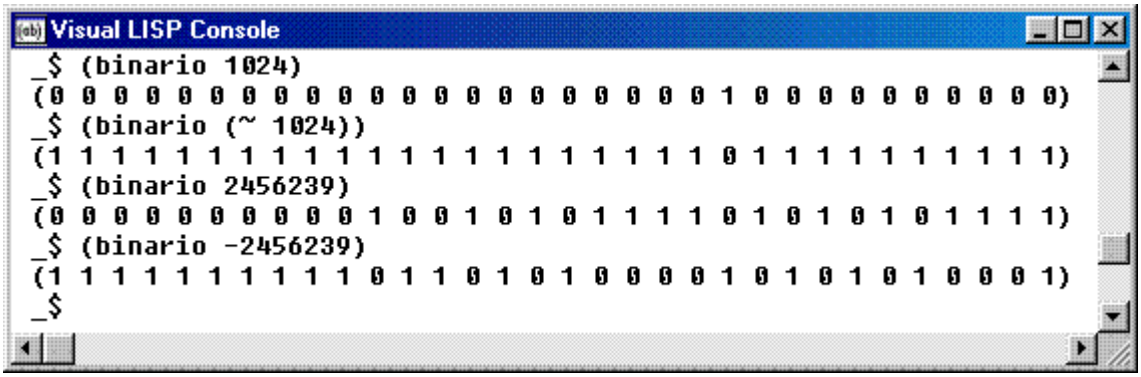

Figura 6-15 Resultados del programa

Si deseáramos el resultado en formato cadena en lugar de lista, bastaría mapear 'ITOA a la lista devuelta para convertir los enteros en cadena y aplicarle después 'STRCAT para unir los caracteres en una cadena única:

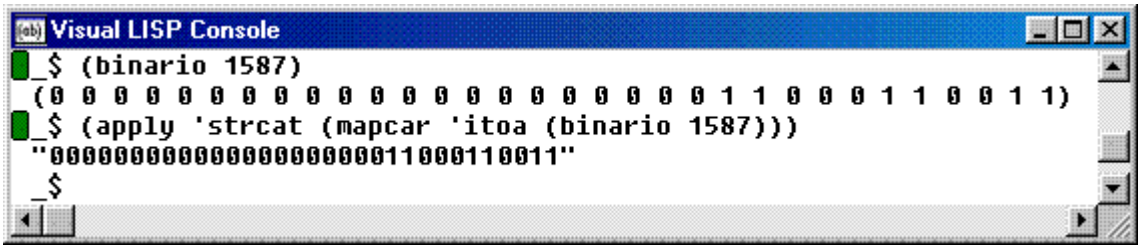

Figura 6-16 Consola de Visual AutoLisp

Dedicaremos la próxima sección a la conversión en el otro sentido, de binario a decimal.

6.16 Conversión Entre Binarios Y Decimales (Ii)

Esta función, complementaria de la anterior, nos permite hacer la conversión en sentido inverso, partiendo de un valor binario y convirtiéndolo en decimal.

Aunque lo más probable es que el tipo de entrada del dato sea numérico, hemos diseñado una función como filtro de entrada que analizará mediante un COND las posibles entradas admitiendo listas y cadenas y rechazando cualquier otro tipo de dato.

En caso de que el dato no sea de uno de estos tres tipos admitidos, se imprimirá un mensaje de error. Si el dato es admitido pero no es de carácter binario (incluye términos distintos de cero ó uno, la función devolverá NIL.

;;;DECIMAL.LSP

```
;;;Recibe un número binario y lo convierte en decimal
;;;debe comprobar el tipo de dato recibido,
;;;que puede ser cadena, número o lista
;;;
;;;Función utilitaria BIT
;;;devuelve el valor decimal del bit en la posición recibida:
;;;
(defun Bit (pos) (lsh 1 (1- pos)))
;;;
;;;
;;;Función utilizada como acumulador del valor de la posición:
;;;
(defun PosBit ()
  (if (null posicion)
    (setq posicion 1)
    (setq posicion (1+ posicion))
  ) ;_ fin de if
) ;_ fin de defun
;;;
;;;PREDICADOS DEFINIDOS PARA ESTA FUNCIÓN:
;;;Como filtro de entrada se emplean tres predicados
;;;definidos expresamente para ella:
;;;STRINGP, STD-DOTTED-PAIR-P y BINARIOP
;;;
;;;Predicado STRINGP
;;;Comprueba si el dato es una cadena
;;;(ver PREDICADOS DEFINIDOS POR EL USUARIO)
;;;
(defun stringp (dato) (equal (type dato) 'STR))
;;;
;;;Predicado STD-DOTTED-PAIR-P
;;;Comprueba de que se trate de un par punteado:
;;;Adaptado de la Standard Lisp Library
;;;de Reini Urban:
;;;
(defun STD-DOTTED-PAIR-P (lst)
  (and (vl-consp lst) (not (listp (cdr lst))))
) ;_ fin de defun
;;;
;;;Predicado Binariop
;;;Comprueba que la lista incluya valores sólo 0 ó 1
;;;
(defun Binariop   (numbin /)
  (apply 'and
          (mapcar '(lambda (term) (or (equal term 0) (equal term 1)))
                  numbin
          ) ;_ fin de mapcar
```

```
  ) ;_ fin de apply
) ;_ fin de defun
;;;
;;;Función utilitaria NUMLISTA
;;;Recibe cualquier número decimal y devuelve los dígitos aislados
;;;en formato de lista
;;;Si el número es real, lo trunca despreciando los decimales
;;;
(defun Numero->Lista (numero / lista)
  (while (> numero 0)
    (setq lista (cons (rem (fix numero) 10) lista))
    (setq numero (/ (fix numero) 10))
  ) ;_ fin de while
  lista
) ;_ fin de defun
;;;
;;;Función utilitaria Cadena->Lista
;;;Recibe una cadena de caracteres y devuelve los caracteres
;;;aislados en formato de lista
;;;
(defun Cadena->Lista (cadena)
  (mapcar
    'chr                          ;3.- convierte los códigos ASCII a caracteres
    (vl-string->list              ;2.- convierte la cadena a lista de códigos ASCII
      cadena
    ) ;_ fin de vl-string->list
  ) ;_ fin de mapcar
) ;_ fin de defun
;;;
;;;Función ConvierteDecimal
;;;Realiza la conversión, al ser la evaluación siempre de izquierda a derecha,
;;;debe invertir la lista para iniciar la comprobación por el bit último de la derecha.
;;;Esta comprobación se hace mediante el mapeado de una función LAMBDA a la
lista,
;;;que comprueba si el número es cero y en caso que no lo sea, inserta el valor
;;;decimal del bit en la lista
;;;Una vez concluido este mapeado, se le aplica la funciom '+ para sumar todos
;;;los valores, con lo que obtenemos el resultado de la conversión Binario->Decimal.
;;;Devuelve un número decimal
;;;
(defun ConvierteDecimal   (numbin / posicion)
  (if (Binariop numbin)
    (apply
      '+                          ;       suma los valores de la lista devuelta por
MAPCAR
      (mapcar
            (function
```

```
            (lambda (x)
              (PosBit)                          ;5.-  valora la variable posicion
              (if       (not (zerop x))
                (bit posicion)
                0                               ;     en caso contrario devuelve cero
              ) ;_ fin de if
            ) ;_ fin de lambda          7.-  y los valores devueltos quedan en una lista
          ) ;_ fin de function
          (reverse numbin)
     ) ;_ fin de mapcar
   ) ;_ fin de apply
   nil
  ) ;_ fin de if
) ;_ fin de defun
;;;
;;;Función filtro de entrada, considerando los posibles tipos de entrada:
;;;Lista, cadena o número
;;;los únicos términos aceptados serán en cualquier caso ceros o unos
;;;de detectarse otro valor, la función devuelve NIL
;;;Otro error derivaría de que la lista fuera un par punteado
;;;Para comprobarlo utilizaríamos la función STD-DOTTED-PAIR-P adaptada de
;;;la Standard LISP library de Reini Urban
;;;
(defun Decimal (numbin /)
  (cond
    ((STD-DOTTED-PAIR-P numbin)
     (terpri)
     (princ numbin)
     (princ " no es un valor admitido.")
     (princ)
    )
    ((listp numbin)
     ;;si es lista, convierte los términos
     (setq
       numbin ;;que sean cadenas en números (o átomos simbólicos si fueran letras)
              (mapcar
                    (function (lambda (term)
                                  (if       (stringp term)
                                    (read term)
                                    term
                                  ) ;_ fin de if
                                ) ;_ fin de lambda
                    ) ;_ fin de function
                    numbin
              ) ;_ fin de mapcar
     ) ;_ fin de setq
     (ConvierteDecimal numbin)
```

```
   )
   ((numberp numbin)
    ;;si es número lo convierte en cadena para después hacerlo lista
    (setq numbin (Numero->Lista numbin))
    (ConvierteDecimal numbin)
   )
   ((stringp numbin)
    (setq numbin (mapcar 'read (Cadena->Lista numbin)))
    (ConvierteDecimal numbin)
   )
   (t
    (terpri)
    (princ numbin)
    (princ " no es un valor admitido.")
    (princ)
   )
  ) ;_ fin de cond
) ;_ fin de defun
```

A continuación algunos ejemplos tomados de la consola de VISUAL Lisp:

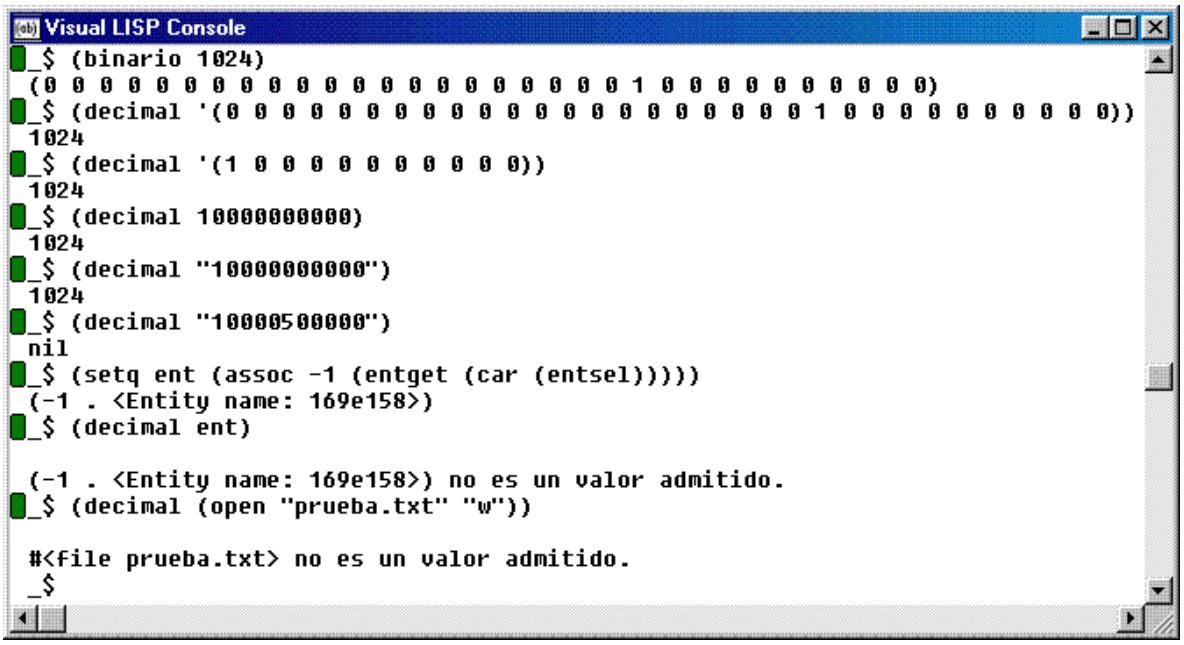

Figura 6-17 Consola de Visual AutoLisp