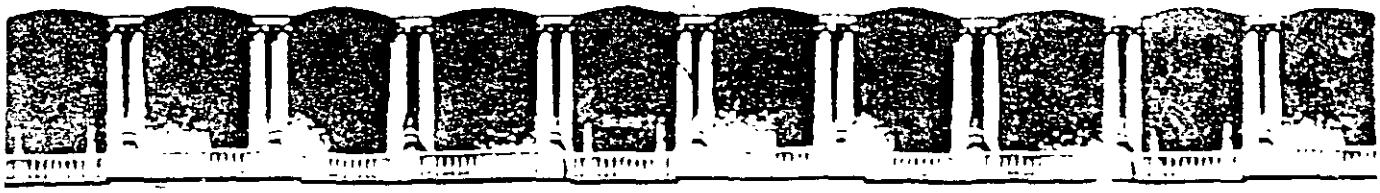


FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

MATERIAL DIDACTICO DEL CURSO

AUTOCAD AVANZADO I

Programación con AutoLisp

Profesor: Ing. Oscar Martín Del Campo C.

Noviembre 2002

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtemoc 06000 México D.F. APDO Postal M-2285
Teléfonos 5510-8955 5510-5121 5521-7087 5521-1987 Fax 5510-0573 5521-4021 AL 25

DIVISIÓN DE EDUCACIÓN CONTINUA DE LA FACULTAD DE INGENIERÍA, UNAM

EVALUACION DEL PERSONAL DOCENTE

Curso: CC078 Programación en Autocad Avanzado I

Instructor: Ing. Oscar Martín Del Campo

Institución: _____

Duración: 40 Hrs.

El propósito de este cuestionario es conocer su opinión acerca del desarrollo. Marque con una "X" considerando el siguiente puntaje. Es muy importante contar con su objetividad.

10 = Excelente 9 = Muy bueno 8 = Bueno 7 = Suficiente 6 = Malo 5 = Deficiente

EVALUACIÓN DEL INSTRUCTOR

Factores a evaluar	10	9	8	7	6	5
1. Mostró dominio del tema						
2. Utilizo un lenguaje claro y sencillo						
3. Propicio la integración del grupo con el propósito de alcanzar el objetivo del curso						
4 Despertó y mantuvo el interés de los participantes						
5. El instructor supervisó adecuadamente los trabajos						
6. Resolvió oportunamente las dudas y los problemas de los participantes						
7. Manejo correctamente los apoyos y recursos didácticos durante su intervención						
8. Ante situaciones conflictivas presentadas por el grupo el instructor fue profesional en su actuación						
9. Ilustro los temas con casos prácticos						
10 Inició y concluyó puntualmente y empleó adecuadamente el tiempo destinado para su exposición						

Comentarios y sugerencias: _____

_____

_____

Factores a evaluar	10	9	8	7	6	5
1 El temario del curso cumplió sus expectativas						
2 El conocimiento adquirido es aplicable en las funciones que desempeña						
3. Los temas tuvieron una secuencia lógica						
4. Las instalaciones fueron adecuadas y cómodas						
5. La coordinación del curso fue adecuada						
6. La duración del curso fue suficiente						
7. Los ejercicios y la dinámica fueron acordes con el contenido del curso						
8. Los temas acordes tuvieron un equilibrio teórico práctico						
9. Los materiales de apoyo y manuales empleados fueron suficientes y de calidad						

Comentarios y sugerencias: _____

_____

_____

PRESENTACION

Hoy en día, es palpable la gran importancia que ha adquirido la computadora para los profesionales que realizan actividades de dibujo y diseño; y son los sistemas CADD, los que han logrado que estos usuarios la utilicen sistemáticamente en sus tareas para obtener mayores grados de eficiencia y desarrollo.

AutoCAD representa uno de los sistemas CADD de mayor difusión, gracias al potencial que ofrece al usuario para alcanzar mayor productividad en el proceso de diseño. *AutoLISP*, lenguaje de programación de AutoCAD, representa una herramienta de vital importancia para alcanzar ese grado de productividad y eficiencia, optimizando la gran cantidad de recursos con que se cuenta y posibilitando el planteamiento de nuevos objetivos que con este lenguaje nos permitirán llegar a mayores logros.

El curso *AutoCAD Avanzado I: Programación con AutoLISP*, pretende introducir al alumno en este lenguaje sin que cuente con antecedentes en la programación o el conocimiento de algún otro lenguaje. Lo que si es indispensable es que se tenga experiencia en el uso de AutoCAD, pues la generación de programas con *AutoLISP* y sus objetivos dependen del conocimiento de todos sus comandos y facilidades.

INTRODUCCION

AutoCAD nos ofrece otras formas para la manipulación de objetos además de la comúnmente utilizada (consistente en dibujar y editar estos símbolos gráficos), se pueden crear sus propios menús, barras de herramientas, cajas de diálogo, así como personalizar las variables del sistema para obtener resultados más completos y rápidos. Pero sin duda uno de los mecanismos más poderosos para manipular el entorno de AutoCAD es el lenguaje de programación *AutoLISP*.

AutoLISP es un lenguaje orientado a la manipulación de símbolos; por ello se encontrará que la forma de programar con él es totalmente diferente a la utilizada en los otros lenguajes de programación convencionales como FORTRAN, BASIC ó C.
En *LISP* se hace uso en gran medida de los paréntesis, que sirven para indicar el inicio y final de una lista. *LISP* procesa listas de símbolos, en vez de datos numéricos como los lenguajes FORTRAN, BASIC y C.

AutoLISP es un subconjunto del *LISP* (lenguaje desarrollado para la investigación en Inteligencia artificial) que significa LISt Processing (Procesado de listas). *AutoLISP* añade algunas funciones especiales que están diseñadas para la manipulación de dibujos de AutoCAD.

Como los sistemas CAD se orientan hacia la manipulación de símbolos gráficos y *AutoLISP* se basa en el uso de símbolos, es un excelente lenguaje para la programación de sistemas CAD.

LISP es un lenguaje que es evaluado en vez de interpretado ó compilado. Los lenguajes interpretados leen el texto del programa línea por línea y lo convierten a instrucciones de máquina, generando un archivo en código de máquina, el cual es ejecutado rápidamente.

Un lenguaje evaluado se encuentra entre uno interpretado y uno compilado. Cuando se encuentra por primera vez un bloque de código, este se convierte en código compacto; si dicho bloque se encuentra de nuevo mientras se ejecuta el programa, el evaluador detecta que ya ha sido evaluado y lo ejecuta.

Hay tres características de LISP que lo distinguen de la mayoría de los otros lenguajes de programación:

- LISP manipula símbolos en vez de números
- Es un lenguaje orientado al objeto en vez de ser un lenguaje procedimental.
- Es un lenguaje que evalúa en vez de interpretar ó compilar.

Resumiendo, las ventajas que se puedan obtener utilizando *AutoLISP* son:

- La sistematización de tareas o procedimientos que agilicen el trabajo con AutoCAD.
- Permite automatizar el desarrollo de proyectos, es decir, realizar un agrupamiento de tareas planificadas que siguiendo una estrategia de desarrollo culmine en un objetivo común, teniendo para esta misma estrategia una o más vías de desarrollo que de antemano ya se a hallan previsto.
- Se tiene una explotación máxima del programa ya que se manipula directamente la base de datos sin pasar por el uso convencional de los comandos básicos de AutoCAD.
- Creación de nuevos comandos que se incorporan a los comandos básicos de AutoCAD.
- Interacción mayor con otros programas de dibujo e inclusive con otros programas de aplicación.

La Programación

Uno de los principales objetivos de este trabajo es presentar algunas de las ideas y principios que se manejan en la actualidad para la optimización del desarrollo de programas de aplicación, atendiendo al proceso de diseño e implementación de un sistema así como al mantenimiento que se requiere una vez que se pone en servicio.

Computadora y lenguaje de programación

Las computadoras electrónicas digitales surgieron en la década de 1940 y han penetrado en casi todos los campos de la actividad humana como herramientas para el almacenamiento y procesamiento de información.

En la ingeniería, la computadora es una ayuda esencial en el proceso de solución de problemas debido a la velocidad con la que puede recuperar y manipular grandes volúmenes de datos. El proceso de solución se le presenta a una computadora en forma de programa, que consiste en una lista de las acciones que se requieren para llegar a los resultados. Los programas escritos por el usuario de la computadora para obtener las soluciones a sus problemas son llamados *software de aplicación.*

El usuario de la computadora, durante el proceso de diseño e implementación del programa que es llamado *programación*, debe especificar las operaciones que van a realizarse. Los *lenguajes naturales* como el inglés, usados para la comunicación humana, no son apropiados para la programación debido a su ambigüedad y falta de precisión. Por el contrario, la programación en un lenguaje de máquina puede ser excesivamente tediosa y podría limitar la capacidad de aplicación de los programas, ya que no se transfieren con facilidad a diferentes computadoras en esta forma.

Por tanto, los usuarios de las computadoras programan en *lenguajes de programación*. Generalmente los más usados son los *lenguajes de alto nivel*, que combinan la precisión de expresión con una cierta aproximación al lenguaje natural por un lado y por otro el problema a resolver. Los programadores que usan estos lenguajes no tienen que tratar directamente con los dispositivos físicos del equipo de cómputo: sus programas se pueden trasladar con facilidad de una máquina a otra. Los *lenguajes ensambladores*, el otro tipo de notación de programa, se aproxima mucho más al lenguaje de máquina de una computadora, con la consiguiente complejidad y limitación que implica.
El objetivo de los lenguajes de programación de alto nivel es facilitar la escritura de programas de manera sencilla, sin involucrarse demasiado en los componentes internos de la computadora; pero se requiere de un medio de traducción. Los programas que se conocen como *traductores* (compiladores), convierten los *programas fuente* (escritos en un lenguaje para el que se ha diseñado el traductor) en *programas objeto*, expresados en el lenguaje de máquina de la computadora en particular.

Al principio los programadores usaban el lenguaje de la máquina. A mediados de la década de 1950, se comenzaron a introducir los lenguajes de alto nivel, con el tal vez más poderoso de éstos, el FORTRAN presentado en su forma original en 1957, desarrollándose años más tarde con la versión FORTRAN IV que cobró auge en las ciencias e ingeniería. Las necesidades de aplicaciones no numéricas trajeron el desarrollo de un nuevo estándar para este lenguaje llamado FORTRAN 77. Otros lenguajes surgidos en la década de 1960 fueron COBOL, ALGOL60, BASIC, PL/1, LISP y posteriormente PASCAL; este último utilizado inicialmente con fines educativos, sirvió para propósitos generales, especialmente en microcomputadoras. LISP fue creado para el desarrollo de aplicaciones en *Inteligencia Artificial*.

Desde la década de 1980 ha venido cobrando fuerza el lenguaje C en el campo de desarrollo de sistemas a la vez que en el de aplicaciones.

La Programación Estructurada

Desde hace algunos años se observaba que los programas, debido a su gran complejidad, adolecían de muchas fallas: no satisfacían las necesidades del usuario, no se producían a tiempo, costaban más de lo estimado, contenían errores, y resultaban difíciles de dar mantenimiento. Esta problemática se pronunció con el auge de las computadoras personales y el consecuente incremento en el desarrollo de sistemas y programas de aplicación.
La Programación Estructurada surge como una necesidad de reducir la complejidad de los grandes programas estableciendo lineamientos para su diseño e implantación, proponién-dose los objetivos siguientes

- Satisfacer los requerimientos especificados
- Facilitar el mantenimiento al programa
- Minimizar el número de errores durante el desarrollo del programa
- Operación resistente a errores
- Facilitar la transportación
- Definir algoritmos con una lógica legible
- Minimizar costos

Los lineamientos que se recomiendan para lograr que un programa cumpla con las normas de la *Programación Estructurada*, y por ende alcance lo objetivos arriba mencionados son:

- **Una entrada – una salida**
  Todo programa deberá tener una sola entrada y una sola salida; es decir, deberá iniciar su flujo siempre en el mismo punto, y por otro lado, independientemente de las diferentes posibilidades de flujo que se manejen, también deberá concluir invariablemente en un mismo punto.

- **Programación descendente**

El flujo de un programa deberá estar en dirección de arriba hacia abajo: a este concepto también se le conoce como programación Top-Down.

- **Documentación**
  Los programas deben estar debidamente documentados. Se considera necesario que un programa contenga una descripción del mismo, así como de los datos y procesos que se manejan para lograr una adecuada depuración y mantenimiento.

- **Programación modular**
  Con el propósito de aminorar la complejidad de un programa, éste deberá descomponerse en módulos que realicen tareas específicas, por lo que se incluyen en cada uno de éstos, aquellas actividades que estén orientadas a un mismo fin. Los módulos ocultan detalles que en cierto contexto no son de interés. Cuando es necesario hacer modificaciones, puede intercambiarse un módulo sin afectar a los otros, ofreciendo seguridad.

- **Uso del seudocódigo**
  A fin de lograr mayor dinámica en la etapa de diseño, el seudocódigo permite escribir el programa sin atender a los requerimientos sintácticos del lenguaje de programación.

- **Utilización de las estructuras básicas de control de flujo**
  Con la finalidad de simplificar la estructura de flujo de un programa, sólo se permiten utilizar las tres estructuras de control básicas: *Secuencia*, *Condición* y *Repetición* (las cuales se tratan más adelante).

Proceso de refinamiento a pasos

Nuestra mayor herramienta en el desarrollo de programas es nuestra capacidad de abstracción. Esto consiste en que, al resolver un problema, se analiza cuidadosamente hasta lograr una representación mental de los elementos esenciales del mismo.
Cada elemento debe tener un propósito bien definido. Este proceso puede aplicarse a su vez a cada uno de los elementos del problema original, es decir,

consideramos a cada elemento como un nuevo problema (subproblema). El proceso puede repetirse para los elementos del subproblema y así sucesivamente.

Inicialmente (en la resolución del problema) estamos concentrados en qué elementos o procesos descomponer el problema, y a medida que procedemos a descomponerlos sistemáticamente, llegamos a concentrarnos en cómo realizar cada proceso. Esta forma de proceder va de lo general a lo particular.

El proceso descrito anteriormente se conoce como proceso de refinamiento a pasos y es central a la *Programación Estructurada*; no es directo ni trivial como pudiera parecer a primera vista; éste es esencialmente un proceso de ensayo y error.

Es evidente que se necesita una notación como herramienta para ir concretando la solución del problema, ya que a medida que se refina sistemáticamente la solución, nuestra limitada capacidad de retención será desbordada por la gran cantidad de detalles involucrados. Dos herramientas muy utilizadas para este propósito son: el *seudocódigo* o seudolenguaje de diseño de programas y el *diagrama de flujo*. El *seudocódigo* y el *diagrama de flujo* deben utilizarse como una extensión natural del proceso de abstracción.

El seudocódigo

El *seudocódigo* o seudolenguaje es un lenguaje intermedio entre el lenguaje nativo del programador y el lenguaje de programación en que se intenta implantar la solución. El *seudocódigo* le permite al programador pensar en la lógica y expresarla en una manera semiformal sin tener que adentrarse en los detalles particulares de un lenguaje de programación.

Básicamente difiere en dos aspectos de un lenguaje de programación:

1).- No existen restricciones sintácticas para el uso del *seudocódigo*. Sólo las estructuras de control básicas y el 'sangrado' para mejorar la claridad del alcance de dichas estructuras, son las únicas convenciones aceptadas.

2).- Cualquier operación se puede expresar a cualquier nivel de detalle, por ejemplo

Incrementar *seccion* o
seccion = seccion + 1

El *seudocódigo* permite al programador tratar el problema a diversos niveles de

abstracción, ya que ésta es la herramienta mediante la cual expresamos la solución de un problema durante el proceso de refinamiento a pasos, esto es, el *seudocódigo* es un lenguaje de diseño de programas (PDL).

El *seudocódigo* es una forma conveniente para documentar los estados de desarrollo del programa, lo cual permite a otros programadores revisar la función del programa antes de implementarlo en un lenguaje de programación, además de facilitar una valoración del estado de desarrollo en cualquier instante del proceso de diseño del programa.

Existen pocas guías generales que hacen del *seudocódigo* efectivo como una herramienta de diseño de programas; éstas son:

- Hacer del *seudocódigo* una extensión del proceso del pensamiento
- Sangrar el código para resaltar la estructura de la lógica
- Dar nombres a los datos de tal manera que reflejen su intención
- Mantener la lógica simple
- Utilizar las estructuras básicas de control permitidas en la *Programación Estructurada*

Diagrama de flujo

De manera complementaria al *seudocódigo*, existe una forma gráfica para la descripción del programa que se denomina *diagrama de flujo*. Con éste, se puede apreciar visualmente la dirección que el flujo va tomando en el desarrollo del programa, dando mayor claridad al proceso de solución que se pretende implementar.

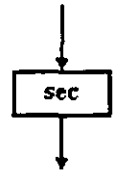
Estructuras básicas de control

Hay tres estructuras básicas de control que se permiten en la Programación Estructurada

1).- *Secuencia*
2).- *Condición*
3).- *Repetición*

La representación gráfica de éstas es:

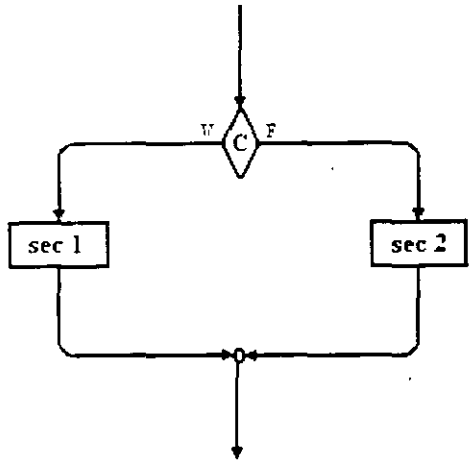
1.- SECUENCIA



sec Secuencia que se ejecuta incondicionalmente cuando el flujo del programa llega a ésta

La *secuencia* es la expresión mínima en la descripción del programa y con la cual se puede representar una operación simple, a la vez que todo un subproceso, según el nivel de detalle y refinamiento que se tenga.

2.- CONDICIÓN



C Condición que determina la dirección que tomará el flujo del programa.

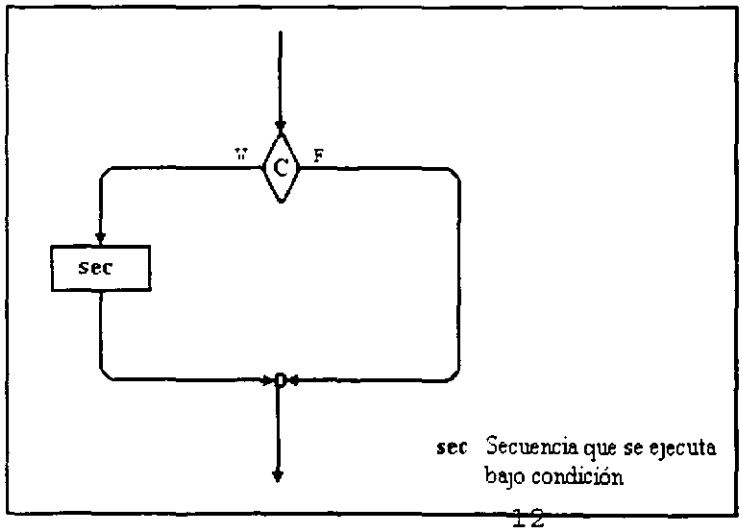
V Dirección del flujo si la condicion es verdadera

F Dirección del flujo si la condición es falsa

sec 1 y **sec 2** Secuencias del programa

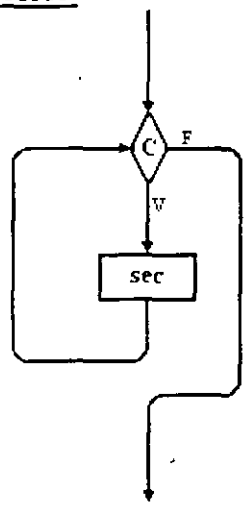
La estructura de *condición* también conocida como *if-then-else*, permite decidir entre dos posibles direcciones del flujo. Para especificar la condición se utilizan las expresiones lógicas; en los casos más sencillos, éstas son relaciones. Se observa que una vez realizada la *secuencia* correspondiente, el flujo regresa al cauce principal indicado por el eje central de flujo.

Un caso particular se tiene cuando sólo se desea condicionar la realización de una *secuencia* ('*if–then-else* sin *else*'):



sec Secuencia que se ejecuta bajo condición

3.- REPETICIÓN



C	Condición que determina si el flujo entra al ciclo
V	Dirección que entra al ciclo si la condición es verdadera
F	Dirección que sale del ciclo si la condición es falsa
sec	Secuencia del programa que se reejecuta en el ciclo mientras la condición sea verdadera.

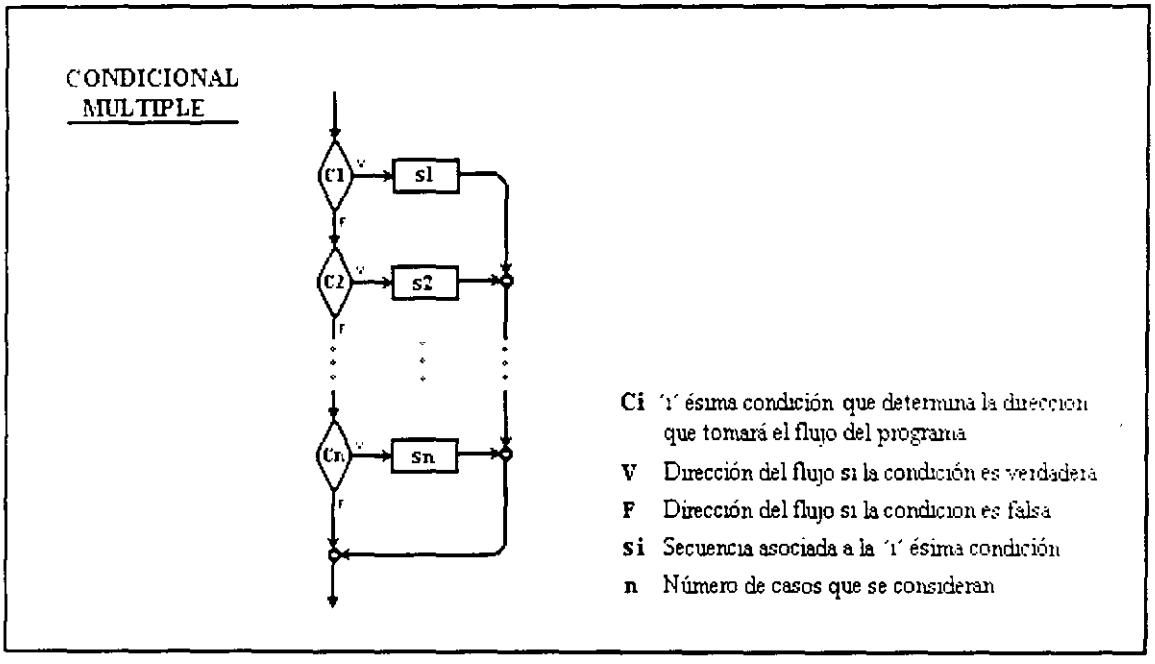
La estructura de *repetición* también conocida como *while*, origina la ejecución repetida de una *secuencia* mientras se cumple una condición. El número de veces que dicha *secuencia* es ejecutada, depende de la condición definida al inicio de la estructura; cuando ésta deja de cumplirse (sea falsa) el flujo sale del ciclo, regresando al cauce principal. Esta ejecución repetida es llamada iteración.

Es necesario al entrar a un ciclo, que la *secuencia* posibilite que la condición deje de cumplirse en alguna determinada iteración, de lo contrario se entraría en un ciclo infinito (*loop*), que puede representar un error de programación.

Algunos lenguajes de programación proporcionan modernas estructuras de control de flujo acorde a la *Programación Estructurada*, e incluso ofrecen algunas estructuras adicionales a las tres básicas, que cumplen con los lineamientos establecidos por ésta.

Una estructura de control que *AutoLISP* incluye y que complementa a las recomendadas por la *Programación Estructurada* es:

Condicional múltiple



Esta estructura de control es muy útil para algunos casos en los que la estructura de *condición* resulta limitada al obligarnos a realizar múltiples anidamientos. En la *condicional múltiple* también conocida como *case*, se pueden considerar varias condiciones, cada una asociada con una secuencia en particular. Se evalúa en orden cada condición; si alguna se cumple, el flujo nos lleva a ejecutar su secuencia asociada, para posteriormente dirigirse al final de la estructura sin tomar en cuenta las demás condiciones que se encuentren por debajo de ésta.

La Programación Orientada a Objetos (POO)

La construcción de software ha recibido la atención de los expertos desde hace mucho tiempo: en la década de 1970 se consiguieron avances significativos en el desarrollo de metodologías de forma sistemática y a bajo costo. Como resultado de esos esfuerzos surgieron técnicas como *La Programación Estructurada* (tratadas previamente en este trabajo) que han sido las herramientas utilizadas por los programadores para construir software durante mucho tiempo en proyectos realmente complejos: sin embargo, en la actualidad la mayoría de los ingenieros de software coinciden en afirmar que sufren de tres grandes deficiencias:

- Los productos que resultan de emplear estas técnicas son poco flexibles.
- Los programadores que las usan tienden a concentrarse en el diseño y la implementación del sistema, sin tomar en cuenta su vida posterior.
- No alientan al programador a aprovechar el trabajo de proyectos anteriores.

La desventaja de utilizar una metodología que se concentra en el diseño inicial del sistema se hace evidente si se toma en cuenta que la vida útil de un producto de software puede ser cinco o seis veces más grande que el lapso en que se desarrolla: por ejemplo, un sistema que se desarrolla en uno o dos años puede mantenerse trabajando durante un periodo que va de cinco a quince años. Los gastos que se hacen durante este último periodo (gastos de mantenimiento) representan alrededor del 70% del costo total del sistema. Es por ello que la extensibilidad (facilidad con que se modifica un sistema para que realice nuevas funciones) debe ser uno de los objetivos primarios de la etapa de diseño.

La fase de mantenimiento es tan importante que cualquier método de diseño debe tener como objetivo principal producir sistemas que faciliten su propio mantenimiento. Para alcanzar este objetivo, los expertos recomiendan una serie de actividades que pueden servir como guía durante el desarrollo del sistema, agrupadas de acuerdo a la etapa de desarrollo en que se deben realizar:

Actividades de análisis

- Establecimiento de estándares (formatos de documentos, codificación, etc).
- Especificar procedimientos de control de calidad.
- Identificar probables mejoras del producto.
- Estimar recursos y costos de mantenimiento.

Actividades de diseño

- Establecer la claridad y modularidad como criterios de diseño.
- Diseñar para facilitar probables mejoras.
- Usar notaciones estandarizadas para la documentación, algoritmos, etc.
- Seguir los principios de ocultamiento de información, abstracción de datos y descomposición jerárquica de arriba hacia abajo.
- Especificar efectos colaterales de cada módulo.

Actividades de implementación

- Usar estructuras de una sola entrada y una sola salida.
- Usar sangrado estándar en las diferentes estructuras.
- Usar un estilo de codificación simple y claro.
- Usar constantes simbólicas para asignar parámetros a las rutinas.
- Proporcionar prólogos estándares de documentación en cada módulo.

Otras actividades

- Desarrollar una guía de mantenimiento.
- Desarrollar un juego de pruebas.
- Proporcionar la documentación del juego de pruebas.

Reutilización de código

La reutilización de código es otro de los factores que no se toma en cuenta en los métodos de diseño tradicionales. Cualquier programador que desarrolla un sistema nuevo debe escribir una buena cantidad de código que ha escrito anteriormente una y otra vez. Las rutinas de búsqueda, ordenamiento, manejo de menús y despliegue de ventanas, entre otras, se repiten continuamente en proyectos diferentes. Los métodos de desarrollo de software deben alentar al programador a utilizar el código escrito previamente por él mismo o por otros programadores.
Normalmente se emplean tres tipos de *reutilización*: de personal, de diseño y de código fuente. En la primera, a un proyecto se le asignan programadores que tienen experiencia en el desarrollo de programas semejantes; la segunda consiste en emplear el diseño de un sistema para desarrollar otro similar; y la tercera forma

de *reutilización* se da cuando un programador utiliza parte de un programa escrito con anterioridad para crear un nuevo programa.

Modularidad

La *modularidad* (ampliamente tratada en *La Programación Estructurada*) es una de las herramientas de diseño más poderosas para facilitar el desarrollo y mantenimiento de sistemas de software. Sin embargo, para ser realmente útil, el concepto de *módulo* debe ser más sofisticado de manera que cumpla con las propiedades siguientes:

- Descomponibilidad: Un problema de diseño puede descomponerse en subproblemas más pequeños que pueden resolverse en forma independiente.
- Componibilidad: Una vez que se cuente con un conjunto de módulos que realizan una función específica, se debe alentar al programador a usar esos módulos para construir nuevos programas.
- Comprensibilidad: El lector de un programa o librería debe ser capaz de entender el funcionamiento de cada módulo sin necesidad de consultar el texto de otros módulos.
- Continuidad: Un cambio pequeño en las especificaciones de un programa debe causar cambios en un sólo módulo o en un conjunto pequeño de ellos.
- Protección: Un error de ejecución en el funcionamiento de un módulo no debe expandirse hacia los demás módulos.

¿Qué es la Programación Orientada a Objetos?

La Programación Orientada a Objetos (POO) es un método de diseño que tiene como objetivo establecer técnicas de desarrollo de software para producir sistemas modulares. Un sistema orientado a objetos se puede entender fácilmente, por lo que su desarrollo, depuración y mantenimiento se facilitan en gran medida.

En primera instancia, la *POO* se basa en una idea relativamente sencilla: un programa de computadora es un modelo que representa un subconjunto del mundo real; la estructura de ese programa simplifica en gran medida, si cada una de las entidades (u objetos) del problema que se está modelando corresponde

directamente con un objeto que se pueda manipular internamente en el programa.

El proceso de representar entidades reales con elementos internos a un programa recibe el nombre de *abstracción de datos*. La abstracción es una descripción simplificada que resalta solamente las características esenciales de un objeto de la realidad, despreciando las características no esenciales. La *abstracción de datos* no se concentra en la representación interna de un objeto (un entero, una cadena de bits, un arreglo), sino en sus atributos (como el nombre, sueldo y edad de un empleado) y en las operaciones que se puedan realizar sobre ese objeto (como calcular el sueldo de un empleado o los impuestos que debe pagar). De esta forma, un tipo de dato abstracto se puede describir concentrándose en las operaciones que manipulan a los objetos de ese tipo, sin caer en los detalles de representación y manipulación de los datos.

ENTORNO DE AUTOLISP

AutoLISP es un programa integrado a AutoCAD que solamente puede ser invocado desde el editor de dibujos; es decir, es un producto que no puede ser independiente de AutoCAD. *AutoLISP* es un programa intérprete (evaluador, propiamente dicho). Entrando al ambiente de AutoCAD, ya puede comenzarse a trabajar y *AutoLISP* es accesado a través del intérprete *AutoLISP*: cuando se teclea cualquier cosa al prompt "command:" el intérprete primero lee la información para ver si se trata de una función de *AutoLISP*. Si es así, *AutoLISP* la ejecuta; en caso contrario AutoCAD se encarga de procesarla.

Como se ve, uno pude interactuar con *AutoLISP* intérprete en ambiente AutoCAD cuando se encuentra el prompt "Command:", por ejemplo:

```
Command: (+ 4.5 3)
```

En el ejemplo anterior se da una expresión de *AutoLISP*, donde se le indica que evalúe la función "+".

Nótese también que la expresión se encuentra entre paréntesis. Esta es la estructura básica para todos los programas en *AutoLISP*. Todo lo que se quiera evaluar con *AutoLISP*, desde la expresión más simple hasta el programa más complejo, deberá escribirse con esta estructura.

El resultado de evaluar una expresión es llamado *valor de la expresión*.

La manera en la que se va a trabajar con *AutoLISP* será la siguiente:

Creación de un programa de instrucciones en *AutoLISP* utilizando para tal efecto el editor que nos ofrece el ambiente de desarrollo de *VisualLISP*. Puede utilizarse cualquier otro editor o procesador de textos, cuidando que si usa un procesador como *Word* guarde el programa como texto sin formato. El programa de instrucciones en *AutoLISP* será un archivo en formato ASCII.

Para este curso se utilizará el editor que nos ofrece *VisualLISP* que se invoca desde el menú *Tools->Visual LISP Editor*

Se pondrá crear en el ambiente de AutoCAD el programa con instrucciones *AutoLISP* pero como serán realizadas en memoria principal se perderá dicho programa una vez interpretado. Procúrese trabajar utilizando un editor y asignando un nombre adecuado al archivo, revisando que la extensión de este archivo sea ".LSP" . Ejemplo de un nombre de archivo con instrucciones *AutoLISP*:

PLACA.LSP

Incorporar el archivo *AutoLISP* a AutoCAD para su interpretación:

```
command: (LOAD "FUNCION")
(funx)
```

El mensaje anterior indica que en el archivo PIEZA.LSP venia la función "funx" que fue interpretada y está lista para ser evaluada desde AutoCAD.

Se procurará que todo lo que se haga en *AutoLISP* sea a través de funciones; y para evaluarlas simplemente se hará escribiendo su nombre junto con los posibles argumentos o datos iniciales que requiera:

```
command: (funx)
```

Recuerde que como función, siempre se hará referencia a ella entre paréntesis.
Un ejemplo de función con parámetros es:

```
command: (funx '(1 1) '(4 4))
```

En el ejemplo anterior la función "funx" requiere como datos de entrada dos puntos o listas.

TIPOS DE DATOS EN AUTOLISP

Así como en los lenguajes de programación tradicionales se definen los diferentes tipos de valores a partir de los cuales se construirá la mitad de cualquier concepto de programación, en *AutoLISP* se mantiene el mismo principio. Un dato o valor define la manera en que la información se podrá representar, así por ejemplo, en AutoCAD gran parte de la información que se necesita manipular son puntos, de tal manera que *AutoLISP* tiene definido un tipo de valor que representa a esta información (en este caso existe el tipo de dato lista).

En *AutoLISP* los tipos de datos que se reconocen son:

ENTEROS

Son los números enteros (sin fracción decimal) que podrán ser positivos o negativos. Cualquier operación que se realice entre enteros resultará invariablemente en otro valor entero. Ejemplo:

```
-32768
+32767
0
(/ 25 2)        ----->     12
```

En el último ejemplo, se define una expresión con la función cociente (/)

donde se dividirá 25 entre 2, como bien sabemos, el resultado matemáticamente hablando será 12.5, pero como en la relación operan dos valores enteros el resultado será otro entero, de tal manera que el resultado será 12 .

REALES

Son todos los números reales conocidos (tradicionalmente son todos aquellos números con punto y fracción decimal). Dentro de estos valores podrán definirse magnitudes mucho más grandes que las definidas por un valor entero con la diferencia que el manejo de valores enteros es más ágil y directa que la manipulación de reales. De la misma forma que para los valores enteros, toda operación entre reales resultará en otro real. Ejemplo:

```
3.1416
19.              ----->      19.0
-0.51012
.1843            ----->      error, debe ser 0.1843
(/ 25. 2.0)      ----->      12.5

observe la siguiente expresión:
( /  25.  2 )    ----->       12.5
```

el resultado será un real, aunque la operación mezcló un valor real (25.) con un valor entero (2), todos los valores reales tienen dentro de los valores numéricos una categoría mayor; de tal manera que en la mezcla de tipos numéricos, siempre resultará un real, si en la relación existiese un real.

Es importante recalcar que cuando se realicen operaciones aritméticas (suma, resta, multiplicación y división) se conozcan perfectamente bien qué tipos de datos numéricos se manejan (reales o enteros) para poder determinar correctamente el tipo de valor numérico que resultará. Se recomienda que no se utilice la mezcla de tipos numéricos, aunque puede existir.

STRING

Este tipo de dato es también conocido como tipo de dato cadena o alfanumérico, normalmente se denomina "string" y define a todos aquellos valores que no son numéricos (ni reales o enteros). Ejemplo:

```
"Proporciona el punto 2"
"Curso AutoLISP"
"123"
(STRCAT "Curso AutoLISP" " DECFI UNAM")
```

Observe que todos los valores "string" se marcan entre comillas. En el último ejemplo se hace uso de la función *STRCAT* con la cual se unen cadenas o "strings"; el resultado de esta función será otro valor "string" cuyo valor será para nuestro ejemplo: "Curso AutoLISP DECFI UNAM".

Generalmente, los valores "string" se utilizan como complemento de alguna instrucción que necesite mostrar al usuario algún mensaje en pantalla.

Todos los valores cadena tienen un valor numérico equivalente, dicho valor se utiliza internamente para su manejo, este valor se conoce como código ASCII.

LISTAS

Son los tipos de datos fundamentales de *AutoLISP* a través de los cuales se hará referencia a:

- Puntos en 3 y 2 dimensiones
- La base de datos de AutoCAD

Son ejemplos de listas :

```
(4.0  5.0  0.0)
(1 . "CIRCLE")
```

En el primer ejemplo se establece el punto de coordenadas X, Y, Z , donde X vale 4, Y vale 5 y Z vale 0.

En el segundo ejemplo se define una lista muy particular que se conoce como sublista. Aquí se tiene una pequeña muestra de como se almacena la información generada por AutoCAD en la base de datos.

Nótese que todas las listas tendrán siempre que marcarse encerradas entre paréntesis y sus elementos vendrán casi siempre separados por lo menos por un espacio en blanco.

Para el manejo de una lista existe una diversidad de funciones que posteriormente se describirán, pero en su estructura se presentarán las siguientes partes:

(4.0 5.0 0.0)	----------	lista normal
	4.0	cabeza de la lista
	(5.0 0.0)	cuerpo de la lista

DESCRIPTORES DE ARCHIVOS

Son referencias indirectas que se establecen para indicar el uso de una archivo ya sea para lectura o para escritura. Estos valores son manejados generalmente con variables:

NOMBRES DE ENTIDAD

Todos los objetos generados en AutoCAD (círculos, líneas, textos, bloques, etc) tiene una referencia interna con la cual son identificados dentro de la base de datos. El *nombre de la entidad* u objeto es la llave para extraer toda la información referente a dicho objeto. Ejemplo:

6000142a

Este *nombre de entidad* es asignado por AutoCAD y generalmente presentará la anterior apariencia, agrupación de números y letras. Estos nombres son obtenidos a partir del uso de funciones adecuadas para la manipulación y localización de objetos así como para la obtención de información desde la base de datos del dibujo.

CONJUNTOS SELECCION

Es el agrupamiento de uno o más objetos de AutoCAD. Recuérdese el uso del comando SELECT de AutoCAD, un conjunto tiene su mismo objetivo: seleccionar un grupo de objetos para realizar con este una determinada operación.

Para definir un conjunto se utilizarán funciones de *AutoLISP* adecuadas que más adelante se mencionarán, dicho conjunto o conjuntos serán identificados por un nombre.

FUNCIONES INTRÍNSECAS

Son todas aquellas funciones no generadas por el usuario y que *AutoLISP* ya tiene definidas. Un programa en *AutoLISP* se compone invariablemente de las siguientes partes:

- Datos o valores a manipularse
- Funciones intrínsecas o de *AutoLISP*
- Funciones creadas por el usuario.

Esta composición siempre estará presente.

EXPRESIONES EN AUTOLISP

Una expresión de *AutoLISP* deberá incluir un operador seguido de los elementos que serán operados. Un operador es una instrucción que toma alguna acción específica; por ejemplo, se tienen los operadores matemáticos (+) para sumar o (/) para dividir.
AutoLISP se refiere al operador como a una función y a los elementos a ser operados como a los argumentos de la función.
En la expresión:

```
(+  4.5  3)
```

el signo "+" es la función y los números 4.5 y 3 son los argumentos de la función. Todas las expresiones de *AutoLISP*, sin importar tamaño, seguirán esta estructura e irán encerradas entre paréntesis:

```
(función arg1 arg2 ..)
```

Los paréntesis en una expresión deberán estar balanceados, es decir que por cada paréntesis izquierdo deberá haber un paréntesis derecho. Si se introduce en el intérprete de *AutoLISP* una expresión desbalanceada, se provocará el siguiente mensaje:

```
n>
```

donde *n* es el número de paréntesis requeridos para completar la expresión. Si manda este mensaje, se deberán dar el número de paréntesis indicado por *n* para que se restablezca el prompt

```
command:
```

Nótese que se requiere al menos un espacio entre los elementos de una expresión. El orden, el tipo de argumentos y el número de estos que siguen a una función varía dependiendo de la función, pero la estructura de la expresión es siempre la función seguida por los argumentos, todo entre paréntesis.

Cada vez que se encuentra una expresión, *AutoLISP* evalúa todo, no sólo la expresión: los argumentos también los evalúa. En el ejemplo anterior, *AutoLISP* evalúa el argumento 4.5 y el argumento 3; en éste caso los números los evalúa a sí mismo.

Debido a que *AutoLISP* evalúa los argumentos, se pueden usar expresiones como argumentos de una función; por ejemplo

```
(+ (- 2  3)  8)    ----->    7
```

VARIABLES

Las variables son localidades de memoria que permiten guardar valores para poder usarlos posteriormente. El valor de la variable puede cambiar en el curso de un programa.
Existen varios tipos de valores que podemos guardar en variables:

Entero (INT)

Real (REAL)

Cadena (STRING)

Lista (LIST)

Archivo (FILE)

Entidad (Entity)

Conjuntos Selección (Selection Set)

Símbolo (SYM)

Función (SETQ)

ASIGNACION DE VALORES A VARIABLES

A las variables se les asignan valores con la función *SETQ*. Como se ha visto, una función puede ser un simple operador, como el de suma, pero también puede consistir de un conjunto de funciones más complejas igual que un programa. Ejemplo:

```
Command: (SETQ  perim  8.5)
```

Para obtener el valor de una variable, dentro del ambiente de AutoCAD, se teclea un signo de admiración "!" y el nombre de la variable:

```
Command: !Perim  -----> regresa el valor 8.5
```

Pruebe la siguiente expresión y analice como trabaja:

```
Command: (SETQ  perim (+ perim  1))
```

Asignando valores cadena:

```
Command: (SETQ  nombre  "Ejemplo")
Command: (SETQ  texto   "12")
```

Los valores cadena "Ejemplo" y "12" se guardan en las variables *nombre* y *texto* respectivamente. Podría pensarse que la expresión *(SETQ perim 8.5)* causará error debido a que *AutoLISP* siempre evalúa los argumentos antes de ejecutar la función. Entonces evaluaría perim, el cual tiene valor nulo (pues no se le ha asignado valor alguno) e intentaría asignarle a éste el valor 1, lo cual sería inconsistente.

¿ Por qué entonces no marca error en la función *SETQ* ?

La función *SETQ* en particular, evita que se evalúe el primer argumento. Otra forma de evitar evaluar un argumento es anteponiendo un apóstrofe a éste; ejemplo:

```
(SET 'perim  8.5)
```

La función *SET* es semejante a *SETQ*, sólo que evalúa todos los argumentos de la función, a menos que se les anteponga un apóstrofo.

A continuación se lista algunas funciones matemáticas más comunes:

a) Que aceptan múltiples argumentos

(+ número número...)	Suma
(- número número...)	Resta
(* número número...)	Multiplicación
(/ número número...)	División
(max número número...)	El mayor número de la lista
(min número número...)	El menor número de la lista
(rem número número...)	Residuo de la división

b) Que aceptan uno o dos argumentos

(1+ número)	Adiciona 1

(1- número)	Resta 1
(abs número)	Regresa valor absoluto
(exp número potencia)	Numero a la potencia dada
(fix real)	Convierte real a entero
(float entero)	Convierte entero a real
(gcd entero entero)	Máximo común denominador
(log número)	Logaritmo natural de número
(sqrt número)	Raíz cuadrada de número

ASIGNACION DE VALORES DE LISTAS

En muchas de las instrucciones de AutoCAD se manejan puntos para definir una posición en el área de dibujo; éstos se definen por coordenadas que no es un simple valor, sino un grupo de valores. Por ser un punto un grupo de valores, en *AutoLISP* deberá usarse una lista para definirlo. Por ejemplo, para almacenar el punto con coordenadas *7, 9* en la variable *punto1*, se puede utilizar la función *LIST* :

```
Command: (SETQ punto1 (LIST  7 9))
Command: (SETQ punto1 '(7 9))
```

Como se ve, la función *LIST* regresa una lista de valores. Así también una variable no sólo acepta valores simples; también pueden almacenarse listas en variables:

```
(SETQ  VX  1.0)
(SETQ  VY  5.2)
(SETQ  p1  (LIST  VX  VY))
!p1          ----->     (1.0  5.2)
```

FUNCIONES CAR Y CADR

Para manejar los elementos de listas, existen las funciones *CAR* y *CADR*. Suponiendo que se quiere manejar sólo la coordenada *X* del punto *p1* teclee:

```
Command: (CAR p1)        -----> 1.0
```

Ahora para acceder a la coordenada *Y*, teclee:

```
Command: (CADR p1)       -----> 5.2
```

éstos valores pueden asignárseles a variables:

```
(SETQ  p1X  ( CAR p1))
(SETQ  p1Y  ( CADR p1))
```

FUNCIONES BÁSICAS

Funciones de lectura

AutoLISP cuenta con una serie de funciones que permiten que se de información desde el área de dibujo. Estas funciones llevan el prefijo *GET*. Algunas de estas funciones son:

GETPOINT
Permite proporcionar vía teclado o con el *mouse* un punto.

GETCORNER
Permite dar la esquina opuesta de una ventana (window). Requiere que se le dé la primera esquina.

GETANGLE
Permite dar un ángulo. Regresa el valor en radianes.

GETDIST
Permite dar una distancia desde el teclado o con el *mouse*.

Ejemplos:

```
Command: (SETQ  punto1  (GETPOINT))
```

En la línea de comandos se queda esperando a que se le teclee las coordenadas de un punto o se defina un punto con el *mouse*.
Los valores de las coordenadas se le asignan a la variable *punto1* en forma de lista. Se puede usar un prompt (mensaje) en las funciones *GET*:

```
Command: (SETQ punto1 (GETPOINT "primer punto":))
```

Esta instrucción espera por un punto, pero al mover el *mouse* se ve una caja elástica según lo movamos, teniendo fija la esquina definida con *punto1*.

Funciones para conversión de datos

(ANGTOS real)	convierte de número a string (cadena)
(ASCII string)	convierte un string (cadena) a su código ASCII
(ATO1 string)	convierte de string (cadena) a entero
(ITOA integer)	convierte de entero a string (cadena)
(CHR integer)	regresa un string (cadena) con el carácter con codigo ASCII dado por un entero
(ATOF string)	convierte de string (cadena) a real

Funciones para el manejo de listas

(CAR lista)	Primer elemento de una lista
(CADR lista)	Segundo elemento de una lista
(CDR lista)	Regresa una lista sin el primer elemento

Ejemplos:

```
(SETQ lista1 (LIST  1  2  3)   -----> lista1
(CAR lista1)                   -----> 1
(CADR lista1 )                 -----> 2
(CADR (CDR lista1))            -----> 3
(CDR  lista1)                  -----> ( 2 3 )
```

Otras funciones para manejo de listas:

(NTH)	Elemento enésimo de una lista
(LENGTH)	Número de elementos
(REVERSE)	Lista invertida
(LAST)	Ultimo valor de la lista
(APPEND)	Junta listas en otra lista

(CONS) Agrega un primer elemento a una lista.

Poner lista asociativa

(ASSOC) De una lista obtiene una sublista a partir de una referencia (1er. elemento de la sublista).

(SUBST) Sustituye elementos en una lista

Ejemplos:

```
(SETQ LISTA (LIST 1 2 3))
(SETQ LISTA1 (LIST '(1 2) '(3 4)))

(NTH 2  lista)              -----> 3
(LENGTH lista)              -----> 3
(LENGTH lista1)             -----> 2
(REVERSE lista)             -----> (3 2 1)
(REVERSE lista1)            -----> (3 4) (1 2)
(LAST lista)                -----> 3
(LAST lista1)               -----> (3 4)
(APPEND lista lista1)       -----> (1 2 3(1 2)(3 4))

(CONS 100  lista )          -----> (100 1 2 3)
(CONS '(5 6) lista1)        -----> ((5 6)(1 2)(3 4))

(ASSOC 1  lista)            -----> ?
(ASSOC 3  lista1)           -----> (3 4)
(ASSOC 4  lista1)           -----> nil
(SUBST 8  1  lista)         -----> (8 2 3)
(SUBST '(5 6) '(1 2) lista1)
                            -----> ((5 6)(3 4))
```

```
(CONS 8 "0")          -----> (8 . "0")
```

Funciones para el manejo de caracteres

(STRCASE)	Convierte cadenas en mayúsculas o minúsculas
(STRCAT)	Junta cadenas
(SUBSTR)	Substrae una parte de la cadena
(STRLEN)	Tamaño de la cadena
(ITOA)	Transforma un entero en cadena
(ATOI)	Transforma cadena en entero
(ATOF)	Transforma cadena en real

Ejemplos:

```
(SETQ TEXTO "A B C D E F" )
(STRCASE TEXTO)          -----> "a b c d e f"
(STRCASE TEXTO T)        -----> "A B C D E F"
(STRCAT TEXTO "1 2 3")
                         -----> "A B C D E F1 2 3"

(SUBSTR TEXTO 7)         -----> "D  E  F"
(SUBSTR TEXTO 3 4)       -----> "B C "
(SUBSTR TEXTO 4 3)       -----> " C "
(STRLEN TEXTO)           ----->  11
(ITOA 45)                ----->  "4 "
(ATOI "114")             ----->  114
(ATOF "8.12")            ----->  8.12
```

Ejecutando comandos de AutoCAD desde AutoLISP

La función *COMMAND* permite invocar a un comando de AutoCAD dentro de una instrucción de *AutoLISP*. Esta función nos permite hacer cualquier cosa que se permita desde el prompt *command:* de AutoCAD. La sintaxis de esta función debe ser acorde a lo siguiente :

- Dar comandos, opciones y textos entre comillas.
- No usar funciones de entrada *GET*.
- No utilizar variables precedidas por la exclamación '!'.
- Llamar sólo comandos básicos de AutoCAD, no se permite comandos del usuario definidos con *DEFUN*.

La función *COMMAND* tiene la siguiente sintaxis :

(COMMAND arg1 arg2 ...)

AutoLISP evalúa primero todos los argumentos (variables o expresiones) y envía enseguida cada uno de los argumentos a AutoCAD. Las expresiones entre comillas son tomadas literalmente por AutoCAD. Las palabras que no estén entre comillas se tomaran como variables de *AutoLISP*. Expresiones precedidas por un apóstrofe no serán evaluadas.

Las constantes numéricas pueden o no ir entre comillas. Ejemplos:

```
(COMMAND "LINE" "1,1" "5,1" "5,4" "c")
(COMMAND "LINE" '(1 1) "5,1" (LIST 5 4) "c")
(COMMAND "LINE" (LIST 1 1) '(5 1) (QUOTE (5 4)) "c")
```

En *AutoLISP*, para indicar un <RETURN> se utilizará la doble comilla :

```
(COMMAND "LINE" "1,1" "5,1" "")
```

La función *COMMAND* siempre regresa como valor *nil*.

Definición de Funciones de Usuario

AutoLISP como ya se ha visto realiza todo el trabajo a través del uso de un sin número de funciones o procedimientos. *AutoLisp* reconoce como funciones dos tipos:

- **Funciones de *AutoLISP* o intrínsecas**
- **Funciones definidas por el usuario**

La posibilidad de crear nuevas funciones hace de *AutoLISP* una excelente herramienta de desarrollo porque permite la conjugación de utilerías de *AutoLISP* con el ingenio y la creatividad del usuario. La forma de definir una función del usuario es con otra función llamada *DEFUN*:

```
(DEFUN cubo (x) (* x x x))
```

En el anterior ejemplo se define la función del usuario *cubo*, que cuenta con un sólo argumento *x*. La función propiamente dicha está definida por otra función :

```
(* x x x)
```

Observemos otro ejemplo :

```
(DEFUN sumcub (x y / A B)
     (SETQ A (* x x x))
     (SETQ B (* y y y))
     (+ A B)
)
```

Se definió una función que presenta la forma más generalizada, aquí se observan :

- Los argumentos, *'x' 'y'*
- Variables locales, *'A'* , *'B'*

La función *DEFUN* tiene como complemento:

- Nombre de la función

- Argumentos de la función
- Definición de las variables locales.

Nombre de la función, es un conjunto de caracteres alfanuméricos que identifican a toda la función al momento de invocarla desde AutoCAD:

```
command: (sumcub 4 5)
```

Argumentos de la función: son los parámetros que representan a los valores que se pasan a la función. Los argumentos podrán tener la siguiente presentación:

- En la definición de la función sólo se marcan los nombres de las variables que asumirán los valores para valuar la función.
- En la llamada a la función se especifican los valores (enteros, reales, cadena, listas, etc.) que pasarán representados por las variables referidas en la definición de la función.

Definición: `(DEFUN sumcub (X Y))`

Llamada: `(sumcub 4 5))`

En este caso *X* asumirá temporalmente el valor de 4 y *Y* el de 5; después de valuar la función *X* y *Y* esperarán asumir otros valores al momento de llamar nuevamente la función.
La definición de las variables locales tiene la idea de optimizar el espacio de *AutoLISP* dedicado a guardar variables, esto es, toda variable definida por alguna función interna o del usuario pasa a ocupar ciertas localidades de memoria principal, si alguna de estas variables no es importante mantenerla con su actual valor, podemos anularla de memoria mandando precisamente como local su definición.
Las variables en *AutoLISP* son:

Globales o generales, es decir, que se pueden referenciar con un mismo nombre desde cualquier función. Son todas las variables de *AutoLISP*.

Locales sólo son utilizadas en una determinada función que así lo defina.

Al finalizar la evaluación de tal función, dichas variables desaparecerán de la memoria.

Ejemplo :

```
(DEFUN sumcub (X Y / A B))
```

En esta definición la función *sumcub* utiliza los siguientes tipos de variables:

Globales : *X* y *Y*
Locales : *A* y *B*

Expresiones de Control

Las expresiones de control son aquellas funciones que permiten realizar tareas de decisión y repetición condicionada. Recuerde las figuras lógicas de la programación estructurada (IF THEN ELSE y WHILE). En una decisión se obtiene siempre un valor lógico (cierto o falso, que en *AutoLISP* son True o nil) y es el que marca qué camino seguir para hallar la solución.

Funciones Lógicas

En *AutoLisp* cuando algo no tiene valor, representa *nil*; si tiene un valor, representa *T* (true, no-nil). A estos dos valores *nil* y *T* se les conoce como valores lógicos de una expresión. Como todo en *AutoLisp* se valúa, cualquier expresión tiene un valor lógico.

Existen las funciones lógicas, las cuales trabajan con los valores lógicos de las expresiones y regresan también un valor lógico:

(AND arg1 arg2...)

Regresa *T* si el valor lógico de todos los argumentos es *true* (*T*); si no, regresa *nil*.

(OR arg1 arg2...)

Regresa *T* si el valor lógico de uno de los argumentos es *true* (*T*); si no, regresa **nil**.

(NOT arg)

Regresa *T* si el valor lógico del argumento es *nil*; si no, regresa *nil*.

Suponiendo que *V1* y *V3* tienen valor y *V2* no lo tiene:

`(AND V1 V3)`	Regresa T
`(AND V1 V2)`	Regresa nil
`(AND V2)`	Regresa nil
`(AND V1 (GETINT "Dame N"))`	Depende de la respuesta a GETINT
`(OR V1 V2 V3)`	Regresa T
`(OR V2)`	Regresa nil

`(NOT V2)`	Regresa T
`(NOT (OR V1 V2))`	Regresa nil

Funciones de Relación

Las funciones de relación evalúan la relación entre dos expresiones. Si la relación es verdadera regresa *T*, si es falsa regresa *nil*.

A continuación se listan las funciones de relación, observe el operador para cada caso:

(< arg1 arg2 ...)
Regresa *T* si cada argumento es menor que el siguiente: si no, regresa *nil*.

(> arg1 arg2 ...)
Regresa *T* si cada argumento es mayor que el siguiente; si no, regresa *nil*.

(<= arg1 arg2 ...)
Regresa *T* si cada argumento es menor o igual que el siguiente: si no, regresa *nil*.

(>= arg1 arg2 ...)
Regresa *T* si el argumento es mayor o igual que el sig; si no, regresa *nil*.

(/= arg1 arg2)
Regresa *T* si los argumentos son diferentes; si son iguales, regresa *nil*.

Las anteriores funciones sólo pueden llevar argumentos numéricos o strings.

(EQ arg1 arg2)
Regresa *T* si los argumentos son idénticos.

(EQUAL arg1 arg2)

Regresa *T* si los argumentos son iguales.

La función *EQ* se utiliza para comparar variables de listas y detecta si les fue asignado el mismo objeto. *EQ* es equivalente a las funciones '=' y *EQUAL* para comparaciones numéricas y de strings.

Suponiendo que X vale 10.0, Y vale 15.0, Z vale 7.0, A vale '(1 1 2) y B vale '(1 1 2):

```
(< X Z)          Regresa nil
(< Z X Y)        Regresa  T
(> 25 Z)         Regresa  T
(<  15.0 Y)      Regresa  T
(>  10 X Y)      Regresa nil
(  7 Z)          Regresa  T
(EQUAL 7 Z)      Regresa  T
(EQ A B)         Regresa nil
(EQUAL A B)      Regresa  T
(/= X 20.0)      Regresa  T
```

Función *IF*

Con esta función se puede condicionar el flujo de un programa. A la estructura de la función *IF* también se le reconoce como *IF-THEN-ELSE*. Recuerde la programación estructurada. La función necesita evaluar una condición; si (*IF*) esta es *T*, ejecuta una primera expresión; si no (else), ejecuta una segunda expresión (es decir, si la condición es *nil*).
La sintaxis de la función *IF* es la siguiente :

```
(IF condición expresión1 expresión2)
```

La condición puede ser cualquier expresión que valúe un valor lógico (*T* o *nil*), pero es recomendable utilizar siempre una función lógica o de relación. La *expresión2* es opcional. Si la condición es *T* se valúa la *expresión1*; si no, valúa la *expresión2* (si existe; si no, valúa *nil*). La función regresa el valor de la última expresión evaluada.
Ejemplo:

Suponiendo que la variable `a`, tiene valor de *10.0*:

```
(IF a "tengo valor" "no tengo valor")
```

La función *IF* valúa la condición, la cual da *T* debido a que `a` tiene valor, por lo tanto la primera expresión es evaluada y como es una cadena, se valúa a sí misma; la función regresa la cadena "tengo valor". Si la variable *b* no tiene valor:

```
(IF b  "tengo valor" "no tengo valor")
```

regresa "no tengo valor"

Si: x=10 y y=20

```
(IF (> x y) (setq menor x) (setq menor y))
```

en este caso, la condición es una función de relación que valúa *T* (verdadero),

puesto que el contenido de *x* (10) es menor que el de *y* (20); por lo tanto se evalúa la primera expresión que asigna a la variable 'menor' el contenido de *x* (10). La función regresa el valor 10.

La siguiente expresión es semejante a la anterior, aunque no funcionan igual. ¿Cuál es la diferencia?:

```
(SETQ menor (IF (< x y) x y))
```

Función *PROGN*

La función *IF* tiene la limitación de evaluar sólo una expresión sí cumpla o no la condición. En ocasiones es necesario evaluar varias expresiones si la condición resulta *T* o *nil*.
AutoLISP proporciona la función *PROGN* para resolver este problema.
La función *PROGN* permite incluir varias expresiones donde una expresión es requerida. *AutoLISP* trata al grupo de expresiones como una sola:

```
(PROGN expresión1 expresión2 expresión3 ... )
```

Se usa generalmente para evaluar múltiples expresiones en un *IF*. La función *PROGN* regresa siempre el último valor evaluado en la expresión. Ejemplo :

```
(IF (= (GETVAR "GRIDMODE") 0 )
    (PROGN
    (SETVAR "GRIDMODE" 1)
    (REDRAW)
    "Malla on"
    );
;Else
    "ya está prendida la malla"
);ENDIF
```

Función *COND*

Esta función trabaja en forma muy parecida a la función *IF*. Los argumentos de esta función consisten de una ó mas expresiones, cada una de ellas incluye una expresión condicional seguido de una ó mas expresiones a ser evaluadas si la condición regresa *T*. La función *COND* evalúa cada una de las expresiones condicionales hasta que evalúe a alguna con *T*; entonces evalúa las expresiones asociadas a esa condición e ignora todas las restantes:

```
(COND
     (condición1  expresión  expresión  ... )
     (condición2  expresión  expresión  ... )
          .
          .
          .
)
```

La función *COND* puede sustituir a la función *IF*, como lo muestra el siguiente ejemplo:

```
(COND
   ((< x y)  (SETQ menor x))
   ((> x y)  (SETQ menor y))
   ( T (SETQ menor x) "valores iguales")
);END
```

Función *REPEAT*

Esta función permite evaluar un conjunto de expresiones repetidamente un número determinado de veces. A esta evaluación repetitiva de un conjunto de funciones se le llama iteración fija o *FOR*. Su sintaxis es:

```
(REPEAT n expresión1 expresión2  ... )
```

donde: ***expresión1, expresión2*** **... serán evaluadas una vez en cada iteración.**

La función *REPEAT* regresa el valor de la última expresión en el proceso de iteración. Ejemplo:

```
(SETQ x1 1 y1 1 x2 1 y2 6)
(REPEAT 10
     (COMMAND "line" (LIST x1 y1)(LIST x2 y2) "")
      (SETQ x1 (+ x1 0.5))
      (SETQ x2 (+ x2 0.5))
 );END
```

El ejemplo anterior se dibujan 10 líneas paralelas con una separación de 6.5 unidades en sentido *x*.

Función *WHILE*

Esta función al igual que *REPEAT* evalúa varias expresiones repetidamente, pero a diferencia del *REPEAT* en el que se especifica el número de iteraciones, en la función *WHILE* el número de iteraciones está en función de una determinada condición. Su sintaxis es:

```
(WHILE condición expresion1 expresión2  ... )
```

condición	Debe ser una expresión condicional
expresión1	
expresión2 ...	Serán evaluadas repetidamente, evaluándose la condición antes de cada iteración, tantas veces mientras la condición regrese el valor *T*.

Usualmente la expresión condicional contiene variables, cuyo valor puede cambiar en el curso de una iteración. A continuación se tiene un ejemplo similar al utilizado para el caso de la función *REPEAT* solo que ahora usando la función *WHILE*:

```
(SETQ x1 1 y1 1 x2 1 y2 6)
(SETQ cont 1)
(WHILE (<= cont 10)
       (COMMAND "line" (list x1 y1) (list x2 y2) "")
        (SETQ x1 (+ x1 0.5) x2 (+ x2 0.5))
        (SETQ cont (1+ cont))
);END
```

Crear la siguiente función y probarla en AutoCAD:

```
(DEFUN C:OPERA (/ C1 D1 D2 r)
   (SETQ d1 (GETREAL "\n Dame el primer dato :"))
   (SETQ d2 (GETREAL "Dame el segundo dato :"))
   (WHILE (AND (/= c1 1)(/= c1 2)(/= c1 3)(/= c1 4))
      (SETQ (1 (GETINT "De la clave de operación: ") )
      (COND
         ((=  C1  1 ) (SETQ  r (+  d1  d2)))
         ((=  C1  2 ) (SETQ  r (-  d1  d2)))
         ((=  C1  3 ) (SETQ  r (*  d1  d2)))
         (T (PRINC "\nNúmero de clave inválido!"))
      );END
    );END
    (PROGN (PRINC "El resultado es:") (PRINC r)(PRINC)
)
);END OPERA
```

MANEJO DE ENTIDADES

Cada entidad de AutoCAD como una línea, arco ó circulo tiene un nombre que es reconocido por AutoCAD, este nombre cambia cada vez que se elabora un dibujo, por lo que no sería adecuado intentar recordar cada uno de ellos.
AutoLISP usa funciones de entidades para preguntar por el nombre de entidades de la base de datos del dibujo.
Realice el siguiente ejercicio:

Cree el Layer *EJEMPLO1* y ubíquese en él para trabajar

Dibuje una línea del punto *(1,1)* al punto *(10,1)*

```
LINE 1,1 10,1
```

Invoque las siguientes funciones de *AutoLISP*:

```
(SETQ ent1  (ENTLAST))
(SETQ datos1 (ENTGET ent1))
```

En la primera instrucción, la función *ENTLAST* regresa el nombre de la última entidad en la base de datos en la forma:

<Entity name: 60000064>

El nombre de la entidad se le asigna a la variable *ent1*.
En la segunda instrucción, la función *ENTGET* regresa una lista con los datos que describen a la entidad que se especifica, en éste caso la entidad *ent1*. Obsérvese que esta función requiere como argumento un nombre de entidad.

La lista que regresa se le asigna a la variable *datos1*:

```
((-1.<Entityname:60000064>)(0."LINE")(8."EJEMPLO1")
 (10 1.000000 1.00000)(11-10.000000 1.000000))
```

AutoLISP permite manipular entidades haciendo referencia a éstas con el nombre de entidad (*Entity name*).

Los datos asociados a una entidad se regresan en una lista con código *DXF* que indica el tipo de dato que está contenido en la sublista. Cada sublista tiene dos partes, la primera es el código *DXF* y la segunda es el valor del dato. El número *0* es el código para el tipo de entidad, que en el ejemplo anterior indica que es "*LINE*". El número *8* es el código para layer, y es indicado en el ejemplo como "*EJEMPLO1*". Los códigos *10* y *11* indican el punto inicial y punto final respectivamente.
Los códigos tienen diferentes significados con diferentes entidades. Se puede manipular todas las partes de cualquier dato de la entidad de un dibujo, lo que hace que *AutoLISP* sea una poderosa herramienta que permite acceso directo a la base de datos del dibujo de AutoCAD.

Funcion *ASSOC*

Esta función permite acceder a parte de una lista de datos de entidad por asociación, a través de *el código DXF. Por ejemplo, si se quiere obtener la sublista que tiene un código DXF* de *10*, *AutoLISP* busca en la lista de datos de la entidad una sublista con código *10* y regresa ésta:
Del ejemplo anterior:

```
(ASSOC 8 datos1)
      regresa la sublista: (8. "EJEMPLO1")

(CDR (ASSOC 8 datos1))
      extrae el dato del grupo layer
      regresa:"EJEMPLO1"
```

Función *TBLSEARCH*

AutoCAD guarda elementos como *blocks*, *layers*, *styles*, *linetypes*, *viewports* y *views* en tablas. La función *TBLSEARCH* busca a través de una tabla y regresa información de ésta. Requiere dos argumentos, la tabla a buscar y el nombre de elementos de la tabla:

```
(TBLSEARCH "LAYER" "EJEMPLO1")

regresa: ((0."LAYER")(2."EJEMPLO1")(70.0)(62.7)
         (6."Continuous"))


(TBLSEARCH "Style" "standard")

regresa:((0."STYLE")(2."STADARD")(70.0)
         (40.0.000000)(41.1.000000)(50.0.000000)
         (71.0)(42.0.200000)(3."TXT")(4." "))
```

MANEJO DE LA BASE DE DATOS

El uso de las características de cada uno de los objetos generados por AutoCAD se efectúa utilizando las siguientes funciones de *AutoLISP*:

a) Selección de objetos

(SSGET)
Define una selección de objetos

(SSGET "W" pto1 pto2)
Defina selección estilizando una ventana de extremos opuesto *pto1* y *pto2*

(SSGET "P")
Define una selección previa

(SSGET "L")
Define una selección utilizando el último elemento incorporado a la base de datos.

(SSGET '(X Y))
Define una selección, marcando un punto referido de un elemento.

(SSGET "C" pto1 pto2)
Define una selección utilizando una ventana cruzada.

(SSGET "X")
Define como selección todos los elementos de la base de datos.

((SSGET "X" (Filtro))
Define como selección todos los elementos de la base que cumplen con una condición (filtro)- listas asociativas. Ejemplo :

```
(SSGET "X" (LIST (CONS 8 "0")))
(SSGET "X" (LIST(CONS 8 "MUEBLES")))
```

b) Manipulación de objetos

(SSLENGTH)

Número de objetos en una selección *SSGET*.

(ENTSEL)

Pide seleccionar un sólo objeto y regresa su nombre con el punto con el que se especificó.

(ENTLAST)

Proporciona nombre del último objeto incorporado.

(SSNAME)

Proporciona los nombres de los objetos de una selección indicando la posición dentro del *SSGET*.

(SSADD)

Selección vacía, asigna valor vacío a una selección.

(ENTNEXT)

Proporciona la referencia al primer objeto incorporado a la base de datos.

(ENTGET)

Proporciona la información de un objeto através de su nombre

Algunos codigos DXF para la base datos AutoCAD:

0 *Tipo de entidad*

2 *nombre del bloque*

6 *tipo de línea*

7 *estilo de texto*

8 *nombre del LAYER*