UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE ESTUDIOS SUPERIORES ACATLÁN

ELEMENTOS DE PROGRAMACIÓN

T E S I N A
QUE PARA OBTENER EL TÍTULO DE

A C T U A R I O

P R E S E N T A

RICARDO SALDAÑA RUIZ

ASESOR: VÍCTOR MANUEL ULLOA ARELLANO

NAUCALPAN, EDO. DE MEXICO, 2006.

AGRADECIMIENTOS

Al tiempo en que toma forma este trabajo, que sólo ha podido ser realizado con el apoyo, comprensión, afán, y diligencia de diversas personas invaluables (sí, todavía, en estos tiempos que hemos coincidido en circunstancias, tengo la osadía de creer en cosas invaluables), en estos momentos no puedo quedarme sin agradecer por TODO en primera instancia a mis Padres: Antonio y Silvia, mis hermanos Silvia y Alejandro y mi tía Carmen por compartirse conmigo.

Sin olvidarme de esa personita en particular por quien tuve que aprender a programar…

Y a Victor, amigo, y maestro en esto, por quien, en primera instancia se ha logrado culminar este trabajo, ¡GRACIAS!

- PAGINA DEJADA INTENCIONALMENTE EN BLANCO -

INDICE

CAPITULO 3 – ESTRUCTURA GENERAL DE UN PROGRAMA

CAPITULO 4 – TÉCNICAS DE PROGRAMACIÓN

CAPITULO 5 – ESTRUCTURA DE DATOS

APENDICE A – BREVE CURSO DE PROGRAMACIÓN EN C

PROLOGO

El presente trabajo se propone servir a manera de introducción a la teoría de los algoritmos y a la programación; por lo tanto, ha de ocuparse primordialmente de uno de los conceptos fundamentales de la matemática, a saber, el de algoritmo. La temática de este interés se encuentra entre la frontera de la lógica matemática y la teoría de las máquinas computadoras.

Estas líneas han nacido por la inquietud de facilitar el aprendizaje y comprensión de la "lógica de la programación"; como premisa se tratan los conceptos de algoritmo, diagrama de flujo, estructura de programa, técnicas de programación y estructura de datos, sin olvidar al final un apéndice de programación en C, tratados en una manera "elemental" sin llegar al estudio exhaustivo de sus componentes.

Por lo que se necesitará del apoyo de maestros para llevarla al campo de la práctica en el que se sublimará el conocimiento de estos conceptos, que actualmente se aplican en casi todos los procedimientos de la actividad profesional.

En este sentido, al interesado en profundizar en estos conceptos se le recomienda la siguiente literatura:

1.- B. A. TRAJTENBROT, "Introducción a la teoría matemática de las computadoras y de la programación", Moscú.

2.- DOMINGO ALMENDAREZ AMADOR, "Circuitos lógicos combinatorios", IPN, México.

3.- ROBERT SEDGEWICK, "Algorithms", Addison-Wesley, USA.

4.- STEVEN S. SKIENA, "The Algorithm Design Manual", Springer-Verlag, New York.

5.- BRIAN W. KERNIGHAN, DENNIES M. RITCHIE, "The C programming language", Prentice Hall, USA.

6.- PETER VAN DER LINDEN, "Expert C programming, deep C secrets", Prentice Hall, USA.

- PAGINA DEJADA INTENCIONALMENTE EN BLANCO -

INTRODUCCIÓN

Antecedentes

Desde los años posteriores a la segunda Guerra Mundial, se ha visto el desarrollo en gran escala de las máquinas computadoras automáticas de alta velocidad que hoy en día se utilizan en la solución y planteamiento de los más diversos problemas (nos enfocaremos en ejemplos matemáticos y lógicos par nuestro actual interés). Aunque en la actualidad los programas (software) con el que disponemos, nos permite una interacción, capaz de alcanzar nuestros objetivos por partes, para este caso, nos centraremos en el caso clásico, aquel en el que la interacción sea sólo en dos instantes.

Es cierto que la lógica matemática ha permitido establecer los importantes teoremas que develan la esencia de los procesos realizados por las máquinas automáticas (o ahora llamadas: máquinas computarizadas). En particular se ha demostrado rigurosamente la existencia de problemas a los que la máquina no puede dar solución (todavía). Ésta se caracteriza por investigar la esencia de conceptos tales como los de "proceso de cálculo", "demostración matemática", "algoritmo" y otros. Es más, algunos años antes de la construcción de las máquinas computadoras actuales, en lógica matemática se habían formulado ya los conceptos exactos de "algoritmos" y el diagrama general de lo que sería una máquina computadora automática; así mismo, se estableció la relación íntima entre tales algoritmos y las máquinas.

Por esta parte, la posibilidad de un tratamiento matemático rápido y seguro, al igual que el análisis de los datos experimentales, crea la premisa propia de la aparición de nuevos métodos de investigación, antes inasequibles (y muchas veces impensables), en muchos dominios de la ciencia.

Y ya que la productividad de las máquinas actuales es "inmediata" y su campo de aplicación se ensancha sin cesar, por ejemplo: resuelven sistemas complicados de ecuaciones, traducen de un idioma a otro (aunque no como uno quisiera), juegan ajedrez, etc. Actualmente, casi todo el mundo reconoce que las maquinas computadoras son un instrumento poderoso para el trabajo intelectual, pues no solamente son capaces de aliviar el esfuerzo intelectual del ser humano, sino que, lo liberan por entero de otras de sus formas mayores y más tensas de aplicación (como trabajos repetitivos), e incluso llevando muchas veces la gran tarea que implica la toma de dediciones masivas (claro, lo que aplica en un trabajo de oficina en este caso), sin olvidar su aplicación en las industrias eliminando en muchos casos el "factor de error humano".

Lo único que no podemos olvidar es que, por el momento las computadoras sólo son las herramientas de nuestra "civilización", ya que necesitamos ponerlas a trabajar en ellas (programarlas), para llegar al momento que (en un futuro cada día más próximo), sean

capaces de aprender por si mismas, hasta entonces sabremos que nuevos problemas tendremos que plantear, y resolver, quien sabe, quizás con un ábaco.

Objetivo

El presente tiene por objeto plantear la relación entre algoritmos y computadoras, mostrar una forma de plantear los problemas (que como se dice, hay muchas maneras de resolver un problema, pero sólo hay un modo de hacerlo… como se debe realizar), buscando ese "modo" se expone la necesidad de investigar, razonar y sobre todo escudriñar los problemas, claro, los sujetos de nuestro interes.

Reciente, al formar parte del alumnado de la carrera de Actuaria y ante la necesidad de no sólo saber utilizar programas de computadora (como procesadores de texto, hojas de cálculo y navegadores de Internet), que para muchos casos resulta suficiente y bastante, pues trasciende que para los Actuarios resulta de capital importancia realizar sus propios programas, tales como aquellos que realicen cálculos masivos o tareas tan especificas (como por ejemplo: cálculos y reportes financieros, actuariales, ejercicios de optimización, etc.), que, como se puede imaginar, no queda de otra, hay que aprender a programar.

De esa necesidad surge este trabajo, un texto que, aunque ya existen muchos, no se encontró uno enfocado en nuestra actividad, la Actuarial. Que aunque llevando un tratamiento modesto, sin llegar al estudio exhaustivo de sus componentes, resultará de interes a los estudiantes que gusten de los "problemas", que al fin y al cabo es lo que hacemos los Actuarios, resolver problemas.

Un vistazo al contenido

Capitulo 1.- Algoritmos y Programas

En este capitulo se precisa y describe lo que es un Proceso, lo que resume a un Algoritmo y sus características, así como los datos, sus tipos y operaciones primitivas, el uso de constantes, variables y expresiones, lo que conlleva a los programas.

Capitulo 2.- Diagramas de flujo

Aquí se muestra por primera vez la noción de Diagrama de flujo, su descripción y sus componentes, las reglas de construcción de estos, el concepto de Pseudo-código y el de Modularización.

Capitulo 3.- Estructura general de un Programa

Empezando por el concepto de Programa, se muestra un primer desarrollo, distintos Lenguajes de programación y su clasificación, además de las partes constitutivas de un programa, los tipos de instrucciones así como los de programas.

Capitulo 4.- Técnicas de Programación

Escudriñando los elementos básicos de un programa, se presentan las "iteraciones", bifurcaciones, los "subprogramas" y las estructuras esenciales de todo programa.

Capitulo 5.- Estructura de Datos

Se de una introducción a la manipulación y estructura de datos, las Listas, los Árboles sus tipos y los recorridos en estos.

Apéndice A – Breve curso de programación en C

En esta sección se presenta un lenguaje no sólo de programación, sino de programadores, al ser C un lenguaje formativo, se pretende lucirlo, así como sus principales características y estructuras, llevándolo al desarrollo de una "librería" y un programa de Matemáticas Financieras.

- PAGINA DEJADA INTENCIONALMENTE EN BLANCO -

CAPITULO 1

ALGORITMOS Y PROGRAMAS

1.1 CONCEPTO Y DESCRIPCION DE UN PROCESO.

Una acción es un suceso o acontecimiento producido por un actor (ejecutante). Tiene la característica de una duración limitada y produce un resultado bien definido y previsto.

El tener una duración limitada en el tiempo implica la existencia de un instante inicial de la acción (t_o) y un instante final (t_i). Para poder reconocer el resultado del sistema debe estar provisto de indicadores que tomen valores diferentes. El valor de estos indicadores se denomina información. El conjunto de los valores de los distintos indicadores en un instante dado (t) del desarrollo del acontecimiento se denomina estado en el instante t del sistema observado. El resultado es el estado del sistema en el instante t_n. El estado del sistema en t_o define los datos de la acción.

Proceso *es una acción que se puede descomponer en otras más simples, o también conjunto de fenómenos organizados en el tiempo y concebidos como activos. Se puede considerar un proceso como un conjunto de acciones elementales que forman un acontecimiento. Procesador es el elemento capaz de ejecutar un determinado proceso de trabajo.*

Los procesos pueden ser: secuenciales y paralelos.

- *Un proceso es secuencial si una acción del mismo no puede empezar antes que la acción en curso esté completamente terminada; en otras palabras: dos acciones no se ejecutan simultáneamente, si no en un orden secuencial.*

- *Un proceso es paralelo si se ejecutan simultáneamente dos o más acciones. A lo largo del libro sólo trataremos los procesos secuenciales que en un instante dado tan solo se pueden ejecutar una única acción.*

Un ejemplo típico de proceso es:

Un cocinero elabora un plato de cocina (en este caso, un estofado de carne para 6 personas): Las acciones a seguir son: calentar el aceite, freír la carne, poner condimentos, etc.

Un ejemplo de proceso secuencial es:

Un estudiante calcula el producto de los números naturales m=25 y n=36.

La descripción del proceso necesitará:

1. La lista de datos.

2. El conjunto de acciones, orden de estas y condiciones que determinan la ejecución de una u otra acción (para este caso: m*n=25*36=900).

Los procesos en los que estamos interesados son aquellos que se pueden repetir y en los cuales el resultado no depende del ejecutor (siempre que éste se atenga perfectamente al punto 2 anterior). El proceso no ha de servir para resolver todos los problemas de cierto tipo (todos los que resultan de tomar m y n como números naturales).

1.2 ALGORITMOS

Algoritmo es una serie de operaciones detalladas y no ambiguas, a ejecutar paso a paso, y que conducen a la resolución de un problema. En otras palabras, es un conjunto de reglas para resolver una cierta clase de problema o una forma de describir la solución de un problema. Por ejemplo, la receta de cocina para hacer <<cordero asado>> es un algoritmo.

El algoritmo es el medio por el que se explica cómo puede resolverse un problema, mediante aproximaciones paso a paso. Se puede formular de muchas formas, siempre y cuando se realice de modo no ambiguo.

Para describir algoritmos de computadoras se han diseñado lenguajes de programación.

Cada una de las acciones de las que consta un algoritmo se llamará secuencia y estas deben ser escritas en término de cierto lenguaje comprensible para el ejecutor (máquina), que es el lenguaje de programación.

El conjunto formado por la representación de datos utilizada y el algoritmo en si, se conoce usualmente con el nombre de **Programa**. En esencia, un programa es la descripción del proceso en un cierto lenguaje, o dicho de otra manera: la secuencia de acciones entendibles por la computadora que conducen a realizar una tarea determinada y el correcto tratamiento de unos datos.

Recordemos la definición de algoritmo dada por el profesor Niklaus Wirth, padre del lenguaje Pascal: **Un algoritmo o programa de computadora** consiste en dos partes esenciales: una descripción de acciones que deben ser ejecutadas y una descripción de los datos que son manipulados por esas acciones. Las acciones se describen mediante las llamadas sentencias y los datos mediante declaraciones y definiciones (PASCAL. User Manual And Report. Springer-Verlang. New York Inc., 1975).

Los programas constan de una serie de tendencias (líneas de información con unas determinadas reglas de sintaxis). Estas partes elementales de un programa que son las sentencias se componen a su vez de instrucciones que son las acciones concretas que bebe realizar la máquina. Con frecuencia la palabra sentencia e instrucciones se confunden o consideran sinónimas. Nosotros preferimos utilizar sentencias para lenguajes de alto nivel -próximos al usuario- e instrucciones para lenguajes de bajo nivel y máquina –próximos a la máquina.

1.2.1 Características de los algoritmos

Se observa, normalmente, que el número de operaciones que realiza un algoritmo (o programa) no se conoce de antemano, aunque será finito siempre que los datos sean adecuados. Por consiguiente, el número de operaciones que es preciso realizar al ejecutar un algoritmo dependerá de los datos del problema y solamente se conocerá al ejecutar éste.

Las características fundamentales de un algoritmo o proceso algorítmico son:

a) Un algoritmo debe ser preciso e indicar el orden de realización de cada paso.
b) Un algoritmo debe estar definido. Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.
c) Un algoritmo debe ser finito. Si se sigue un algoritmo, se debe terminar en algún momento.

La definición de un algoritmo debería describir tres partes: entrada, proceso y salida. Un algoritmo implica generalmente alguna entrada (algo que existe y es utilizado por el algoritmo); en un algoritmo de receta de cocina, la entrada está constituida por los ingredientes y los utensilios empleados. Un algoritmo produce también resultados denominados salida. La salida de la receta será la terminación del plato (cordero asado).

Un algoritmo describe la transformación de la entrada en la salida.

1.3 DATOS, TIPOS DE DATOS Y OPERACIONES PRIMITIVAS

El primer objetivo de toda máquina (procesador) es el manejo de información o datos.

La representación de los datos utilizada determina la forma del algoritmo; realmente, éstos dependen del tipo de información sobre la que actúa. En consecuencia, la identificación de los objetos (datos) que manipula el algoritmo debe realizarse sin ninguna ambigüedad para que la máquina sepa en cada momento que tipos de datos manipula y cómo debe operar con ellos. El sistema de identificación de los datos que llamaremos definición de la estructura de datos, es, al menos, tan importante como los algoritmos que los transforman.

El lenguaje de programación no es más que una notación que describe las estructuras de datos y los algoritmos. Los datos con los cuales trabaja un programa se sitúan en objetos llamados variables. Al igual que todos los objetos de un programa, las variables llevan un nombre llamado identificador que sirve para referenciar su valor o contenido. Este valor puede ser examinado tan a menudo como sea necesario, borrado o reescrito. La acción que consiste en atribuir un valor a una variable se llama asignación. Los datos y sus valores tratados por un programa pueden ser de diferente naturaleza (números, caracteres, etc.). El tipo de una variable es el conjunto de valores que puede tomar.

La mayoría de las computadoras son capaces de trabajar con distintos tipos de datos: numéricos y no numéricos (series o cadenas de caracteres alfabéticos).

1.3.1. Datos numéricos

Los datos numéricos se representan en dos formas: números enteros y números reales. La computadora maneja de modo diferente ambas formas.

Los enteros corresponden a números completos; no tienen componente decimal o fraccionario y pueden ser negativos o positivos. Ejemplos de enteros son:

-3245	45
4672	-321
35	5

El rango normal de los números enteros suele ir de –32768 a +32767.

Los reales tienen siempre un punto decimal; las fracciones se almacenan en la computadora como números decimales porque no existe otra forma de almacenar numeradores u denominadores separados. Al contrario de los enteros que suelen tomar valores en un rango determinado, los números reales pueden tomar, teóricamente, cualquier valor de la recta numérica real y ser positivos o negativos. Los siguientes ejemplos son números reales:

34.6	4567.34
231.512	313.216
- 8.31	8.74

En notación de computadora la coma decimal no existe y es siempre debe ser sustituida por un punto.

Al objeto de poder representar números reales muy grandes o muy pequeños (3463.227.314 ó 0.00000000003845) se ha diseñado una notación denominada científica o de coma, punto flotante que tiene el siguiente formato:

$n = mxb \uparrow e$	$n = m \times b^e$

Donde:

m	es la mantisa.
e	es el exponente, igual a un entero.
b	es la base del sistema de numeración.
$\uparrow$	es el símbolo de exponenciación.
x	es el símbolo de producto.

En el sistema decimal el formato resulta ser:

$n = mx10 \uparrow e$	$n = mx10^e$

Aunque el rango se dijo antes indefinido, lógicamente a efectos prácticos tiene limitaciones. Normalmente m puede contener de 4 a 6 u 8 dígitos y e dos dígitos positivos o negativos.

Ejemplos típicos de números reales en coma flotante son:

0.34567 x 10↑23
34.567 x 10↑15
0.386 x 10↑-7
3.4567x 10↑14

↑ (Es el símbolo de la exponenciación) se sustituye normalmente por la letra e, y entonces resultará:

0.34567 e23
34.567 e15
0.386 e-7
3.4567 e14

La ocupación de la memoria de los números enteros y los números reales es distinta. Así en el caso de los micros IBM PC, los tipos de números aceptados son:

enteros: rango –32768 a +32767 (ocupan 2 bytes de memoria).
reales: simple precisión (7 dígitos de precisión).
doble precisión (15 dígitos de precisión).

1.3.2 Datos no numéricos

Existen fundamentalmente dos tipos: datos alfanuméricos y datos lógicos. Los datos alfanuméricos se agrupan en series o en cadenas de caracteres (caracteres alfabéticos A, B,...., X, Y, Z, a, b,...., x, y, z; los dígitos 0, 1, 2,...., 8, 9; caracteres especiales #, $, -, etc.).

Los datos lógicos son aquellos que pueden tomar dos valores <<verdadero>>y <<falso>>.

Una síntesis de los tipos de datos utilizados por los procesadores es la figura 1.1

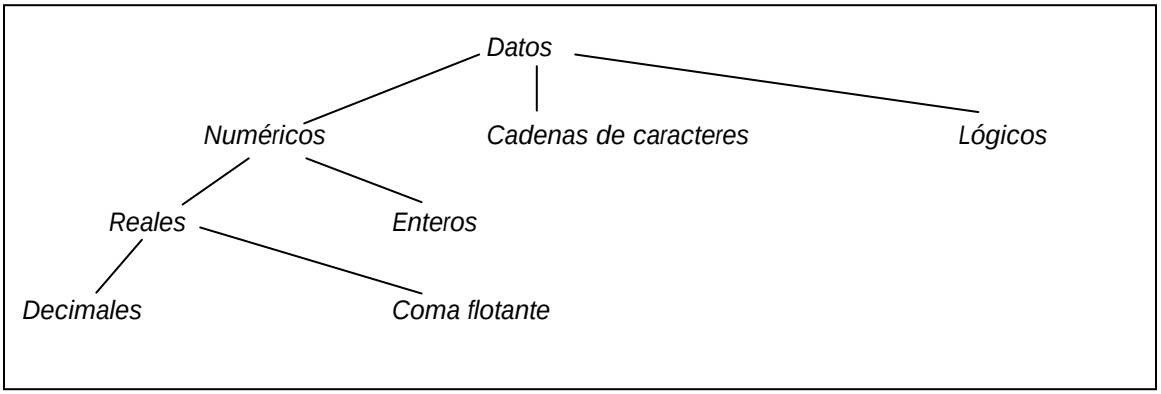


Figura 1.1. *Tipos de datos de los procesadores.*

1.4 CONSTANTES, VARIABLES Y EXPRESIONES.

Una **variable** es un objeto cuyo valor es variable. Además de este valor toda variable posee dos atributos: un nombre (invariable, denominado en ocasiones, caso del lenguaje Pascal, identificador) que sirve para designarla y un tipo (invariable) que describe la posible utilización de la variable. Al definir una variable se ha de precisar su nombre y su tipo. El nombre como ya se ha indicado anteriormente se suele conocer como identificador.

El valor de las variables puede ser modificado mediante la ejecución del programa. Dependiendo de los lenguajes las variables pueden ser: enteras, reales, de caracteres y lógicas (booleanas) y tomarán los valores vistos en el apartado anterior.

Una **constante** es un objeto de valor invariable. Este valor no cambia durante el proceso. Para expresar una constante se escribe explícitamente su valor, por ejemplo, 5, -70 ó 3.141519. La mayoría de los lenguajes permiten diferentes tipos de constantes, siendo los más comunes enteros, decimales, caracteres y booleanos (lógicos).

1.4.1 Constantes.

Constante entera (integer): es un número con un valor entero, positivo
Negativo, por ejemplo, 3, -4, 0.

Constante real: Una constante real o decimal es un número escrito con un punto decimal. Obsérvese que 3.0, aunque su valor es un número completo o entero, se considera una constante decimal.

Constante de caracteres: es un conjunto de uno o varios caracteres. Normalmente los caracteres disponibles son letras mayúsculas, minúsculas, dígitos, signos de puntuación y otros símbolos especiales.

1.4.2 Variables.

Enteras (integer): su representación interna está formada por una secuencia de bits en un código binario (complemento a uno, complemento a dos, código de exceso, signo y magnitud).

Las variables enteras pueden ser declaradas explícitamente por el programador.

BASIC	Pascal
A%, NOTAS%	**var** x, y, z: integer;

Reales: la codificación interna se representa en coma flotante o en notación científica.

BASIC	Pascal
A, NOTAS	**var** x, y, z: real;
A! B#	**var** u, v, w: real;

Caracteres (char): es un conjunto de caracteres.

BASIC	Pascal
A$, B$ NOMBRES$	**var** a, b: char;

Boolean: puede tomar los valores de álgebra de Boole (0,1) o bien (cierto, falso // true, false)

```
Pascal
var  a, b: boolean;
```

1.4.3 Expresiones

Las expresiones son combinaciones de constantes variables, símbolos de operación, paréntesis izquierda y derecha y nombres de funciones especiales. Las mismas ideas se utilizan en notación matemática tradicional, por ejemplo:

m(n+5)+ raíz de p

En la expresión anterior, los paréntesis indican el orden del cálculo y raíz cuadrada.

Cada expresión tiene un valor, que se determina tomando los valores de las variables y constantes implicadas y ejecutando las operaciones indicadas.

Las expresiones se clasifican en dos tipos:

- *Expresiones aritméticas*
- *Expresiones booleanas*

1.4.3.1. Expresiones aritméticas.

La mayoría de los lenguajes permiten diferentes tipos de expresiones. Expresiones aritméticas son análogas a las fórmulas matemáticas. Las variables; constantes son numéricas (enteros o reales) y las operaciones son las aritmética clásicas.

+ *suma*
- *resta*
* *multiplicación*
/ *división*

*Los símbolos +, -, *. /. Se denominan operadores cuando actúan sobre determinados datos:*

*AB se suele escribir en un lenguaje de programación como A*B*

Los paréntesis se utilizan también para agrupar términos juntos y asegurar que las operaciones se ejecutan en el orden correcto. En la expresión M*(n+5), la constante 5 se suma en primer lugar al valor n, y su resultado se multiplica por M. Cualquier ambigüedad se resuelve asociando a cada operador una prioridad o procedencia y ejecutando las operaciones en orden de prioridad. (En el capitulo 3 se ampliarán estos conceptos.)

1.4.3.2. Expresiones booleanas

Un segundo tipo de expresión es la expresión booleana, cuyo valor puede ser o bien verdadero o bien falso. Un medio de generación de expresiones booleanas es combinar constantes y expresiones por medio de los operadores booleanos **and, or** y **not**.

Las reglas de funcionamiento de estos operadores racionales o de comparación (=, >, <, >=, <=, < >) se analizarán en detalle en el Capitulo 3.

Algunos ejemplos de expresiones son:

```
A + (B / C) * D
19 mod 4
A > B
(A - 5) < (B – 6)
```

La asignación de una constante a una variable se realiza de acuerdo al formato siguiente:

```
V = constante          en lenguaje BASIC
V = constante          en lenguaje PASCAL
```

El segundo miembro de la asignación puede, de hecho, ser cualquier expresión en la que se tenga una combinación de variables, constantes y operadores (suma, resta, multiplicación, división....). El resultado de la evaluación de tal expresión será el valor que se le asigne a la variable indicada.

```
V: = 45 + 73 + 3
```

Se le asigna a V el valor 121

```
V: = a + b – m
```

A V se le asigna la resultante de la operación a + b – m, donde a, b y m son los valores que en ese momento tengan las variables así nombradas.

1.5. LOS PROGRAMAS

Se podría considerar la programación como el conjunto de actividades y operaciones realizadas por el personal informático tendentes a instruir a la máquina para que pueda realizar las funciones previstas en el algoritmo. La programación se manifiesta en los

programas (el concepto de programa se introdujo en el apartado 1.2. El desarrollo de un programa abarca diferentes etapas, de las cuales la escritura puede ser menos significativa.

La primera etapa al escribir un programa es la definición del problema. Una vez definido el problema se puede diseñar la solución. El diseño general elegido se desarrolla posteriormente en forma de algoritmo; un método muy utilizado para el desarrollo es el método arriba-abajo (top-down), que consiste en partir de una idea general y definir cada paso posterior con más detalle hasta llegar a la resolución del problema. En esta descomposición de la idea general, se suele recurrir, a veces, a algoritmos normalizados.

Tras diseñar el algoritmo, se pasa a la escritura en un lenguaje de programación. El proceso de escribir las sentencias reales de un lenguaje de programación se denomina codificación. Es importante no realizar la codificación hasta tanto el algoritmo no esté prácticamente definido. La codificación es una parte de la programación.

Una vez codificado el programa, se ejecuta y se comprueban sus errores. La operación de detectar y corregir errores se denomina depuración. Las comprobaciones sucesivas del programa permitirán detectar la presencia de errores, pero no detectarán su ausencia, ya que pueden existir otros errores que no se hayan encontrado por no haberse probado. Tras la depuración final del programa y la ejecución sin errores, es preciso documentar el programa. Es necesaria una documentación interna con comentarios convenientes a los largo del programa que permitan ayudar a futuras modificaciones, y documentación externa basada en la descripción completa del algoritmo, organigrama, tablas de desición, etc.

La vida del programa se continúa en la fase de mantenimiento, que consiste en las operaciones necesarias para mantener el programa al día, es decir posibilitar que el programa cumpla con sus objetivos pese a las variantes o modificaciones que sus datos, acciones, etc. puedan sufrir con el paso del tiempo.

- PAGINA DEJADA INTENCIONALMENTE EN BLANCO -

CAPITULO 2

DIAGRAMAS DE FLUJO

2.1 INTRODUCCIÓN

La realización de trabajos mediante computadora, como cualquier otra actividad (ingeniería, arquitectura, etc.) requiere un método que explique de un modo ordenado y secuencial hasta los últimos detalles a realizar por la máquina.

Al igual que para construir una casa, la empresa constructora no comienza por el tejado, sino que se encarga a un arquitecto el diseño de unos planos, al jefe de proyectos un calendario de actividades, etc., una aplicación informática, sobretodo si tiene cierta complejidad, no debe comenzar nunca por la codificación de programa, sino que exige una serie de fases previas destinadas a conocer todos los aspectos del problema planteado y estudiar las posibles soluciones. El conjunto de todas las fases necesarias para el completo desarrollo de una aplicación informática recibe el nombre de análisis y analista se denomina la persona encargada de realizarla.

El análisis de sistemas se encarga del funcionamiento de un sistema informático, incluyendo en él, todos los medios informáticos y materiales (hardware/software) y humanos, así como de la organización que controla su funcionamiento. El analista de sistemas es la persona encargada de controlar todo el sistema informático y estudiar la necesidad de nuevos medios humanos o informáticos, cuidando el buen funcionamiento de los existentes. Así mismo el analista de sistemas deberá proponer las soluciones optimas ante cualquier problema de funcionamiento que se plantee en el sistema informático y sugerir el desarrollo de nuevas aplicaciones. Así se puede decir que el analista de sistemas es el responsable de todo el proceso informático, tanto en un aspecto estático (uso del hardware/software existente y puesta al día del sistema) como en su aspecto dinámico (adquisición de hardware/software, nuevos desarrollos, contratación/formación de personal, etc.).

El análisis de aplicaciones es el estudio de nuevas aplicaciones en un sistema informático y la persona encargada de su realización es el analista de aplicaciones, que recibirá los encargos del analista de sistemas o en el caso de pequeños equipos informáticos (como puede ser la gama de computadoras personales PC, XT y AT) directamente del propio usuario. En el entorno del PC, es muy frecuente que el usuario pretenda ser al mismo tiempo el analista de sus propias aplicaciones. Esta obra va dirigida esencialmente a las personas que desarrollan aplicaciones bien en su faceta de analista o de programador que al igual que en el caso citado, se suelen confundir en los pequeños equipos.

Así pues, centraremos nuestra atención en la metodología de la programación que a grandes rasgos se compone de las siguientes etapas:

a) Toma de datos

El analista o programador deberá recibir una descripción clara y detallada de la aplicación que debe desarrollar y en caso de no percibir suficiente información, debe solicitarla en la medida necesaria para el desarrollo de su trabajo. Las Tablas de Decisión dan un buen método de toma o captura de datos.

b) Modularización

Esta fase consiste en la descomposición sucesiva del problema en módulos o subproblemas cada vez más concretos y detallados. Estos módulos normalmente se programarán y desarrollarán independientemente y luego se enlazarán.

c) Representación gráfica de las operaciones a realizar.

En esta etapa se realizará una representación gráfica, clara y detallada que refleje la secuencia en que deben ser ejecutadas las diferentes operaciones por la máquina. Estas representaciones gráficas son herramientas utilizadas para el análisis de la programación. Se clasifican en tres grandes bloques: diagramas de flujo u organigramas, pseudo-códigos y tablas de decisión. A lo largo de este capitulo y los restantes del libro se explicarán en detalle el uso adecuado de estas herramientas de programación.

d) Codificación en un lenguaje de programación.

Una vez que el diagrama de flujo o el algoritmo de resolución del problema ya está definido se pasa a la fase de codificación del programa en el lenguaje elegido y la obtención del programa fuente.

e) Preparación de un conjunto de datos.

Es necesario un conjunto de datos que permitan probar el programa cuando se ejecute.

f) Ejecución y corrección de errores del programa.

Según sea el tipo de lenguaje elegido así se realizarán las diferentes fases de la traducción del programa a lenguaje o código máquina (programa objeto). En el Capitulo 3 se estudiarán los lenguajes de programación.

g) Puesta a punto final del programa

El programa se considera terminado cuando se han finalizado pruebas, y ensayada su fiabilidad con los conjuntos de datos seleccionados y otros nuevos que se quieran elegir, y no se encuentran ya errores de ningún tipo.

h) Documentación del programa

La documentación de un programa es todo el material escrito que se ha ido produciendo simultáneamente a la elaboración del programa. La puesta a punto final del programa debe ir aparejada siempre con la documentación del mismo.

La documentación de un programa debe ser de dos tipos: documentación para personal informático y documentación de usuario.

h.1) La documentación para el personal informático.

Debe incluir toda la información técnica del programa o conjunto de programas que conforman una aplicación informática. A su vez esta documentación puede ser interna y externa. La documentación interna es la que se incluye como comentarios en los listados de los programas y la documentación externa es la ajena al listado del programa en si. La información contenida en la documentación debe ser la necesaria para la comprensión del programa y para su mantenimiento y puesta al día caso de que así se requiera.

Una buena documentación debería incluir como mínimo los siguientes elementos:

- Algoritmos y diagramas de flujo de los diferentes módulos de aplicación así como las relaciones entre ellos.
- Listado del programa/s de la aplicación.
- Definición de variables y ficheros de cada módulo, con indicación de los que son comunes a los diversos módulos.

h.2) Documentación para el usuario.

Debe incluir una descripción general de las tareas que realiza el programa, así como la descripción detallada de las instrucciones que sean necesarias para su instalación, puesta en marcha y funcionamiento; así como consejos, recomendaciones de uso, explicación de los mensajes de errores y modo de solucionarlos, entrada y salida de datos, menús de opciones, etc.

2.2 DIAGRAMAS DE FLUJO

Como ya se ha comentado, el planteamiento de un problema que pueda llegar a una solución, requiere la aplicación de una lógica que evolucione secuencialmente.

La resolución de todo problema exige tres grandes elementos: datos del problema, resultados solicitados y algoritmo de resolución

Los datos del problema son información de partida y sobre la que normalmente no se puede actuar con la excepción de su manipulación correcta. Los resultados constituyen la información de salida y estarán íntimamente relacionados con la información de entrada. El algoritmo de resolución es el conjunto de operaciones (matemáticas, lógicas, etc.) o manipulaciones que se deben realizar con los datos para llegar a la obtención de resultados.

Los algoritmos se suelen representar en forma narrativa, pero cuando tienen su aplicación más directa es cuando se convierten en diagramas o gráficos de programación, y son la representación gráfica de la solución del problema que se desea mecanizar.

Un diagrama de programación es la representación gráfica de los procedimientos y la secuencia u orden en que deben ejecutarse; en resumen la representación gráfica de la solución de un problema o de un procedimiento.

Se pueden considerar tres tipos fundamentales de diagramas de programación, conocidos también como diagramas de flujo u organigramas:

a) Diagramas del sistema o de configuración

Son diagramas destinados a describir el flujo de información entre los distintos soportes físicos de un sistema informático. Reflejan las operaciones normales para el desarrollo del proceso, que realizan los componentes utilizados en un programa.

b) Diagramas de macroprocesos o bloques

Representan la estructura en los módulos o bloques que se han realizado del problema a resolver. Incluye también el flujo entre los diversos módulos, así como el orden de ejecución de los mismos. Estos diagramas están relacionados con el proceso.

c) Diagramas de detalle u ordinograma

Son las órdenes en secuencia que se deben dar a la máquina para la resolución del problema.

2.3 SÍMBOLOS UTILIZADOS EN LOS DIAGRAMAS.

Los diagramas que se realizan durante el desarrollo de una aplicación informática deben ser claros, concisos, esquemáticos y, especialmente, independientes del lenguaje de programación que se vaya a utilizar. Así mismo deben ser comprensibles para cualquier analista o programador que los examine, procurando no presentar excesiva complejidad.

El Instituto de Normalización americano (ANSI) (American National Standard Institute) ha diseñado un conjunto de símbolos y signos estándar que prácticamente han sido adoptados internacionalmente.

Los símbolos utilizados varían según sean de aplicación a diagramas de sistemas o diagramas de bloques y diagramas de detalle. Aunque describiremos los dos conjuntos de símbolos, este trabajo está orientado básicamente hacia los diagramas del flujo de macroprocesadores o bloques y los de detalle (ordinogramas).

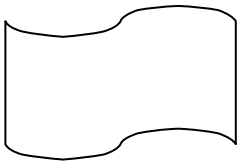
2.3.1 Diagramas de flujo de sistema

Los símbolos utilizados sirven para representar operaciones manuales y automáticas con los diferentes dispositivos del sistema informático. Se denominan también organigramas del sistema o de la máquina; no refleja las grandes funciones que debe desarrollar en forma automática el sistema de proceso de datos, pero en cambio, expresan de modo

claro el número de dispositivos de entrada y salida que deben estar disponibles para la ejecución de cada programa.

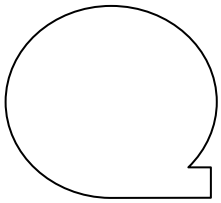
Los siguientes símbolos son usados en los diagramas de sistema:

Tarjeta perforada (lectora/perforadora de fichas)

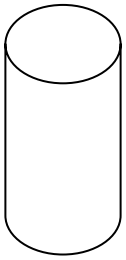


Cinta perforada (lectora/perforadora de cinta de papel)

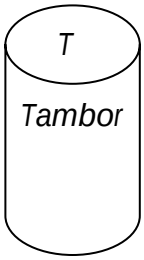
Lectora de documentos ópticos/impresora de documentos ópticos

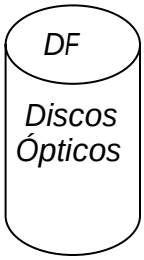


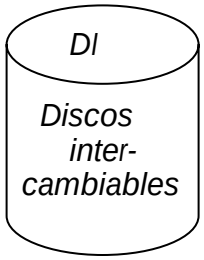
Cinta magnética (lectora/grabadora de cinta magnética)

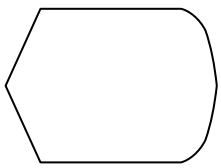


Disco magnético (lectura o grabación de un fichero soportado en disco)

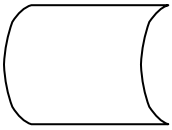




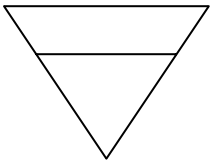




Pantalla/CRT (salida de datos a través de pantalla).



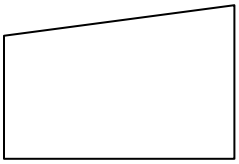
Almacenamiento de datos en línea (no-line)



Almacenamiento de datos fuera de línea (off-line)

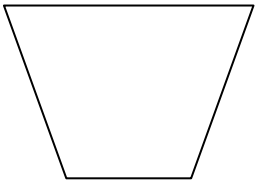
Impresora (salida de datos en forma impresa) /Trazador gráfico (plotter)

Operación o proceso, procesador.

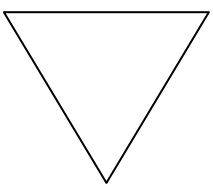


Terminal o consola (introducción manual de datos desde el teclado o consola del operador

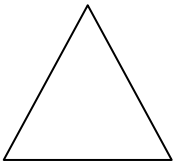
Operación mecánica con los datos fuera de línea, en una máquina auxiliar no controlada por el procesador



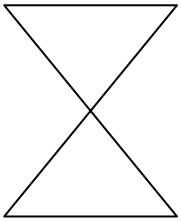
Operación manual con los datos fuera de línea, sin utilización de máquinas (guillotina de papel...)



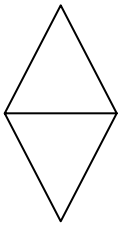
Fusión o mezcla de so o más ficheros en uno solo (Merger)



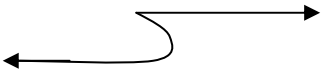
Extracción de datos de un fichero



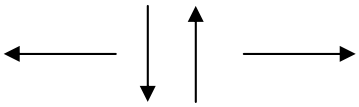
Manipulación de uno o varios ficheros fuera de línea (intercalación)



Clasificación u ordenación de los datos de un fichero.



Red de transmisión (transmisión de datos a través de sistemas de transmisión de datos).

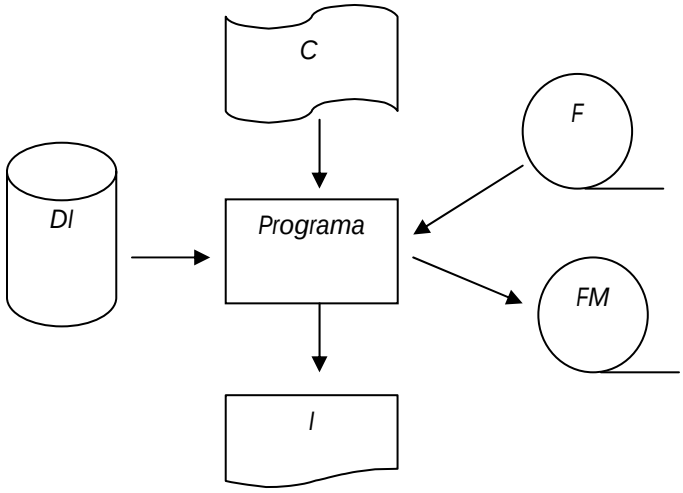


Dirección del proceso o flujo de datos.

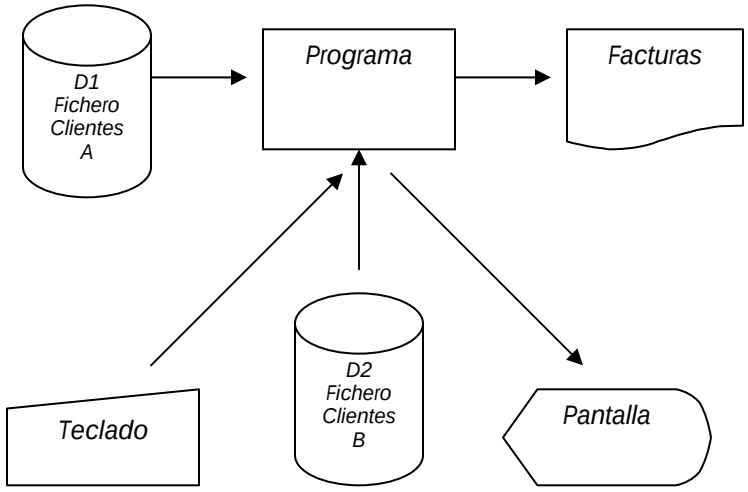
Línea conectora (indica secuencia de operaciones o transmisión de datos entre unidades o elementos de información).

Ejemplos

1. *Una aplicación realiza la puesta al día de un fichero de datos fijo F, soportado en cinta magnética, a través de los datos del fichero C, soportado en una cinta perforada. El proceso realiza una consulta a un fichero de entrada/salida (E/S) que contiene los datos en discos intercambiables. Los resultados del proceso se presentarán en: a) el fichero M con los datos fijos actualizados; b) un listado de los posibles errores detectados impresos en papel mediante impresora (se realizarán las operaciones a través de un fichero de movimientos M).*



2. *Una aplicación consiste en la emisión de facturas a partir de los datos introducidos por teclado y los datos del cliente almacenados en un fichero de disco. La factura se debe presentar en pantalla y una vez dada la conformidad, escribirla en impresora.*



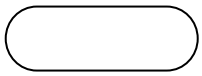
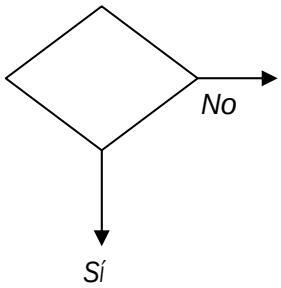
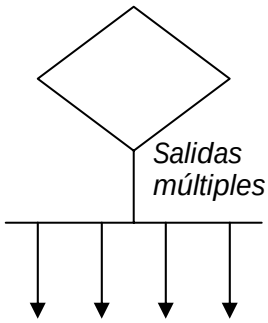
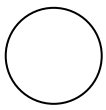
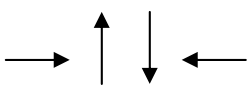
2.3.2 Diagramas de flujo de detalle/organigramas

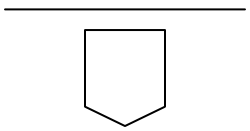
Los ordinogramas o diagramas de flujo de detalle y de macroprocesos deben mostrar las operaciones que realiza un programa, con el detalle necesario para que una vez confeccionados, se pueda realizar la etapa siguiente de la programación (codificación).

Dependiendo del nivel de lenguaje utilizado, orientado a la máquina u orientado al problema, variará el grado de detalle de los ordinogramas. Para lenguajes muy evolucionados como el COBOL, en general, es suficiente representar únicamente los grandes bloques de tratamiento: entrada, salida, proceso y decisión sin tener que detallar las operaciones elementales. En los lenguajes ensambladores o de bajo nivel, como cada instrucción simbólica se transforma en una o varias instrucciones de

lenguaje máquina, es preciso un ordinograma de detalle que muestre las operaciones elementales que debe efectuar la máquina.

Símbolos utilizados en diagramas de detalle / ordinogramas:

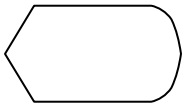
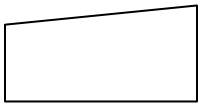
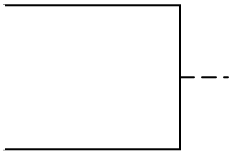
Símbolos principales	Función
	*Terminal (representa el comienzo, "inicio", y el final, "fin", de un programa. Puede representar también una parada o interrupción programada que sea necesario realizar en un programa.*
	*Entrada/Salida (cualquier tipo de operación, de introducción de datos en la memoria desde los periféricos "entrada", o registro de la información procesada en un periférico "salida").*
	*Proceso (cualquier tipo de operación definida que pueda originar cambio de valor, formato o posición de la información almacenada en memoria: operaciones aritméticas, de transferencia de datos, etcétera).*
	*Decisión (indica operaciones lógicas o de comparación entre datos –normalmente dos– y en función del resultado de la misma determina cuál de los distintos caminos alternativos del programa se debe seguir; normalmente tiene dos salidas –respuestas Sí o No– pero puede tener tres o más según los casos).*
	*Decisión múltiple (en función del resultado de la comparación se seguirá uno de los diferentes caminos de acuerdo con dicho resultado).*
	*Conector (sirve para enlazar dos partes cualesquiera de un ordinograma a través de un conector en la salida y otro conector en la entrada). Se refiere a la conexión en la misma página del programa.*
	*Indicador de dirección o línea de flujo (indica el sentido de ejecución de las operaciones).*



Línea conectora (sirve de unión entre dos símbolos).

Conector (conexión entre dos puntos del organigrama situado en páginas diferentes).

Llamada a subrutina o a un proceso predeterminado (una subrutina es un módulo independiente del programa principal, que recibe una entrada procedente de dicho programa, realiza una tarea determinada y regresa, al terminar, al programa principal).

Símbolos secundarios	*Función*
	*Pantalla (se utiliza en ocasiones en lugar del símbolo de E/S).*
	*Impresora (se utiliza en ocasiones en lugar del símbolo de E/S).*
	*Teclado (se utiliza en ocasiones en lugar del símbolo de E/S).*
	*Comentarios (se utiliza para añadir comentarios clasificadores a otros símbolos del diagrama de flujo. Se pueden dibujar a cualquier lado del símbolo).*

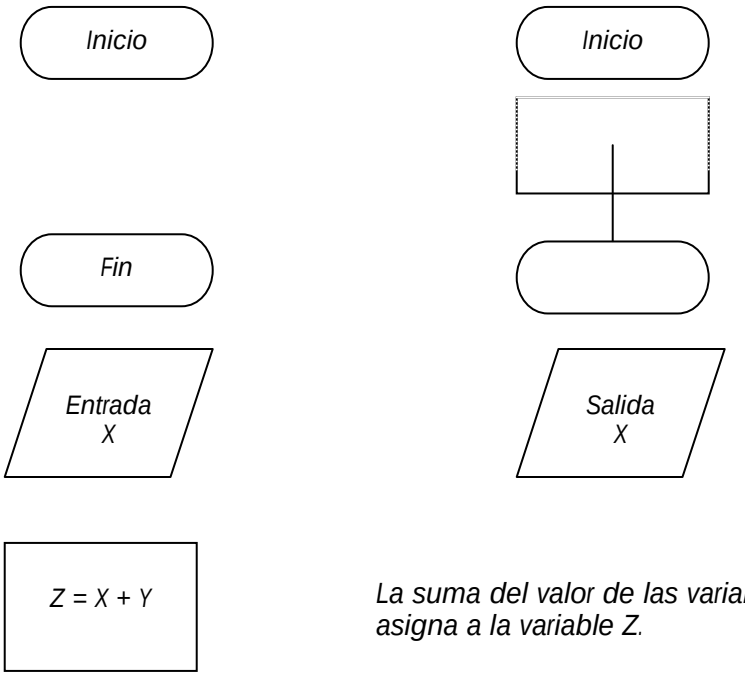
Es conveniente que los ordinogramas y los organigramas –cuando así lo exijan– utilicen notaciones normalizadas para operaciones de cualquier tipo.

Relación de símbolos utilizados en diagramas de detalle / ordinogramas:

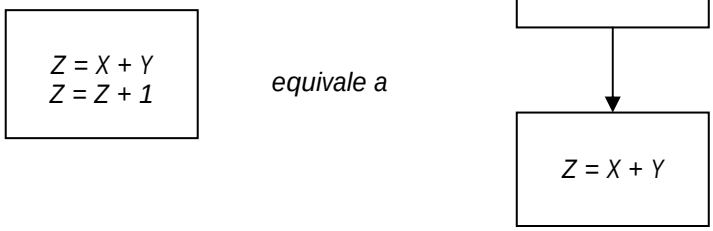
Operaciones aritméticas	*Significado*
→	*Movimiento de unas posiciones de memoria a otras o cambios en*
←	*campos de información.*
+	*Suma*
-	*Resta*
*	*Multiplicación*
/	*División*
↑ *o bien* ^	*Exponenciación*
Operaciones de relación	
<	*Menor que*
=	*Igual*
>	*Mayor que*
≤	*Menor o igual que*
≥	*Mayor o igual que*
<>	*Diferente (menor o mayor que)*

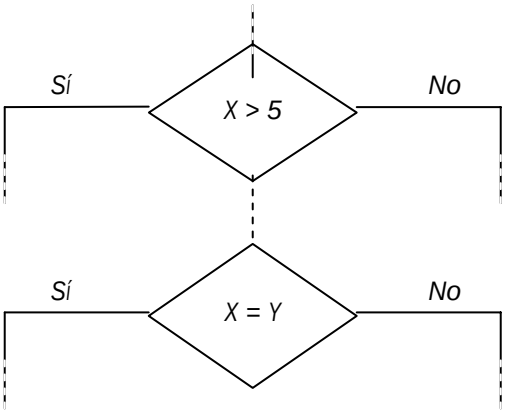
□ *	*No menor que (mayor o igual)*
≠ *	*No igual que (diferente)*
□ *	*No mayor que (menor o igual)*
:: y <>	*Símbolo de comparación*

Ejemplos de símbolos en ordinogramas:



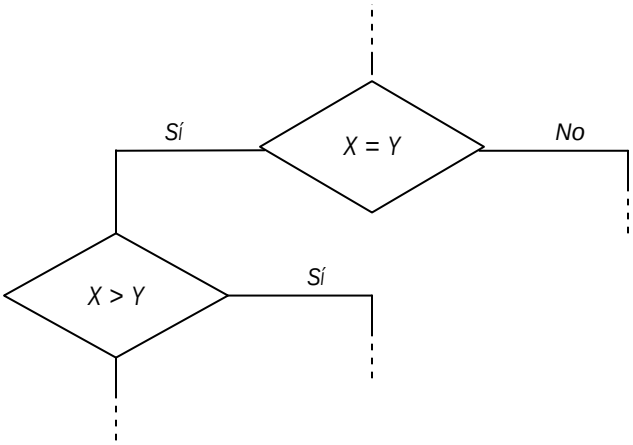
La suma del valor de las variables X e Y se asigna a la variable Z.





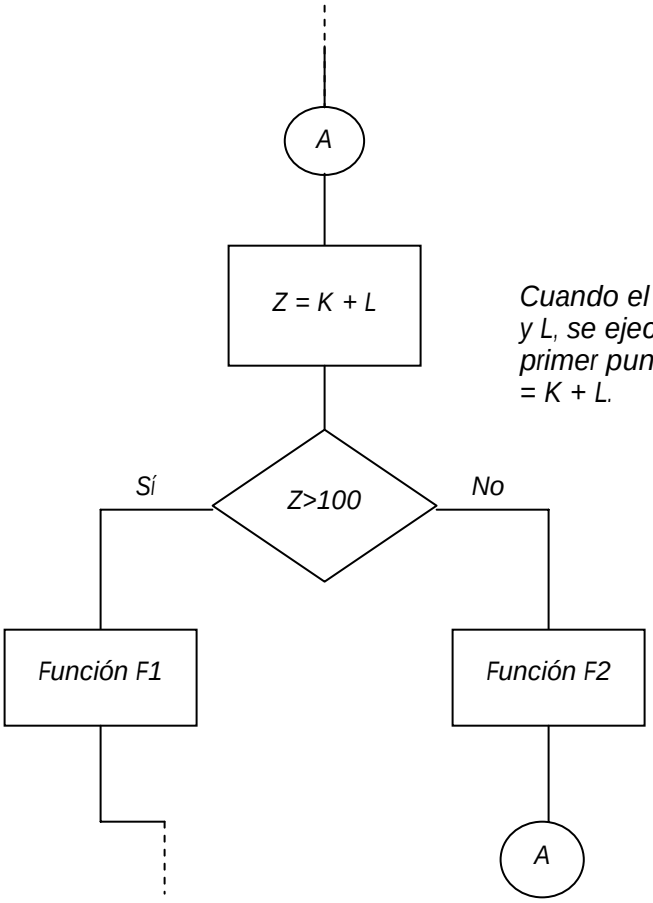
Si el valor de X es mayor que 5 seguir el camino rotulado Sí, en caso contrario el camino No.

Si el valor de X es igual a Y seguir el camino rotulado Sí, en caso contrario el camino No.

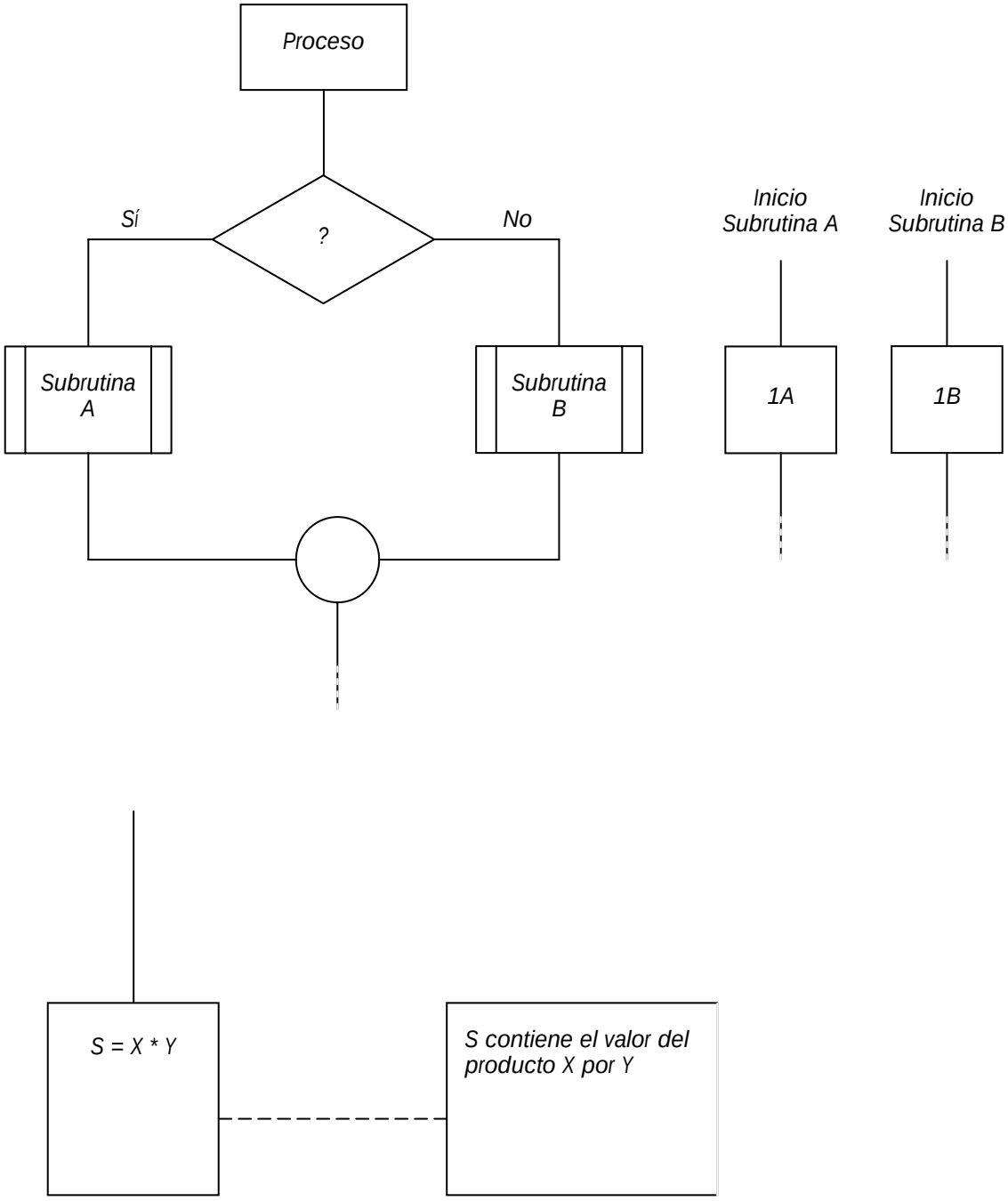


Si el valor de X es igual que el de Y seguir el camino rotulado Sí, en caso contrario el camino No.

Si el valor de X es mayor que el de Y seguir el camino rotulado Sí, en caso contrario el camino…



Cuando el valor de Z sea igual a la suma de los valores de K y L, se ejecuta la función F2 y a continuación se retorna al primer punto conector A, o sea, realización de la operación Z = K + L.

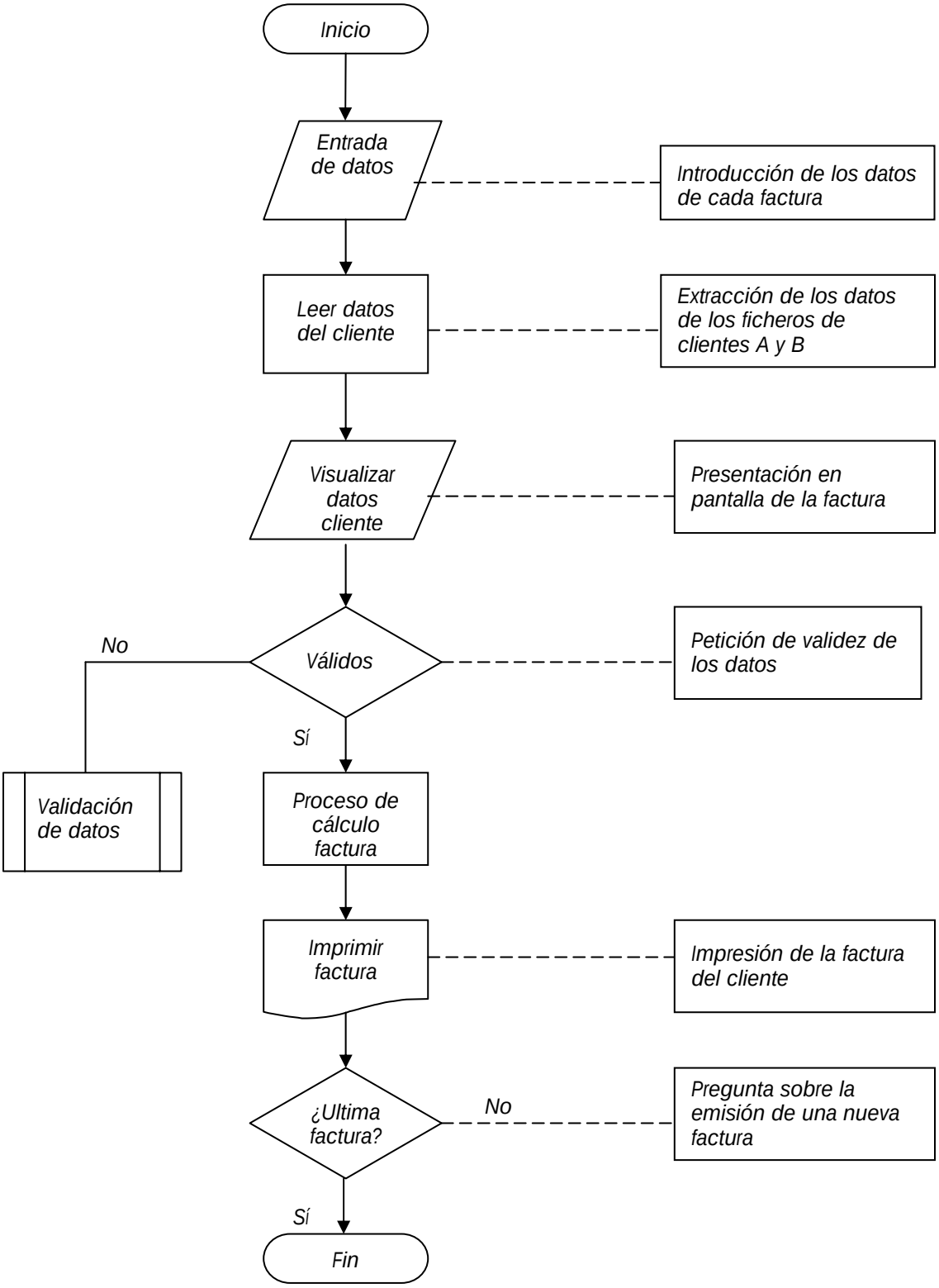


Ejemplo.- *Diagrama de flujo de macroprocesos o bloques.*

Emisión de facturas del ejemplo 1

El problema se subdivide en módulos de menor entidad (entrada de datos por teclado, lectura de datos del fichero de clientes, visualizaciones en pantalla, validez de los datos, impresión de facturas, etc.)

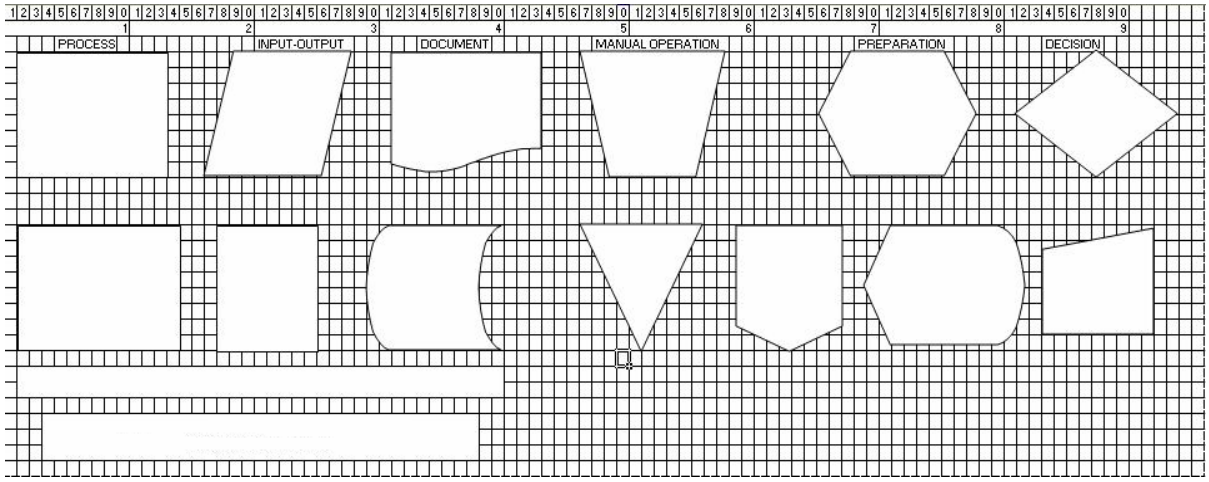
El diagrama de flujo de detalle podría ser el siguiente:



2.3.3 Plantillas y hojas de codificación

En el comercio existen a la venta plantillas para la realización de diagramas de flujo.

Generalmente son piezas rectangulares planas y transparentes, en las cuales figuran recortados los símbolos vistos en los párrafos anteriores. El uso de las plantillas es similar al de cualquier otra plantilla de dibujo, es decir, basta con deslizar el lápiz o rotulador por el contorno del símbolo correspondiente para dibujar dicho símbolo.



Plantilla para confección de diagramas de flujo.

Aunque la confección de formatos de pantalla o de salida en impresora se pueden realizar en cualquier folio de papel (cuadriculado o no), se dispone normalmente en los centros de proceso de datos de formularios que facilitan los diseños así como hojas de codificación, donde de suelen representar normalmente las posiciones físicas de los caracteres en pantalla (25 filas por 80 columnas, es decir 25 x 80 caracteres) de modo que facilitarán considerablemente el diseño de formatos de texto en pantalla o de salida en impresora.

2.4 DIAGRAMAS DE FLUJO ESTRUCTURADOS

Aunque en los capítulos siguientes se profundizará en los conceptos de diagramas de flujo estructurados, en este apartado realizaremos un pequeño anticipo de los mismos, al objeto de que el lector se introduzca de modo general en el importante concepto de estructuración de datos, diagramas y programas.

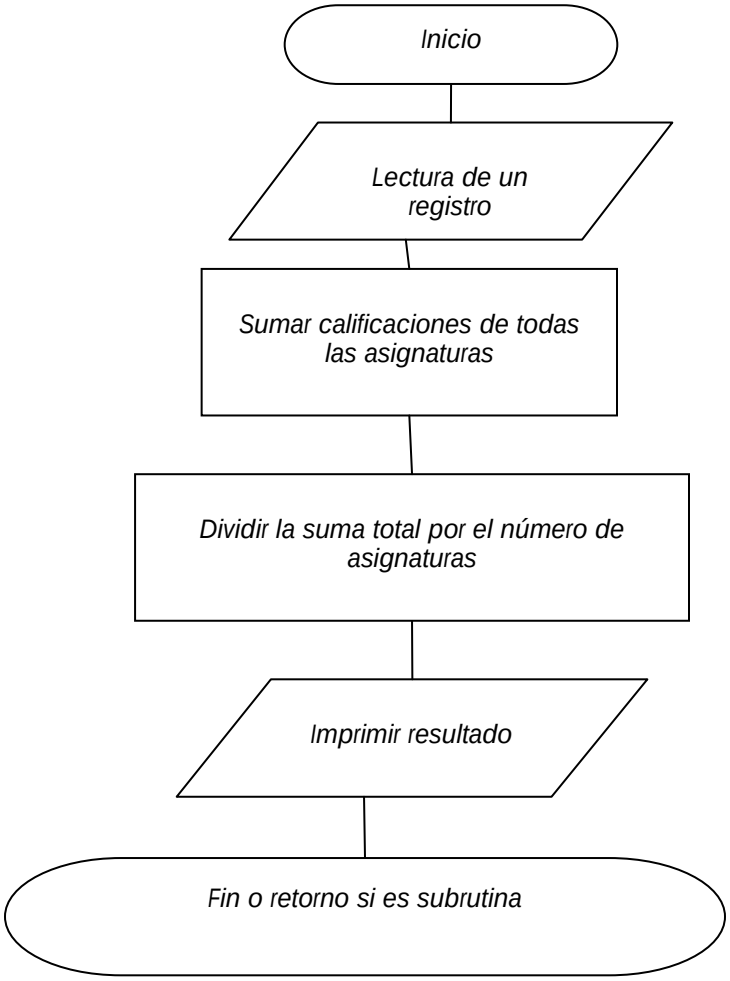
En general, un ordinograma contará con las siguientes fases:

1. *Inicio*
2. *Entrada de datos*
3. *Proceso*
4. *Salida de datos*

5. Fin

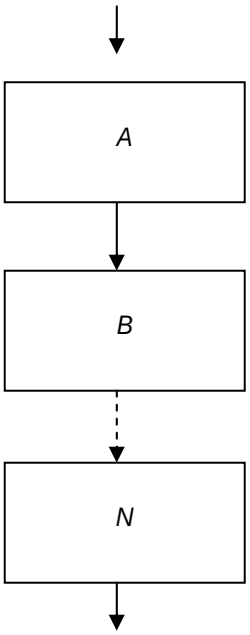
Aunque las cinco partes existen prácticamente siempre, el orden de las fases 2, 3 y 4 puede variar e incluso confundirse unas fases con otras.

Así por ejemplo, si se desea obtener la nota (calificación) media de un alumno correspondiente a las diferentes asignaturas de su curso escolar, cuyos datos escolares (registros o fichas) están grabados en un fichero en disco, un ordinograma podría ser:

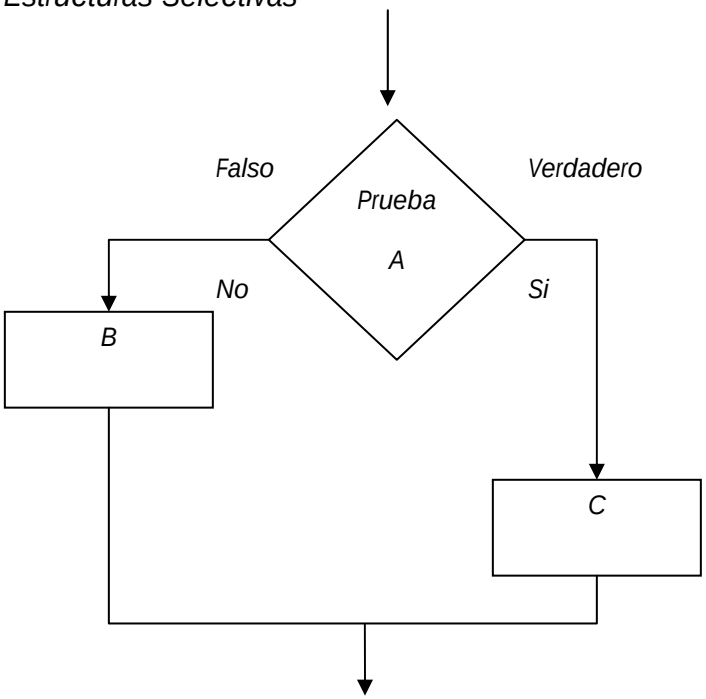


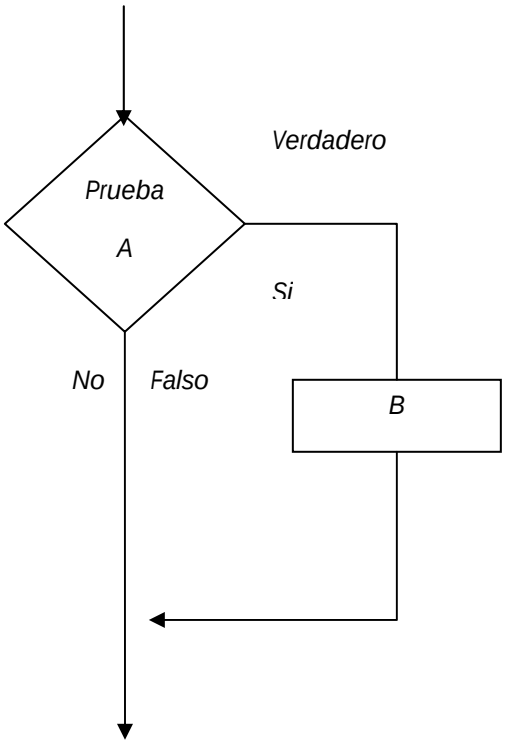
Las estructuras lógicas básicas necesarias para confeccionar un programa se reducen a tres: secuenciales, selectivas y repetitivas.

Estructuras Secuenciales



Estructuras Selectivas

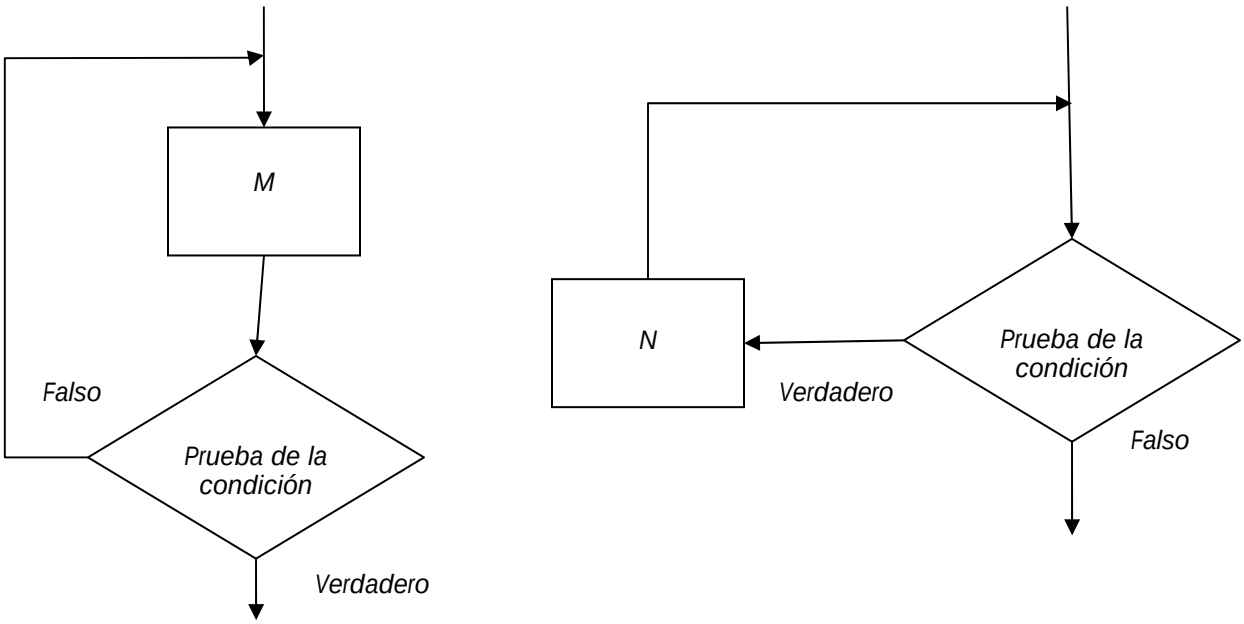




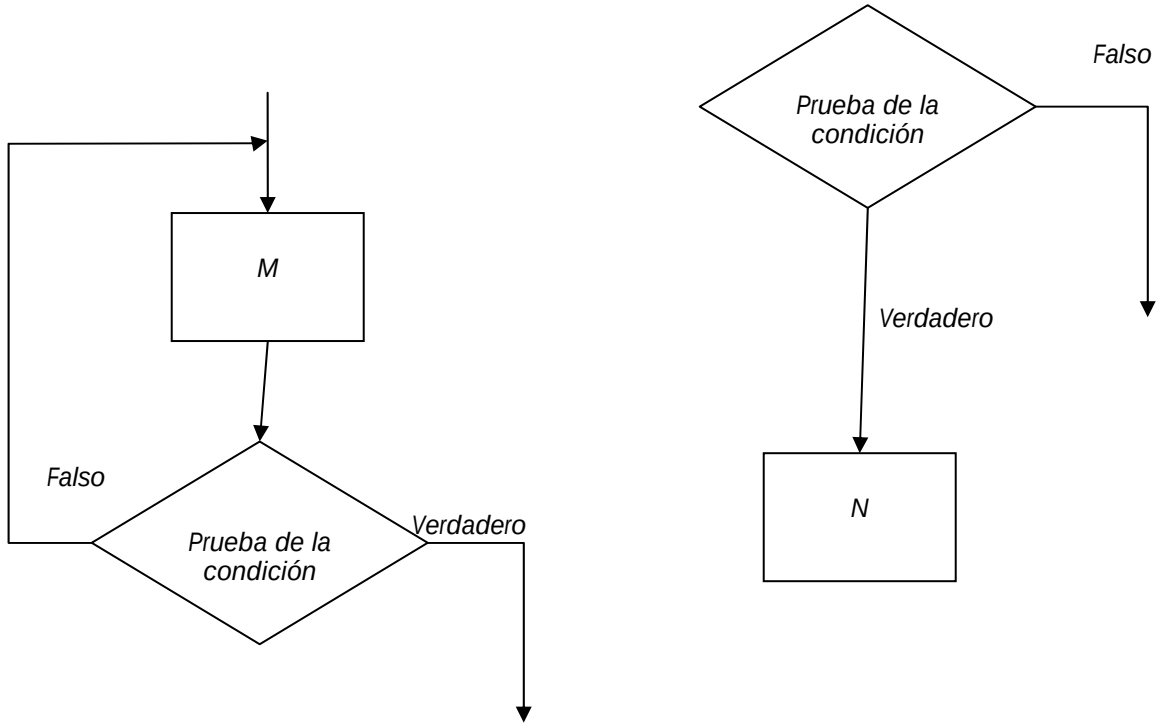
Constan esencialmente de sucesivos pasos, uno detrás de otro.

Requieren una prueba o comparación de ciertas condiciones seguidas de rutas alternativas en el programa.

Estructuras Repetitivas

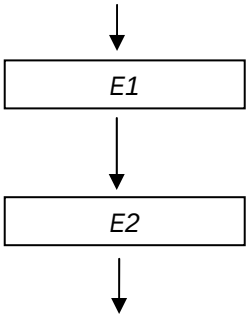


O bien representadas de la siguiente forma:

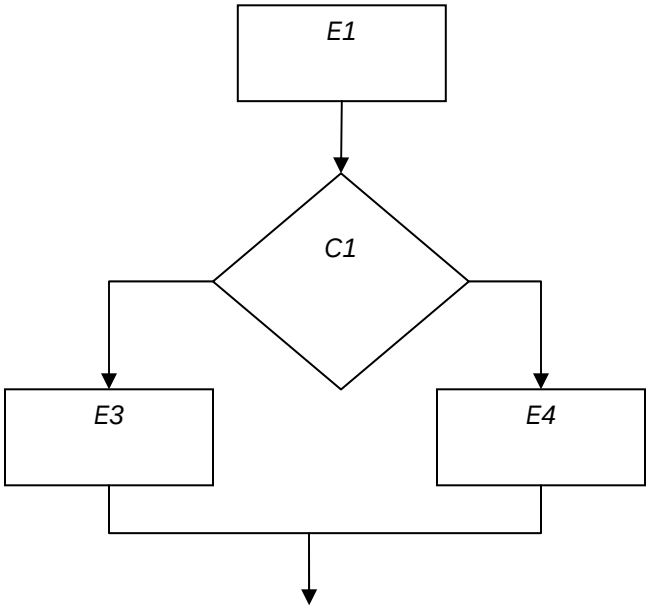


Cada una de las estructuras tiene una sola flecha de entrada y una sola flecha de salida y cada una de ella es legible de arriba hacia abajo (diseño top-down); por consiguiente estas estructuras llamadas básicas o fundamentales pueden componerse sustituyendo cada bloque básico por una de dichas estructuras.

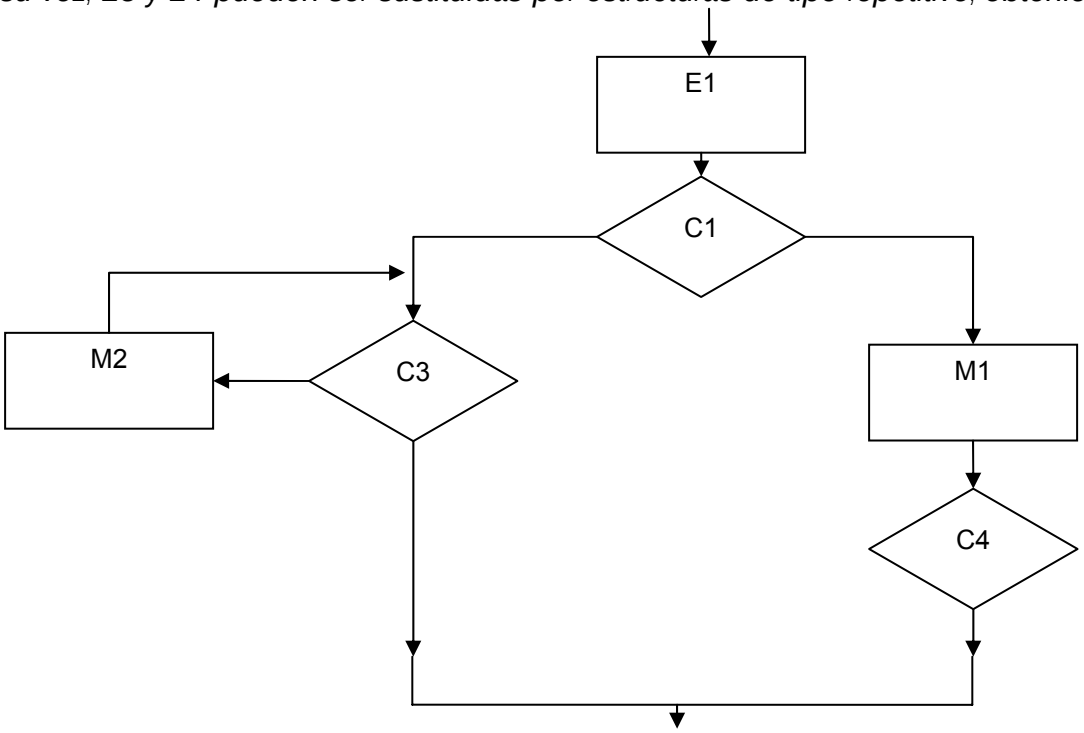
Así por ejemplo, en la estructura secuencial.



Si se sustituye E2 por una estructura del tipo selectivo, se tendrá:



A su vez, E3 y E4 pueden ser sustituidas por estructuras de tipo repetitivo, obteniéndose:



A los diagramas de flujo construidos de esta forma se les llama diagramas de flujos estructurados o simplemente diagramas estructurados. Estas técnicas forman parte del concepto denominado programación estructurada y que hoy en día tiene una gran aceptación.

2.5 REGLAS PARA LA CONSTRUCCIÓN DE DIAGRAMAS DE FLUJO

La finalidad de un diagrama de flujo es describir de modo gráfico la estructura lógica del programa y que posteriormente permita una fácil codificación en lenguaje específico.

El proceso para la construcción de un diagrama de flujo no supone un método rígido y de hecho la mayoría de los programadores terminan acuñándose el propio. Sin embargo se pueden enunciar unas reglas de tipo general, útiles para cualquier nivel

La elaboración de un diagrama de flujo debe comenzar con un diagrama de bloques que represente las relaciones fundamentales y posteriormente ir descomponiendo las operaciones fundamentales en otras más sencillas, pero sin tratar de que los ordinogramas se correspondan instrucción a instrucción con el funcionamiento real del programa, ya que en ese caso obtendríamos unos diagramas de flujo excesivamente recargados y no serían útiles.

Reglas prácticas

1. Todo diagrama debe tener un principio (inicio) y un fin, al objeto de que pueda ser utilizado como submódulo de otro módulo de nivel superior.

2. Las líneas de conexión o de flujo deben ser siempre rectas y, si es posible, que sean sólo verticales y horizontales (no cruzarse, ni inclinadas); para conseguir lo anterior se debe recurrir a conectores, numerados convenientemente y sólo en los casos estrictamente necesarios.

3. Las líneas que enlazan los símbolos entre sí deben estar todas conectadas. Cada línea o flecha debe entrar en un bloque, en un símbolo de decisión, terminar en «Fin» o unirse a otra flecha.

4. Se deben dibujar todos los símbolos de modo que se pueda seguir el proceso visualmente de arriba abajo (diseño «top-down») y de izquierda a derecha.

5. Realizar un gráfico claro y equilibrado, procurando que el flujo central del diagrama sea la parte central de la hoja de papel.

6. Evitar la utilización de terminología especifica de un lenguaje de programación o máquina, sobre todo en las expresiones donde se tiene tendencia natural a ello.

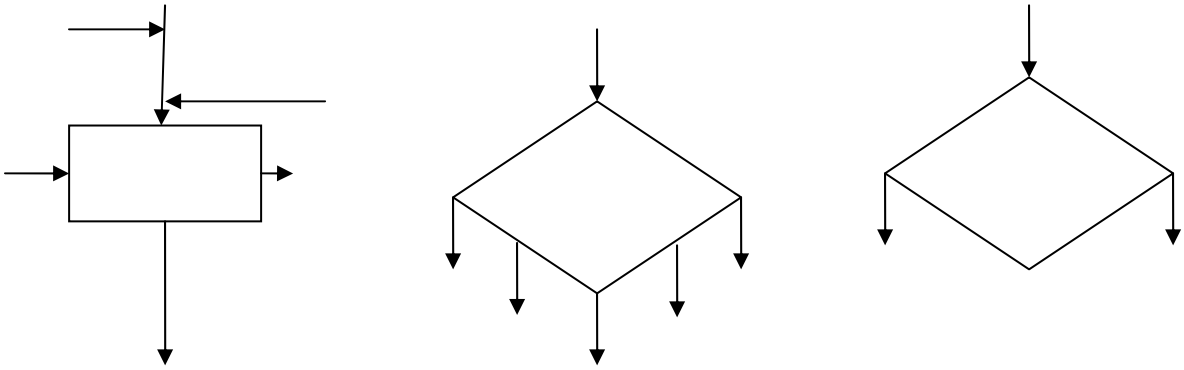
7. Se debe dejar un bloque o dos de proceso libres al comienzo del diagrama, para reservar posiciones de memoria para variables, acumuladores, inicialización de subíndices de listas y tablas (arrays) conmutadores (switch), etc.

8. Indicar con comentarios al margen o mediante el símbolo gráfico comentarios, las variables utilizadas y su descripción, procurando no abusar de su uso. Diferenciar las variables propias del proceso de las pseudovariables o variables ficticias (contadores, conmutadores o interruptores).

9. *En las operaciones lógicas recurrir preferentemente a la lógica positiva que a la lógica negativa (es siempre más claro «si A = B» que «si no es A <> B»)*

10. *A cada bloque o símbolo se accede por arriba y/o por la izquierda y se sale por abajo y/o por la derecha.*

    Las entradas pueden ser varias pero la salida es única excepto en los casos de símbolos de decisión.



11. *Realizar todas las anotaciones o comentarios marginales al diagrama para que éste sea comprensible no sólo por la persona que lo ha elaborado sino también por cualquier persona ajena al mismo, sobre todo con el paso del tiempo y para cuando se necesite una actualización o modificación del diagrama.*

12. *Siempre que sea posible, es conveniente que el diagrama no sobrepase una página; si no es posible, numerar adecuadamente las hojas del diagrama y utilizar los correspondientes conectores de páginas que indiquen sin dudas la dirección correcta del flujo (de donde viene y a donde se dirige).*

2.5.1 Comprobación de diagramas

Terminado el diagrama de flujo, se deben tomar un conjunto de datos significativos y comenzar la lectura del mismo en el orden arriba abajo/izquierda derecha y seguir paso a paso todos los símbolos, con sus operaciones correspondientes, introduciendo los datos tomados inicialmente en los momentos oportunos y ver como responde el diagrama de flujo y si los resultados obtenidos son correctos y coherentes. No se debe pasar a la fase de codificación, mientras que el diagrama de flujo no haya sido comprobado su funcionamiento con datos significativos.

Tras la ejecución de un programa por la computadora, los ordinogramas deben ayudar en el proceso de depuración (detección, localización) y corrección de errores).

2.5.2 Ventajas e inconvenientes de los diagramas de flujo

En el análisis de la programación los beneficios que se pueden derivar del uso de los diagramas de flujo se pueden sintetizar en los siguientes:

1. Rápida comprensión de las relaciones.
2. Análisis efectivo de las diferentes secciones del programa.
3. Los diagramas de flujo pueden utilizarse como modelos de trabajo en el diseño de nuevos programas y sistemas.
4. Comunicación con el usuario.
5. Documentación adecuada de los programas.
6. Codificación eficaz de los programas.
7. Depuración y pruebas ordenadas de programas.

Las limitaciones o inconvenientes se pueden englobar en:

- Los diagramas complejos y detallados suelen ser laboriosos en su planteamiento y dibujo.
- Las acciones a seguir tras la salida de un símbolo de decisión, pueden ser difíciles de seguir si existen diferentes caminos.
- No existen normas fijas para la elaboración de los diagramas de flujo que permitan incluir todos los detalles que el usuario desea introducir.

2.6 PSEUDO-CODIGO

Otra herramienta muy útil en el análisis de programación es el pseudo-código. Pseudo o seudo, significa «falso», «imitación» y código se refiere a las instrucciones escritas en un lenguaje de programación; pseudo-código no es realmente un código sino una imitación y una versión abreviada de instrucciones reales para las computadoras.

Las tres herramientas que utilizan los programadores son: diagramas de flujo, tablas de decisión y pseudo-códigos. A lo largo de esta obra se tratarán las tres herramientas detenidamente. Aunque se utilizan las tres son sin duda los diagramas de flujo y los pseudo-códigos los que más utilizan los programadores.

El pseudo-código es una técnica para expresar en lenguaje natural la lógica de un programa, es decir, su flujo de control. El pseudo-código no es un lenguaje de programación sino un modo de plantear un proceso de forma que su traducción a un lenguaje de alto nivel sea sencillo para un programador. La fase de confección de pseudo-códigos es inmediatamente anterior a su codificación en el lenguaje de programación elegido. En la actualidad el pseudo-código es una técnica muy utilizada sobre todo en la programación de lenguajes estructurados como Pascal, o bien utilizando técnicas de programación estructurada en otros lenguajes.

Los pseudo-códigos utilizan palabras clave como **DO** (Hacer), **IF-THEN-ELSE** (Si-entonces-sino), **ENDIF** (Fin de si), **REPEAT-UNTIL** (Repetir hasta), **REPEAT-WHILE** (Repetir-

mientras) o bien **DO-UNTIL** *(Hacer-hasta), etc. El ejemplo del programa de cálculo de la calificación media de un alumno en pseudo-código podría ser:*

Lectura *registro o ficha de estudiante*

Mientras *no se encuentre «fin de fichero»*
Sumar notas
Dividir la suma total entre número de asignaturas
Impresión del registro
Fin-mientras

2.7 DIAGRAMA DE NASSI/SHNEIDERMAN (N/S)

Un diagrama N-S es como un diagrama de flujo con las flechas omitidas y cajas o bloques contiguos. Las acciones sucesivas se escriben en cajas sucesivas. Como en un diagrama de flujo, diferentes acciones pueden ser escritas en una caja. La diferencia entre esta forma y las otras dos formas (diagrama de flujo y algoritmo) parece trivial. Sin embargo, esto es debido a que los algoritmos son bastantes sencillos.

Este sistema de representación permite tener una visión mucho más estructurada de ellos y por consiguiente mayor facilidad al introducirlos al lenguaje de una computadora.

Estos diagramas tratan de optimizar los programas, ya que permiten gran flexibilidad, al permitir fáciles correcciones, modificaciones o ampliaciones al diagrama original.

El diagrama se lee de arriba-abajo. Cada bloque ejecuta una operación específica que se puede documentar o describir con la precisión que se desee.

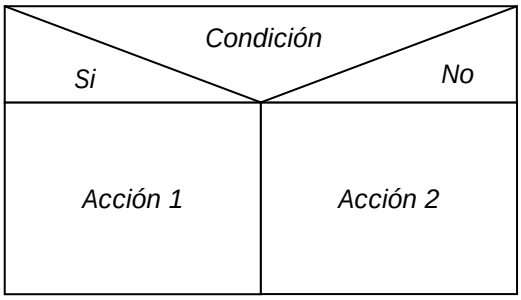
Los diagramas de las estructuras lógicas son los siguientes:

Secuencial

Acción A
Acción B
. . .
Acción N

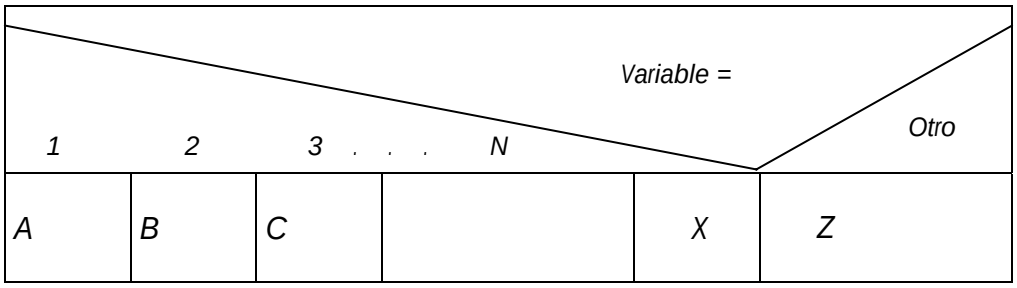
Alternativa

Selección doble: Si la condición es verdadera se ejecuta la acción 1 y si es falsa la acción 2.



If
Then
Else

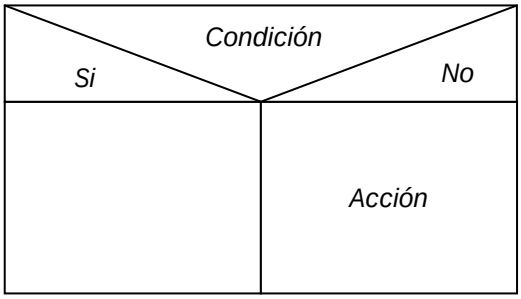
Selección múltiple: dependiendo del valor de la variable se ejecuta el proceso especificado y en caso de que la variable no esté en el rango 1 a n se ejecuta el bucle Z (vacío o no).



Case

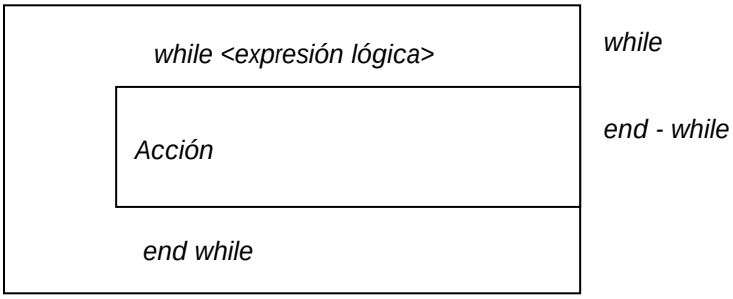
end-case

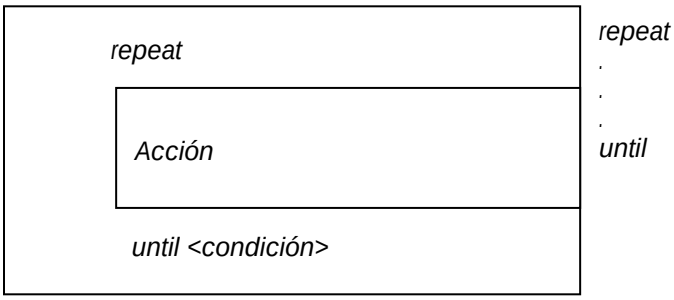
Selección simple: Si la condición es verdadera se ejecuta la acción y si es falsa las acciones siguientes:



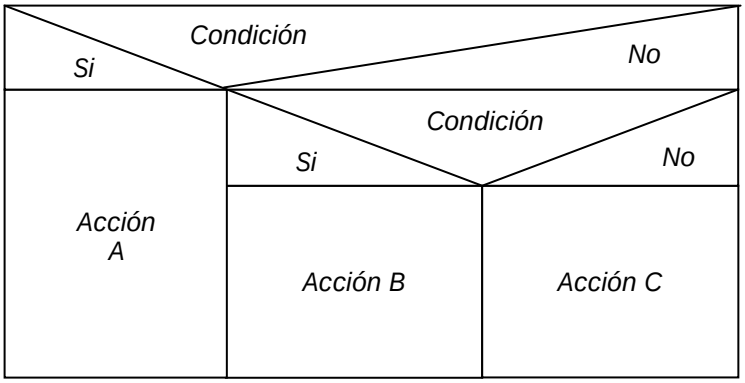
If
.
.
.
then

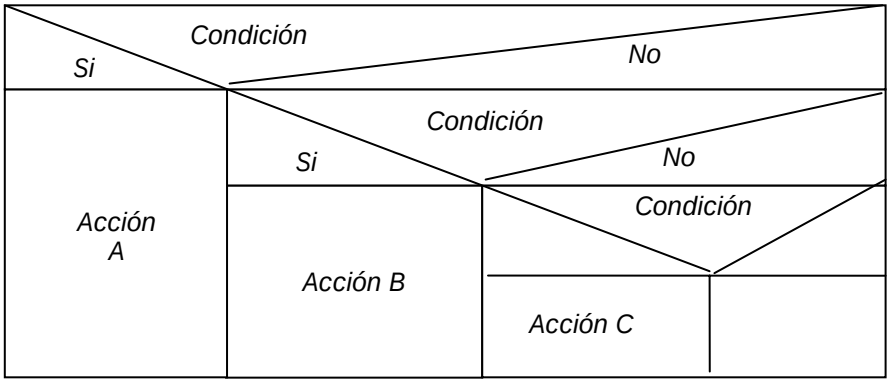
Repetitivas

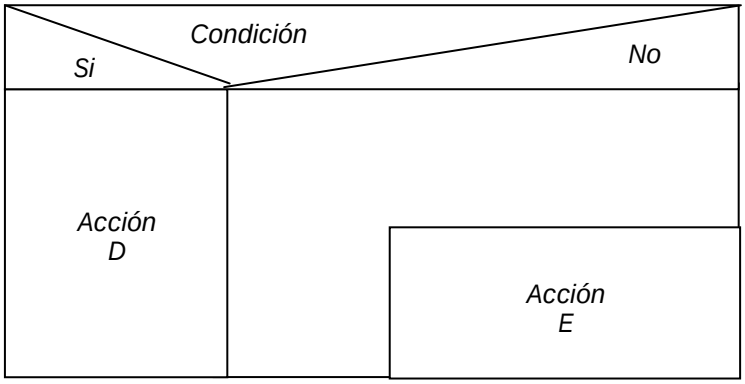




Estructuras más complejas:







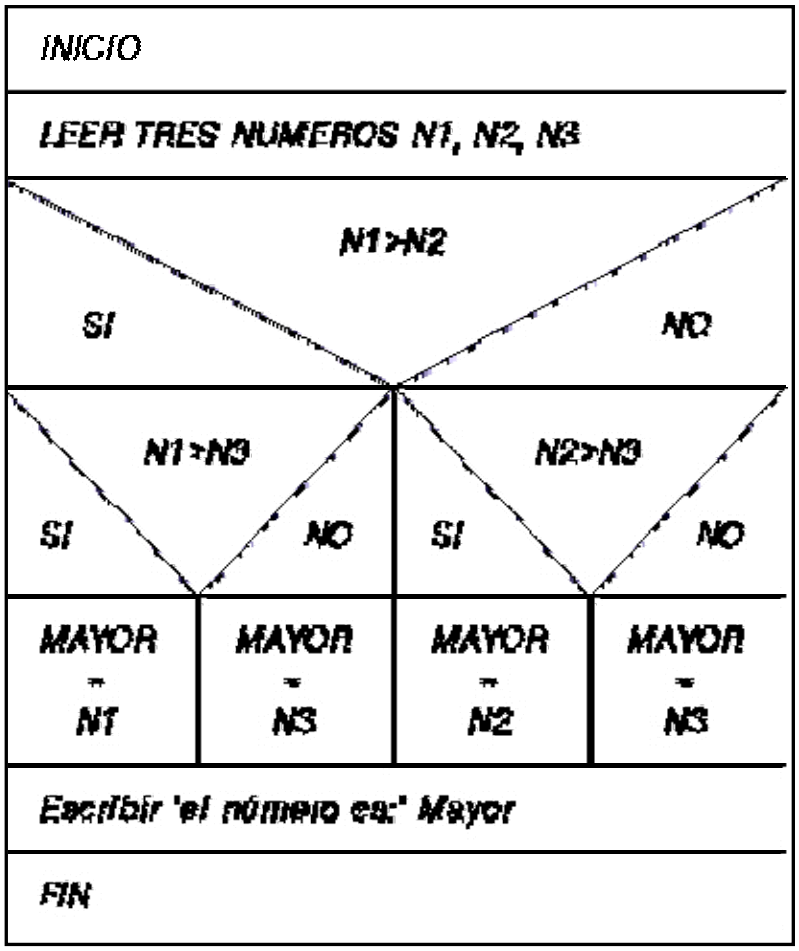
Ejemplo

Diseñar un algoritmo y el diagrama N-S correspondiente que resuelva el siguiente problema: Leer 3 números por teclado, hallar el mayor y presentarlo en pantalla.

Solución:

```
Inicio

    Leer 3 números N1, N2, N3

Si N1>N2

    Entonces

        Si N1>N3

            Entonces    MAYOR = N1

            Sino        MAYOR = N3

        Fin _ si
    Sino

        Si N2>N3
            Entonces    MAYOR = N2

            Sino        MAYOR = N3

        fin_si
fin_si

Escribir "el número mayor es": MAYOR

Fin
```

DIAGRAMA N-S (A CONTINUACIÓN)



2.8 MODULARIZACION

El sistema más idóneo para la resolución de un problema es la descomposición en módulos más pequeños, que serán objeto de estudio y desarrollo independiente para posteriormente enlazar en un solo programa.

La modularización de un problema debe seguir un proceso arriba-abajo («top-down»), que consiste en abordar el problema a nivel general y posteriormente hacer sucesivas modularizaciones que realizarán tareas de mayor detalle. Las normas a tener en cuenta es que el proceso de modularización incluirá: descripción de los módulos, relaciones entre ellos, entradas y salidas a cada módulo.

CAPITULO 3

ESTRUCTURA GENERAL DE UN PROGRAMA

3.1 CONCEPTO DE PROGRAMA

La programación de computadoras es el proceso de planificar una secuencia de instrucciones que ha de seguir una computadora. Un programa es la secuencia de instrucciones que indica las acciones que ha de ejecutar la computadora.

El concepto de programa almacenado en la memoria fue ideado por John Von Neumann en 1946 y desde entonces todas las computadoras utilizan esta característica que las hace más flexibles y permite realizar cambios de un programa a otro.

En general, el desarrollo de un programa para la resolución de un problema tendrá en líneas generales las fases siguientes:

Análisis: *Definición del problema.*

Algoritmo: *Desarrollo de la secuencia lógica de pasos para la resolución del problema.*

Prueba de algoritmo: *Seguir los pasos del algoritmo y ver si resuelven realmente el problema.*

Codificación: *Conversión del algoritmo en un programa escribiéndolo en un lenguaje de programación.*

Edición, ejecución y prueba: *Introducir el programa en la memoria, ejecutarlo y probar sus resultados, corrigiendo los errores hasta su puesta a punto final.*

Uso y mantenimiento: *Manejo y actualización del programa.*

En el apartado siguiente ampliaremos los conceptos anteriores.

Uno de los objetivos fundamentales de este trabajo es la programación y ello en resumen significa realizar por el programador las fases de la Figura 3.1.

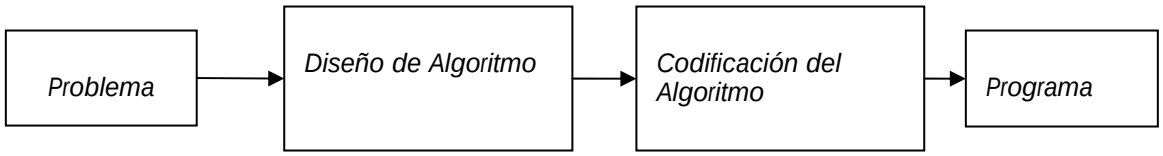


Figura 3.1. *Fases de la Programación.*

No se debe pasar nunca a la codificación o escritura del programa sin haber realizado las fases de análisis del problema y resolución del algoritmo. Muchos estudiantes –incluso profesionales– tienden a eliminar la fase de diseño del algoritmo y ello supone un grave error, pues si bien en ocasiones puede ahorrar tiempo y producir un resultado satisfactorio, a la larga puede producir errores y aumentar el tiempo de la fase de ejecución y puesta a punto del programa.

3.1.1 Desarrollo de un programa

Dependiendo de la naturaleza del programa a resolver se puede considerar el equipo de desarrollo de un programa, y así éste puede constar de una sola persona o un equipo de personas dirigidos por un programador jefe del equipo. En cualquier caso, la descripción general de cada programa incluirá al menos los siguientes conceptos:

- *Tipo de lenguaje de programación (COBOL, Pascal, BASIC, etc.)*
- *Descripción del programa con indicación de las tareas a realizar y del algoritmo de resolución.*
- *Frecuencia de procesamiento (diaria, semanal, en línea, etc.)*
- *Entradas y salidas del programa.*
- *Especificaciones detalladas de cálculos, tablas, etc.*
- *Limitaciones y restricciones (orden de entrada/salida de datos, tiempos de respuesta, etc.).*

Una vez que tales conceptos se conocen, es el momento de escribir los programas. Las personas que escriben y depuran los programas se llaman **programadores**.

Los pasos que exige el desarrollo de un programa –como se ha descrito anteriormente– se suelen sintetizar en los siete siguientes:

1. *Planificación del problema: descripción y análisis.*
2. *Desarrollo de las especificaciones del programa.*
3. *Codificación del programa.*
4. *Depuración del programa.*
5. *Verificación del programa.*
6. *Documentación del programa.*
7. *Mantenimiento del programa.*

Planificación del programa

Un equipo de analistas y usuarios debe decidirse exactamente lo que el programa debe hacer, qué datos debe procesar y qué información producirá.

Desarrollo de las especificaciones del programa

El programador especifica las funciones del procesamiento de los datos que el programa debe ejecutar. Las relaciones entre las funciones se establecen en una serie de diagramas de flujo.

Codificación del programa

El programador escribe el código fuente del programa. Tal código fuente consta de los pasos del programa descritos en un lenguaje de computadora. En la computadora el código fuente se traduce en un programa que la computadora puede ejecutar.

Depuración del programa

El programador ejecuta el programa para detectar y corregir errores. A esta acción se le denomina depurar el programa.

Verificación del programa

El programador comprueba el programa para asegurarse que produce la información requerida. Durante esta fase se podría necesitar modificar el programa.

Documentación del programa

El programador describe el funcionamiento y uso del programa en una documentación técnica y de usuario.

Mantenimiento del programa

Si el tipo de información requerida necesitase cambios, el programa puede tener que ser modificado. Así mismo, los usuarios del programa pueden descubrir errores o introducir a su vez cambios que la experiencia les dicte, y por consiguiente debe modificarse el programa. En resumen, el programador debe mantener el programa, corrigiendo cualquier error o introduciendo las necesarias modificaciones para que el programa continúe durante todo el tiempo de su vida activa, siendo útil para cumplir las necesidades del usuario.

3.2 LENGUAJES DE PROGRAMACION

Al igual que los lenguajes humanos, tales como el inglés o el español, los lenguajes de programación poseen una estructura (gramática o sintaxis) y un significado (semántica). La gramática española trata de los diferentes modos (reglas) en que pueden ser combinados los diferentes tipos de palabras para formar sentencias o frases aceptables en español. La gramática de Pascal se refiere a las diferentes reglas en que pueden combinarse las sentencias (instrucciones) de Pascal para formar un programa válido en Pascal. Los lenguajes de computadoras tiene menos combinaciones aceptables que los lenguajes naturales, sin embargo, estas combinaciones deben ser utilizadas correctamente; ello contrasta con los lenguajes naturales que se pueden utilizar aunque no sigan las reglas gramaticales e incluso aunque no sean comprendidos.

3.2.1 Concepto de lenguaje, vocabulario y reglas sintácticas

Un lenguaje de programación es un conjunto de reglas, símbolos y palabras especiales que permiten construir un programa.

La sintaxis es el conjunto de reglas que gobiernan la construcción o formación de sentencias (instrucciones) válidas en un lenguaje. La semántica es el conjunto de reglas que proporcionan el significado de una sentencia o instrucción del lenguaje. Solamente las sentencias correctamente sintácticas pueden ser traducidas por un lenguaje de programación, y los programas que contienen errores de sintaxis son rechazados por la computadora. Cada lenguaje de programación posee sus propias reglas sintácticas.

El vocabulario de un lenguaje es un conjunto de símbolos (en ocasiones se denominan símbolos terminales). Los símbolos usuales son: letras, dígitos, símbolos especiales (, ; : / & + - *, etc.), palabras reservadas o claves –**if** (si), **then** (entonces), **repeat** (repetir), **for** (o), **begin** (inicio), **end** (fin).

Las reglas sintácticas son los métodos de producción de sentencias o instrucciones válidas que permitirán formar un programa. Estas permiten reconocer si una cadena o serie de símbolos es correcta gramaticalmente y a su vez información sobre su significado o semántica de tal manera que deben definir los conceptos de sentencia (instrucción), expresión, identificador, variables, constantes, etc., y deben permitir de modo fácil verificar si una secuencia de símbolos es una sentencia, expresión, etc., correcta del lenguaje.

Para definir las reglas sintácticas se suelen utilizar dos tipos de notaciones: la formalización de Backus-Naur-Form (**BNF**) y los diagramas o grafos sintácticos.

3.2.1.1 La notación BNF

La notación o formalización BNF (Backus-Naur-Form) es uno de los métodos empleados para la definición de reglas sintácticas y se concibió para que permitiera decidir en forma algorítmica cuándo una sentencia es válida o no en un lenguaje. Esta notación fue ideada por P. Naur que junto con otro grupo de científicos desarrollaron en 1960 el lenguaje de programación ALGOL 60. Aunque los diagramas sintácticos se utilizan con mayor profusión sobre todo en su aplicación a lenguajes estructurados como Pascal, dedicamos este párrafo a la notación BNF por su interés histórico en el campo de la programación de computadoras comerciales.

La notación BNF utiliza esencialmente cuatro metasímbolos (símbolos externos al lenguaje o utilizados raramente)

"::=" "|" "{" "}"

Los significados de los diferentes símbolos son:

"::="	significa	"esta definido como"
"|"	significa	"o bien"
"{" "}"	significa	"repetición de los símbolos encerrados entre las llaves"

Ejemplos:

*<expresión> ::= a-b | a*b <expresión> se define como la serie o cadena de símbolo "a–b" o bien la cadena "a*b"*

<expresión> ::= <letra> + <letra> + <letra> + …(a)

<letra> ::= a | b | c | d |…|z (b)

| a la expresión se obtiene uniendo o concatenando una letra con el signo "+" con otras letras.

b letra es cualquiera de las letras del alfabeto.

<identificador> ::= <letra> { <letra> | <digito> }

identificador es una letra seguida de cualquier serie de letras o dígitos.

*<Símbolo de operación aritmética> ::= + | - | * | / | ^*

*Símbolo puede ser + (suma), - (resta), * (multiplicación), / (división), ^ (exponenciación).*

De acuerdo con las anteriores reglas sintácticas se pueden formar las siguientes cadenas de símbolos válidos:

expresiones	*abs*	*mnp*	*abs*	*man*
identificadores	*ab5*	*beta4*	*denco17*	

Sin embargo no son válidos los identificadores siguientes:

5b ab beta4^ 6 demo*

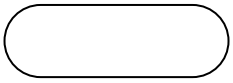
En consecuencia utilizando la notación BNF se pueden formular las diversas reglas sintácticas, identificadores, expresiones, variables, etc.

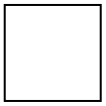
3.2.1.2 Diagramas sintácticos

Una de las herramientas más útiles –en la actualidad su uso es muy frecuente en el ámbito universitario y en el científico– para describir con precisión las reglas sintácticas son los diagramas sintácticos.

Los diagramas o grafos sintácticos son una serie de símbolos unidos por flechas o caminos. Los posibles caminos representan las secuencias o sucesiones posibles de símbolos.

Existen dos tipos de símbolos: terminales y no terminales

terminales  *o bien* 

no terminales  *o bien* 

Una regla sintáctica se compone de una cabecera o encabezamiento con el nombre de elemento que define y a continuación el grafo o diagrama. Los diagramas se leen siguiendo las flechas desde la entrada hasta la salida, o lo que es igual, comenzando por la izquierda y siguiendo la dirección de la flecha. En los puntos de bifurcación puede ir por cualquier Identificador

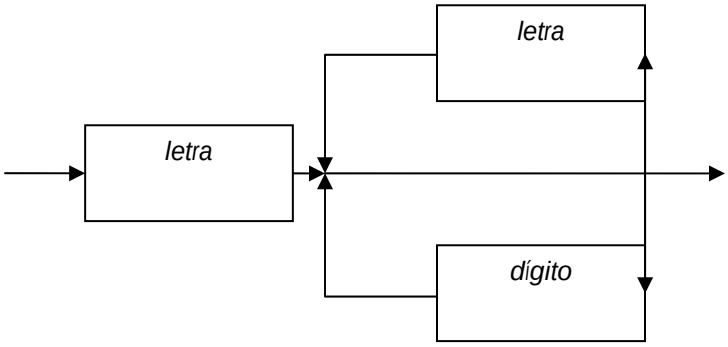


Diagrama 3.1a. *Identificador.*

El diagrama sintáctico también se puede representar del modo siguiente:

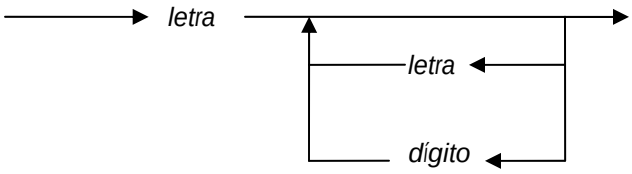
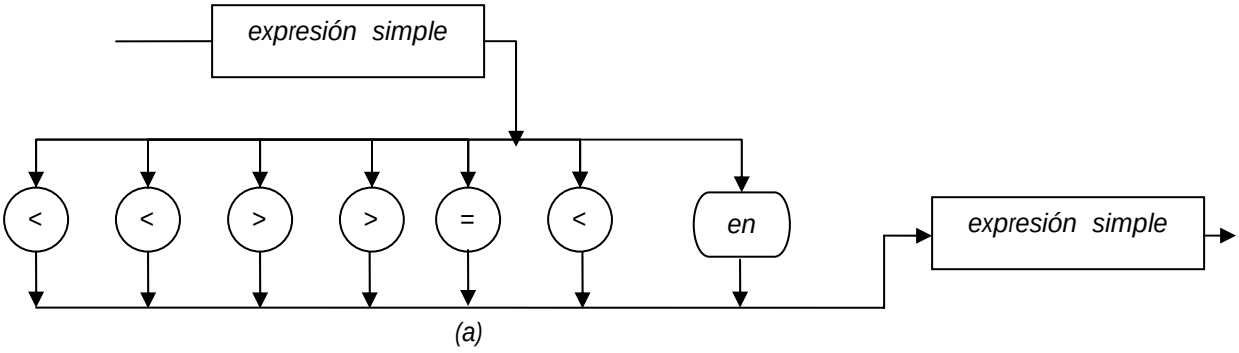


Diagrama 3.1b. Identificador.

En notación BNF el diagrama 3.1. sería:

<identificador> ::= <letra> { <letra> | <digito> }

Expresión:



(a)

o bien:

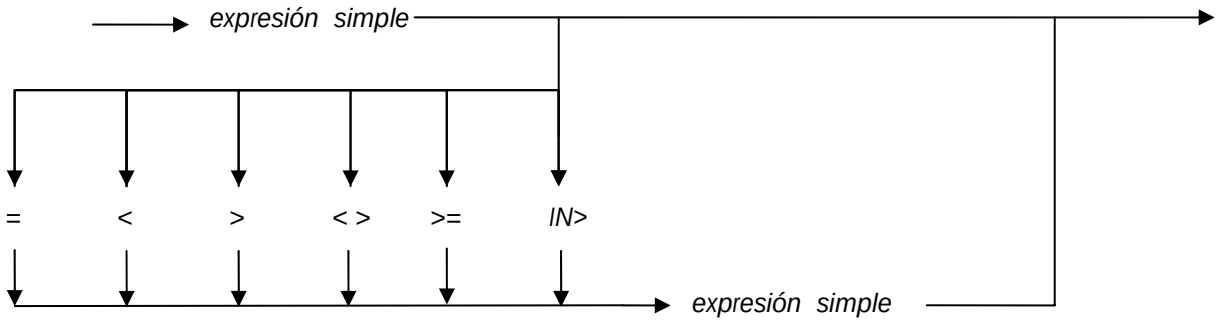
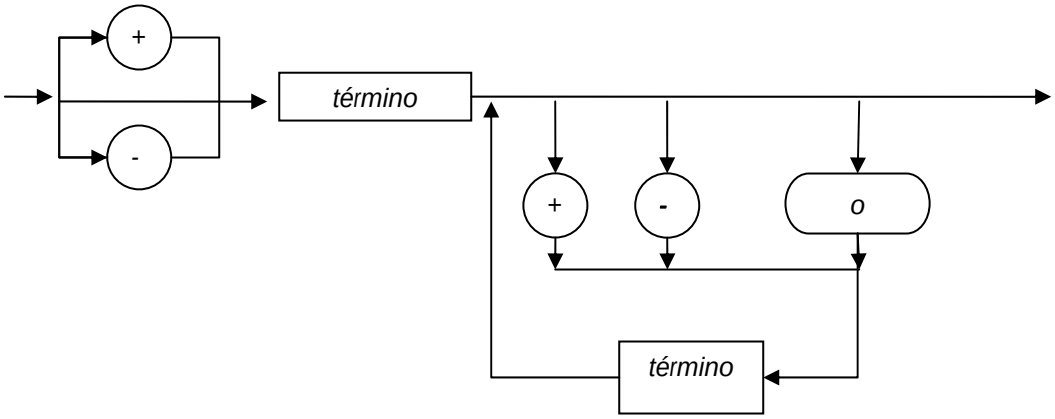
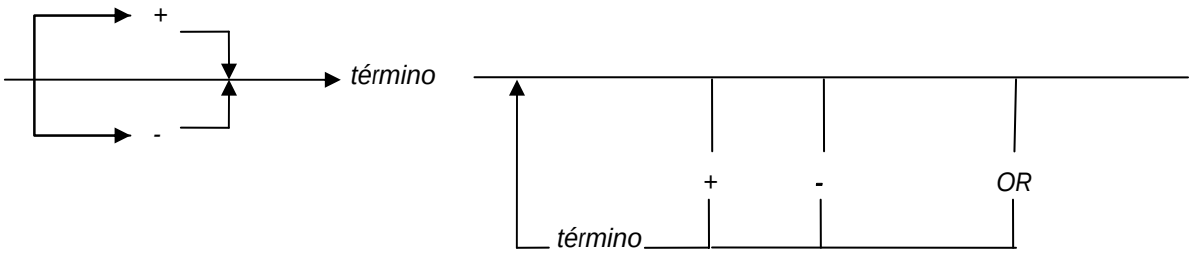


Diagrama 3.2. Expresión.

Expresión simple:



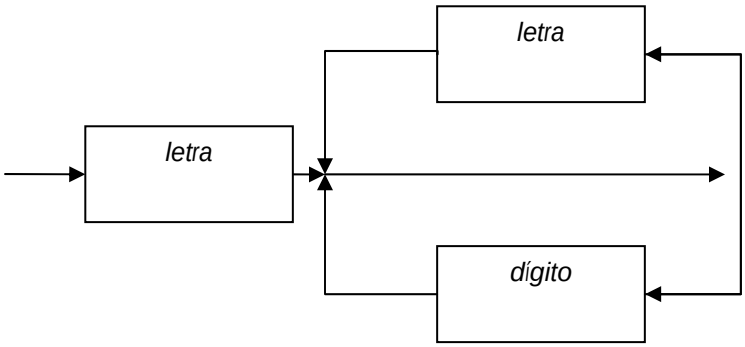
o bien:



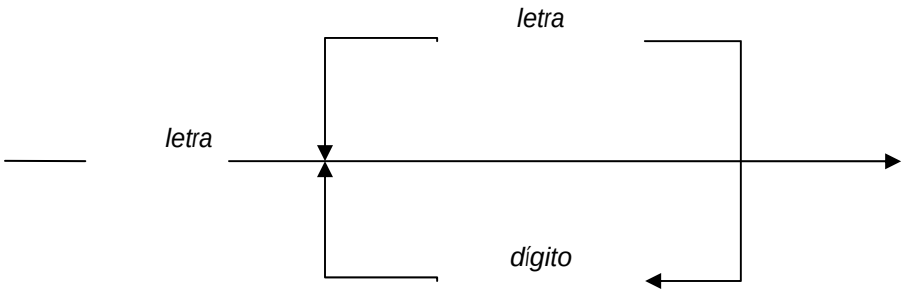
Expresiones válidas son:

x *x – y + z* *x + y * (x – y)/(Z - W)*

Variable numérica (en lenguaje BASIC):



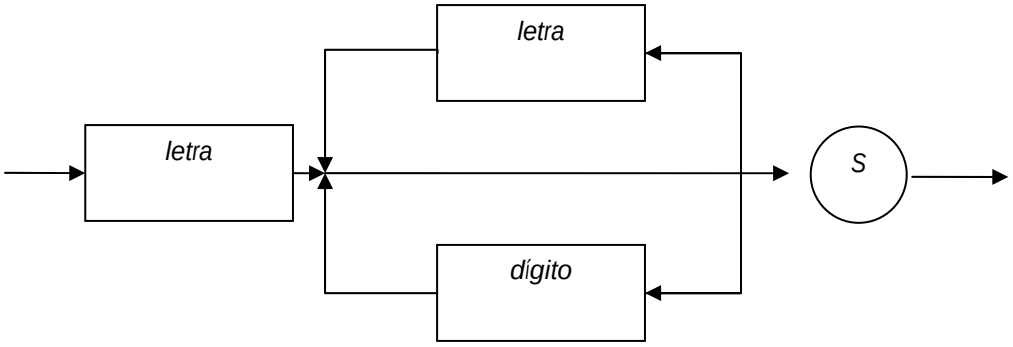
o bien



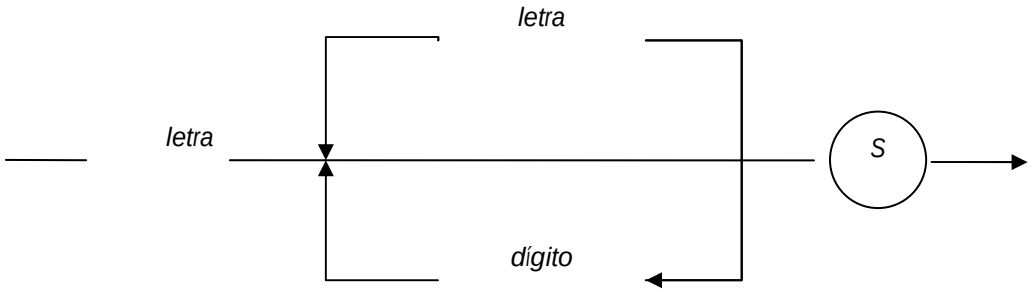
Variables numéricas válidas son:

Demo 5 *SEVILLS* *nombres 17*

Variable alfanumérica (en lenguaje BASIC):



o bien:



Nombres de variables alfanuméricas o de cadenas válidas son:

demo 5 $ ABS Nombre$

Sentencia REM (en lenguaje BASIC)

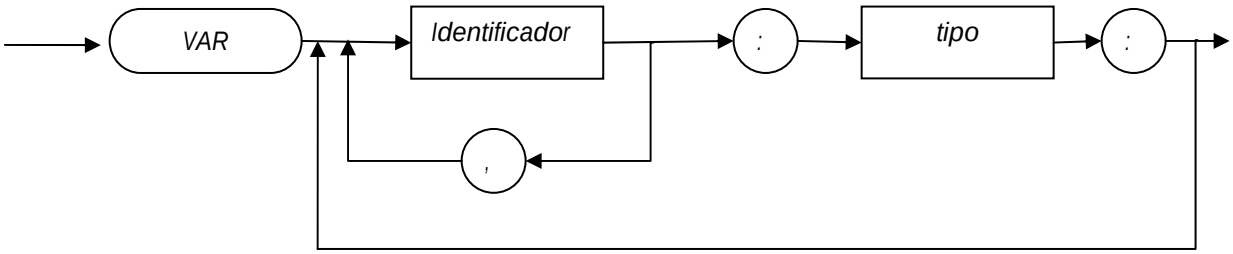
o bien:

Sentencias REM válidas son:

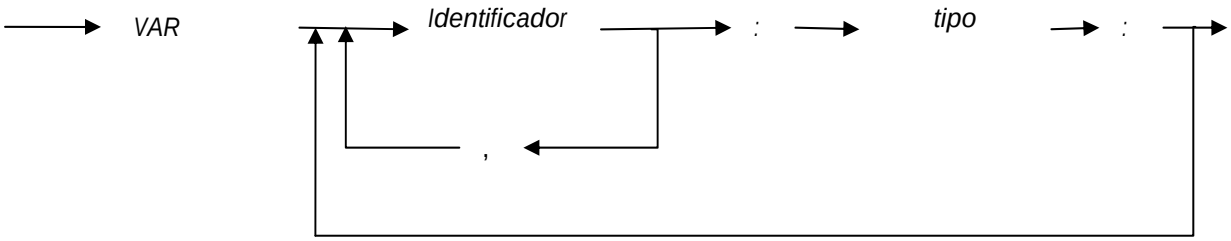
REM esto es una prueba

REM adelante mis valientes

Variable (en lenguaje PASCAL):



o bien:

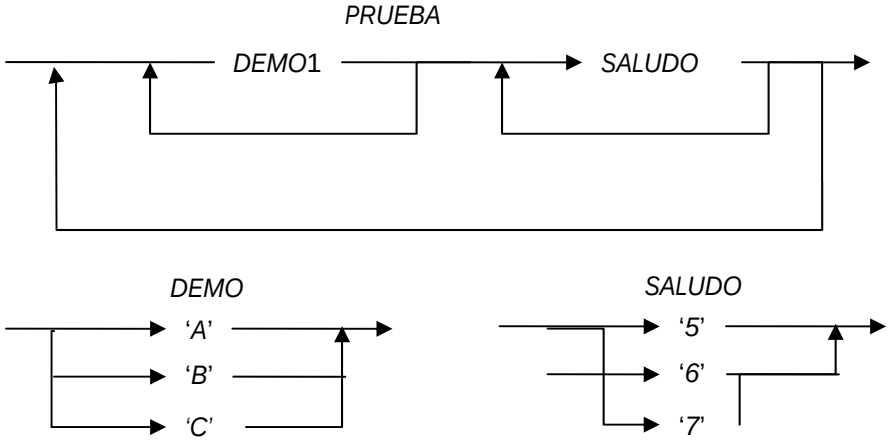


declaraciones válidas de variables son:

```
VAR     A, B: INTEGER :
        CONTADOR : REAL;
        CADENA : CHAR;
```

Ejercicio de aplicación:

Dados los diagramas sintácticos correspondientes a hipotéticas expresiones "PRUEBA", "DEMO","SALUDO" de un lenguaje de programación, averigüe nombres válidos de expresiones.



Nota: Los símbolos 'A', 'B',... equivalen a: A, B, es decir son símbolos terminales.

3.2.2. Clasificación de los lenguajes: bajo nivel y alto nivel

Existen centenares de lenguajes de programación para computadoras y cada uno tiene diferentes versiones. Cada lenguaje tiene sus ventajas e inconvenientes junto con sus defensores y detractores. Algunos lenguajes son ideales para la programación de un tipo de problemas (gestión, científicos...) y otros han sido diseñados para resolver otros problemas diferentes (educación, investigación, etc.).

Los lenguajes se suelen clasificar en términos de niveles que constituyen una jerarquía de los lenguajes de programación, relacionada con el número de instrucciones necesarias para realizar una tarea específica. Los lenguajes de programación se clasifican como de ***bajo nivel*** *y* ***alto nivel****.*

El nivel de un lenguaje de programación es indirectamente proporcional al número de instrucciones necesarias para realizar una tarea especifica; esto es, un programa destinado a calcular e imprimir la media de las calificaciones de las asignaturas de un estudiante, necesita muchas más instrucciones en un lenguaje de bajo nivel que en uno de alto nivel. Los lenguajes de bajo nivel están más próximos a la máquina que los lenguajes de alto nivel próximo al usuario.

PROGRAMAS EQUIVALENTES EN DIFERENTES LENGUAJES (UNIVAC 1100)

FORTRAN (ALTO NIVEL)	*OCTAL (MAQUINA)*	*ASSEMBLER (ASSEMBLY)*	
TEST=0	*7413 13 00 0 000000 0003* *0000 00 00 0 000000*	*LMJ* *+*	*X11,NINTRS* *0000,0*
OUTPUT=1	*0500 00 00 0 000000 0000*	*SZ* *LA* *SA*	*TEST* *A0,(1.000000+0)* *A0,OUTPUT*
1 TEST = TEST+1	*1000 00 00 0 000004 0000* *7600 00 00 0 000000 0000*	*IL LA* *FA*	*A0,(1.000000+0)* *A0,TEST*
OUTPUT=OUTPUT+TEST	*0100 00 00 0 000000 0000* *7602 00 00 0 000001 0000*	*SA* *FM*	*A0,TEST* *A0,OUTPUT*
IF(TEST.LT.100)GOTO 1	*0100 00 00 0 000001 0000* *1000 02 00 0 000000 0000* *7601 02 00 0 000005 0000* *7402 02 00 0 000016 0001*	*SA* *LA* *FAN* *JP*	*A0,OUTPUT* *A2,TEST* *A2,(1.000000+2)* *A2,$+2*
PRINT 2,OUPUT	*7401 02 00 0 000005 0001* *7413 13 00 0 000000 0004* *0000 04 00 0 000002 0000* *0000 04 10 0 000000* *0000 02 00 0 000001 0000* *7201 00 00 0 000000 0005* *7413 13 00 0 000000 0006*	*JNZ* *LMZ* *+* *+* *+* *SLJ* *LMJ*	*A2,1L* *X11,NPRT$* *0100,2F* *0110,0* *0040,OUTPUT* *N102$}* *X11,NSTOP$*
2 FORMAT(F15.0) *END*	*0000 00 00 0 000006 0000*	*+*	*(0050505050505)*

Fuente: Actividades escolares y Guía de Referencia para usar con el Computer Parts Kit, Cambrige.MN: Educational Computer Shoppe,1978.

3.2.2.1 Lenguajes BAJO NIVEL

El procesador (unidad central de proceso; CPU) de la computadora no puede ejecutar directamente sentencias escritas en un lenguaje simbólico (alto nivel), basado en símbolos; el procesador solamente puede ejecutar instrucciones más simples, llamadas instrucciones de máquina. Una instrucción de máquina es un conjunto de ceros y unos, es decir el procesador sólo entiende lenguaje binario.

Cada procesador tiene su juego o conjunto de instrucciones que dependerá de la estructura física o electrónica de la computadora, o lo que es igual cada computadora tiene solo un lenguaje de programación que puede ser ejecutado: **el lenguaje máquina**.

EL LENGUAJE MAQUINA

Un programa máquina (o escrito en lenguaje o código máquina) es un conjunto de instrucciones de máquina escritas con ciertas reglas sintácticas que constituyen el **lenguaje máquina**. El programa escrito en lenguaje máquina es el único que entiende la computadora. Evidentemente los programas escritos en lenguaje máquina son complicados, largos y difíciles de escribir. Imagínese instrucciones máquinas como:

```
1000110000110001
0110001100111110
```

El programador debe hacer un gran esfuerzo para recordar tales instrucciones, si es que llega a recordarlas. Los programas escritos en lenguaje máquina, aunque fueron los que primero aparecieron y por ello se denominan lenguajes de la primera generación, son difíciles de interpretar o corregir. A estos inconvenientes es preciso añadir que, como se ha comentado, estos lenguajes sólo sirven para un determinado procesador y generalmente una sola máquina (aunque dos computadoras estén basadas en el mismo procesador –microprocesador en el caso de las computadoras personales– no tienen porqué coincidir su estructura y por consiguiente su programación). Así resultará que el lenguaje máquina sólo servirá para una solo máquina que, junto con los inconvenientes citados, hacen que este lenguaje, entendible por la máquina, no lo sea por el programador. Para solucionar este problema en un nivel superior se diseñaron los lenguajes ensambladores basados en codificación de las instrucciones mediante nemotécnicos (o nemónicos) que son un conjunto de letras y números más simples de utilizar que el código máquina.

EL LENGUAJE ENSAMBLADOR (Assembler)

Los lenguajes máquina como los ensambladores son únicos para un procesador particular (en el caso de los microprocesadores, Z-80. 6502, 8088, 68000, etc.). La gran diferencia entre los dos tipos reside en el medio en el que se representan las instrucciones por el programador. En lugar de las pesadas series de 1 y 0, el lenguaje ensamblador utiliza símbolos reconocibles, llamados nemotécnicos para representar las instrucciones (Figura 3.2.).

0000005A	1850		LR 5.0
5C	5C40	C16E	M 4,=F'A'
60	1A	1A47	4,7
62	5ª64	0000	6,0(4)
66	5064	0000	6,0(4)
6A	B7A8	C04E	10,8, LOOP
Direcciones De memoria	Programa objeto		Programa ensamblador

Figura 3.2. Programa ensamblador y objeto.

Los lenguajes ensambladores constituyen para muchos historiadores de software, la segunda generación de los lenguajes de programación.

*En las décadas de los años 60 y 70 los lenguajes ensambladores se utilizaron mucho para desarrollo de programas de aplicación y siempre para desarrollo de software del sistema (sistemas operativos, utilidades, etc.). Aunque los programas eran muy largos, muchos programadores preferían la programación en ensamblador por considerar que utilizaban las características del sistema de computadora de un modo más eficaz. En la década de los 80 se popularizó el uso de los lenguajes en ensamblador de los microprocesadores más populares Z-80, 6502 y 8085 en 8 bits; 8086/8088, 68000, 80286 en 16 bits, y 80386 en 32 bits. El desarrollo de los lenguajes de programación de alto nivel de la cuarta generación de lenguajes (**C**, esencialmente) está comenzando a sustituir a los lenguajes ensambladores en el desarrollo de software del sistema. En cualquier forma. Hoy día, el porcentaje más alto de la programación –en general– se realiza en lenguajes de alto nivel.*

3.2.2.2. Lenguajes de ALTO NIVEL

De lo expuesto en el apartado anterior se deduce el interés por otros lenguajes de programación generales (y que a ser posible no dependan de la máquina utilizada) con los cuales se puedan describir algoritmos para ser ejecutados en el mayor número de computadoras. Tales lenguajes, denominados ***lenguajes de alto nivel****, permiten programar sin necesidad de conocer el funcionamiento interno de la máquina.*

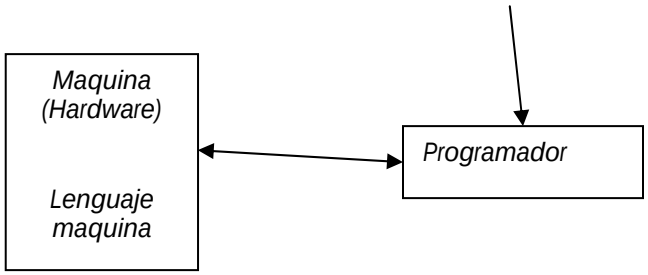
Los lenguajes de alto nivel poseen una potencia considerable en sus instrucciones, de modo que una instrucción en lenguaje de alto nivel equivale a varias en lenguaje máquina. Así por ejemplo:

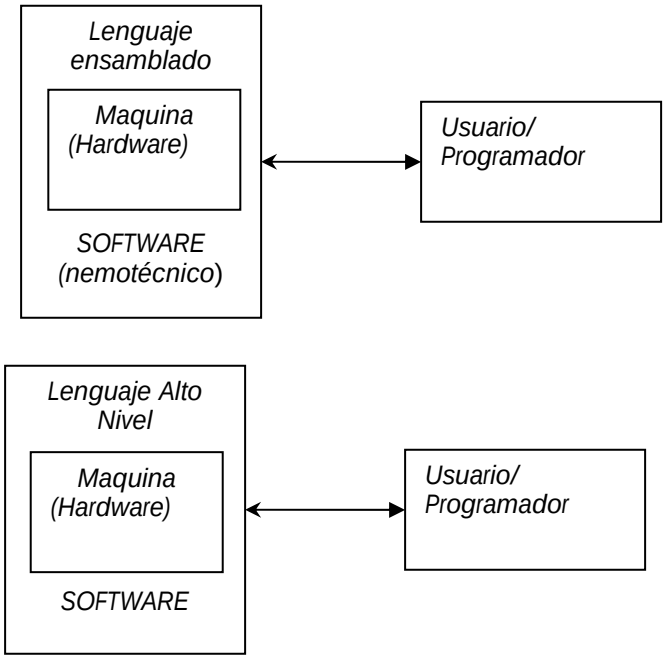
*M = A + B (I, J) + C * LOG (X)*

puede suponer muchas instrucciones (posiblemente centenares); sin embargo la instrucción es fácilmente entendible por el programador.

En esencia las diferencias con el lenguaje ensamblador, residen en que los lenguajes de alto nivel utilizan instrucciones muy potentes (normalmente entendibles en idioma inglés –no nemotécnicas, IF, THEN, WHILE, FOR, DO, PRINTF, etc.), independientes de la máquina sobre la que operan.

Un esquema sintetizador del funcionamiento en bloques de los lenguajes se muestra en las Figuras 3.4:





Figuras 3.4. Funcionamiento en bloques.

Los lenguajes de alto nivel se pueden dividir en diferentes grupos:

- *Lenguajes orientados a los procedimientos.*
- *Lenguajes orientados a los problemas*
- *Lenguajes de consulta*
- *Lenguajes generadores de aplicaciones.*

LENGUAJES ORIENTADOS A LOS PROCEDIMIENTOS

Son los más utilizados y se subdividen a su vez según su aplicación en:

Lenguajes científicos: FORTRAN, APL.
Lenguajes de gestión: COBOL, RPG.
Lenguajes de aplicaciones múltiples: BASIC, Pascal, Ada, C.
Lenguajes educativos: Logo, Pilot y Prolog.
Lenguajes de la inteligencia artificial: Lisp.

3.2.3 Intérpretes y compiladores

En cualquier lenguaje de alto nivel en que se escriba un programa, éste debe ser traducido a lenguaje máquina antes de que pueda ser ejecutado. Esta conversión de instrucciones de alto-nivel a instrucciones a nivel máquina se hace por programas de software del sistema, denominados compiladores e intérpretes. Estos programas especiales se denominan en general traductores.

Cuando se prepara un programa, el programador lo escribe en un lenguaje de computadora de los citados en el párrafo 3.2.2. Normalmente, el programa se introduce

en la computadora utilizando un editor (un programa procesador de palabras, propio del sistema operativo de la máquina –caso de EDLIN en el sistema operativo MS-DOS de las computadoras IBM PC y compatibles– o ajeno al sistema, Personal Editor, por ejemplo en el mismo caso de MS-DOS. Un editor permite al programador teclear el programa línea a línea, para introducirlo en la RAM y guardar o grabar una copia en disco. La versión del programa escrita en un lenguaje de computadora es el **programa** *o* **código fuente** *del programa general. Antes de que el programa se pueda ejecutar, se debe traducir desde el código fuente al lenguaje máquina llamándose al programa traducido programa o* **código objeto***. El proceso de traducción y su conversión en programa objeto difiere según que el programa sea compilador o intérprete.*

COMPILADORES

Un compilador es un programa que traduce el programa fuente (conjunto de instrucciones de un lenguaje de alto nivel, por ejemplo COBOL o Pascal) a programa objeto (instrucciones en lenguaje máquina que la computadora pueda interpreta y ejecutar). Un compilador independiente (o un intérprete, como se verá en la siguiente sección) se requiere para cada lenguaje de programación; esto es, para ejecutar programas en COBOL o Pascal, necesitará un compilador COBOL o un compilador Pascal.

El compilador efectúa sólo la traducción, no ejecuta el programa.
El proceso de compilación en Pascal se muestra en la Figura 3.4.
Una vez compilado el programa, el resultado en forma de programa objeto será directamente ejecutable.

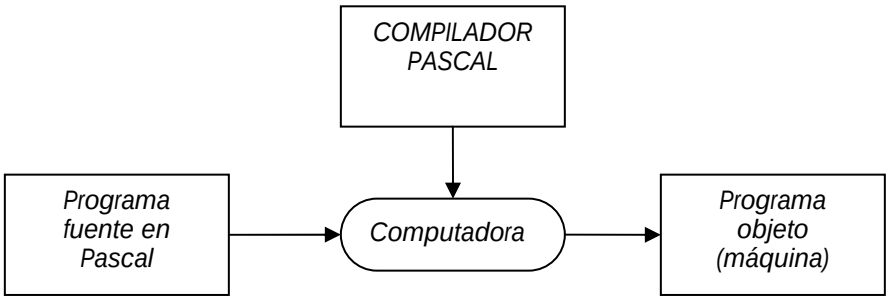


Figura 3.4. *Compilación de un programa en Pascal*

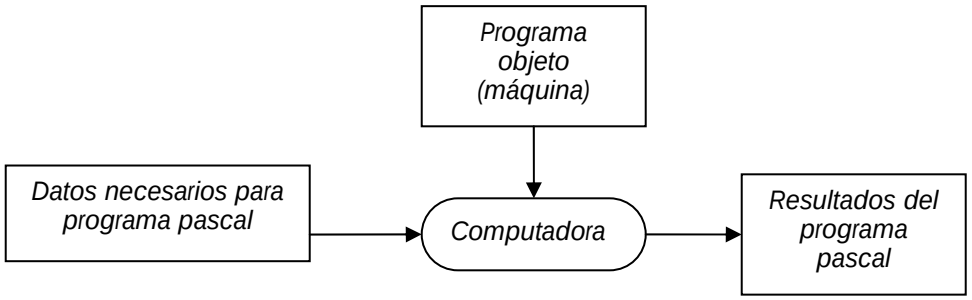


Figura 3.5. *Ejecución de un programa.*

Para asentar las ideas sobre la compilación consideremos un segundo ejemplo, en este caso, con un programa en COBOL. Los pasos a seguir en el proceso de compilación son:

1. *Introducir el programa en COBOL con ayuda de un editor.*

2. *Llamar al programa COBOL, normalmente situado en disco y cargarlo en la memoria principal junto con el programa fuente COBOL. Comenzar la compilación.*

3. *Si el programa fuente contiene un error de sintaxis en una instrucción (una instrucción con formato no válido, por ejemplo en COBOL la instrucción DISPLY «ALELUYA» tiene un error de sintaxis, ya que DISPLY se ha escrito mal, su verdadero nombre es DISPLAY), el compilador al intentar traducir esa instrucción produce un mensaje de error. Algunos compiladores detienen la traducción con el primer error mientras otros continúan la traducción para encontrar otros errores.*

   El programador debe encontrar, entonces, el error en el código fuente, comenzando de nuevo la compilación y continuar una y otra vez hasta que el programa se compila felizmente. Normalmente es necesario compilar (ejecutar) el programa varias veces, antes de que el programa haya sido traducido correctamente.

   Sin embargo, a veces le ocurrirá que la compilación termina sin errores y sin embargo no se obtienen los resultados apetecidos; en este caso será debido a errores lógicos de programación como por ejemplo el intento de división por cero (lógicamente imposible en la máquina) que produce error.

4. *El compilador COBOL ha traducido el programa fuente en un programa máquina llamado programa objeto. El programa objeto es la salida del proceso de compilación. En este momento, el programa objeto reside en memoria y se puede ejecutar con el nombre que se la haya dado, o bien grabarlo en un disco para poder utilizarlo directamente en otra sesión de trabajo, pero ya no será necesario el proceso de compilación.*

Por consiguiente, para poder ejecutar el programa en otra sesión de trabajo, deberá almacenar el programa objeto en un almacenamiento secundario – disco o disquete– para su recuperación posterior. En la mayoría de los sistemas de computadoras grandes, esta operación se suele hacer automáticamente.

Por consiguiente cada vez que necesite ejecutar su programa COBOL, no necesitará realizar el proceso de compilación; bastará simplemente con llamar al programa objeto almacenado en el dispositivo de memoria auxiliar, cargarlo en la memoria principal y después ejecutarlo.

El programa objeto está ya en código máquina y no se requiere compilación. El programa objeto sin errores se suele denominar ejecutable.

Algunos compiladores traducen sólo programas completos, mientras otros traducen secciones de un programa. Esta característica es especialmente útil en diseño de grandes programas. Los programadores pueden romper el programa en secciones o módulos y compilar/verificar cada sección independientemente.

Cuando esto se realiza, es necesario utilizar un **enlazador** *o* **encadenador** *(«linker»), también llamado* **editor de enlaces** *(«link editor») que une («cose») las secciones traducidas, juntas en un solo programa. Los enlazadores se utilizan también con programas que tiene una sola sección, para añadir el código de recepción del control del sistema operativo cuando el programa comienza y devolver el control al sistema operativo cuando el programa termina. La operación del editor de enlaces y el compilador se muestra en la Figura 3.6.*

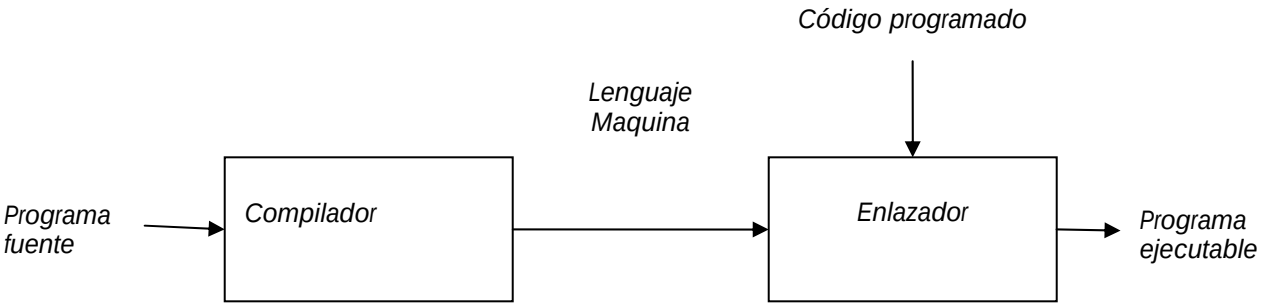


Figura 3.6. *Editor de enlaces.*

Si desea hacer cambios en los programas fuente originales escritos en COBOL, Pascal, etc., deberá recuperar el programa fuente del dispositivo de memoria auxiliar (disco, disquete, etc.), realizar los cambios –mediante el editor–, volver a compilar y crear un nuevo programa objeto, que será la última versión actualizada y que le recomendamos ponga un nombre diferente para poder recordarla con el paso del tiempo (es frecuente denominar a las sucesivas versiones con nombres como este GRAFICOS1, GRAFICOS2, etc.).

Los programas que se ejecutan, con frecuencia se almacenan y ejecutan como programas objetos, normalmente separados de los programas fuentes originales.

INTERPRETES

Los lenguajes de programación además de ser compilados pueden ser interpretados. Un intérprete es un programa que procesa los programas escritos en un lenguaje de alto nivel, sin embargo, está diseñado de modo que no existe independencia entre la etapa de traducción y la etapa de ejecución. Un intérprete traduce cada instrucción o sentencia del programa escrito en lenguaje a código máquina e inmediatamente se ejecuta, y a continuación se ejecuta la siguiente sentencia. Ejemplos de intérpretes son las versiones BASIC que se utilizan en la mayoría de las microcomputadoras bien en forma residente (en memoria ROM), bien en forma no residente (en disco).

El intérprete está diseñado para trabajar con la computadora en modo conversacional o interactivo; en realidad se le dan órdenes al procesador a través del intérprete con ayuda –por ejemplo– de un teclado y un programa denominado editor.

El procesador ejecuta la orden una vez que ésta es traducida, si no existe ningún error de sintaxis y se devuelve el control al programador con indicación de mensajes (errores de sintaxis, de ejecución, etc.).

El intérprete no traduce todo el programa fuente en un solo paso, sino que ejecuta cada instrucción del programa fuente antes de traducir y ejecutar la siguiente. Éste se sitúa en memoria principal (RAM), junto con el programa del usuario. De este modo el programa no se ejecutará directamente traducido a lenguaje máquina, sino a través de la interpretación que se producirá al ejecutarse el programa interpretador en una especie de traducción simultánea. El proceso de interpretación se puede resumir en la Figura 3.7.

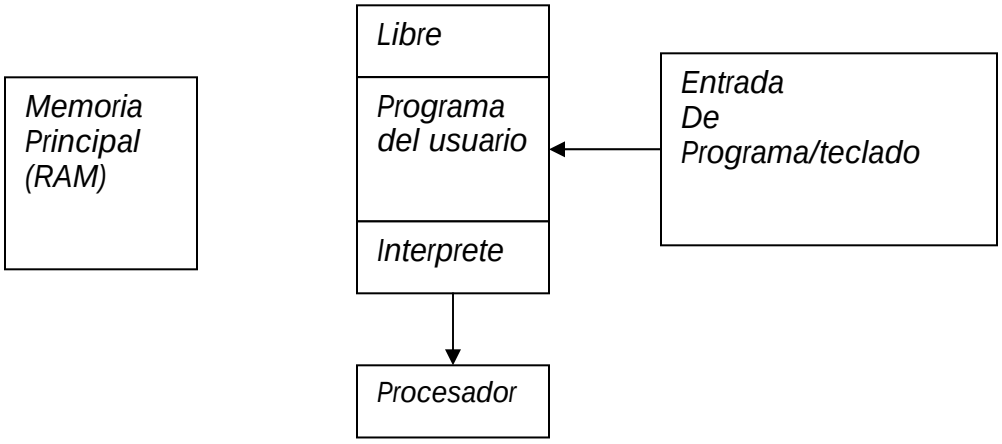


Figura 3.7. Proceso de Interpretación

Un intérprete suele incluir, casi siempre, el editor para introducir el programa fuente. Ello permite un paso fácil entre la edición del programa (código fuente) y la ejecución del programa en sí. Los diagramas de flujo de la siguiente figura muestran el uso de un intérprete y un editor.

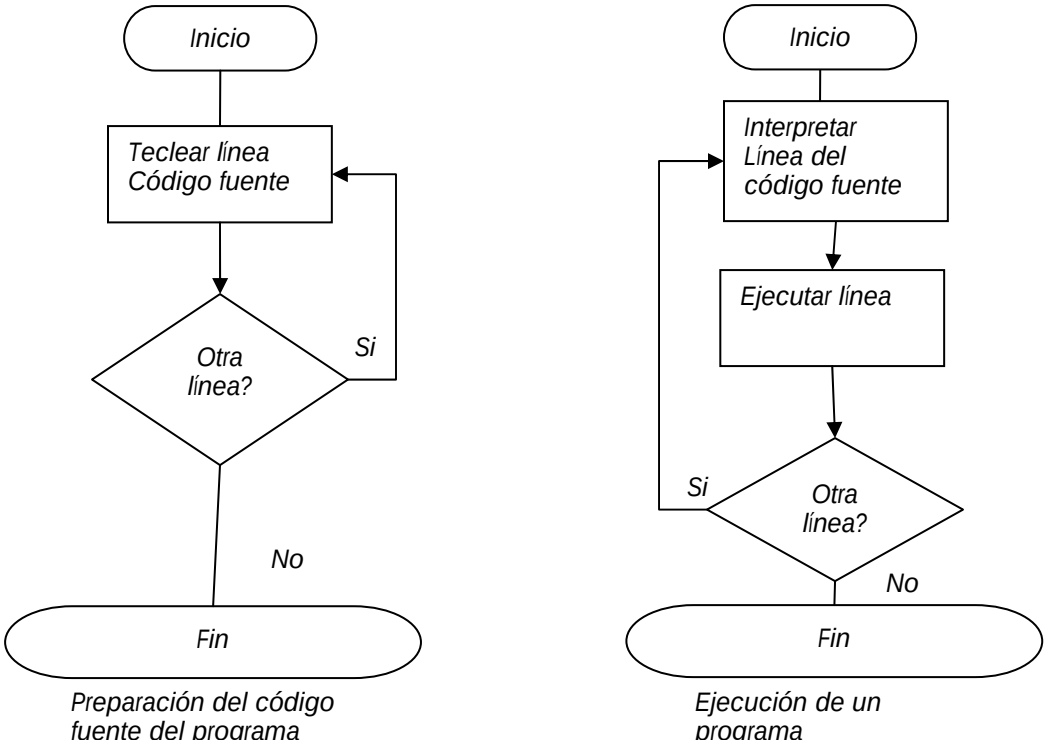


Figura 3.9.

3.2.3.1 Comparación entre intérpretes y compiladores

Los intérpretes y compiladores tienen ventajas e inconvenientes derivados de sus peculiares características.

La principal ventaja de los intérpretes sobre los compiladores reside en que el análisis sintáctico del programa se puede ir realizando a medida que se introduce por teclado o bien cuando se interpreta. El intérprete muestra un error e incluso un mensaje de diagnóstico que indica la naturaleza del problema. El programador puede ir directamente al error y corregirlo; de este modo los programas pueden ser comprobados y corregidos durante el desarrollo de los mismos. Las restantes características a destacar que diferencian un intérprete de un compilador se resumen a continuación.

Ejecución

La modificación del programa fuente es muy sencilla en el intérprete, mientras que en el compilador, además de la modificación con el programa editor, será necesario crear un nuevo programa compilado. Esto significa que para comprobar el funcionamiento de un programa, una vez modificado, habrá que volver a compilarlo.

Las ejecuciones sucesivas de un programa compilado no necesitan traducciones del programa fuente. Como el intérprete no produce un programa objeto, se debe realizar el proceso de traducción cada vez que se ejecute un programa.

Escritura

Un programa en intérprete es más fácil de escribir que en compilador, entre otras razones porque el programa intérprete lleva un editor incorporado.

Ocupación en memoria

Un intérprete es más pequeño que un compilador y por ello puede ser utilizado en una máquina con menos memoria. Sin embargo esta propiedad es engañosa, ya que al tamaño del programa hay que añadir el tamaño del propio intérprete que deberá estar en memoria; en cada caso particular puede ser diferente.

Rapidez

La ventaja fundamental de un programa escrito en lenguaje compilador es su rapidez, ya que en caso de la interpretación cada instrucción debe ser analizada e interpretada antes de su ejecución.

La ejecución del código máquina (programa objeto) es más rápida que la interpretación de un lenguaje de alto nivel. Cuando un programa está compilado, el tiempo de ejecución es pequeño por estar codificado en lenguaje máquina, sin embargo en un programa en lenguaje intérprete, el programa fuente se debe traducir cada vez que se ejecute, y por ello el tiempo de ejecución será la suma de la traducción a código máquina y la propia ejecución. Z-80 proporcionó los siguientes resultados:

Operación	Tiempo (en segundos)		
Interpretación	10	40	60
Compilación	300	300	300
Ejecución del código compilado (programa objeto)	1	1.5	2

Sin embargo, se puede dar la paradoja siguiente: el tiempo de compilación, corrección de errores, vuelta a compilar, etc. puede ser mayor que el tiempo empleado en ejecutar directamente un programa de forma interpretada (donde los errores se corrigen sobre la marcha).

Conclusiones:

Los lenguajes compiladores presentan la ventaja considerable frente a los intérpretes de la velocidad de ejecución, por lo que su uso será mejor en aquellos programas probados en los que no se esperen cambios y que deban ejecutarse muchas veces. Así mismo, en general, ocuparán menos memoria en el caso de programas cortos ya que en el caso de interpretación, el interpretador que tendrá un tamaño considerable debe residir siempre en memoria.

Los lenguajes intérpretes encuentran su mayor ventaja en la interacción con el usuario, al facilitar el desarrollo y puesta a punto de programas, ya que los errores son fáciles de detectar y sobre todo de corregir. Por el contrario, como el intérprete no produce un programa objeto, debe realizar el proceso de traducción cada vez que se ejecuta un programa y por ello será más lento el intérprete.

En la actualidad los programadores suelen aprovechar con frecuencia las ventajas tanto de intérpretes como de compiladores. En primer lugar desarrollan y depuran los programas utilizando un intérprete interactivo (por ejemplo, MBASIC intérprete); después compilan el programa terminado a fin de obtener un programa objeto (por ejemplo, MBASIC compilador).

3.3 PARTES CONSTITUTIVAS DE UN PROGRAMA

Después que se ha tomado la decisión de desarrollar un programa, el programador debe establecer el conjunto de especificaciones que deben contener el programa: entradas, salidas, algoritmo de resolución que incluirá las técnicas para obtener las salidas a partir de las entradas. Así pues, se pueden considerar tres grandes bloques en el diseño de todo programa:

- *Entrada de datos.*
- *Algoritmo de resolución del problema/codificación.*
- *Salida de resultados.*

Conceptualmente un programa puede ser considerado como una caja negra como se muestra en la Figura 3.9.

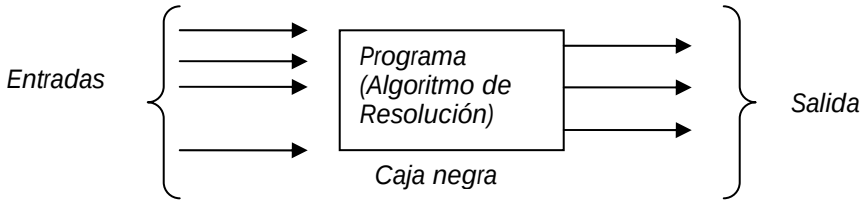


Figura 3.10. Bloques de un programa

La caja negra o algoritmo de resolución, en realidad, es el conjunto de códigos que transforman las entradas de programa (datos) en salidas de programa (resultados). Al establecer las especificaciones del programa, el programador debe conocer, en primer lugar, cuales son las entradas del programa y cuales son las salidas del programa, antes de que pueda especificar el contenido de la caja negra.

3.3.1 Entrada de datos

El programador debe establecer las entradas al programa o conocer de dónde provienen, así como el momento en que se requieren en el programa. Si el procesamiento de datos es interactivo, las entradas pueden proceder del usuario mientras el programa se está ejecutando; en otros casos pueden estar contenidas en ficheros de papel o microfilm propios del usuario o externos. En esencia los datos procederán de un dispositivo periférico de entrada.

Al proceso de introducir la información de entrada (datos) en la memoria de la computadora se denomina entrada de datos.

El programador debe realizar una lista de todas las entradas requeridas por el programa, la fuente de cada entrada y el formato en el que existe actualmente cada una de ellas. El programador debe determinar la frecuencia con que son requeridas las diferentes entradas por el programa: diariamente, semanalmente, mensualmente o en intervalos irregulares. Además, el programador debe determinar el lugar donde se prepararán, si las entradas serán en línea (on-line), fuera de línea (off-line) y si las entradas serán procesadas por lotes (modo«batch»).

Las computadoras reciben la secuencia de ejecución de las instrucciones a través del programa. Esta secuencia la seguirán las máquinas exactamente, y ejecutarán las sucesivas instrucciones, por ello el programador debe prever todas las situaciones posibles y evitar que un dato pueda ser solicitado sin que haya sido previamente introducido en la memoria.

3.3.2 Salida de resultado.

El programador debe listar todas las salidas previstas del programa, así como el formato requerido por el usuario. Algunas de las características que deben reunir las salidas son:

- Salida en pantalla o en papel impreso.
- Grabación de resultados en cinta o disco.
- Diseño o presentación de resultados: listas, tablas, informes, gráficos, etc.
- Frecuencia de salidas de resultados.

La respuesta a las premisas anteriores permitirá codificar correctamente las instrucciones de salida y elección adecuada del dispositivo periférico de salida: pantalla, impresora, unidad de cinta magnética, unidad de disquete, unidad de disco, trazador gráfico (plotter), etc.

3.3.3 Algoritmos de resolución / codificación.

Una vez que las entradas y las salidas han sido determinadas, el programador debe decidir como obtener las salidas deseadas a partir de las entradas dadas.

La caja negra que se denominó algoritmo de resolución en esencia constara de dos etapas:

1. Diseño del modelo de resolución del problema.
2. Algoritmo de resolución de problema.

En la etapa 1 se establece el modelo de preciso para la resolución del problema, para lo cual se tendrá en cuenta los datos de entrada y los resultados que se desean obtener; su estudio y desarrollo llevará al algoritmo de resolución del problema que deberá codificarse en un lenguaje de programación de alto o bajo nivel de los estudiados en el apartado 3.2.2.

El algoritmo de resolución se suele expresar previamente a la codificación en el lenguaje de programación, en alguna de las siguientes representaciones:

- Pseudo-código.
- Diagrama de flujo.
- Diagrama estructurado o N-S (Nassi-Shneiderman).

3.4 TIPOS DE INSTRUCCIONES

Las instrucciones disponibles en un lenguaje de programación, dependen del tipo de lenguaje bajo nivel o alto nivel. La diferencia esencial entre ambos lenguajes: próximos al usuario (alto nivel) y próximo ala maquina (bajo nivel), se manifiesta en el juego o repertorio de instrucciones (conjunto especifico de instrucciones del lenguaje). Así para conocer en profundidades las instrucciones en ensamblador se requiere conocer la estructura del procesador (registros, acumuladores, direcciones en memoria, etc.), mientras que en los lenguajes de alto nivel -salvo varias excepciones- no será necesario conocer la estructura interna de la máquina. Como ya se ha comentado el repertorio de instrucciones en lenguaje ensamblador es específico del procesador (o microprocesador

–en su caso–, z-80, 8088, 68000, etc.) mientras que en un lenguaje de alto nivel son especificas de dicho lenguaje.

Una clasificación de las instrucciones –llamadas también sentencias en los lenguajes de alto nivel– podría ser la siguiente:

- *Entrada / salida*
- *Asignación / movimiento*
- *Aritméticas*
- *Lógicas*
- *Bifurcación o transferencia de control*
- *Especiales*

En un sentido global las instrucciones se pueden dividir en dos grandes grupos:

1) Las que manipulan los datos.

2) Las que gobiernan la lógica del programa (es decir el orden de ejecución de las instrucciones del programa.

3.4.1 Instrucciones de entrada / salida

Permiten la transferencia de información desde los periféricos de entrada (teclado, unidad de cinta, unidad de disco, etc.) a la memoria de la computadora y desde ésta a un periférico de salida (pantalla, impresora, unidad de cinta, unidad de disco, etc.).

El proceso de introducción de datos en la memoria desde un dispositivo periférico de entrada se denomina lectura o carga de los datos, y al proceso de extracción de datos de la memoria y su envío a un dispositivo periférico de salida se le denomina escritura, grabación o conservación (memoria externa), impresión (impresora o trazador gráfico).

Instrucciones de lectura (entrada)

Las instrucciones de lectura afectan alas unidades o dispositivos llamados de lectura: pantalla (terminal), unidad de cinta magnética, unidad de disco o disquete, lectura de tarjetas perforadas, digitalizador, etc.).

Ejemplos de estas instrucciones en pseudo-códigos podrían ser:

Leer *A, B, C*	*Lectura de los valores numéricos correspondientes las variables A, B y C.*
Leer *425, 321*	*lectura de las constantes numéricas 425 y 321.*

En lenguajes de alto nivel estas instrucciones son:

BASIC	Pascal
INPUT	READ
LINE INPUT	READLN
READ	

Instrucciones de escritura (salida)

Afectan a las unidades de salida: pantalla, disco trazador gráfico, etc.

Ejemplos de estas instrucciones en pseudo-códigos y lenguajes de alto nivel son:

Pseudo-código	Escribir A, B, C	escritura de los valores numéricos correspondientes a las variables A, B y C.
	Escribir 53, 64	escritura de las constantes Numéricas 53 y 64.
BASIC	PRINT A, B, C PRINT USING	
Pascal	WRITE WRITELN	

3.4.2 Instrucciones de asignación / movimiento

Las instrucciones de asignación son fundamentales en casi todos los lenguajes de programación. Permiten asignar valores o variables del programa:

A = 5	el valor numérico 5 se asigna a la variable A (BASIC). equivalente a A = 5 en lenguaje BASIC; es opcional el uso de LET en BASIC.
A: = 5	equivalente a A = 5 en lenguaje Pascal.
A: = B + 5	equivalente a A = B + 5 en lenguaje Pascal.

Estas instrucciones ejecutan al cálculo del lado derecho y asigna el valor obtenido a la variable de la izquierda; el lado derecho de la instrucción puede ser cualquier expresión legal; el lado izquierdo debe ser el nombre de una variable. En el lado izquierdo no puede haber una expresión. La instrucción de asignación no tiene que confundirse con una ecuación matemática. Así:

A + 5: = B-2 en Pascal A + 5= B-2 en BASIC

No es un formato correcto, aunque la ecuación matemática correspondiente:

A + 5= B-2

Tenga sentido. Por el contrario

I: = I + 1 *o bien* *I = I + 1*

Es aceptable como instrucción de asignación (el valor de la variable I se incrementa en una unidad) y no tiene sentido matemático.

Las instrucciones de movimiento tienen el mismo sentido que la asignación y permite transferir la información o contenido de un campo o posición de memoria a otro, manteniendo siempre intacta la información en su primera posición (emisor) y variando el contenido de la segunda posición (receptor).

M ⟶ *N* *mueve el contenido del campo M al campo N*

Suponiendo que M = 32225 y N = 724, la instrucción anterior M ⟶ *N dejaría los campos M y N con los valores*

M = 32225 *y* *N = 32225*

3.4.3 Instrucciones matemáticas

Realizan el cálculo de operaciones aritméticas tales como sumar, restar, multiplicar, dividir o funciones como seno, coseno, tangente, etc.

M = P + Q	*suma el contenido de los campos o variables P y Q, almacenando el resultado en M*
A + B/2	*división de B entre 2 y asignación del resultado a A.*
Y = SIN (X)	*cálculo del seno del ángulo X y almacenamiento del resultado en Y.*

La siguiente tabla representa los operadores o símbolos de las operaciones aritméticas básicas:

Operador	Significado
+	Suma
-	Resta
*	Multiplicación
/	División
\ (div)	división entera (19 \ 5 = 3)
MOD	Resto (19 MOD 5 = 4)

Tabla 3.1 *operadores aritméticos*

3.4.4 Instrucciones lógicas y de relación

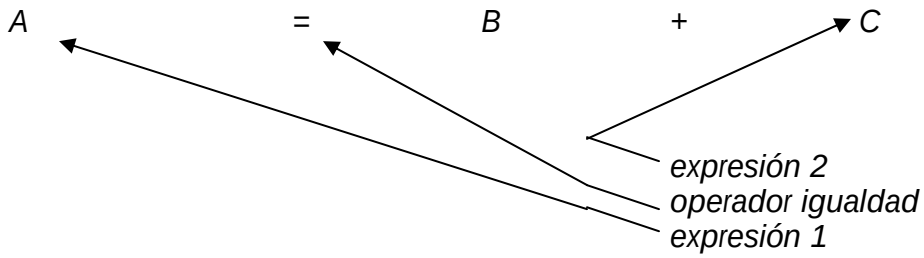
En numerosas ocasiones se necesita comparar dos expresiones aritméticas entre ellas con la finalidad de tomar una decisión en función del resultado de dicha comparación. Este tipo de operaciones se realiza con operadores de la relación que se indican en la Tabla 3.2:

Operador	Significado
<	menor que
>	mayor que
=	igual
>= ó =>	mayor o igual
<= ó =<	menor o igual
<> ó ><	diferente
<	no es menor
>	no es mayor

Tabla 3.2 *operadores de relación*

Estos operadores relacionan dos expresiones aritméticas entre si con el siguiente formato:

Expresión 1	OPERADOR DE RELACION	Expresión 2



Al conjunto anterior se le suele denominar condiciones

A > B
C > D-E

El resultado de evaluar una condición puede tomar sólo dos valores: verdadero y falso. Así por ejemplo

A > B si A = 7 y B = 3 entonces A > B es verdadero
si A = 14 y B = 27 entonces A > B es falso.

Si en un programa se elige una determinada condición para realizar una tarea concreta, ésta sólo se realizará cuando dicha condición sea verdadera o falsa –según la condición–. Además de los operadores de relación existen otro tipo de operadores condicionados conocidos como lógicos que permiten realizar las operaciones lógicas o booleanas AND (y), OR (o), NOT (no) y en ocasiones XOR (OR exclusiva) e IMP (implicación).

Las operaciones lógicas básicas AND, OR y NOT se representan por unos cuadrados conocidos como tablas de verdad y que se indica en la Tabla 3.3.

A	B	A AND B	A OR B	NOT A	NOT B
F*	F	F	F	V	V
F	V	F	V	V	F
V*	F	F	V	F	V
V	V	V	V	F	F

** F = falso o el valor lógico 0*

** v = verdadero o el valor lógico 1*

***Tabla 3.3**. Tablas de verdad de los valores operadores lógicos*

Si en lugar de dos condiciones A y B hubiese tres, el rango de valores diferentes que podrían tomar es

A	B	C
F	F	F
F	F	V
F	V	F
F	V	V
V	F	F
V	F	V
V	V	F
V	V	V

O bien considerando valores digitales 1(v) y 0(f) resultaría

A	B	C
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Así la tabla de verdad completa en el caso de tres condiciones C1, C2, y C3, y dos operaciones lógicas AND y OR.

C1	C2	C3	C1 AND C3	C2 OR C3	C1 AND (C2 OR C3)
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	0	0	1
1	0	1	1	1	1
1	1	0	0	1	1
1	1	1	1	1	1

El operador de negación NOT (NO) transforma la certeza o falsedad de cada suceso en su opuesto de modo que si el suceso A es verdadero NOT A es falso y de igual forma si A es falso, NOT A es verdadero.

Como aplicación de la condición NOT, los siguientes ejemplos representan la misma condición:

NOT (A = B)	A <> B
A = B	**NOT** ((A > B) **OR** (A <B))
A > B	**NOT** ((A = B) **OR** (A < B))

3.4.5 Instrucciones de control o transferencia de control

Permiten variar (romper) la secuencia de ejecución de un programa al saltar (bifurcar) a otra parte del mismo. Las instrucciones de este tipo se basan en una simple comparación o en los resultados de operaciones lógicas.

Las bifurcaciones producidas por las instrucciones de control pueden ser con salto adelante en la secuencia normal del programa o con salto atrás. Los tipos de bifurcaciones en cuanto a las condiciones a cumplir son: incondicionales y condicionales; las bifurcaciones incondicionales se realizan siempre que se ejecuta la instrucción, mientras que las bifurcaciones condicionales sólo se realizarán al cumplirse una determinada tarea.

3.4.6 Instrucciones especiales

Las instrucciones especiales varían de unos lenguajes a otros, y las que citaremos aquí no significan en modo alguno que las posean todos, sino que según el lenguaje o su versión, podrán existir o no. El grupo, más importante de instrucciones especiales se clasifican en:

- *Edición*
- *Impresión*
- *Conversión*
- *Ordenación*
- *Declarativas*
- *Modificación de direcciones*
- *Comunicación de E / S (entrada /salida)*
- *Gráficas*

Edición

Facilitan los formatos de presentación de resultados tanto en modo texto como en modo numérico. Ejemplos típicos son: Dibujo de rótulos en zonas especificas, supresión de ceros no significativos, inclusión de signos positivos, negativos o monetarios, justificación de resultados a derecha o izquierda, etc.

Impresión

Son una mezcla de instrucciones de salida y edición que facilita la escritura o impresión de resultados en una hoja de papel de impresora.

Conversión

En las computadoras es usual trabajar con diferentes sistemas o códigos de numeración: hexadecimal, binario, decimal y octal –son más frecuentes–, así como códigos de caracteres: ASCII y EBCDIC, también son los más comunes. Existen instrucciones de conversión que facilitan las conversiones y pasos de un sistema o código a otro.

Ordenación

Se utiliza para clasificación creciente o decreciente de datos.

Declarativas

Permiten asignar las zonas de memoria encargadas de almacenar los datos y definir el tipo de información almacenada. Así mismo se consideran en este tipo las instrucciones que con fines de documentación interna se incluyen en los programas para una mejor legibilidad de los mismos.

Modificación de direcciones

Actúan sobre las diversas posiciones de memoria, modificando su contenido.

Comunicación de E/ S

Actúan sobre los caminos o canales de comunicaciones, para facilitar la comunicación entre el procesador y periféricos externos (normalmente dispositivos electrónicos).

Gráficos

Instrucciones especiales para manipulación de gráficos. BASIC y Pascal son lenguajes que poseen características gráficas.

3.5 TIPOS DE PROGRAMAS

Los programas escritos para computadoras, en general, se pueden clasificar en tres tipos o estructuras fundamentales:

- *Lineales.*
- *Cíclicos.*
- *Alternativos (decisión ales)*

En realidad es difícil que un programa de complejidad –e incluso sencillo– se componga de una sola estructura y normalmente una mezcla de los tres tipos es lo que suele constituir un programa.

3.5.1 Programas lineales

Son aquellos en los que no existen instrucciones de bifurcación y por consiguiente las instrucciones se ejecutan en la misma secuencia es que han sido codificadas.

Estos programas se denominan también secuencias dado que siguen exactamente la secuencia especificada.

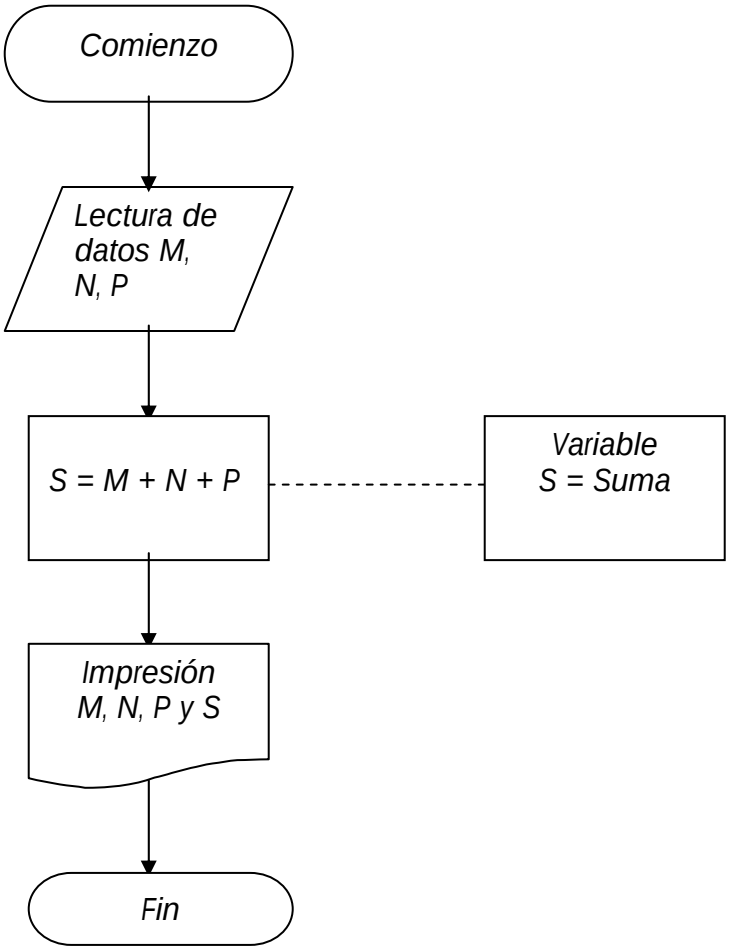
Es normal que estos programas consten de las siguientes fases:

- *Lectura o entrada de datos.*
- *Proceso.*
- *Impresión o salida de resultados.*

Ejemplo 1: *Diagrama de flujo para lectura de tres números M, N, P en una sola operación e impresión se los mismos y de su suma en una impresora.*

Datos de entrada:	*M, N, P*		
Proceso:	*Suma = M + N + P*	*;*	*S = M + N +P*
Salida de resultados:	*impresión de M, N, P y S*		

El diagrama de flujo sería:

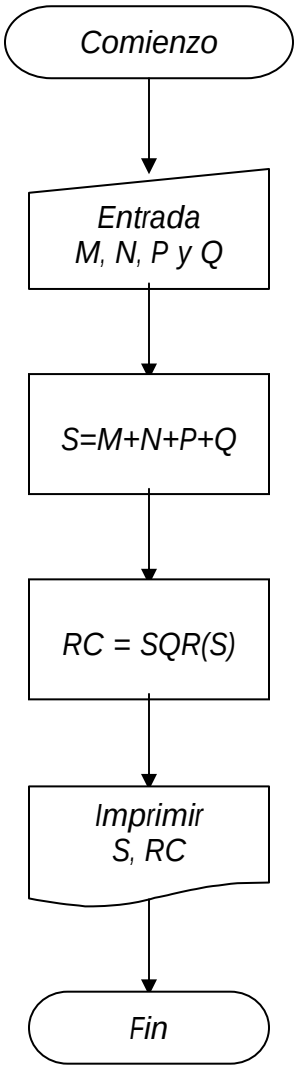


***Ejemplo 2**: Diagrama de flujo que calcula la suma de cuatro números M, N, P y Q, y a continuación extrae la raíz cuadrada de dicha suma. El valor de la suma S y la raíz cuadrada RC, se deben imprimir.*

Nota: los números M, N, P y Q son positivos.

Datos de entrada:	*M, N, P y Q*
Proceso:	*S = M + N + P + Q*
	RC = M + N + P + Q = S
Salida de resultados:	*Impresión de S y RC.*

El diagrama de flujo sería:



3.5.2 Programas cíclicos

Son aquellos programas en los que un grupo de instrucciones se ejecuta un número determinado de veces –de modo cíclico– hasta que se cumple una cierta condición que indica el fin de las ejecuciones de dichas instrucciones. El conjunto de las instrucciones que se repiten cíclicamente se denominan bucle, lazo o ciclo.

La estructura de un programa cíclico suele constar de los siguientes bloques o fases:

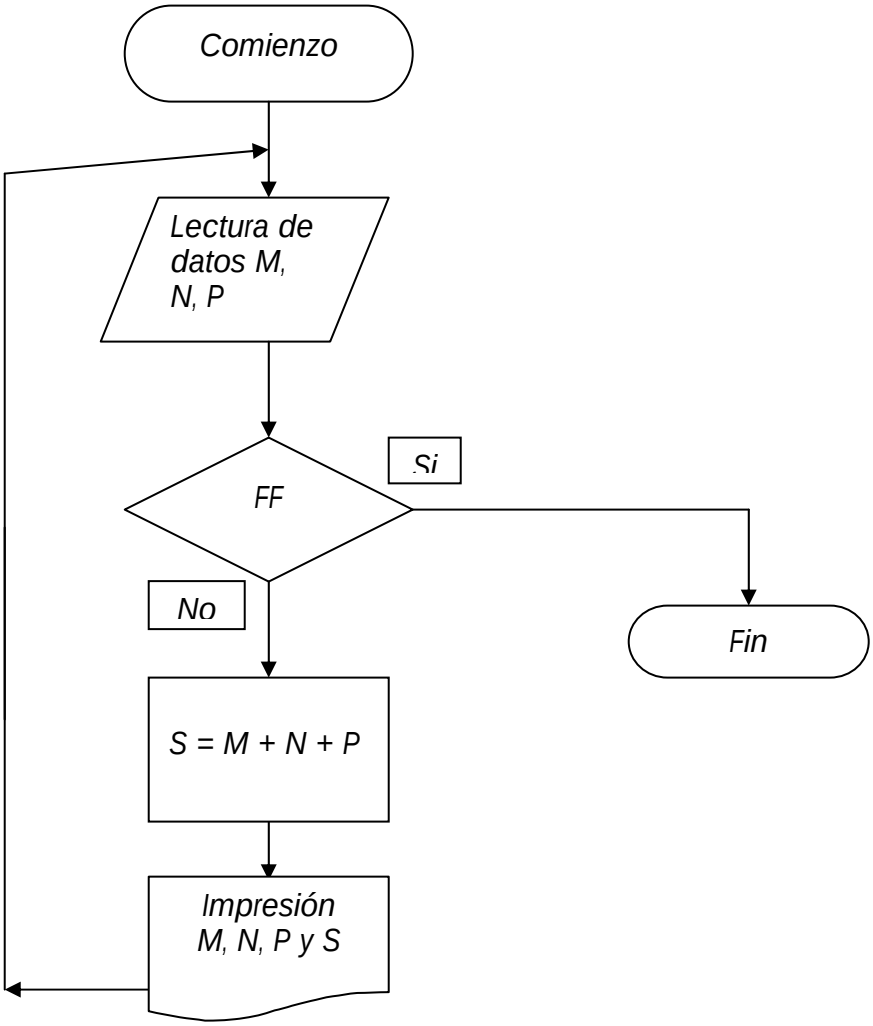
- Entrada de datos e instrucciones previas.
- Lazo o bucle (conjunto de instrucciones que se repiten y ejecutan un número determinado de veces)
- Instrucciones finales o resto del proceso.
- Salida de resultados.

Estos programas precisan instrucciones de bifurcación o control que permite la entrada y salida del bucle.

***Ejemplo 3**: Supongamos que los datos del ejemplo 1 provienen de un fichero o archivo que tiene un número determinado de grupos de tres números, y que se desea obtener la suma e impresión de todos los grupos.*

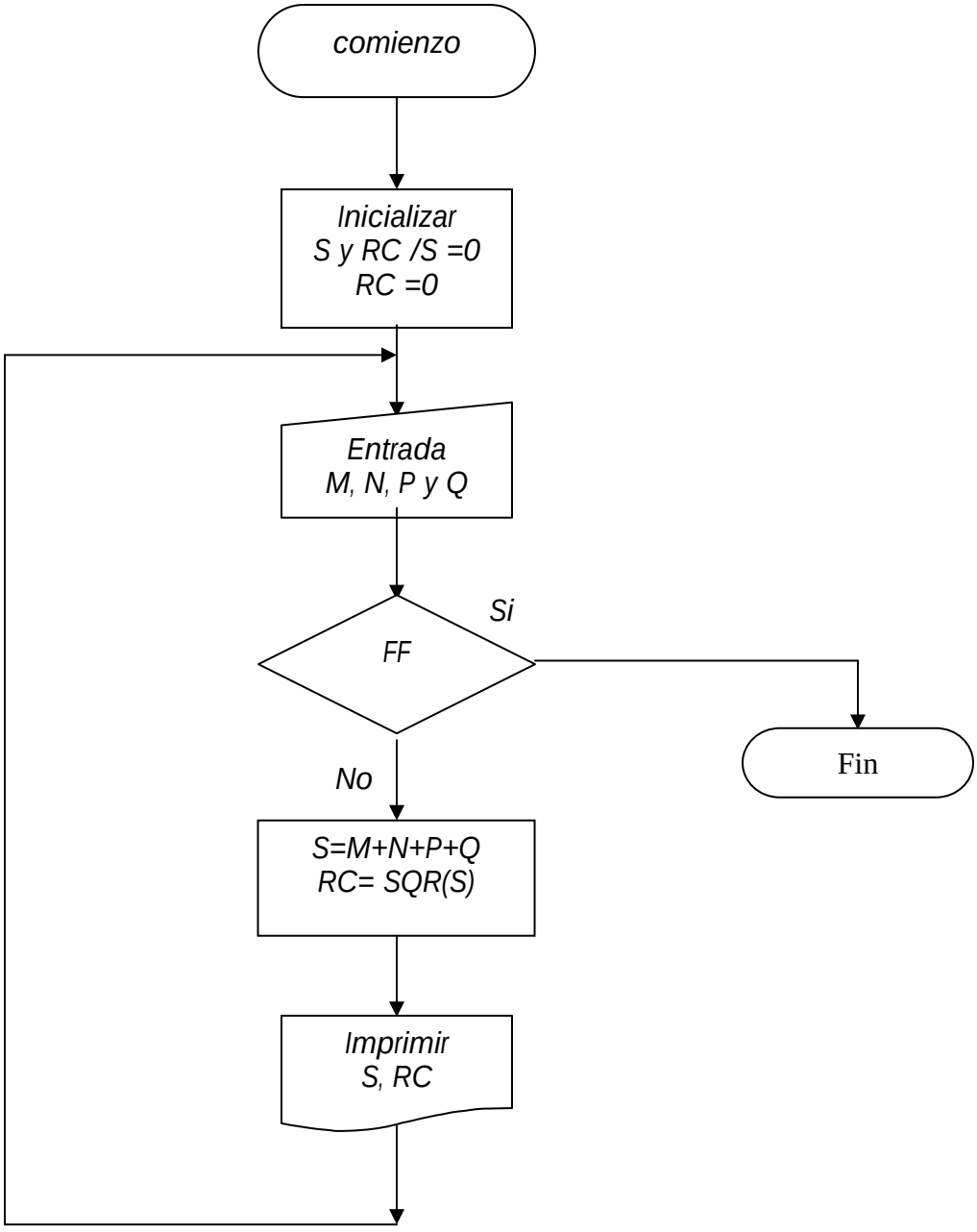
En este caso al tratarse de un fichero, éstos suelen contener siempre un campo en su última ficha que se denomina fin de fichero (FF) y que cuando se encuentra en la lectura supone que se ha alcanzado el final del fichero y por consiguiente no se debe seguir leyendo.

El diagrama de flujo modificado es:



***Ejemplo 4**: Igual que el ejemplo 2, suponiendo que los cuatro elementos M, N, P y Q se encuentran en un fichero.*

*En realidad los tres diagramas adolecen técnicamente de un defecto: la no inicialización de las variables o campos: **S** y **RC**. Estas variables antes de efectuar la operación de suma o raíz cuadrada se les debe dar un valor inicial –en este caso cero– y normalmente esta asignación del valor inicial se suele colocar al principio del diagrama de flujo.*



3.5.3 Programas alternativos

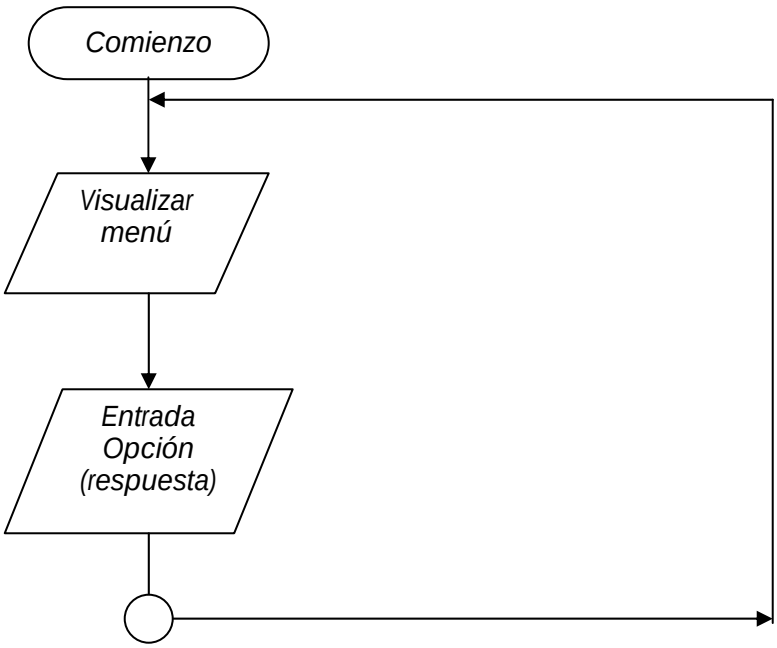
Son aquellos que permiten la ejecución de diferentes operaciones, dependiendo de que se cumpla (o no) determinadas condiciones que se producen en los datos de entrada o durante el proceso. Según la condición que se cumple se realiza una serie de instrucciones diferentes.

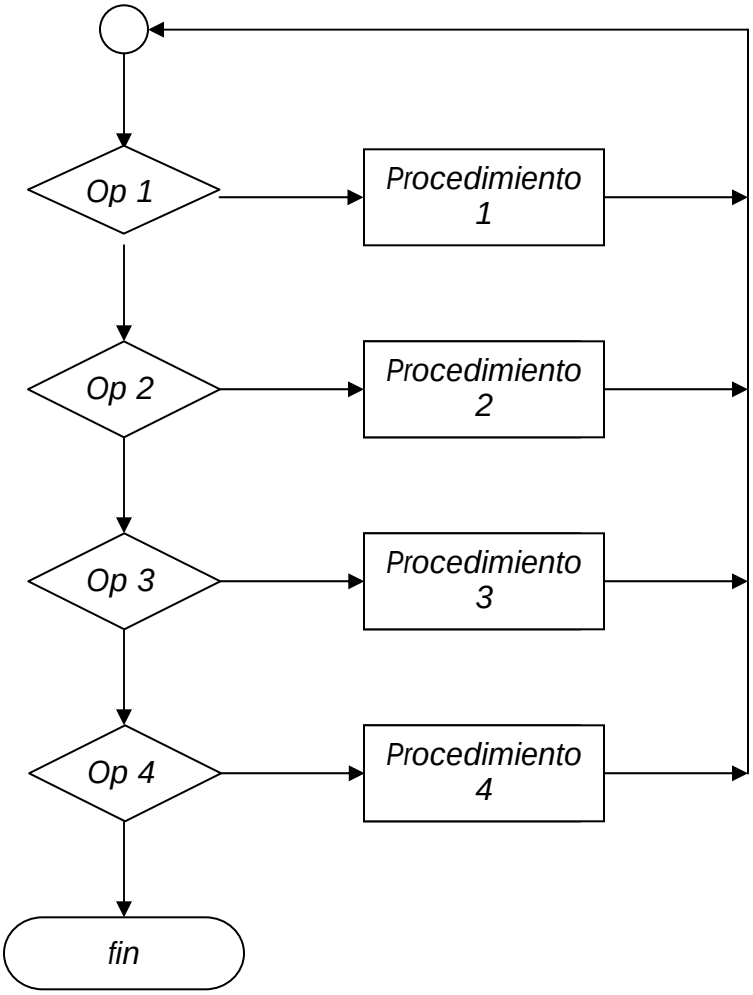
Ejemplo 5. *Se trata de diseñar un diagrama de flujo de modo que se presente un menú al usuario con cuatro opciones posibles; cada opción representa un procedimiento –conjunto de instrucciones independientes–. Después que se ejecute el procedimiento seleccionado, el usuario debe ser capaz de ejecutar otro procedimiento cualquiera –incluso el mismo– o bien de terminar el programa.*

Operaciones a realizar:

- *Visualizar menú de opciones.*
- *Introducir respuesta (selección, opción).*
- *Ejecución procedimiento seleccionado.*
- *Visualizar menú.*
- *Seleccionar otro procedimiento o fin.*

El diagrama de flujo restante se muestra en la figura 3.10.

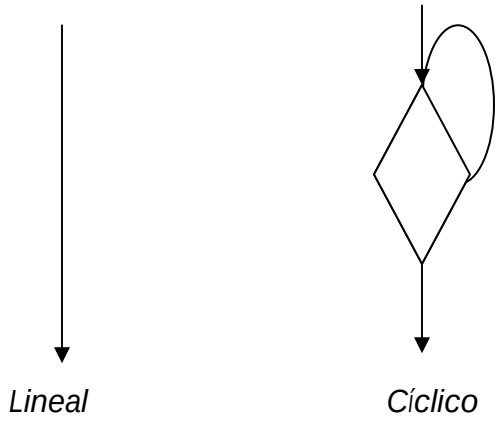


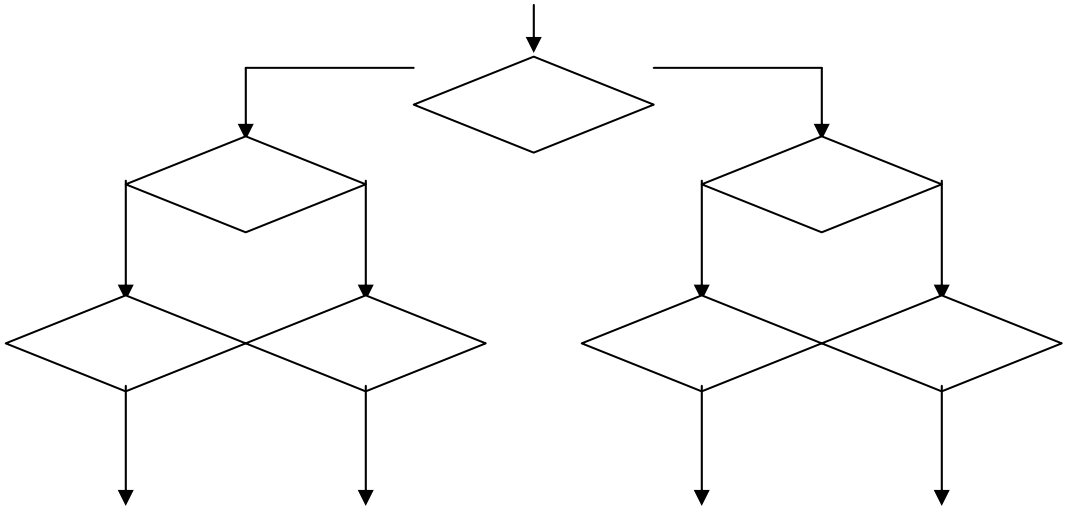


Figuras 3.10. *Programa alternativo.*

3.5.4 Otras representaciones gráficas

A veces se suelen utilizar otras representaciones gráficas que, si bien son muy esquemáticas, pueden dar una idea exacta de las estructuras o secuencias sobre todo desde un punto de vista global.





Alternativo

- PAGINA DEJADA INTENCIONALMENTE EN BLANCO -

CAPITULO 4

TÉCNICAS DE PROGRAMACIÓN

4.1 ELEMENTOS BÁSICOS DE UN PROGRAMA

En los capítulos anteriores hemos introducido al lector en conceptos tales como algoritmos, diagramas de flujo, programas, lenguajes de programación, etcétera. Así, hemos visto cómo en programación se debe diferenciar entre el diseño del algoritmo y el diagrama de flujo que resuelve un problema y su implementación en un lenguaje particular. En consecuencia, debemos diferenciar claramente entre los conceptos de programación y el medio en que son realizados en un lenguaje específico. Sin embargo, una vez que sepamos los conceptos de programación y saber cómo utilizarlos, el aprendizaje de un lenguaje es relativamente fácil. Las técnicas especificas de programación son los objetivos de este capítulo.

Como ya conocemos, las reglas para combinar los elementos básicos de un lenguaje forman la sintaxis del lenguaje. Existen un número determinado de **palabras reservadas**, *que sólo pueden ser utilizadas de un modo limitado: instrucciones o sentencias de programación. (En Pascal además de las palabras reservadas como sentencias, existen los* **identificadores estándar** *que tienen un significado, o bien ser elegidos por el programador.)*

Los elementos básicos cuya correcta combinación permite construir un programa son:

- *Palabras clave e identificadores,*
- *Constantes,*
- *Variables,*
- *Expresiones,*
- *Sentencias de asignación.*

4.1.1 Palabras clave e identificadores

Las palabras clave o reservadas constituyen las instrucciones (órdenes –comandos–, sentencias, funciones y operadores) intrínsecas al lenguaje de programación y son la parte fundamental de su sintaxis.

Los identificadores tiene un significado predefinido (por ejemplo, en Pascal, **sbs**, **input**, **read**, **eoln**, **reset**, **char**, *etc.) o bien ser elegidos por el programador como son los casos de los nombres de variables, programas, etc.*

Las palabras reservadas no pueden ser elegidas como identificadores o nombres de variables. Son palabras reservadas:

BASIC	Pascal
GOTO, READ, INPUT, END…	and, array, type, case, begin…

4.1.2 Constantes

Una constante es una cantidad cuyo valor no cambia durante el proceso, es decir, es un elemento fijo de datos.

Para expresar un constante es preciso escribir su valor, por ejemplo, 1, -25, 43 ó 3.141592. la mayoría a de los lenguajes permitan diferentes tipos fe constantes, siendo las más comunes: enteros, decimales, caracteres y constantes booleanas o lógicas.

Constante entera (integer)

Una constante entera es un número con valor entero, positivo o negativo,

5	-124	465	+12456

Las comas y espacios no se deben utilizar para separar grupos de dígitos en enteros. El número 31.425.427 se debe escribir 31425427.

Constante real (real)

Un decimal o constante real es un número escrito con un punto decimal (número decimal).

41.35	-0.561	32.821	4.0

Obsérvese que si bien 4.0 por su valor es un entero, se considera como constante decimal. Los números reales se pueden expresar en notaciones de punto (coma) fijo y en notación de punto flotante.

Punto fijo	3.141592	0.00000254	-324.567
Punto flotante	0.314192e+1	4.5e-8	-12.4567e2

o sus equivalentes (se representa la potencia de 10)

	0.3141592x10↑1	4.5x10↑-8	-12.4567x10↑2

La notación exponencial de un número real se representa por ↑

Constante decimal: e + n

Donde la n es la potencia de diez a la que se tiene que elevar la constante decimal. En realidad esta constante es un tipo particular de las constantes reales:

e + 5 *equivale a* 10^5

Constantes de caracteres (char)

Es un carácter perteneciente al conjunto de caracteres disponibles. Normalmente, los caracteres disponibles son letras mayúsculas y minúsculas, dígitos, símbolos de puntuación y otros símbolos. Las constantes de caracteres se organizan en series o secuencias de caracteres denominadas cadenas (strings).

En lenguaje Pascal, las constantes de caracteres se escriben encerrando los caracteres en un solo signo de comillas o mejor en apóstrofo.

'A' *'B'* *'*'* *'Hola Flanagan'*

En lenguaje BASIC las cadenas se escriben entre comillas

"A" *"B"* *"*"* *"Hola Flanagan"*

Constante bolean

La constante bolean puede tener dos valores posibles: verdadero (true) y falso (false). Los valores bolean son muy útiles en programación.

4.1.3 Variables

Las variables representan también elementos de datos variables. Mientras que una constante siempre representa un elemento fijo de datos, las variables representan elementos que pueden cambiar durante la ejecución de un programa (por ejemplo, por la acción de una instrucción de lectura de datos). Las variables se refieren en los programas por nombres simbólicos o identificadores.

Dependiendo del lenguaje, existen diferentes tipos de variables, tales como enteras, reales, caracteres, etc.

BASIC	Pascal
Variables numéricas	
• Enteras	integer (enteras)
• Simple precisión	real (real)
• Doble precisión	char (carácter)
Variables de cadena de caracteres	boolean (lógicas)

Tabla 4.1

Una variable o mejor, un cierto tipo de variable puede tomar sólo valores de ese tipo.

Una variable de carácter, por ejemplo, puede tener como valor sólo caracteres, mientras que una variable entera sólo puede tomar valores enteros. Cualquier intento de asignar un valor de distinto tipo a la variable producirá un error.

4.1.4 Expresiones

Se pueden evaluar datos y por consiguiente obtener nuevos valores, evaluando expresiones tales expresiones son combinaciones de constantes, variables, símbolos de operación- operadores-, paréntesis de apertura y cierre y nombres de funciones especiales.

Las expresiones matemáticas tienen igual sentido, por ejemplo;

$$X\,(Y + 5) - 4 * Z + x$$

Aquí los paréntesis indican el orden de calculo, + y – representan adición y resta, y significa la raíz cuadrada de x.

Cada expresión tiene un valor, que se determina tomando los valores de las constantes y variables de la expresión y ejecutando las operaciones indicadas.

4.1.4.1 Expresiones aritméticas.

Las expresiones aritméticas son análogas a las formulas matemáticas. Las variables y constantes implicadas en la expresión son numéricas (enteras o reales) y las operaciones vienen expresadas por los operadores matemáticos.

^	Potenciación	(213=8)
+	Suma	(5+4=9)
+	Resta	(8-3=5)
·	Multiplicación	(5·8=40)
/	División	(15/2 +7.5)
\	División entera	(17 DIV 3= 5 ó bien 17\3=5)
MOD	módulo entero (resto)	(17 MOD 3=2)

Tabla 4.2. Operaciones aritméticas.

Los paréntesis se utilizan también para agrupar términos y asegurarse que las operaciones se ejecutan en el orden correcto.

*A * (B-7) Se efectúa primero B-7 y luego el resultado se multiplica por el valor de B*

En el caso de que se presenten diferentes símbolos y se produzca una ambigüedad en el orden de las operaciones, existe una prioridad o precedencia asociada con cada

operador y la ejecución de las operaciones se realiza por orden de prioridad comenzando por la más alta. En caso de igualdad en prioridad, se procesan los operadores en orden de izquierda a derecha.

La prioridad suele ser similar en los diferentes lenguajes, y se indica en la siguiente tabla:

Operador	Prioridad
^	(Mas alta)
*, /	
+, -	
\, MOD	(más baja)

Tabla 4.3. Prioridad de operadores Aritméticos.

En consecuencia,

A * B +7	es igual que	(A* B) + 7
P + (M/N) +Q	es igual que	P + M/N + Q
((A * B) /C) * D	es igual que	A * B/C * D

Además de los operadores es posible utilizar funciones matemáticas estándar del sistema o definidas por el usuario (en el apartado 4.7 se explicara el concepto de función) como

ABS (X)	valor absoluto de X
SQR (X)	raíz cuadrada de X
SIN (X)	seno de x (en radianes)
LN (X)	logaritmo neperiano de X

En las funciones, X (argumento) puede ser entero o real.

4.1.4.2 Expresiones Booleanas

Una expresión booleana es una expresión en donde existen entre otros elementos, operadores de relación o lógicos y su valor es siempre verdadero o falso. Un sistema para generar expresiones Booleanas es combinar operadores booleanos y relacionales con otros elementos.

AND	A **AND** B	la expresiones verdadera solo si A y B son verdaderos, en caso Contrario es falso
OR	A **OR** B	la expresión es verdadera si A o B es verdadero, y falsa si A y B Son falsos.
NOT	**NOT** A	la expresión es verdadera sólo si A es falso, y viceversa.

Tabla 4.4. Operadores Lógicos.

Ejemplos:

A = verdadero *B = falso*

NOT (A) OR B = Falso OR falso = falso

NOT (A OR B) = NOT (verdadero OR falso) = NOT verdadero = falso.

NOT (A AND B) = NOT (verdadero AND falso) = NOT falso= verdadero

Las expresiones booleanas se pueden expresar también con operadores de relación o comparación (Tabla 4.5)

=	igual
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
<>	No igual

Tabla 4.5. Operadores de relación.

Los operadores de relación se utilizan para comparar expresiones. El formato general de tal comparación es.

Expresión 1	OPERADOR RELACIONAL	Expresión 2

Ejemplos

A =5	B =14	A>B	falso
A =4	B =10	(A – 3) > (B – 5)	falso
A =4	B =10	(A – 3) < (B – 5)	verdadero

Las reglas de prioridad se aplican también en este caso. Todos los operadores de relación tienen menor prioridad que los operadores aritméticos.

REGLAS DE PRIORIDAD en Pascal/ BASIC

COMO EJEMPLO HEMOS ELEGIDO Pascal y BASIC para dar las reglas de prioridad, aunque en los restantes lenguajes suelen ser similares.

.

BASIC	Prioridad Máxima	Pascal
()	↓	()
-		
^		not
*, /		*, /, mod, div, and
+, -		+, -, or
\, MOD		
=, < >, <, >, <=, >=		<, <=, >,>=, =, <>, in
NOT		
AND		
OR		

Tabla 4.6. Prioridad total de operadores

Todos los operadores de una misma línea tienen igual prioridad. Dentro de una misma línea – prioridad igual- , la evaluación de operadores se hace de izquierda a derecha. Los operadores de una línea tienen prioridad sobre los de línea inferior.

4.1.5 Sentencias de Asignación.

Para ejecutar cálculos se necesitan sentencias que le indiquen a la computadora qué acciones ha de ejecutar. La herramienta básica es la sentencia de asignación (en el apartado 3.4.2 se estudio el concepto de asignación). Las sentencias de asignación son una parte fundamental de casi todos los lenguajes de programación, permiten dar el valor de una expresión a una variable. Recordemos que.

A: = B + 1 (Pascal) *A = B + 1 (BASIC)*

Significa que se han de ejecutar las operaciones del lado derecho y asignar el valor obtenido a la variable de la izquierda. El lado derecho de la sentencia puede ser cualquier expresión legal y el lado izquierdo debe ser u nombre de variable. En general, la expresión de la derecha debe tener un valor del mismo tipo que la variable de la izquierda. Si se dan varias sentencias de asignación se supone que se ejecutan en el orden en que están escritas.

No se puede tener una expresión en el lado izquierdo. La sentencia de asignación no debe confundirse con una ecuación matemática o de igualdad aritmética. Por consiguiente: A + 5: = B – 6, no es un formato correcto, aunque la ecuación matemática A + 5 = B – 6 tiene sentido. Por el contrario

N:= N + 1 *N = N + 1*

Es aceptable como sentencia de asignación y no como ecuación matemática; significa que el valor de la variable N se debe incrementar en 1.

El formato de la sentencia de asignación varia de un lenguaje a otro, si bien la idea básica permanece.

Pascal

Variable: = expresión

BASIC estándar

Variable = expresión

O bien

LET variable = expresión

BASIC HP (ANSI)

LET variable 1, variable 2,..= expresión

Variable 1, variable 2,... = expresión

Nota: la expresión en cualquier lenguaje puede ser numérica, de caracteres o boolean.

Ejemplos

1. Escribir como sentencias de asignación las fórmulas del área y longitud de un círculo de radio R.

a↑ = R 2
L = 2 R

Pascal	*BASIC*
*AREA: = 3.141592*RADIO*RADIO:**	*AREA= 3.141592* RADIO *RADIO*
*LONGITUD: = 3.141592*2*RADIO;*	*LONGITUD= 6.283184*RADIO*

** Pascal exige un punto y coma a la terminación de la sentencia.*

2. sentencias de asignación con expresiones de caracter o boolean.

Pascal

DEMO: = 'C';	*correcta si DEMO es de tipo CHAR*
NOMBRE: = true;	*correcta si NOMBRE es de tipo boolean*
M:= (B > C);	*es correcta si Mes de tipo boolean*

4.1.6 Otros elementos.

Además de las constantes, variables, expresiones y sentencias de asignación los programas se componen de otra serie de elementos tales como bucles, bifurcaciones, contadores, acumuladores, interruptores así como los subprogramas que constituyen módulos independientes dentro de los programas.

Todos los elementos básicos unidos adecuadamente mediante las estructuras básicas constituirán los programas.

En lo que resta de capitulo estudiaremos los demás elementos y estructuras básicas que van a permitir el diseño adecuado del algoritmo para la resolución de un problema que convenientemente codificado se convertirá en un programa.

CAMPOS DE MEMORIA.

A lo largo de un proceso se necesita, con frecuencia, almacenar datos, para su posterior recuperación. El almacenamiento de los datos y las operaciones con ellos exige su introducción en memoria, en áreas de almacenamiento específico de la computadora. Las áreas de almacenamiento se denominan campos de memoria.

Los campos de memoria se dividen en los siguientes tipos:

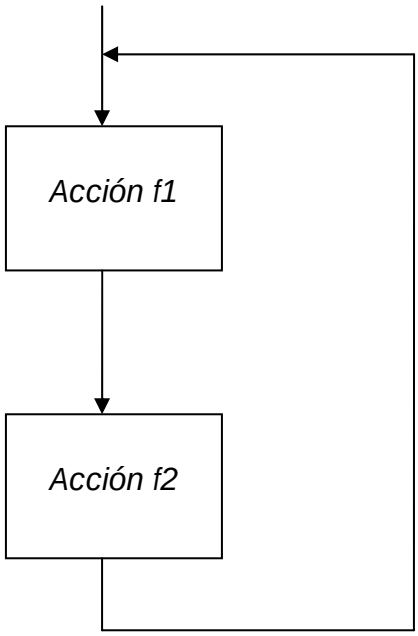
- **Campos de entrada:** *contienen los datos de entrada que aún no han sido utilizados durante el proceso.*
- **Campos de salida:** *contienen los datos de salida que van a visualizar en pantalla o escribir en algún dispositivo.*
- **Campos de trabajo:** *contienen los datos o resultados parciales que se almacenan hasta que son requeridos por el programa.*

Las variables, contadores, etc., que veremos en los párrafos siguientes, se almacenan e campos de memoria.

4.2 BUCLES E ITERACIONES.

Un bucle es un proceso en el que se ejecutan una serie de operaciones un número determinado de veces; las operaciones un numero determinado de veces; las operaciones serán siempre las mismas, pero con datos y resultados diferentes. En el caso de un programa de computadora, el bucle o lazo es un conjunto de instrucciones que deben ser ejecutadas un cierto número de veces, en un proceso iterativo o repetitivo; el bucle constará de una entrada y una salida; la entrada se producirá con una o varias instrucciones y la salida del bucle –fin del proceso repetitivo- se producirá cuado se cumpla una condición.

Un bucle se representa gráficamente así:



Si no se pone condición de salida se permanecerá dentro del bucle indefinidamente, y se conoce este tipo de bucles como bucle infinito o bucle sin fin. La salida del bucle exigirá el cumplimiento de una condición, por ejemplo que una determinada variable tome un determinado valor, X = 999, X = 0, etc.

Una iteración es la repetición controlada de la secuencia de acciones internas al bucle.

En general, un bucle constará de las siguientes partes:

- Preparación o arranque del bucle: una o mas instrucciones que pueden ser: asignación de valores a constantes, contadores a cero, dimensionado de listas o tablas, etc.

- Cuerpo del bucle: grupo de instrucciones que integran realmente el bucle para cumplir el objetivo especificado y que se repiten mientras no se cumple la condición.

- Modificación del bucle: conjunto de instrucciones que modifican el bucle, haciendo progresar su ejecución hasta su terminación final; se suele realizar con contadores, totalizadores.

- Comprobación de la condición: suele ser una instrucción para averiguar si se ha producido la condición que determinar la salida del bucle.

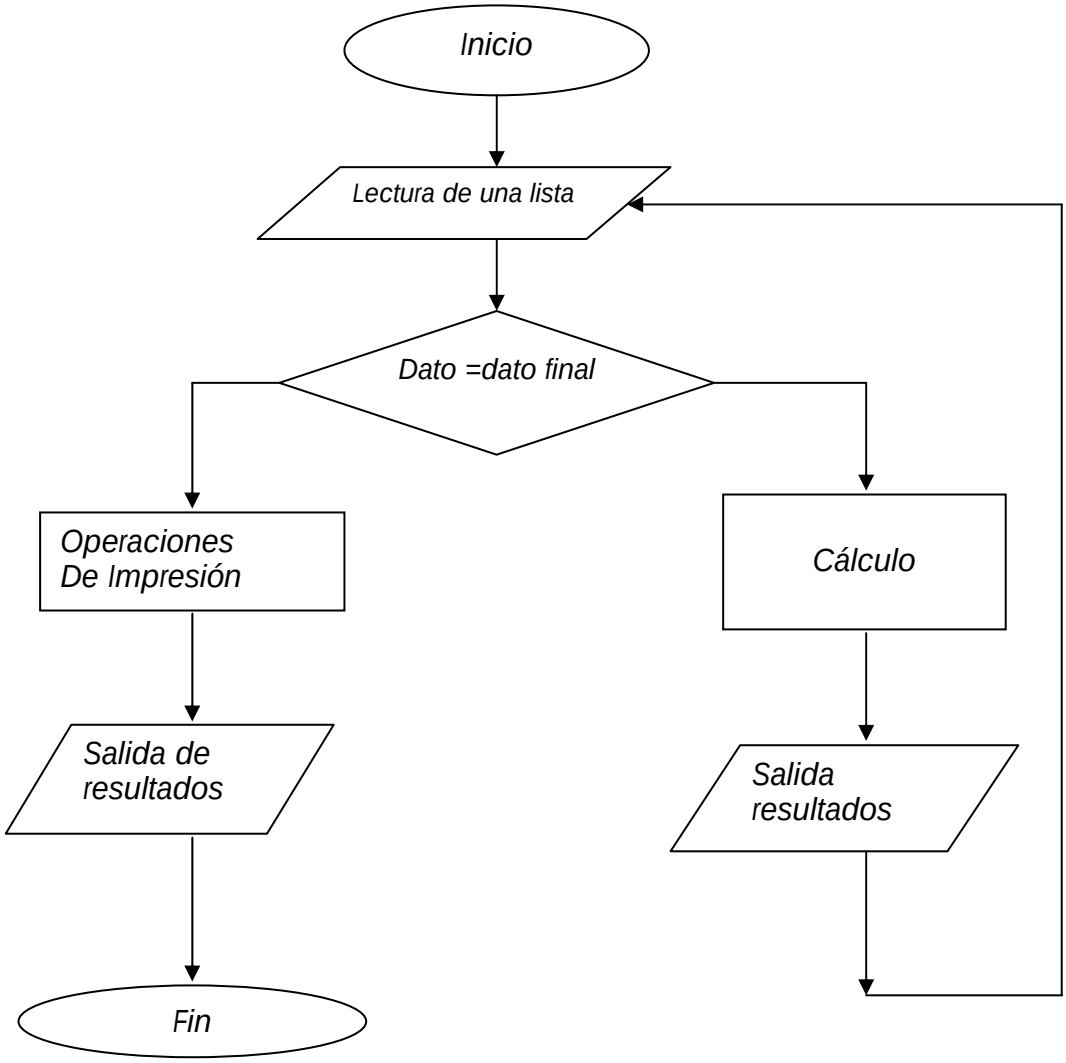
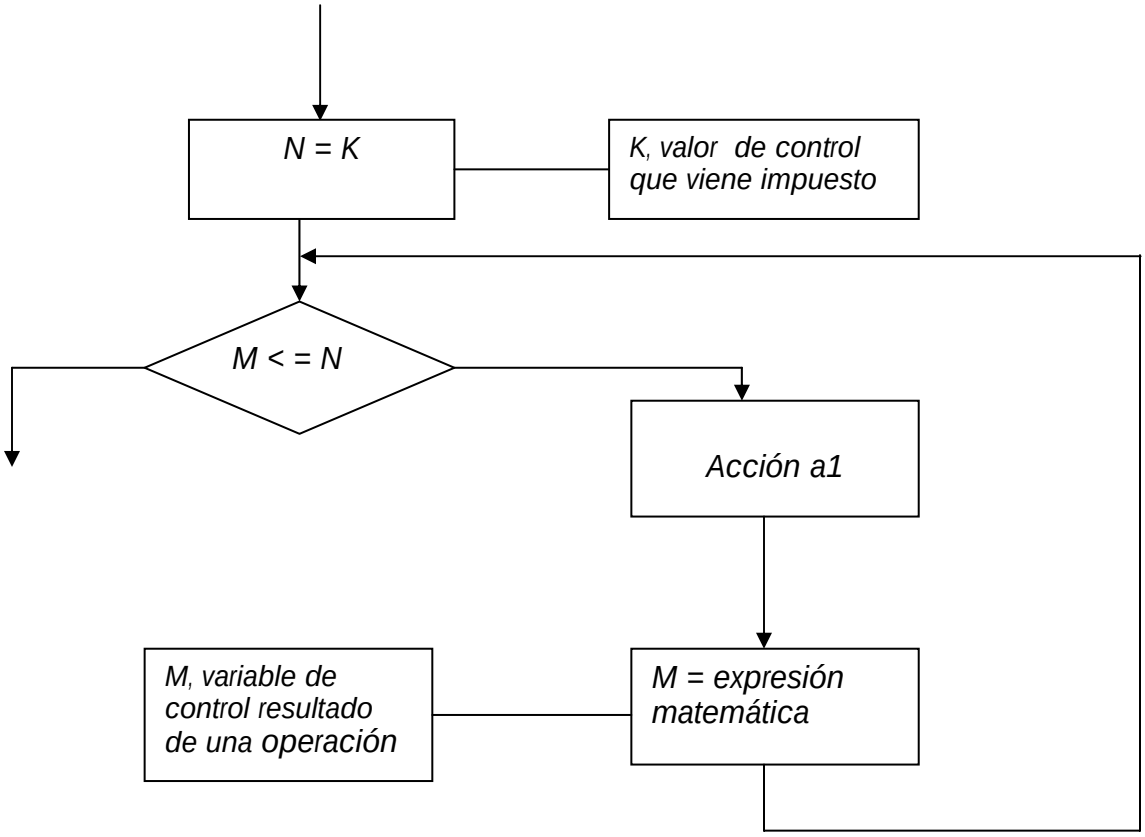


Figura 4.1. Organigrama típico de un bucle.

El valor de los valores de control que intervienen en la condición pueden ser variables y cuyos valores dependan de resultados anteriores a la condición (caso de K) o de operaciones posteriores a la condición (caso de M).



4.3 CONTADORES.

En los procesos repetitivos se necesita normalmente contar los sucesos o acciones internos del bucle, como pueden ser: registros o elementos de un fichero y número de iteraciones a realizar por el bucle. Para realizar esta tarea se utilizan los contadores, cuya construcción es una de las técnicas corrientes en la realización de cualquier diagrama de flujo.

Un contador es un campo de la memoria que está destinado a contener los diferentes valores que se van incrementando o decreciendo en cada iteración.

El campo contendrá las sucesivas sumas parciales que se van realizando en la ejecución de las sucesivas repeticiones. El incremento en las sumas parciales es siempre constante, pudiendo ser positivo o negativo, es decir el contador irá incrementando o decreciendo. El contador se representara en un programa con una variable.

En las instrucciones de preparación del bucle se realiza la inicialización del contador o contadores. La inicialización de un contador consiste en ponerle valor inicial de la variable que representa el contador:

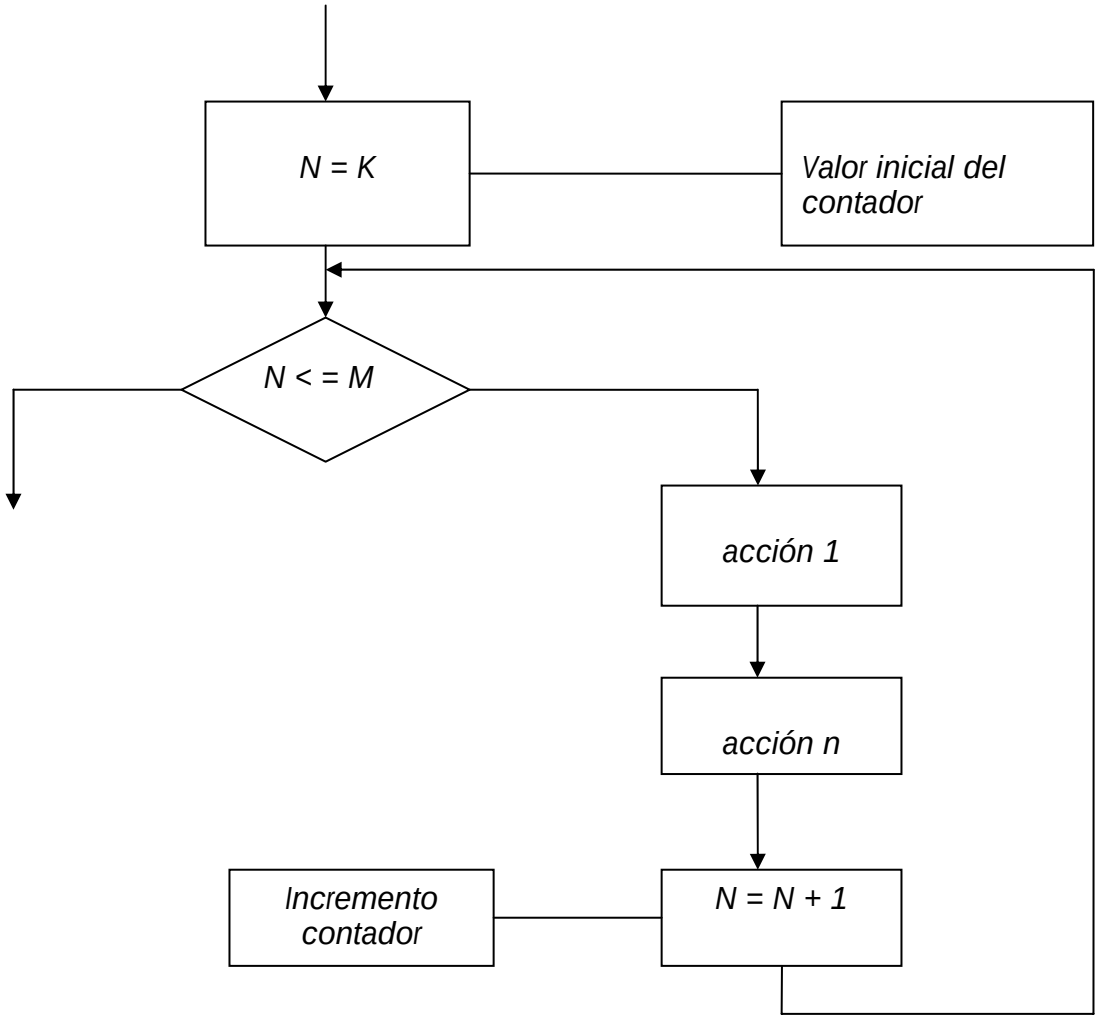
N = 5 I = 3 P = 0

Por consiguiente, el contador se representara por una instrucción de asignación del tipo

N = N + 1 N = N – 1

Siendo 1 el incremento del contador.

Un ejemplo de la aplicación de un contador en un organigrama es:



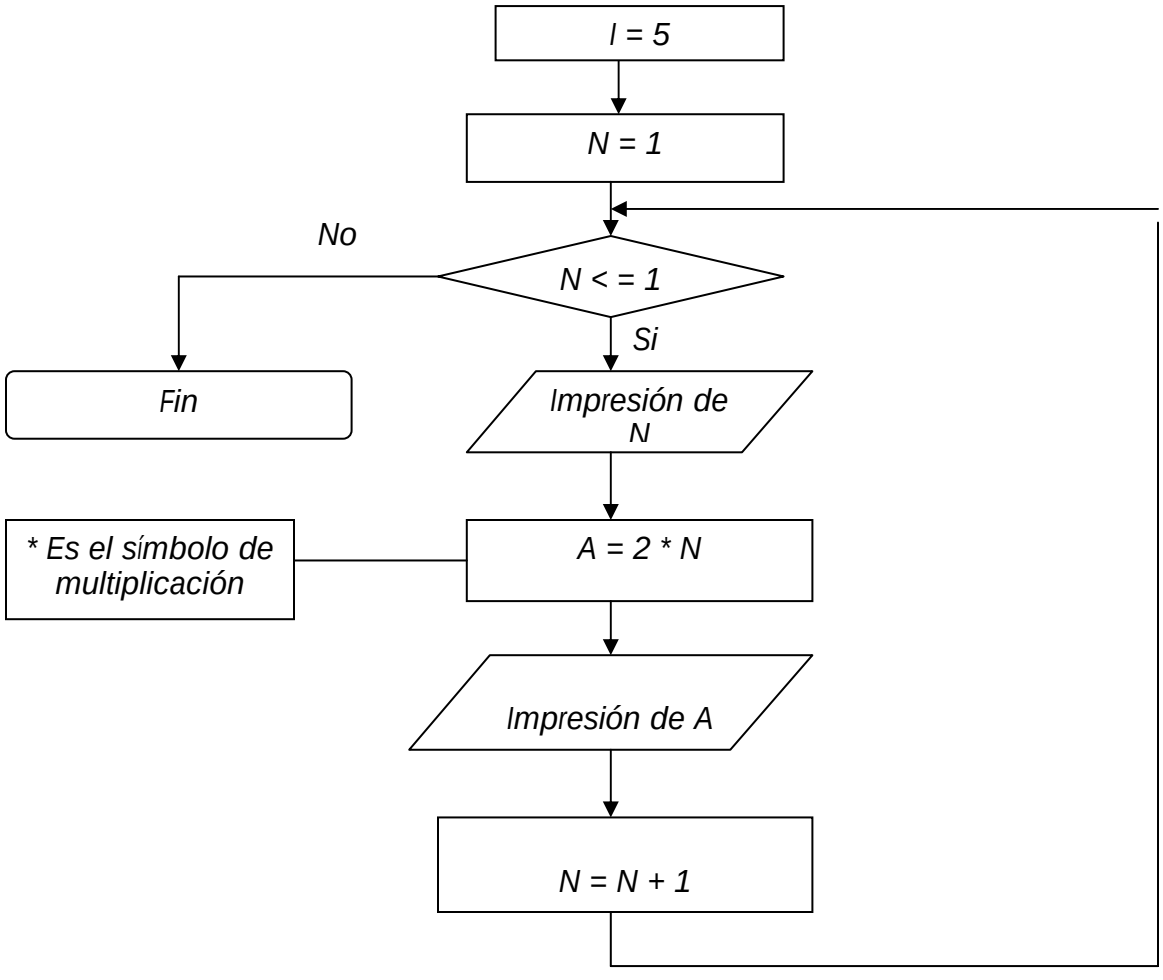
En el ejemplo anterior, los sucesivos valores de N se muestran en la tabla 4.7 para los diferentes valores de I.

caso	valor inicial N	incremento I	valores sucesivos de N
A	0	1	1, 2, 3,4,.....
B	5	1	6, 7, 8, 9,....
C	6	3	9, 12, 15, 18,...
D	8	-1	7, 6, 5,4, 3,...
E	15	-3	- 12, - 9, - 6, - 3,…

Tabla 4.7. Valores sucesivos de N.

caso	valor inicial	incremento I	valor de M	salida cuando M =
a	0	1	10	11
b	5	1	25	26
c	6	3	15	16
d	8	- 1	- 4	- 5
e	15	- 3	0	- 1

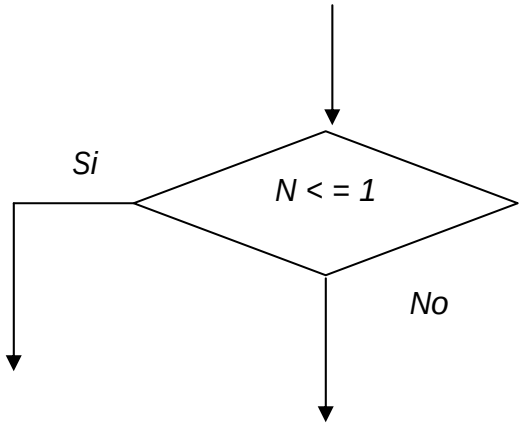
Un ejemplo especifico de un bucle, con la variable N como contador y la constante I = 5 como incremento del contador.



El valor de N se incrementa de 1 en 1 (los valores sucesivos son 2, 3, 4,...) y la salida se produce cuando N vale 6. Las sucesivas salidas de A y N son:

N	*A*
1	*2*
2	*4*
3	*6*
4	*8*
5	*10*

Si por el contrario la condición es:



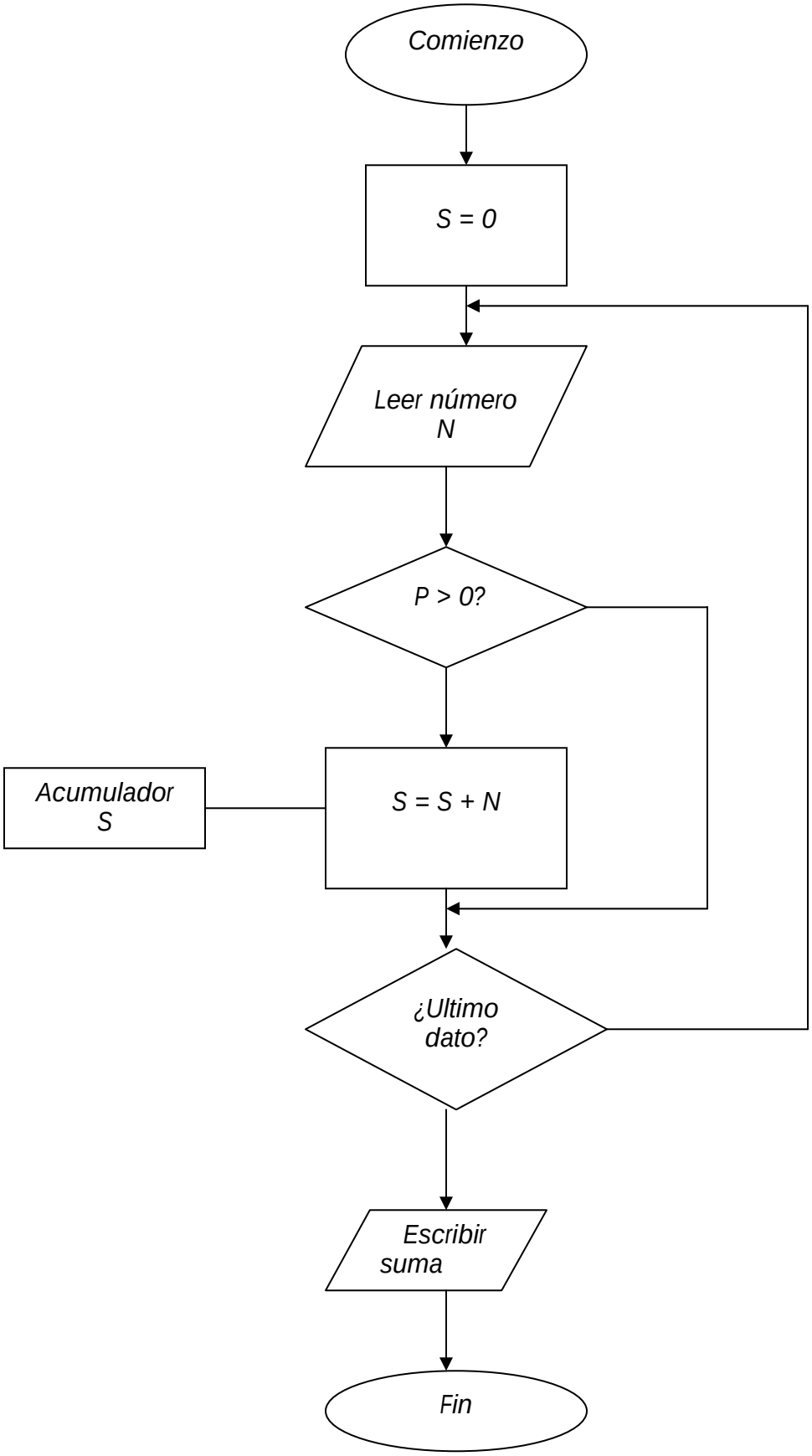
El bucle no se realizara nunca y se saldrá de la condición terminándose el diagrama, ya que en la primera pasada como N = l e l = 5, resulta que N < 5 y por consiguiente la respuesta es <<no>>.

4.4 ACUMULADORES.

Un acumulador o catalizador es un campo o zona de memoria cuya misión es almacenar cantidades variables resultantes de sumas sucesivas. Realiza la función de un contador con la diferencia que el incremento o decremento de cada suma es variable en lugar de constante como en el caso del contador.

Se presenta por S = S + N donde N es una variable y no constante.

El diagrama de flujo lee un conjunto de datos numéricos y obtiene la suma de todos aquellos números que sean positivos:


Comienzo
S = 0
Leer número N
P > 0?
Acumulador S
S = S + N
¿Ultimo dato?
Escribir suma
Fin

Ejemplos de Aplicación:

Diagrama 1. Diagrama de flujo que suma todos los números contenidos en un fichero (lista de datos).

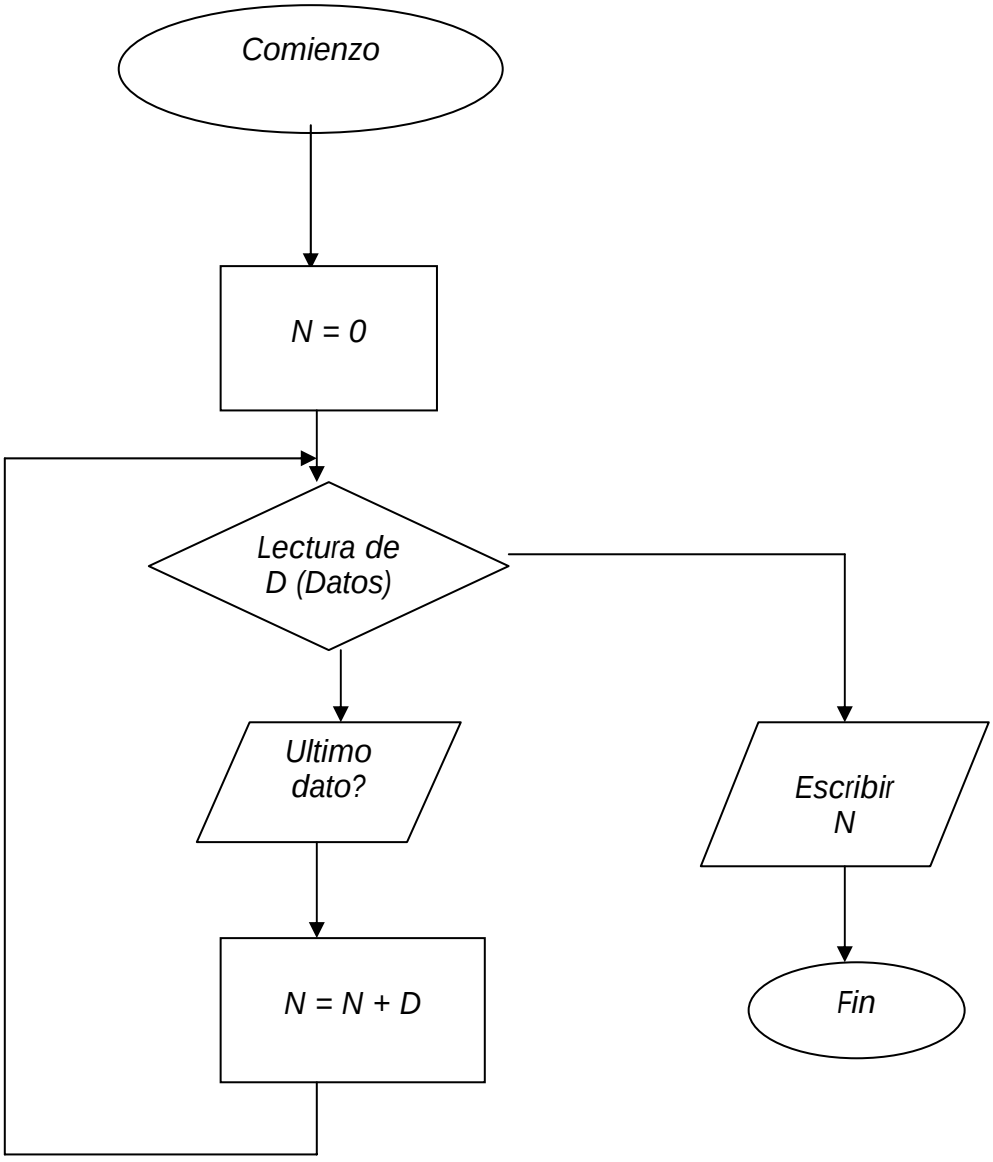
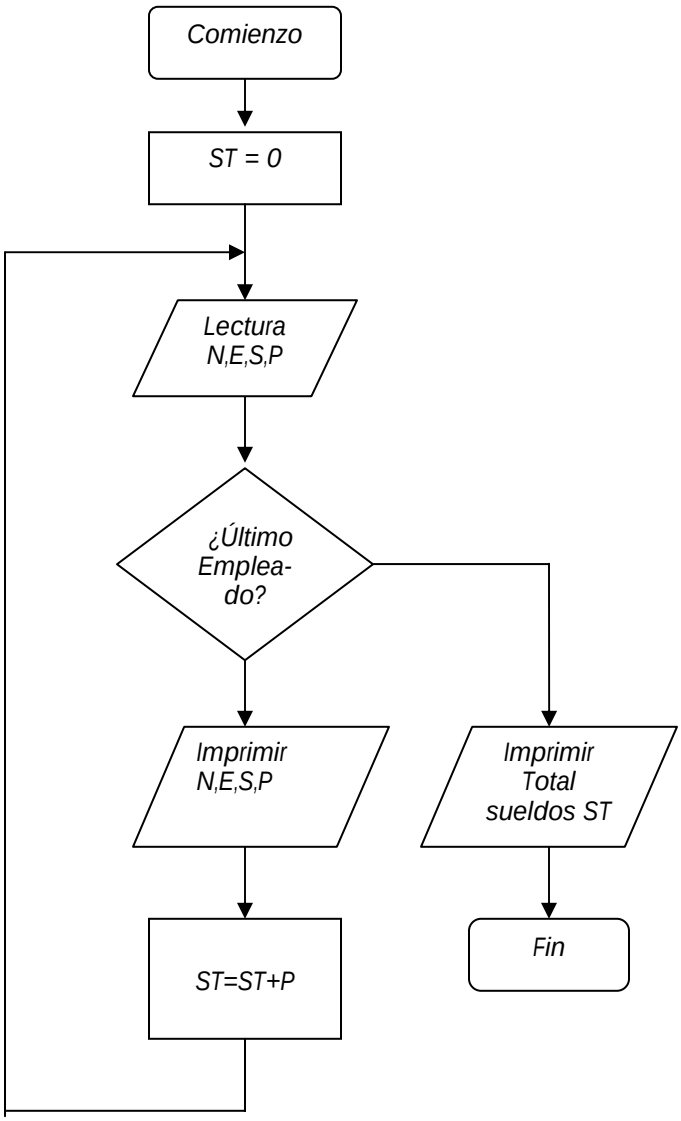


Diagrama 2. Lectura de fichero de datos de empleados de una empresa (nombre, edad, número de seguridad social, sueldo), e impresión de los datos personales de cada empleado así como la suma total de sueldos que paga la empresa:



4.5 BIFURCACIONES.

Las instrucciones de un programa se ejecutan, como ya sabemos, de un modo secuencial. Sin embargo, en numerosas ocasiones es preciso romper el orden secuencial de ellas y bifurcar, saltar o transferir el control a otras instrucciones del programa que no sean consecutivas a las que en ese momento se ejecutan.

Las instrucciones que siguen a la que se salta se volverán a ejecutar secuencialmente hasta la aparición de otra instrucción de bifurcación o el fin del programa. Como se verá posteriormente en le caso de las subrutinas, en le caso de instrucciones de bifurcación

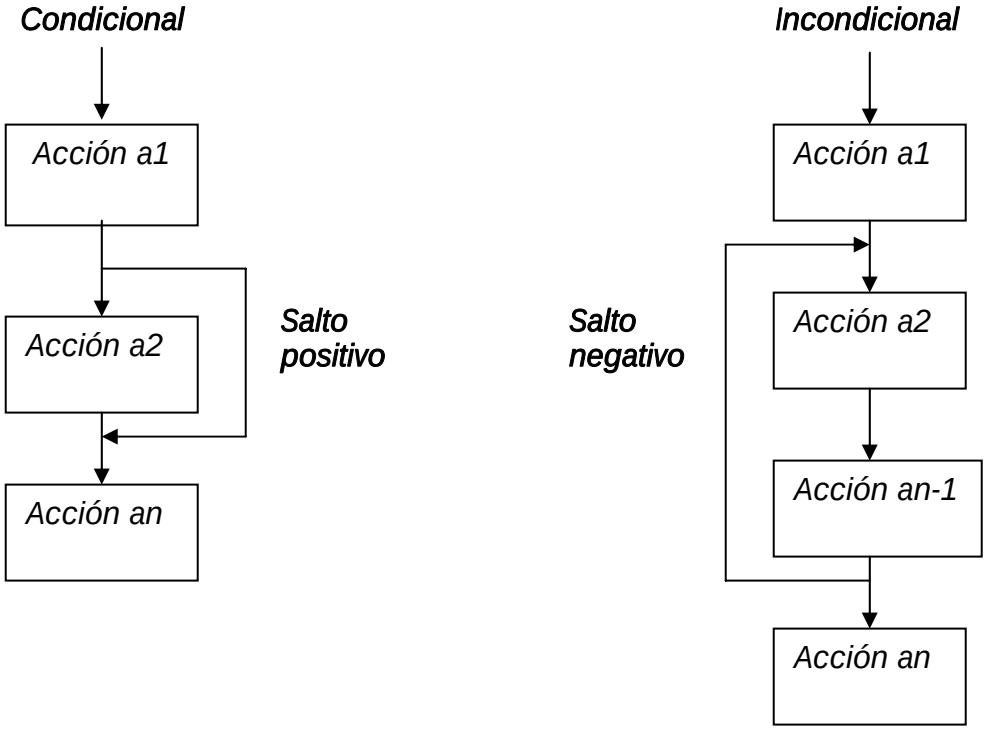
es posible conservar la dirección de retorno al programa principal, tras ejecutar las instrucciones a donde se haya transferido el control. Los saltos o transferencias del control pueden ser positivas (hacia delate) o negativas (hacia atrás), lo que puede implicar la no ejecución de instrucciones ---saltos positivos--- o repetición de instrucciones ---saltos negativos---.

Existen dos tipos de bifurcaciones:

- **Condicionales:** la bifurcación depende del cumplimiento de una determinada condición; cuando se cumple la condición el control del programa bifurca a la instrucción especificada; si la instrucción es falsa el programa continúa ejecutándose en la siguiente instrucción en el orden secuencial previsto.

- **Incondicionales:** la bifurcación se realiza siempre que el programa pase por la instrucción sin el cumplimiento de ninguna condición.

La expresión de ordinogramas de las instrucciones de bifurcación son:



Otra clasificación de las bifurcaciones es en función de la existencia de retorno o no a la instrucción siguiente a la que realiza el salto.

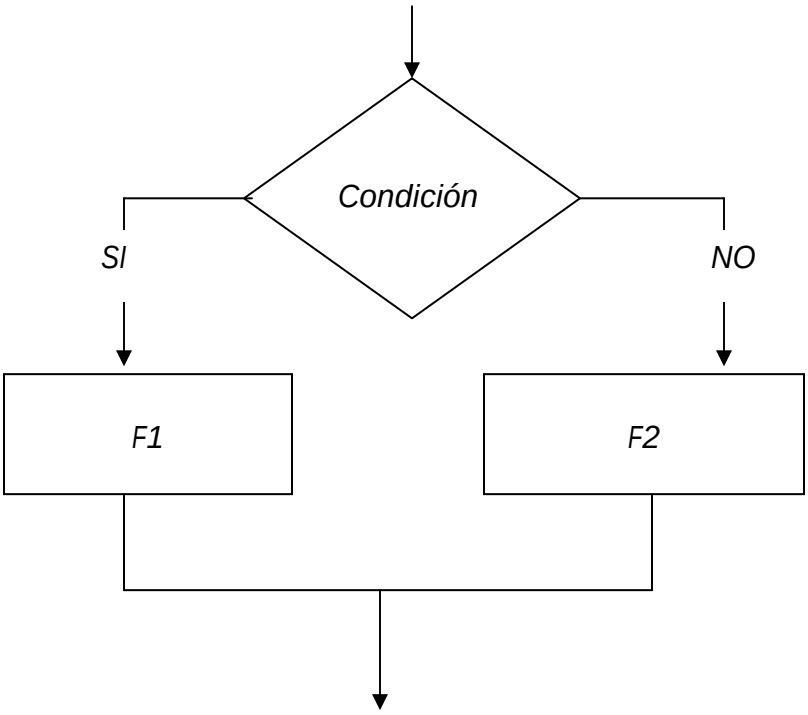
Existen dos tipos de clasificar las bifurcaciones:

- **Controladas:** son aquellas que una vez terminadas de ejecutarse las instrucciones siguientes a la bifurcación, el control del programa retorna a la

instrucción siguiente a la que produjo la bifurcación; para ello se guardará la dirección de retorno en la memoria (en unas zonas o memorias llamadas pilas) cuando se realiza el salto y al terminar las instrucciones de transferencia se recupera dicha dirección de retorno para poder continuar en ella el flujo normal del programa.

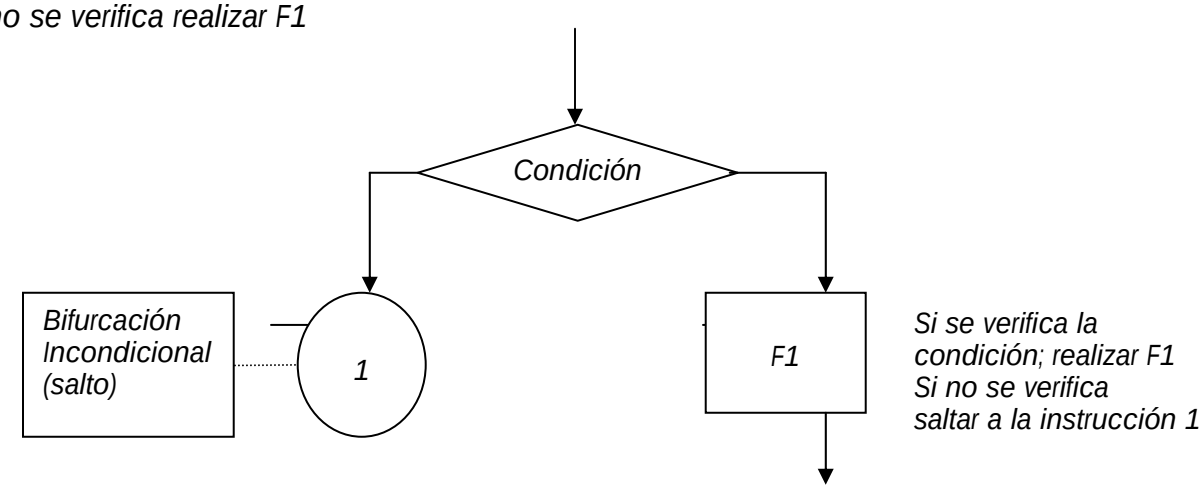
- ***Dirigidas**: son aquellas que para retornar el control es necesario generar una nueva bifurcación que dirija el flujo del programa a la siguiente instrucción en la secuencia ordinaria.*

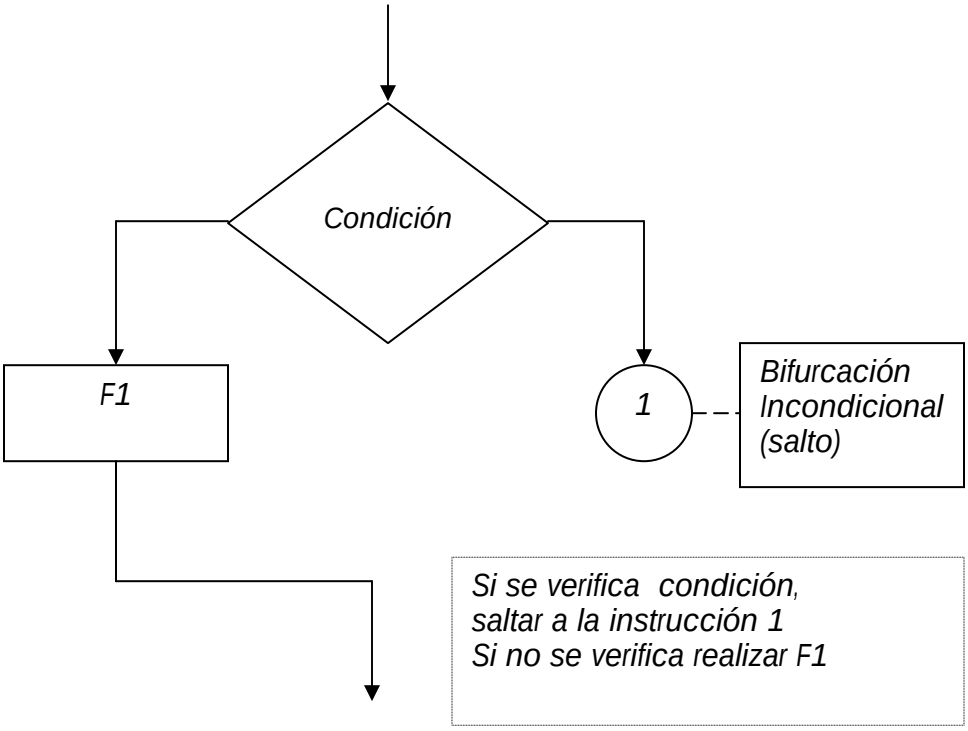
Condicional

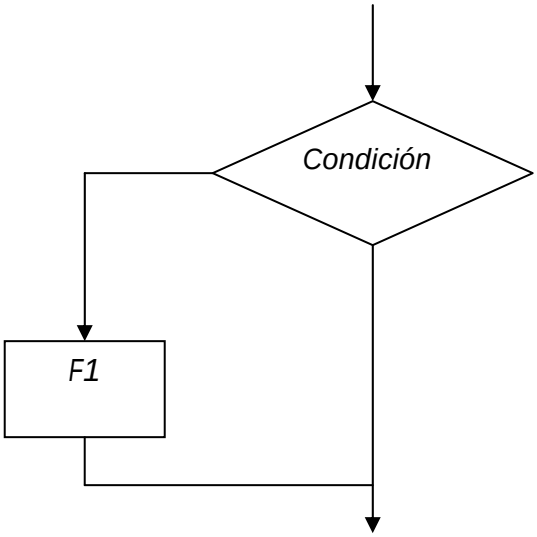


SI se verifica la condición realizar la condición realizar F2

SI no se verifica realizar F1

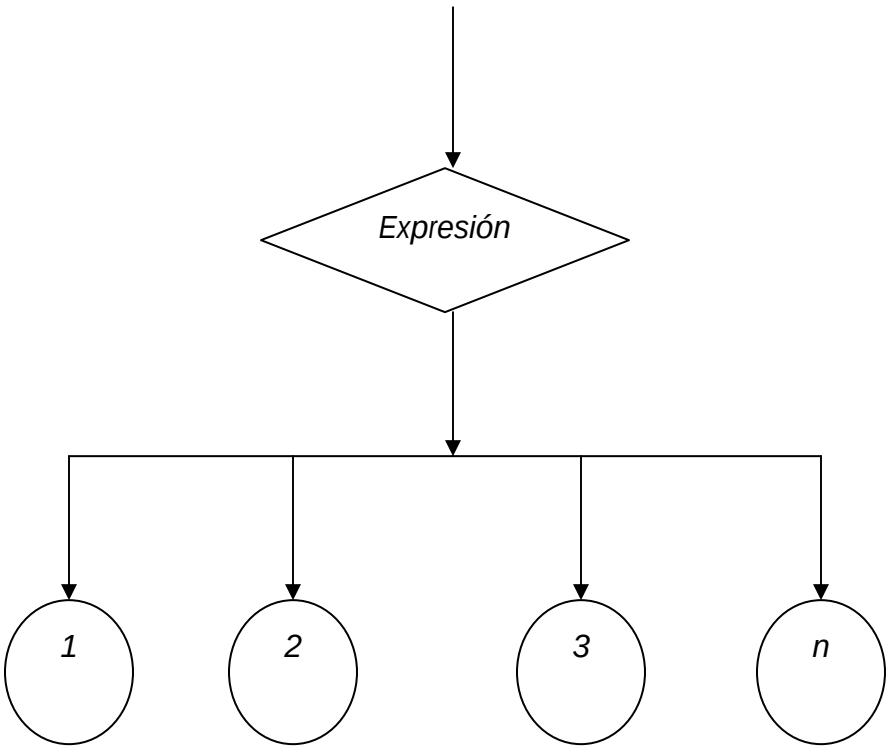






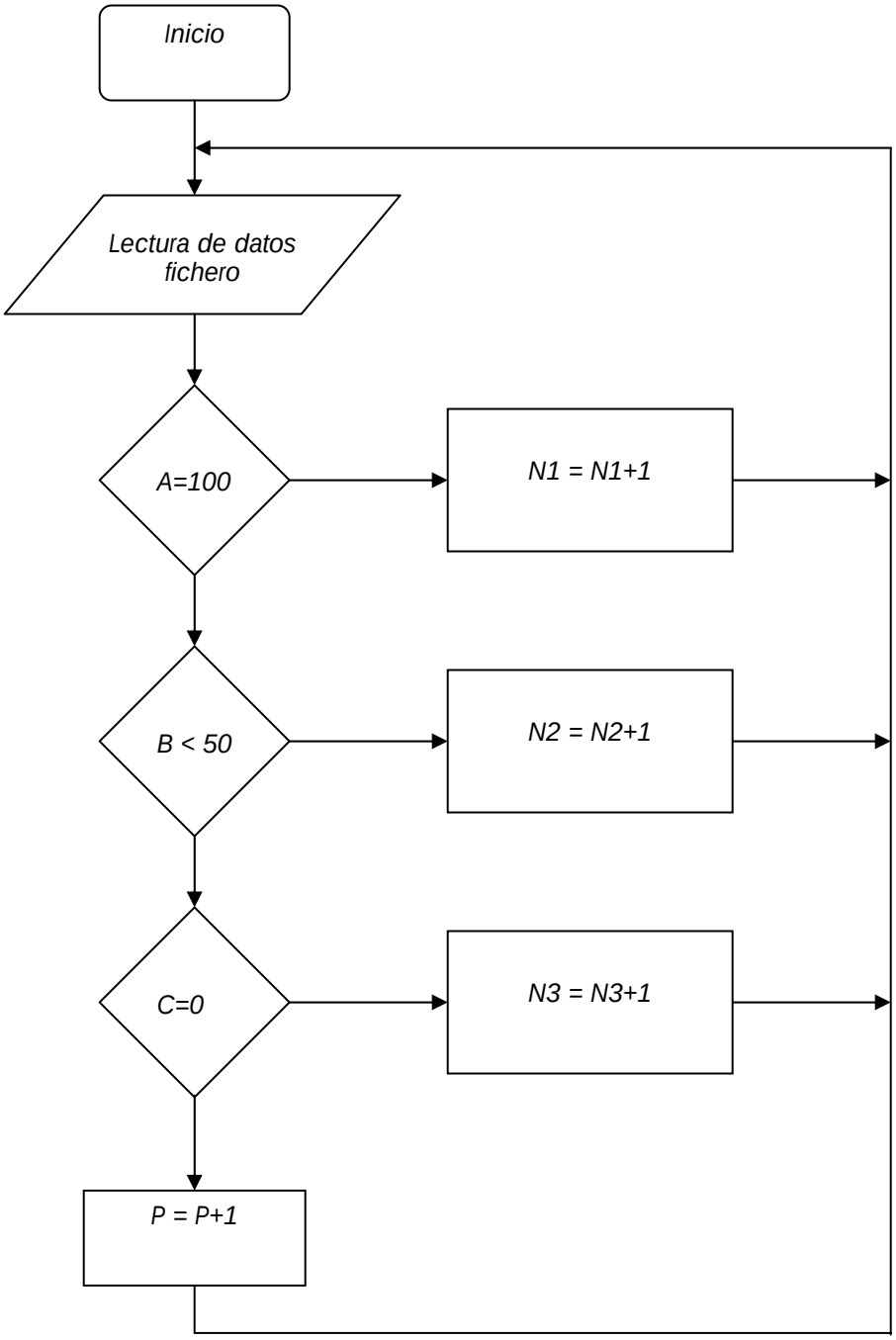
Si se verifica la condición
realizar F1

Si no continuar secuencia



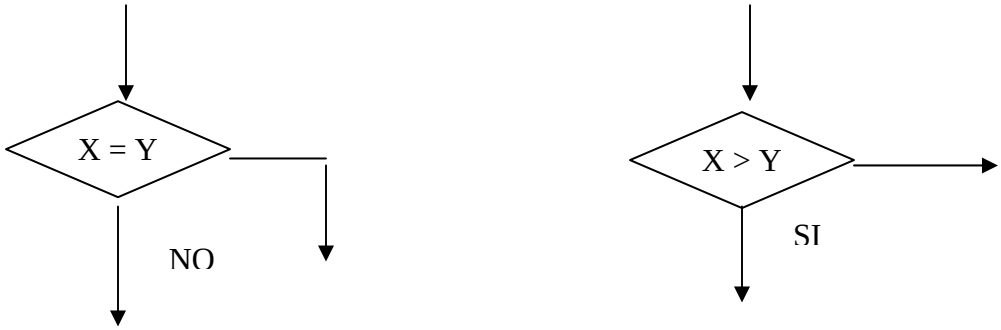
4.5.1 Bifurcaciones anidadas

Las bifurcaciones condicionales y no condicionales se pueden anidar (situarse en el interior de otras) como indica el siguiente diagrama de flujo (próxima pagina);



Nota: las condiciones en las bifurcaciones pueden ser expresiones aritméticas, lógicas o racionales.

Relacionales.



4.6 INTERRUPTORES O CONMUTADORES.

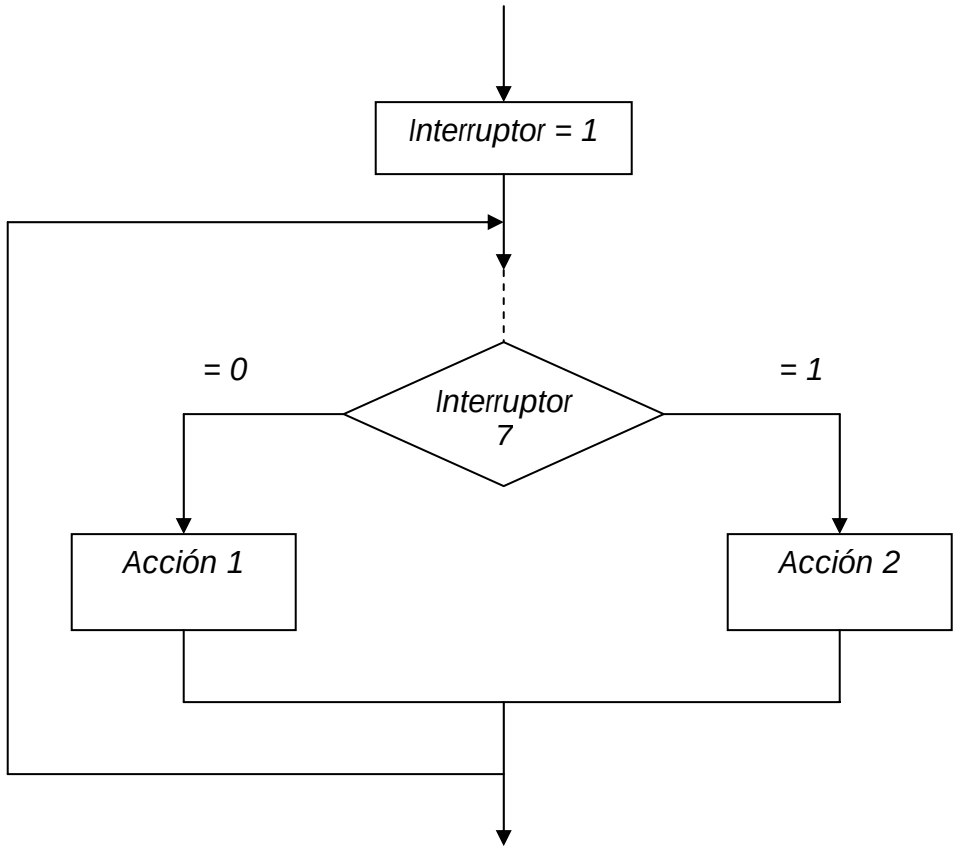
Un interruptor o conmutador ("switch") ---a veces se les llama centinelas, banderas, "flags"---es un campo de memoria (variable) que toma diversos valores a lo largo de la ejecución del programa y que permiten comunicar información de una parte a otra del mismo, es decir variar la secuencia de ejecución de un programa, dependiendo del valor que tenga en cada momento. Los dos únicos valores que puede tomar un interruptor son 1 y 0 (encendido y apagado, abierto y cerrado).

Los interruptores suelen intervenir en las bifurcaciones condicionales cuando el salto se debe demorar a un momento en que no se disponga la información original, en lugar de realizarlo inmediatamente que está presente la condición.

La técnica del interruptor es:

Reserva de una posición de memoria para contener una constante que posteriormente se utilizará en operaciones de comparación. A esta constante (IN, CON, SW, etc.) se le puede dar el valor 1 (interruptor on, cerrado) o el valor 0 (interruptor off, abierto), pero previamente se la ha asignado un determinado significado, por ejemplo IN = 1 puede significar que la ficha leída de un registro ocupa posición par y si IN = 0 significa que la ficha leída es impar. Otro ejemplo de posibles significados podría ser: IN = 1, condiciones iniciales constantes, IN = 0, condiciones iniciales variadas. El interruptor puede activarse (asignarle un valor) tras realizar una prueba y en función del resultado; así por ejemplo si se esta examinando un fichero de estudiantes, y el interés reside en los estudiantes de Medicina, se puede asignar IN = 1 si estudia Medicina el IN = 0 en el resto de los casos. De este modo el proceso de los datos de los estudiantes de Medicina se identificará rápidamente, preguntando si IN = 1.

En esencia un interruptor permite dirigir el flujo de los procesos con dos ramas paralelas, en un sentido o en otro dependiendo de la ocurrencia de una situación determinada (IN = 1) o de la contraria (IN = 0).

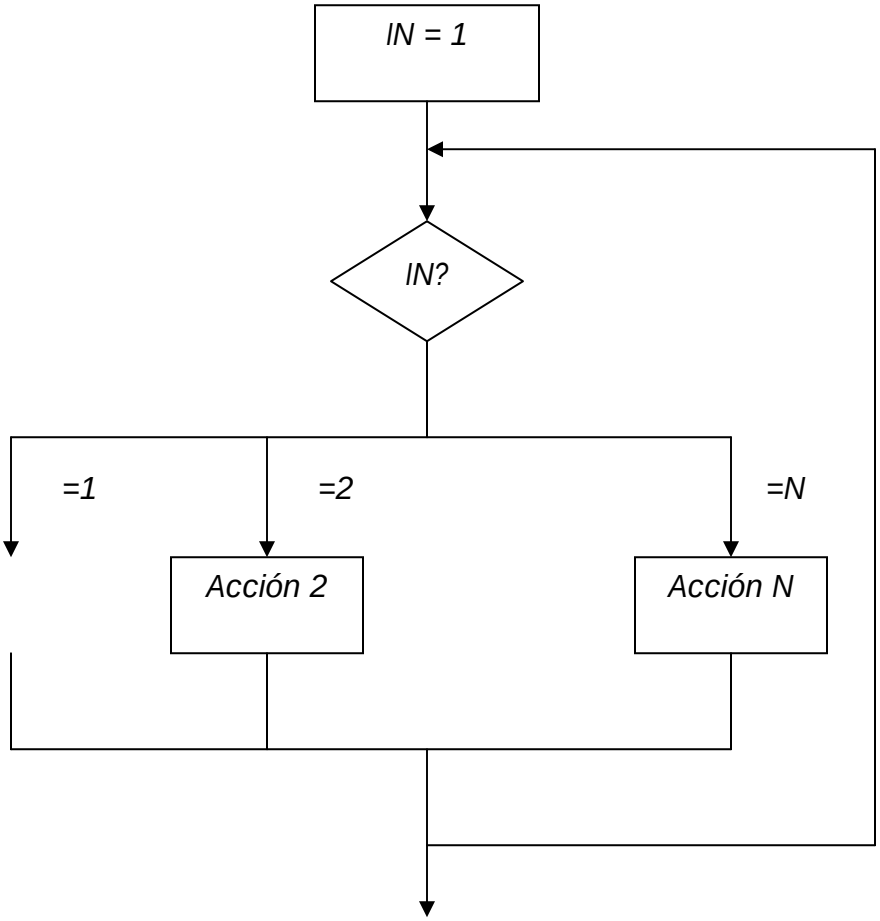


El diagrama que representa el interruptor es:

En el caso de tratamientos de ficheros secuenciales en lenguaje BASIC, existe un caso típico de interruptor: la función EOF. Esta función toma el valor 0 si el fichero no se ha terminado de leer ---quedan registros o fichas--- y el valor -1- cuando se ha leído todo el fichero y no quedan fichas o registros.

Variantes del interruptor

El uso del interruptor binario (dos estados) se puede ampliar a un interruptor múltiple, en el caso de que el programa tuviese n ramas en paralelo. En este caso dependiendo del estado de un suceso determinado que pudiese tomar n estados distintos, se podría utilizar un interruptor múltiple.



4.7 SUBRUTINAS O SUBPROGRAMAS.

La programación estructurada consiste en descomponer sucesivamente un problema en módulos que se tratarán y programarán independientemente unos de otros.

Consideramos las siguientes dos situaciones:

1. *En cierto programa, diferentes listas o tablas (arrays) de números se desean clasificar por orden ascendente y luego descendente---es decir, primero la lista A, luego la B y luego la C---.Conocemos un algoritmo de clasificación ---por ejemplo el "bublesort"(burbuja) o el "quicksort" (rápida) por lo que escribiremos en primer lugar el código (programa) para clasificar A, a continuación se repite el proceso , con los cambios adecuados, para clasificar B y C. Esto esta claro, pero requiere el uso de códigos casi iguales para las tres listas. Podríamos ahorrar esfuerzos si fuera posible escribir el código solamente una vez y a continuación aplicarlo a A, B y C (o a las listas que se desee).*

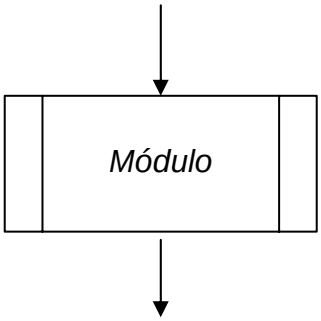
2. El programa jefe o el analista de aplicaciones de un gran proyecto desea dividir el trabajo entre varios programadores. Para evitar confusiones, él o ella deben hacer cada parte tan independiente como sea posible de los demás. ¿Cómo puede un programa, que es una tarea sencilla, ser dividida en dos subproblemas independientes?

Un medio de solucionar estos dos problemas es dividir el programa en partes que puedan ser desarrolladas por separado y eventualmente integrarse en una unidad. Tales partes independientes se llaman ***módulos*** *o* ***subprogramas****.*

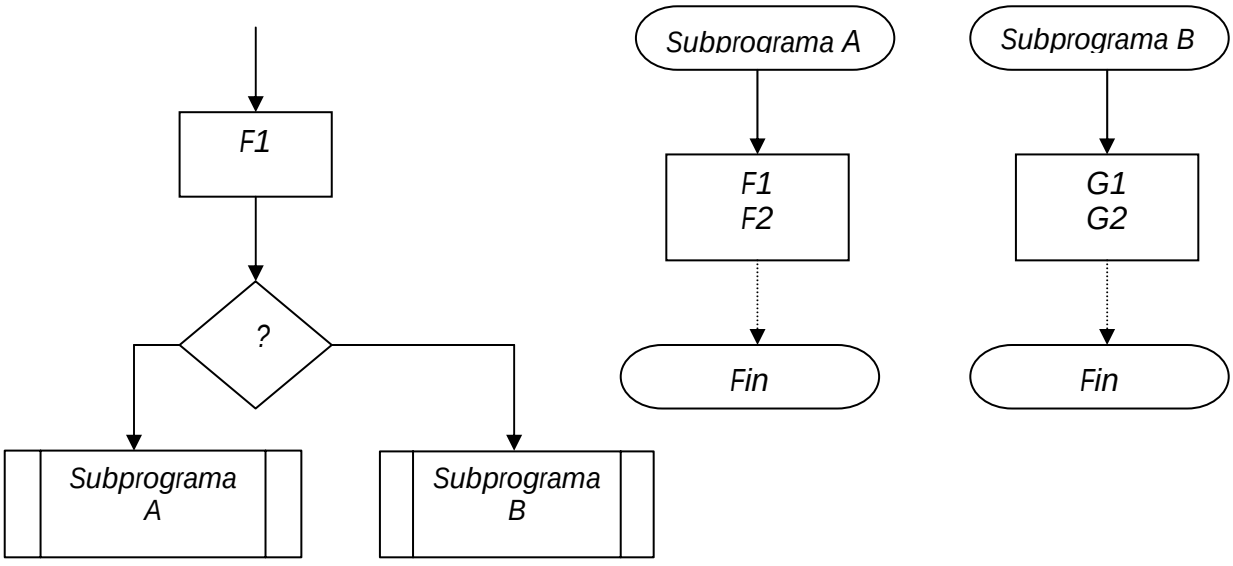
El tratamiento independiente de un módulo se puede hacer normalmente por 2 motivos distintos:

- *Los módulos se repiten varias veces dentro de un mismo programa;*
- *Los módulos efectúan una tarea de gran importancia cada uno de ellos.*

Así un módulo se puede representar por:



En la integración del programa completo o principal se podrá representar gráficamente.



Un subprograma puede ejecutar las mismas acciones que un programa: (1) aceptar datos, (2) ejecutar cálculos y (3) devolver o proporcionar resultados. Sin embargo un subprograma se utiliza por el programa para un fin en específico. El subprograma recibe datos del programa y le devuelve resultados. Podemos imaginar que el subprograma es el jefe de una empresa o una unidad militar que da ordenes a un subordinado y cuando la tarea ha sido terminada éste le informa del resultado de la misma. Se dice que el programa llama o invoca "call) al subprograma. El subprograma ejecuta una tarea, a continuación retorna o devuelve "return", el control al programa. Esta acción puede suceder en diferentes partes del programa. Cada vez que el subprograma es llamado, el control vuelve a el lugar de donde fue hecha la llamada ---bifurcación controlada (figura 4.2).

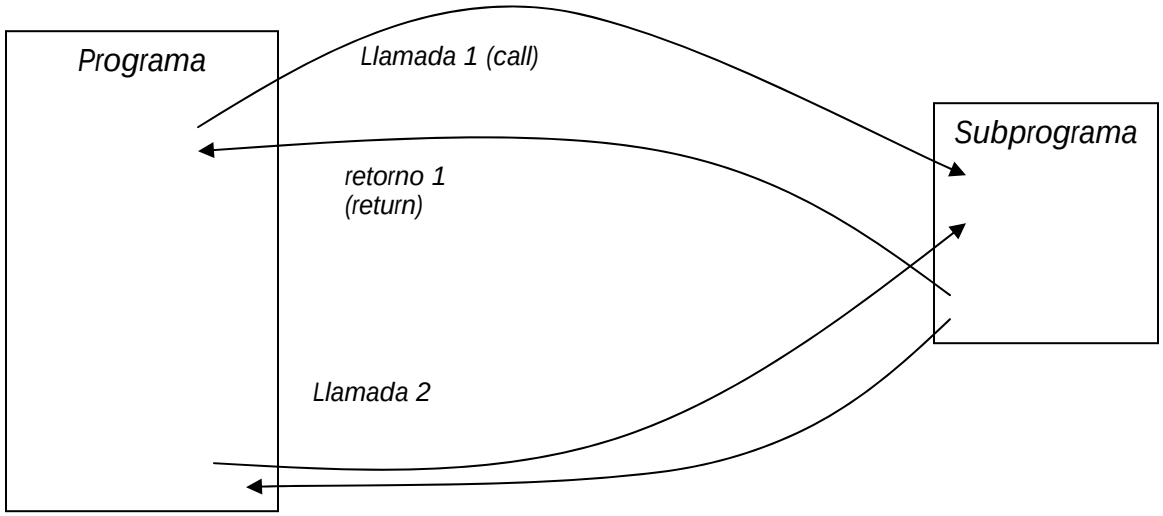


Figura 4.2

Un subprograma puede también tener a su vez uno o varios subprogramas que se suelen conocer como subprogramas anidados o bien de niveles jerárquicos.,

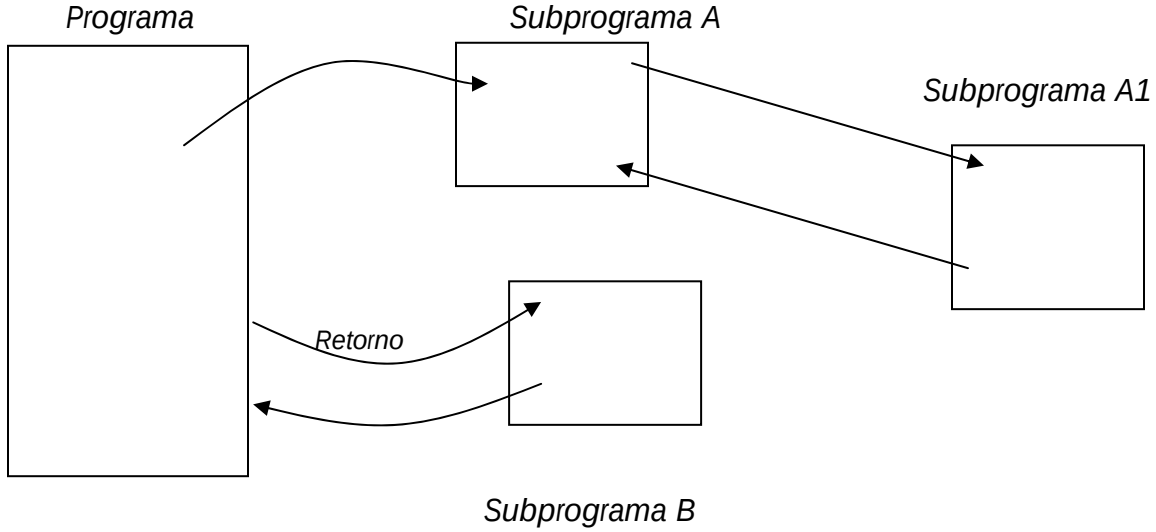


Figura 4.3

El número posible de llamadas sucesivas entre subprogramas normalmente vendrá limitado por la memoria de la computadora.

El subprograma, como se ha comentado, esta constituido por un grupo de instrucciones que no forman parte del programa principal. Sin embrago, aunque cada subprograma es funcionalmente independiente (incluso ensamblado o compilado) no funciona nunca solo, debe siempre ser llamado por el programa principal y dejará de actuar cuando haya terminado su función y el control se retorne al programa principal.

El subprograma puede ser compilado o ensamblado (en el caso de lenguajes intérpretes –BASIC-, también puede ser grabado con independencia y posteriormente con sentencias tipo CHAIN o MERGE, ser llamado para encadenarse o fusionarse con el programa principal) independientemente del programa principal, lo que presenta la ventaja de poder ser utilizado con diferentes programas principales y la consiguiente facilidad en la depuración, puesta a punto y modificaciones al tener longitud más pequeña –normalmente- que el programa principal.

Aunque los subprogramas son independientes del programa principal, evidentemente se necesitará un medio de comunicación entre el programa principal y subprograma, y viceversa. Así en el caso de leguajes como Pascal, True BASIC, Macintosh BASIC, Logo,..., la llamada se realiza mediante la definición previa de dichos subprogramas (denominados procedimientos en estos lenguajes debido a su funcionamiento es distinto de las subrutinas) y luego simples llamadas por su nombre; en el caso de lenguajes más formales, como FORTRAN, COBOL o BASIC, las llamadas se realizan de un modo mas explicito, mediante instrucciones.

FORTRAN	*CALL*
COBOL	*PERFORM*
BASIC	*GOSUB (o CALL, USR o SYS si los subprogramas están en lenguaje máquina).*

Aunque hasta este momento nos hemos referido a un solo tipo de programas –los que se conocen en la jerga informática por subrutinas-, en realidad existen dos tipos de subprogramas:

- *Procedimientos o subrutinas*
- *Funciones*

Las funciones y procedimientos han de ser definidos y su llamada se realizará por su nombre, mientras que las subrutinas serán llamadas mediante instrucciones específicas.

Las definiciones de funciones especifican cálculos que producirán valores necesarios para evaluar expresiones. Las definiciones de procedimientos especifican cálculos subsidiarios a un programa, esto es, cálculos que producen la acción o valor requerido.

Para manipular programas adecuadamente, necesitamos comprender los siguientes aspectos de su funcionamiento:

- *¿Cómo se definen los subprogramas? Existen unas reglas de construcción o definición de los subprogramas de modo similar a las que existen para la construcción de programas. Cada lenguaje tiene sus propias reglas específicas.*

- *¿Cuándo se ejecutan los subprogramas? La ejecución de subprograma se invoca o llama durante la ejecución de un programa. Una función se referencia --no se llama explícitamente--- si debe ser evaluada. Un procedimiento o subrutina se invoca si la instrucción específica o el nombre que define el procedimiento se ejecutan.*

Por ejemplo, si una función interviene la función y = sin (x)

*Y = Z * sin (X)*

Se está invocando el subprograma correspondiente.

- *¿Cómo se proporciona la información a los subprogramas? En general, los subprogramas reciben información a través de parámetros formales (o variables falsas "dummy", tales como x, n... en fig. 4.4). Cada invocación de un subprograma proporciona información por medio de parámetros actuales (o argumentos, tales como 3.4 y 2 en fig. 4.4. Los parámetros actuales o efectivos especifican los valores actuales utilizados para parámetros formales en la definición de subprogramas. La relación o correspondencia entre parámetros formales y actuales constituyen un mecanismo de paso a definir.*

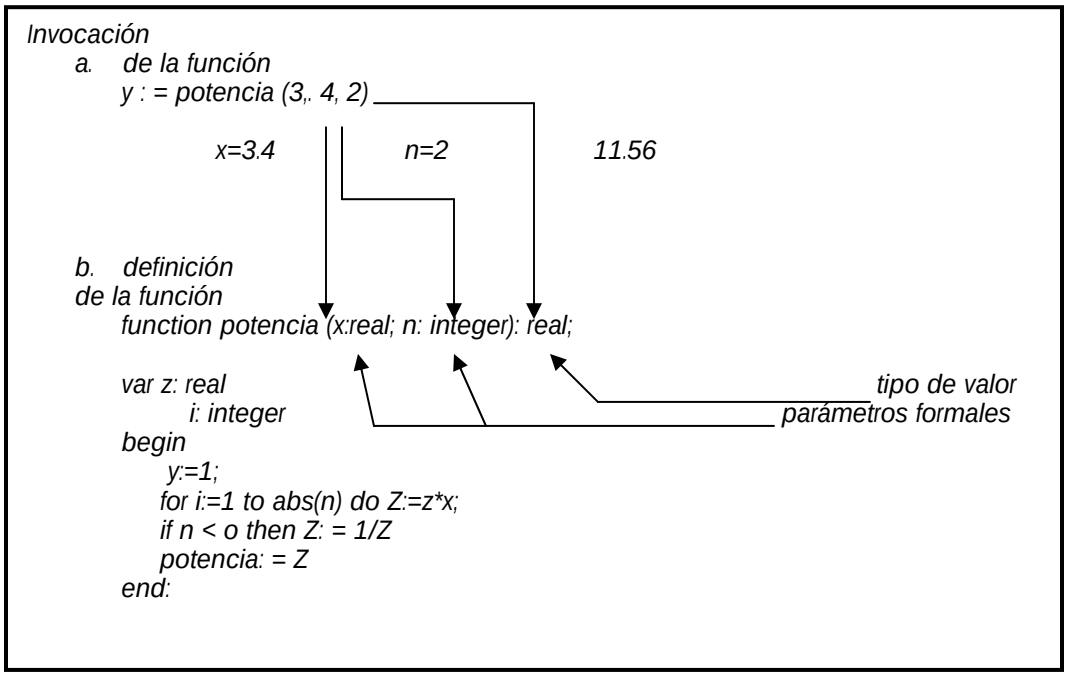


Figura 4.4

- *¿Cómo proporcionan la información los subprogramas? La función proporciona información en forma de un valor, ---por lo que en alguna parte del programa debe existir una instrucción de asignación---, por ejemplo 11.56 en la fig. 4.4, que podrá ser utilizada en la evaluación de cualquier expresión.*

Una diferencia entre funciones y subrutinas, es que:

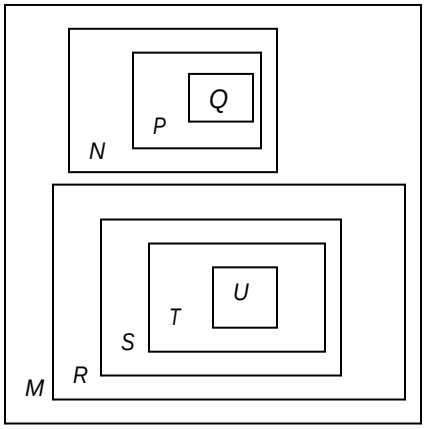
- *Las funciones devuelven un valor.*
- *Las subrutinas pueden devolver diferentes valores.*

- *¿Cómo interactúan las variables en subprogramas independientes? Las variables se clasifican en locales y globales. Una variable local es una variable declarada dentro de un subprograma y distinta de las variables con el mismo nombre declaradas en otra parte del programa. Una variable es global cuando está definida a un nivel más amplio que las locales; normalmente se define en el programa principal.*

El significado de una variable se limita al procedimiento en que está declarada. Cuando otro subprograma utiliza el mismo nombre, se crea otra variable diferente, es decir, el nombre se refiere a una posición diferente en memoria. Se dice que las variables son locales al subprograma en que está declarada. La parte del programa en el que una variable tiene acción (identidad, definición y ejecución del programa) se conoce como ámbito, rango o campo de definición ("scope") de esa variable.

Si un subprograma asigna un valor a una de las variables locales, este valor no es accesible a otros subprogramas ---es decir, no puede utilizar este valor---. Sin embargo, en ocasiones, es conveniente permitir que el identificador (nombre de la variable) tenga el nombre en diferentes subprogramas ampliando su campo de definición. En este caso se utilizan variables globales cuyo ámbito es más amplio que un subprograma.

En algunos lenguajes de programación (BASIC entre ellos), las variables declaradas en un subprograma pueden retener sus valores entre llamadas sucesivas del subprograma. En Pascal no se transmiten los valores, pero es posible su transferencia mediante la correspondencia entre variables definidas en un ámbito son accesibles en el mismo y por lo tanto son accesibles desde los procedimientos interiores. En la fig. 4.5 se muestra el campo de definición de las variables definidas en los procedimientos M a T.



Variables Definidas En los Procedimientos	*Son accesibles desde los procedimientos*
M	*M, N, P, Q, R, S, T, U*
N	*N, P, Q*
P	*P, Q*
Q	*Q*
R	*R, S, T, U*
S	*S, T, U*
T	*T, U*
U	*U*

Figura 4.5

4.7.1 Subrutinas /procedimientos

Una subrutina o procedimiento (subrutinas cerradas o externas, se denominan en ocasiones) es un módulo o subprograma independiente que ejecuta un proceso específico y que es llamado en un momento dado.

Las subrutinas pueden ser llamadas desde el programa principal tantas veces como sea preciso, devolviendo el control al programa principal cuando se termina la ejecución. Las subrutinas tienen la ventaja de ser totalmente independientes del programa principal, se puede almacenar en memoria en cualquier orden y además no necesita posiciones contiguas en esa memoria. Otra ventaja consiste en la no limitación de espacio de memoria. Sin embargo no se puede asociar ningún valor con el nombre de la subrutina o procedimiento y por lo tanto no puede aparecer en una expresión.

Subrutinas típicas en los lenguajes de programación son las definidas o llamadas del siguiente modo.

FORTRAN	BASIC
Definición con SUBROUTINE	llamada con GOSUB, ON-GOSUB
Llamada con CALL	

Los procedimientos en el lenguaje Pascal se define mediante la palabra reservada "procedure". Un procedimiento se llama escribiendo su nombre, es decir, DEMO, para indicar que se llama a un procedimiento denominado DEMO. Cuando se invoca su nombre, los pasos que lo definen se ejecutan y a continuación el control retorna al programa que llama.

Otra diferencia esencial entre procedimiento y subrutina, reside en el hecho que los procedimientos pueden llamarse a si mismos por poseer los lenguajes dotados de ellos (Logo, Pascal, True BASIC, etc.) una propiedad denominada recursión o recursividad (esta propiedad en Pascal se extiende también a las funciones).

Ejemplo de aplicación

Se dispone de una subrutina denominada CLASIFICACION que permite ordenar o clasificar una lista de números. Un programa principal trata con diversas listas numéricas que han de ser clasificadas cuando así se requiere en el flujo del programa. Un esquema gráfico del proceso podrían ser los diagramas a, b de la figura 4.6.

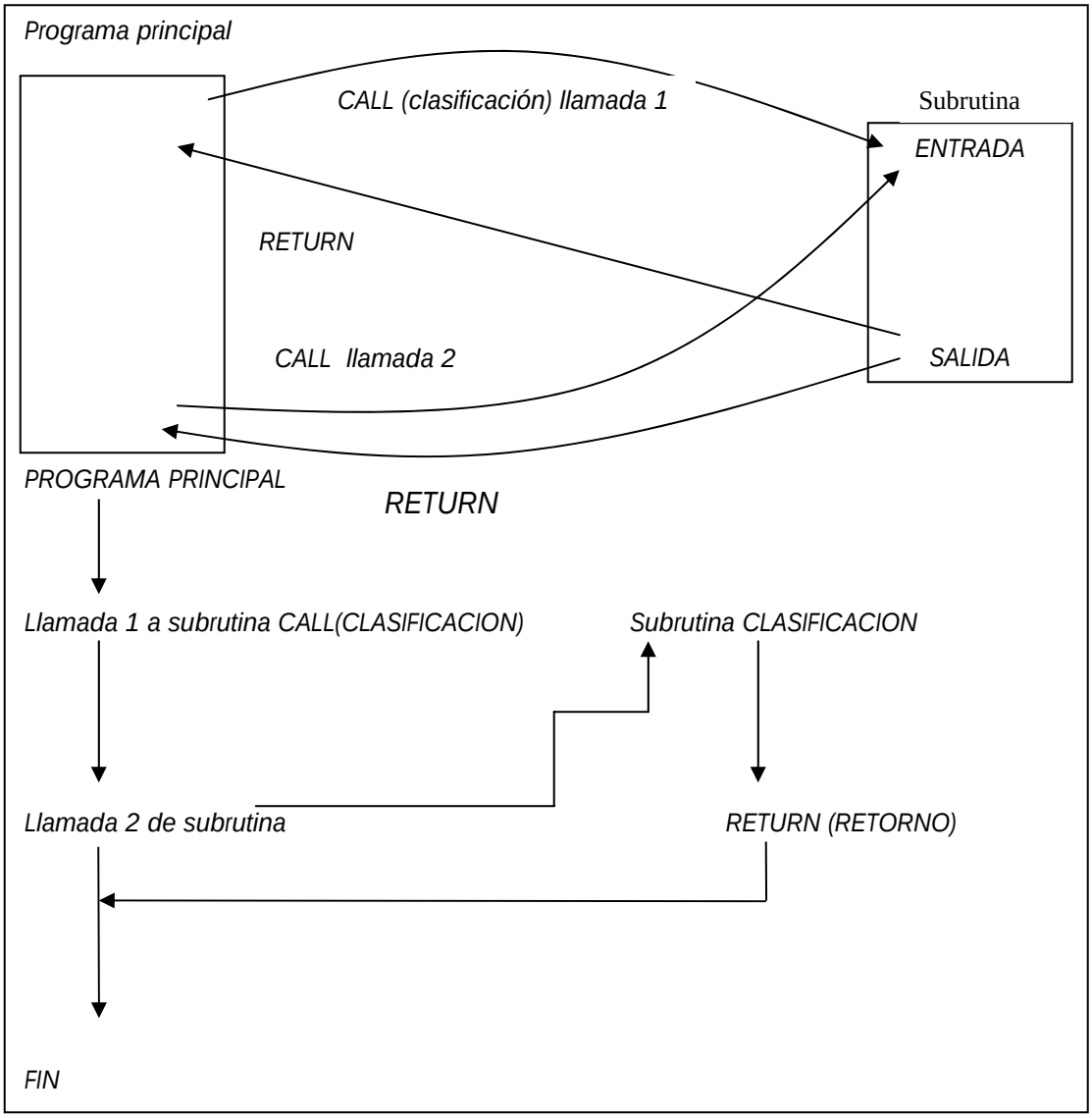


Figura 4.6

Las subrutinas pueden tener una entrada y una salida, que es lo usual, sin embargo, los subprogramas puede tener diferentes puntos de entrada, cada uno de ellos con una etiqueta o nombre, y el programa principal podrá transferir el control a los distintos puntos de entrada, debiendo incluirse en la instrucción de llamada el nombre o etiqueta del punto de entrada elegido.

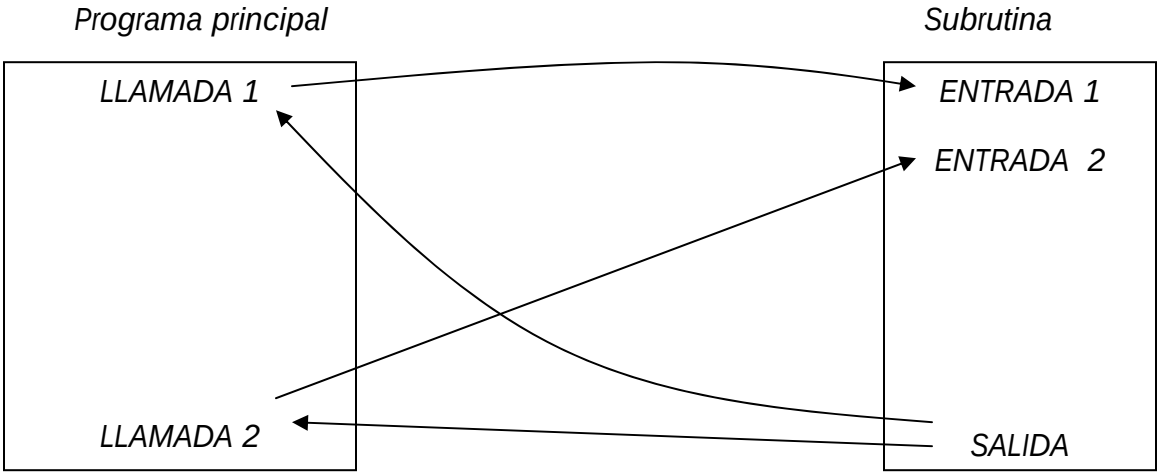


Figura 4.7

Nota: un subprograma puede ser llamado desde cualquier parte del programa principal o de otros subprogramas, mientras que un programa principal nunca puede ser llamado desde una subrutina, sólo es posible retornar a el a la terminación de la misma.

4.7.2. Funciones

Las funciones o subprogramas abiertos son un módulo o parte del programa que realiza un determinado cálculo que se utilizará en una expresión del módulo que llama a la función. Se diferencia del procedimiento o subrutina en que la subrutina, cuando se le invoca o referencia entrega un resultado único, mientras que los procedimientos o subrutinas pueden entregar 1, 2, 3,...n valores. Las funciones suelen tener pocas instrucciones, no tienen secuencias de llamada y su ejecución es muy rápida.

La mayoría de los lenguajes tiene un número de funciones intrínsecas, estándar o "incorporadas". Como: sin(x), cos(x), tan(x), sqr(x) donde los argumentos (x) de la función pueden variar a lo largo del programa; el cálculo del valor de la función se realiza en cada momento sustituyendo el argumento por su valor en ese punto.

Además de las funciones estándar es posible definir otras diferentes: en Pascal con "function" y en BASIC con DEF FN.

En matemáticas, una función es una regla que, cuando se le proporciona uno o más valores ---llamados argumentos--- produce un resultado ---el valor de la función para los argumentos dados---. Por ejemplo funciones pueden ser:

(a) f (x) = x / (5 – x) *(4.1) función con argumento x*

Para evaluar f (x) es preciso dar a x un valor actual, con este valor se calcula el resultado; con x = 3, se obtiene el valor 3 / (5 – 3) = 3 / 2 = 1.5, que se expresa escribiendo:

f (3) = 1.5

Una función puede tener varios argumentos:

(b) f (x, y) = 3 x + 4 y (4.2) función con argumento x, y

Pero, sólo un valor se obtiene como resultado para cada con junto de argumentos (x e y en el caso anterior).

Estas y cualesquiera otras funciones podrán ser definidas en cada lenguaje siguiendo una serie de pasos específicos.

BASIC

```
DEF FN A(X) = X / (5 – X)          (4.1)
DEF FN A(X,Y) =  3*X + 4*Y         (4.2)
```

PASCAL

```
function F ( X: real ): real;
begin
        F: = X / (5 – X)           (4.1)
end
 y

function F ( X: real; Y: real ): real;
begin
        F: = (3*X) / (4*X)         (4.1)
end
```

4.8 ESTRUCTURAS BÁSICAS.

Para poder conseguir un programa estructurado que sea claro, simple fiable y fácil de leer, será necesario utilizar estructuras de control que permitan a cualquier programa o subrutina dividirse en segmentos independientes.

El teorema de la estructura (también denominado Teorema de Jacopini Bohm, 1966) establece que un programa propio puede ser escrito utilizando solamente tres tipos de estructuras de control; estas tres estructuras básicas son:

- *Secuencial*
- *Selectiva*
- *Iterativa*

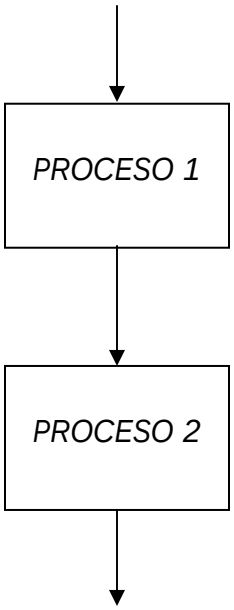
Un programa se define como propio si cumple las siguientes características:

- *Posee un solo punto de entrada y otro de salida para control del programa.*
- *Existen caminos desde la entrada hasta la salida que se pueden seguir y que pasan por todas las partes del programa.*
- *Todas las instrucciones son ejecutables y no existen lazos o bucles infinitos (sin fin).*

4.8.1 Estructura secuencial (DO-END / inicio-fin)

En la estructura secuencial los pasos del proceso se ejecutan en secuencia, uno después de otro. Las acciones o procesos se representarán del modo siguiente:

Diagrama de Flujo



Pseudo-código

- *1*
- *2*

PROCESO 1
PROCESO 2

- *1*
- *2*
- *3*

Los procesos o acciones pueden ser desde instrucciones sencillas hasta módulos completos la sucesión de las acciones puede ser alterada mediante un salto o bifurcación, empleando la sentencia GOTO.

Se suele utilizar a veces el termino DO-END para definir el pseudo-código de un estructura secuencial. ***DO*** *(hacer) para significar el comienzo de la secuencia y* ***END*** *(fin) que representa el fin de la parte secuencial. En Pascal se utiliza* ***begin-end****.*

DO		***HACER*** *(o bien INICIO)*
Acción 1	*o bien*	*acción 1*
Acción 2	*en español*	*acción 2*
Acción 3		*acción 3*
*		*
*		*
END		***FIN***

En ocasiones se suele utilizar **DO SEQUENTIALLY** *(Hacer o iniciar secuencia) y* **END SEQUENCE** *(fin secuencia).*

DO SEQUENTIALLY	*INICIO SECUENCIA*
Acción 1	*acción 1*
Acción 2	*acción 2*
Acción 3	*acción 3*
•	*
•	*
END SEQUENCE	*FIN SECUENCIA*

Como se ve en los pseudo-códigos anteriores, es recomendable utilizar identación (sangría o margen anterior de las acciones) dentro de las estructuras, ya que ello facilita la lectura del programa al mostrar de una forma gráfica las relaciones existentes entre las distintas instrucciones. La identación suele ser beneficiosa en la codificación de programas cualquiera que sea el lenguaje de trabajo.

4.8.2 Estructuras selectivas.

Una estructura es selectiva cuando sólo uno de los procesos alternativos (acciones o funciones) posibles se puede seleccionar tras el cumplimiento de alguna conducción determinada. La elección entre varias alternativas exige una toma de decisiones, por ello a veces a las estructuras selectivas se las conoce como de alternativa simple, alternativa doble o de alternativa múltiple según el número de opciones o alternativas que se pueden tomar.

Los puntos donde se deben tomar las decisiones se denominan puntos de decisión y en los diagramas de flujo se representan con el símbolo "rombo", de modo que uno de los vértices se conecta con el proceso anterior y el otro u otros con el resto de los procesos o acciones.

Las estructuras selectivas se clasifican en:

- *condiciones o alternativa simple*
- *alternativas o alternativa doble*
- *selectivas o alternativa múltiple.*

4.8.2.1. Estructura condicional (IF- THEN / si- entonces)

Realiza la ejecución condicional de una acción; es decir la acción o proceso solo se realiza si se cumple una determinada condición.

El diagrama de flujo es el siguiente:

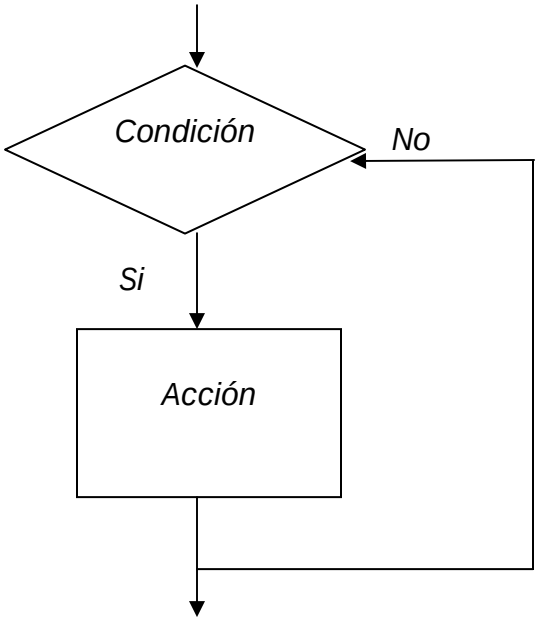


Diagrama de flujo IF – THEN

Al llegar el control al punto de decisión se ha de observar el estado del sistema. Si se satisface la condición (respuesta sí) se ejecuta la acción o proceso y en caso contrario (respuesta no) no se ejecuta. Los pseudo-códigos y sus diversos formatos son:

Inglés:

1. IF condición
THEN
Acción
END IF

2. IF condición THEN acción END IF

3. IF condición THEN

4. IF condición THEN acción

5. IF condición
THEN
Acción

Español:

1. si condición
Entonces acción
Fin _ si

2. Si condición entonces acción fin _ si

Los restantes formatos son equivalentes; basta sustituir la palabra IF por si, THEN por entonces y END IF por fin _ si.

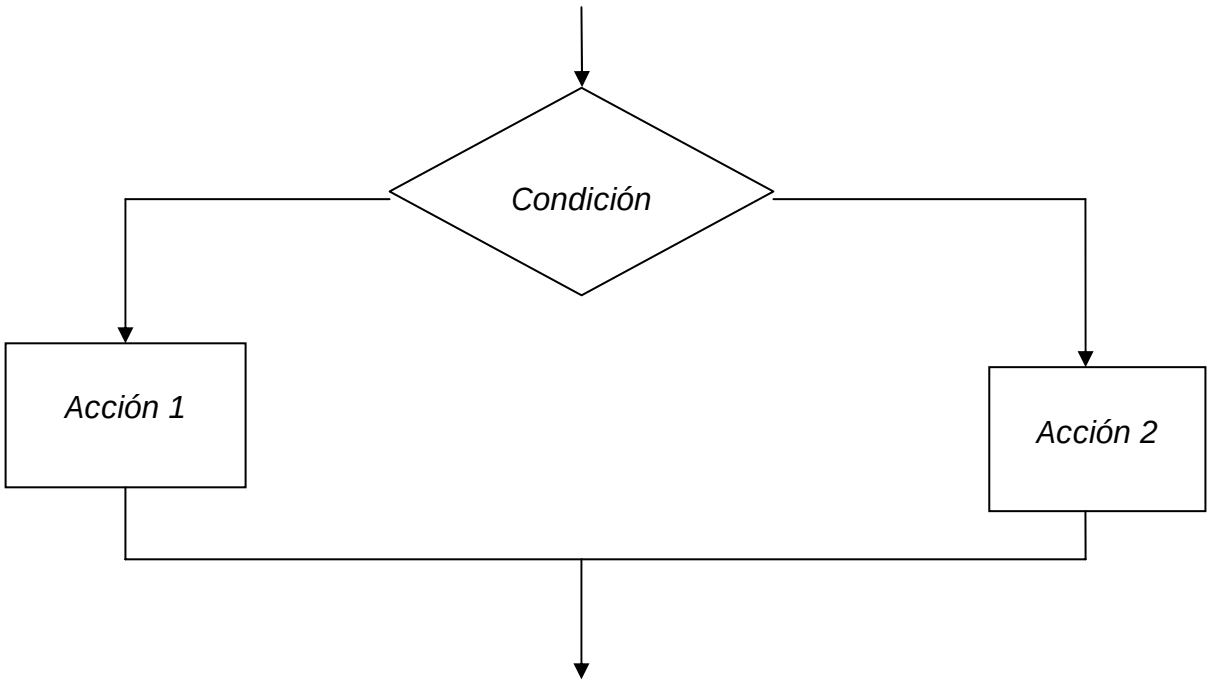
Si la condición se cumple (resultado verdadero, si) se ejecuta la acción; en caso contrario (resultado falso, n), no se ejecuta la acción y se salta, continuando la secuencia

Nota:

1. Las estructuras se han escrito con letras mayúsculas y minúsculas deliberadamente. La razón es que los libros y manuales de programación así como los editores de texto de las computadoras utilizados para escribir los programas, en la actualidad permiten la escritura.
*2. En ocasiones las palabras **END IF** y **fin _ si** se suelen abreviar en **ENDIF** y **fin si**.*
3. La acción puede ser una o varias.

4.8.2.2. Estructura alternativa (IF- THEN- ELSE / si-entonces-sino)

La estructura condicional es muy limitada, y será necesario utilizar estructuras que permitan la elección entre diferentes acciones o procesos. Si las opciones son dos, la estructura se denomina alternativa. El diagrama de bloques es:



Si la condición es verdadera (si) se realiza la acción 1 y si es falsa (no) se realiza la acción 2.

Los pseudo-códigos son:

Inglés:

1. *IF condición*
   Then
   Action 1
   ELSE
   Action 2
   END IF

2. *IF condición THEN acción 1 ELSE acción 2*
3. *IF condición*

THEN *acción 1*
ELSE *acción 2*
END IF

Español:

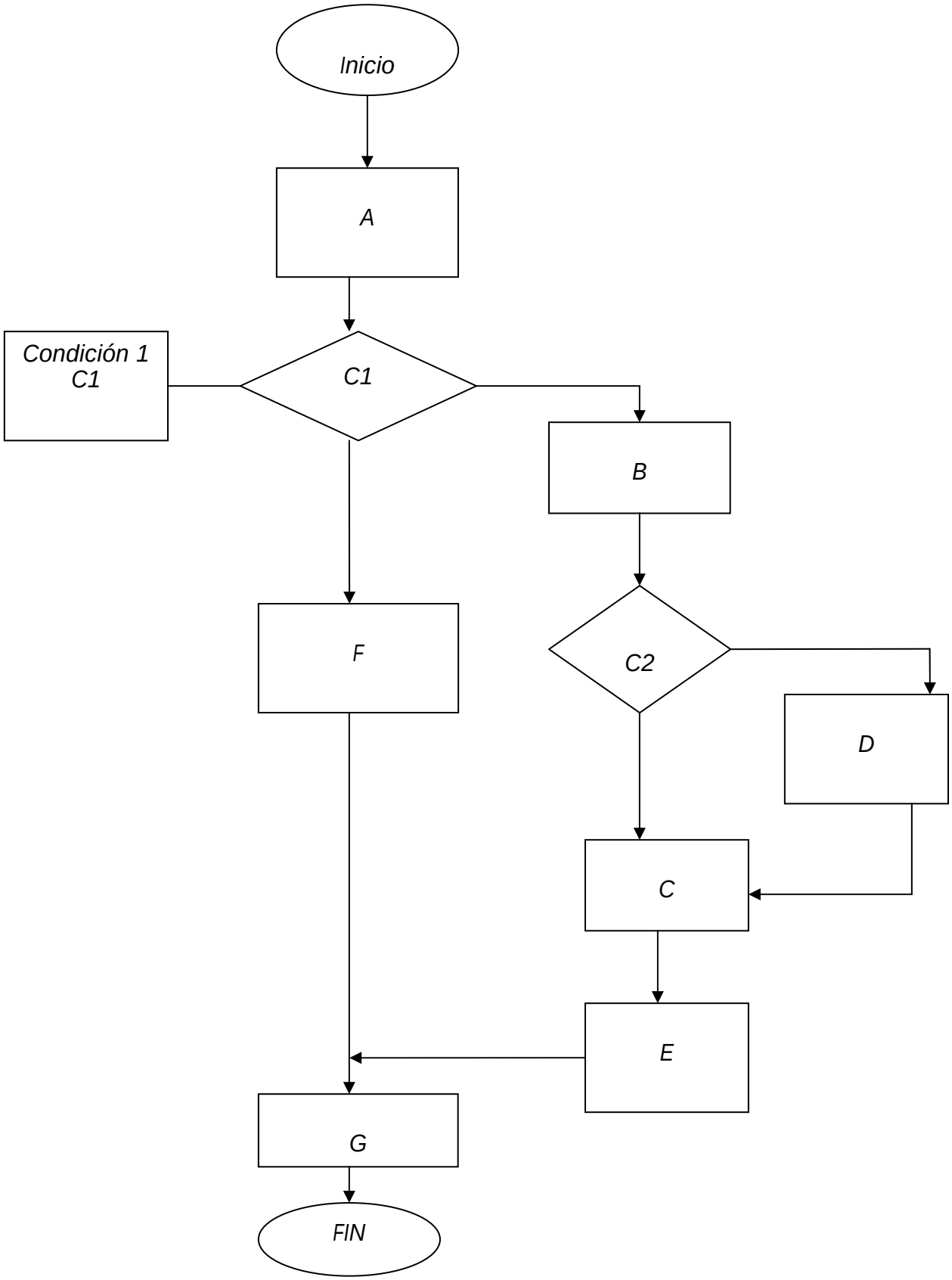
1. **SI** *condición*
   Entonces *acción 1*
   Si no *acción 2*
   Fin _ si
2. **si** *condición* **entonces** *acción 1* **sino** *acción 2*
3. **si** *condición*
   Entonces
   acción 1
   Sino
   acción 2
   Fin _ si

La palabra **THEN** *(entonces) señala la alternativa para procesar la acción 1 se la condición es verdadera. La palabra* **ELSE** *(sino) señala la alternativa* **para** *procesar la acción 2 si la condición es falsa.*

Ejercicio de aplicación

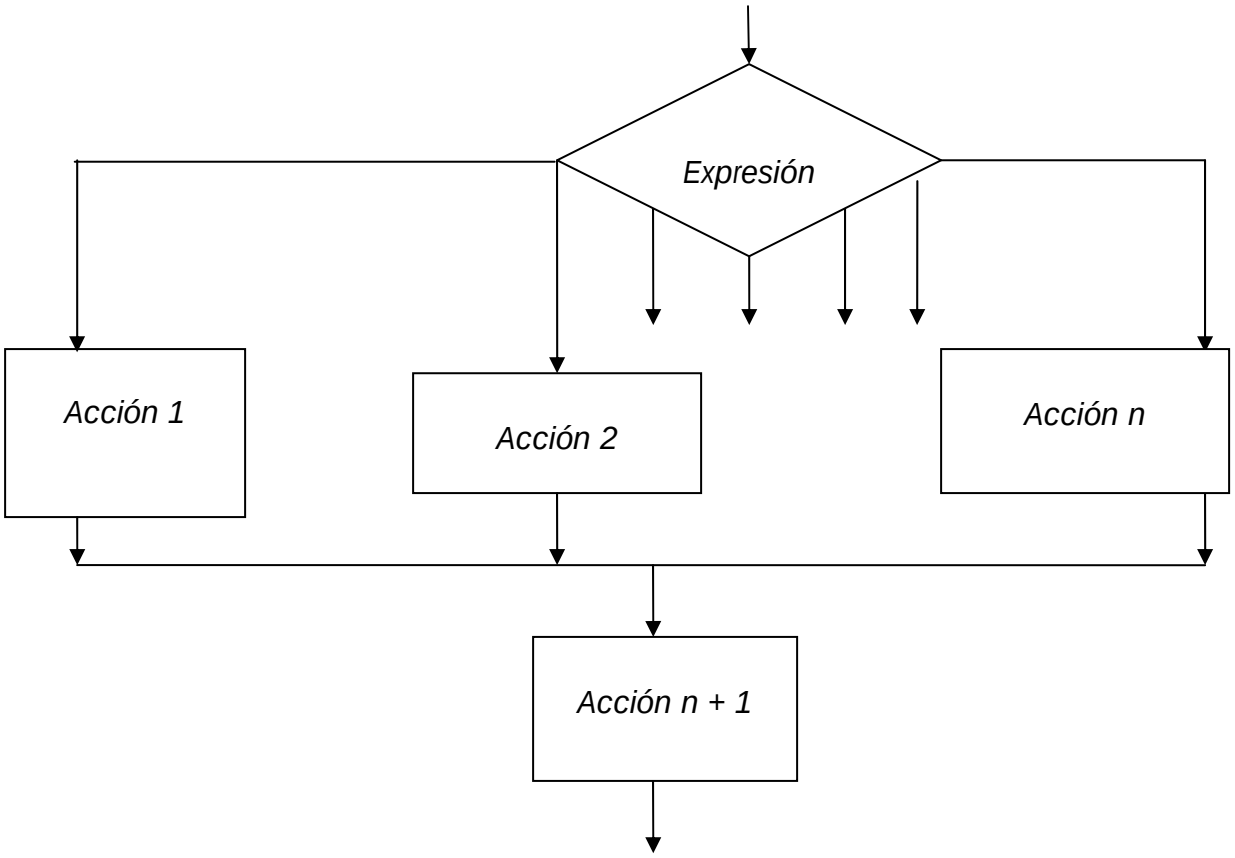
Un diagrama de flujo puede tener todo el grado de complejidad que se desee, ya que puede contener varias estructuras lógicas. Un ejemplo de ello puede ser el diagrama de la página siguiente cuyo correspondiente pseudo-código sería:

Inicio
acción A
Si *C1*
Entonces *acción B*
Si *C2*
Entonces *acción D*
Sino *acción C*
Fin _ si
acción E
Sino *acción F*
Fin _ si
acción G
Fin

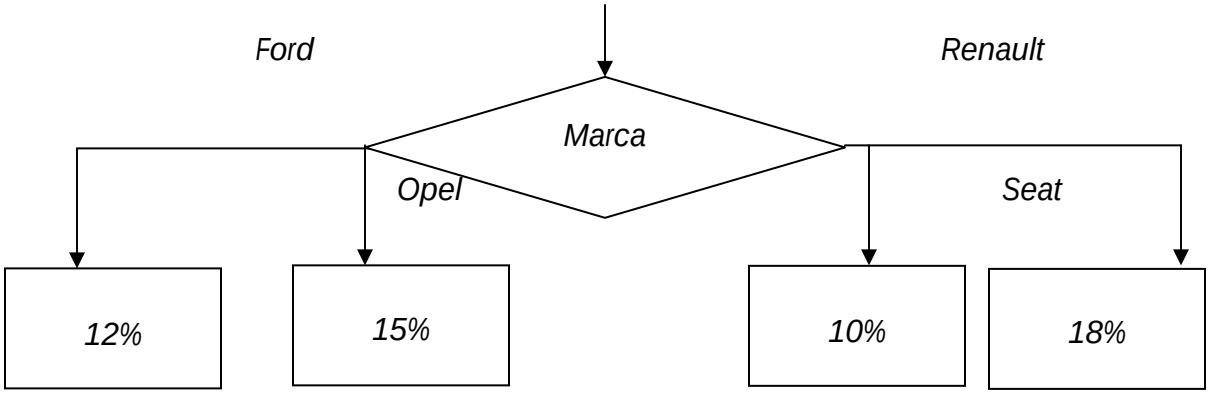

Inicio
A
Condición 1
C1
C1
B
F
C2
D
C
E
G
FIN

4.8.2.3 Estructura selectiva (CASE)

La estructura selectiva permite asociar un conjunto de condiciones a un conjunto de acciones que se excluyen mutuamente. La estructura se conocedor CASE (caso) y permite una desviación del flujo de control hacia múltiples procesos en función del resultado de la evaluación de una expresión o indicador. Así, si el resultado toma el valor 1 se realiza la acción 1, si toma el valor 2 la acción 2, si toma el valor n, y si no es ninguno de los valores 1 a n, se realizara la acción siguiente n + 1.



Un ejemplo aclaratorio podría ser la tasa de descuentos de un concesionario de automóviles en función de la marca.



Los pseudo-códigos son:

Inglés:

CASE expresión OF
Valor 1: acción 1
Valor 2: acción 2
……
Valor n: acción n
END CASE

Español:

Según expresión hacer
Valor 1: acción 1
Valor 2: acción 2
……
Valor n: acción n
Fin _ según

Variante del Formato

Se puede considerar un caso nuevo en las alternativas, y en los restantes valores no seleccionados específicamente.

En ese caso el pseudo-código sería:

Según expresión hacer
Valor 1: acción 1
Valor 2: acción 2
……
Valor n: acción n
Otro: acción n + 1
Fin _ según
En el caso de que se seleccione otro valor se ejecutara la acción n + 1.

CASE basando en IF-THEN-ELSE

En el caso de que la estructura CASE no tuviese una sentencia equivalente en su lenguaje de programación, se podría realizar su misión mediante estructuras condicionales (IF-THEN) y estructuras condicionales (IF-THEN) y estructuras alternativas (IF-THEN-ELSE).

CASE a base de IF-THEN-ELSE (si-entonces-sino)
Si valor 1
Entonces acción 1
Sino
Si valor 2
Entonces acción 2
Sino
Si valor n
Entonces acción n
Sino acción n + 1
Fin _ si
……
Fin _ si
Fin _ si

4.8.3 Estructuras iterativas

Las estructuras iterativas permiten ejecutar una acción un número determinado de veces, es decir repetir esa acción una o mas veces consecutivamente, se suelen llamar lazos o bucles; todas las instrucciones incluidas en los bucles se repiten un número determinado de veces. Se denomina iteración a cada una de las diferentes pasadas o ejecuciones de todas las instrucciones contenidas en el bucle.

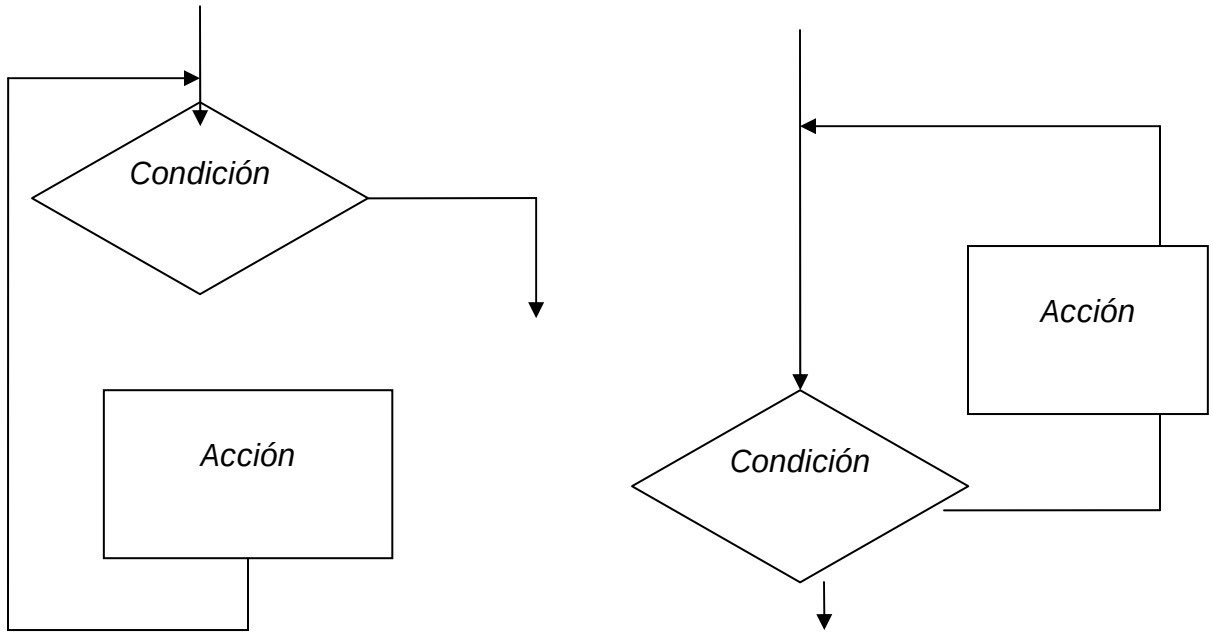
Existen dos variantes de estructuras iterativas: **DOWHILE** *(mientras –hacer) y* **DOUNTIL** *(repetir-hasta que). La diferencia entre ambas es que la condición se situé al principio de la secuencia de instrucciones (bucle DOWHILE) o que la condición se situé la final (bucle DOUNTIL), como se ve en los diagramas.*

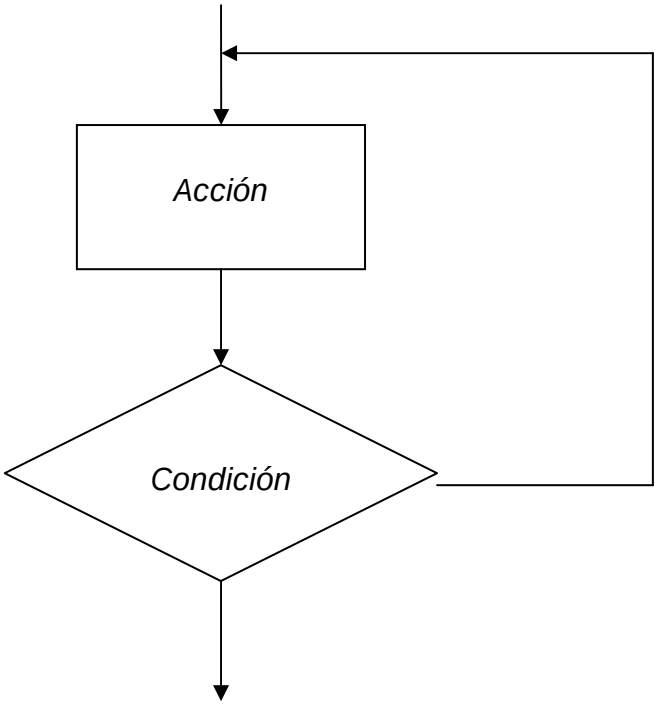
La diferencia entre DOWHILE y DOUNTIL es que en DOWHILE el bucle continuo mientras que la condición es verdadera (la condición se comprueba antes de ejecutar la acción, si es falsa la acción no se ejecuta) y el bucle se detiene cuando la condición es falsa.

En DOUNTIL el bucle continua hasta que la condición se hace verdadera (la condición se comprueba después de ejecutar la acción, o sea que la acción se ejecutara al menos una vez con independencia de que sea la condición verdadera o falsa). Otra diferencia como ya se ha comentado es que la condición se comprueba al principio del bucle en DOWHILE y al final del bucle en DOUNTIL.

Las estructuras DOWHILE y DOUNTIL se conocen también como estructura REPEAT WHILE y REPEAT UNTIL (en el lenguaje Pascal estas estructuras son **WHILE-DO** *y* **REPEAT-UNTIL**, *las versiones de BASIC que soportan estas estructuras* **WHILE-WEND y REPEAT-UNTIL**).

Como se muestra en los siguientes ejemplos:





Los pseudo-códigos de las estructuras DOWHILE y DOUNTIL son los siguientes:

DOWHILE

Inglés:	*Español:*
DOWHILE *condición*	**mientras** *condición* **hacer**
acción 1	*acción 1*
acción 2	*acción 2*
.....	.....
ENDDO	**fin _ mientras**

El pseudo-código significa "mientras que la condición sea verdadera hacer las/s acción/es, cuando sea falsa terminar el bucle". Puede ocurrir que el bucle no se ejecute ni una vez en el caso de que la condición no se cumpla inicialmente.

DOUNTIL

Inglés:	*Español:*
DOUNTIL *condición*	**repetir**
acción 1	*acción 1*
acción 2	*acción 2*
.....	.....
ENDDO	**hasta _ que** *condición*

Nota: *tanto en DOWHILE como en DOUNTIL se necesita que el bucle contenga al menos una instrucción que cambie la condición que controla el bucle. Si no la hubiera el bucle continuaría indefinidamente.*

4.8.3.1 Otras estructuras iterativas

Las estructuras DOWHILE y DOUNTIL repiten el bucle mientras o hasta que se cumpla una condición; ello significa que el bucle se repite un número no determinado de veces. En ocasiones, sin embargo, se conoce a priori el número de iteraciones a realizar, y en esas ocasiones, aunque también se podría utilizar DOWHILE y DOUNTIL, es preferible utilizar otra estructura iterativa denominada REPEAT –FOR (FOR-NEX-STEP en el lenguaje BASIC y FOR-TO-DO/ FOR-DOWN TO-DO en el lenguaje Pascal).

Los formatos de los pseudo-códigos son:

Inglés:

***FOR** variable = expresión 1 **TO** expresión 2 {**STEP** expresión 3}**
DO
acción
ENDFOR

***{STEP** expresión 3} es opcional y es un incremento*

El significado del pseudo-código es: repetir la acción desde un valor inicial de la variable de control dada por expresión 1 hasta alcanzar el valor final dado por expresión 2. Si no se indica nada en contra con STEP se supone que los incrementos son positivos de 1 en 1 (caso de enteros); caso de querer variar los incrementos será preciso indicarlo con STEP expresión 3, siendo el incremento positivo o negativo el valor de expresión 3.

Español:

***Desde** var. = exp. 1 **hasta** exp. 2 {**incremento** exp. 3}*
Hacer
Acciones
Fin _ desde

***Nota:** algunos autores prefieren traducir FOR por la palabra para en lugar de desde*

En realidad – como ya se ha comentado- la estructura FOR es un caso especial de DOWHILE- DOUNTIL en el que se conoce a priori el número de veces que se ejecutará el bucle; por ello se suele denominar estructura repetitiva en lugar de iterativa, para diferenciarla de las otras estructuras.

Las estructuras WHILE y FOR pueden, en ciertos casos, no realizar ninguna iteración del bucle, mientras que UNTIL ejecutara el bucle al menos una vez.

4.8.4 Estructuras especiales. Goto

*En los apartados anteriores hemos visto como utilizar **if-then-else** y el bucle que genera para crear caminos alternativos en el cálculo. En ocasiones, es conveniente utilizar una estructura con una larga y controvertida historia, la sentencia o estructura **goto**.*

En un programa sin bucles o estructuras **if**, *las estructuras -el programa- se ejecutan en secuencia y se dice que el flujo es normal. A veces, se rompe el flujo normal por la transferencia o bifurcación a otra instrucción que no esta en secuencia, como es el caso de los bucles, o bien, transfiriendo el control directamente a alguna instrucción especifica. Para este caso se necesitan:*

1. *Una estructura para ejecutar la transferencia del control*
2. *Un medio de referencia de esa estructura en el programa.*

Este segundo requisito se satisface permitiendo que las instrucciones sean numeradas o etiquetadas con palabras. El primer requisito es la estructura **GOTO** (**goto**).

GOTO

Es una estructura (enunciado simple lo denomina Wirth, el padre de Pascal) que indica que el proceso adicional debe continuarse en alguna otra parte del programa.

El formato de la estructura GOTO (ir a) es

GOTO etiqueta	*GOTO número de línea*

Se produce la transferencia a la instrucción cuya etiqueta o número de línea se da en el formato. A partir de ese punto se reanuda el flujo normal del proceso.

Los lenguajes BASIC y FORTRAN utilizan la estructura GOTO, Pascal también la posee pero su uso esta muy restringido.

El uso indiscriminado de GOTO en los programas es nocivo porque dificulta su legibilidad. Si existen muchas bifurcaciones con GOTO será necesario recurrir a numerosas etiquetas o números de línea y ello puede producir errores, por duplicado, ausencia de etiquetas, etc.

¿Cuando es útil GOTO?

Junto con la estructura **IF** *se puede utilizar para construir bucles.*

Por ejemplo, el bucle:

```
DOWHILE condición
        Acción 1
        Acción 2
ENDDO
```

Puede ser sustituido por:

```
1: IF NOT condition THEN GOTO 2;
        Acción
        GOTO 1;
2… siguientes instrucciones
```

Sin embargo la composición anterior no parece clara.

La estructura GOTO se puede utilizar para transferir el control a un bucle, o desde dentro de un bucle al exterior, pero no puede transferirse desde fuera al interior de un bucle.

De hecho, los buenos programadores necesitaran utilizar la estructura GOTO muy raramente. Algunos incluso llega a decir que no se debe utilizar nunca (Diestra entre ellos), sin embargo esta idea nos parece excesiva. Una estructura GOTO debe reservarse para situaciones no usuales o comunes en que la estructura natural de un algoritmo debe quebrarse. Una buena regla es evitar los saltos para expresar iteraciones regulares o la ejecución condicional de instrucciones. La presencia de GOTO en lenguajes como Basic aunque es necesaria se debe tratar de sustituir por otras estructuras equivalentes, alternativas o iterativas. En Pascal, Wirth (su autor) dice que la presencia de GOTO en un programa Pascal suele ser indicación de que el programa no ha aprendido aun "a pensar" en Pascal.

Imaginemos una estructura como:

```
IF condición
        THEN GOTO 1;
.
.
Acciones
IF condición
        THEN GOTO 1;
.
.
1: end
```

Puede ser sustituida eliminado GOTO por

```
IF NOT condición
        THEN
.
.
Acciones
.
.
IF NOT condición
        THEN
```

Sin embargo, con GOTO la aplicación era más directa.

Una buena aplicación de GOTO puede ser en recuperación de errores, para indicar códigos de errores, síntomas o desviaciones de flujo cuando se comete un error en el proceso.

4.8.5 Síntesis de estructuras de control

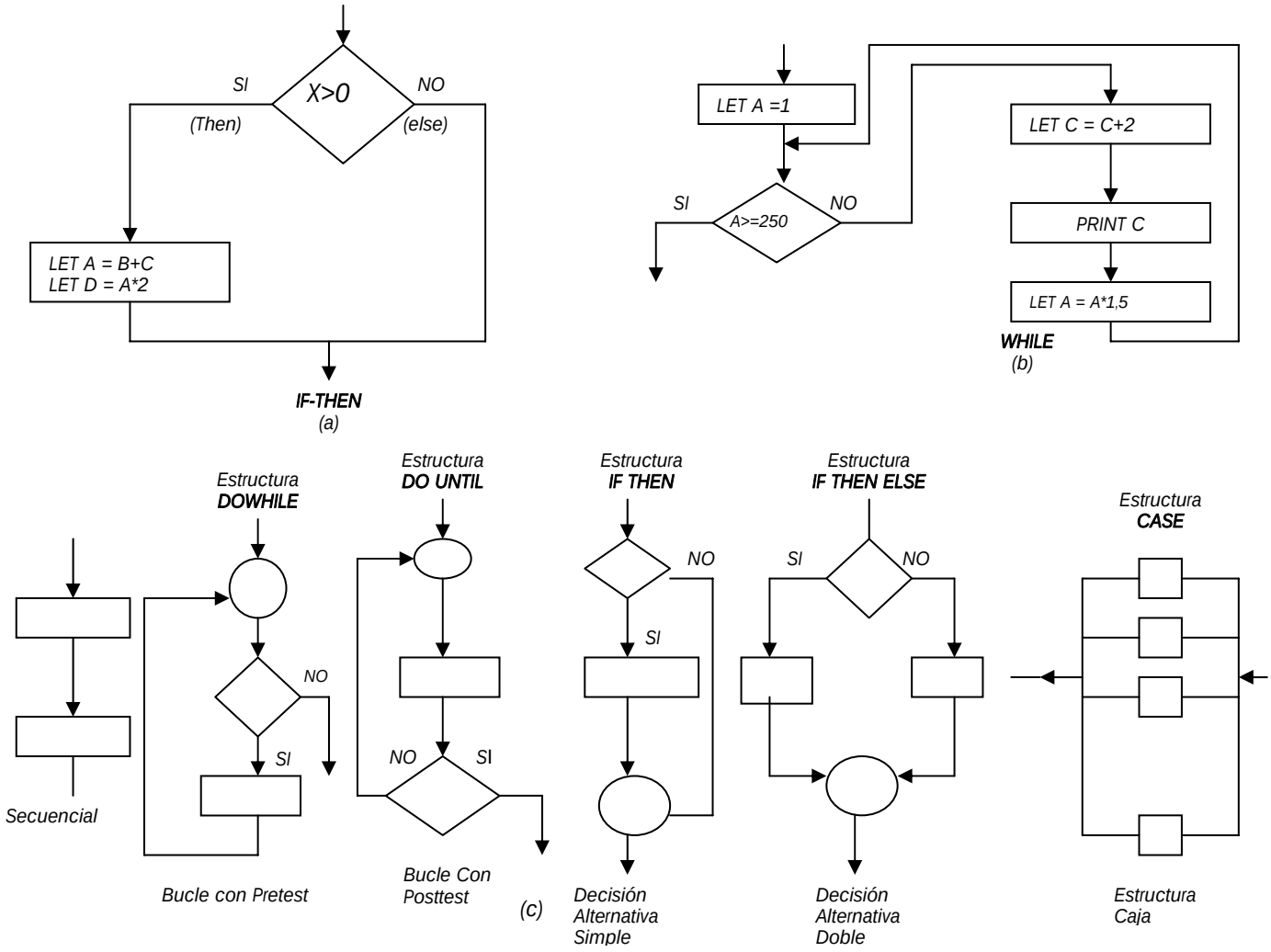


Figura 4.8. Estructura de control: (a) IF-THEN;(b) WHILE; (C) estructuras de control básicas en programación estructurada.

- PAGINA DEJADA INTENCIONALMENTE EN BLANCO -

CAPITULO 5

ESTRUCTURA DE DATOS

5.1 INTRODUCCIÓN.

El programador para llegar a expresar la solución de un problema debe expresarlo en forma de un algoritmo que luego le permita una fácil codificación. En esta consideración la información se basa en el estudio de algoritmos-secuencia paso a paso de las diferentes operaciones que se deben realizar para obtener un determinado resultado-; sin embargo, un algoritmo manipula datos y para que se pueda aplicar correctamente necesita que esos datos estén relacionados de alguna forma particular. Esta nueva idea nos lleva a otra concepción de la informática, el estudio de los datos, su estructura y su manipulación o abreviadamente el concepto de estructura de datos o estructura de la información.

Estructura es la relación que existe entre los distintos elementos de un grupo. En informática la relación que existe entre elementos de datos es **una estructura de datos**.

Uno de los problemas más serios con los que se enfrentara el programador es, precisamente, la estructuración y almacenamiento de la información. El programador deberá analizar con mucho cuidado la estructuración, almacenamiento y recuperación de datos, comparándolos con eficacia del algoritmo elegido para su manipulación. En caso contrario la solución dada para la resolución del problema puede resultar muy costosa en espacio de memoria o en tiempo de trabajo de máquina.

En este capitulo se estudiaran las diferentes formas (estructuras) que los datos pueden tomar y las aplicaciones en que se utilizan las diferentes estructuras.

Estas estructuras de datos se almacenaran en la memoria principal o interna o en los soportes externos de almacenamiento de información. La memoria secundaria o externa-casetes, cintas, discos, etc.

5.2 LOS DATOS

Los datos o información que manipule una computadora serán diferentes según la aplicación en que se utilicen, y una de las características diferenciales de los lenguaje de programación es el tratamiento de los datos. Así cuando un científico interesado en números reales y complejos y en matrices trabaja en programación utilizara FORTRAN. En contraste, los datos o información utilizados en la gestión comercial están normalmente reunidos en ficheros, y por ello se emplea COBOL que proporciona grandes facilidades para la descripción y estructura de ficheros. La mayoría de los lenguajes están diseñados para manipular ciertos tipos de datos más adecuadamente.

La tabla 5.1 muestra las notaciones mas utilizadas para el usuario y las características de los datos manipulados

Lenguaje	Notación normal del usuario	Características de los datos
Ada	Programas	Los usuarios pueden diseñar sus propias estructuras de datos
ALGOL	Diagrama de flujo, pseudo-código	Numéricos
APL	Símbolos matemáticos ampliados	Matrices
BASIC	Diagramas de flujo y expresiones matemáticas elementales	Caracteres y números
COBOL	Inglés	Ficheros jerárquicos
FORTRAN	Formulas matemáticas	Numéricos
LISP	Lógica	Conectados intrincadamente (listas)
PASCAL	Diagramas de flujo, pseudo-código	Los usuarios pueden diseñar sus propias estructuras
PL/1	Variadas	Numérica y comercial
RPG	Inglés restringido	Informes
SIMULA	Descripción de sistemas	Dependiente del tiempo
SNOBOL	Patrones de palabras	Cadenas de caracteres

Tabla 5.1 *Características de lenguajes de programación*

La información tratada por el ser humano puede proceder de su memoria-como es el caso del alfabeto, la dirección de su casa o despacho, o el método para utilizar el teléfono-, o bien de la búsqueda de la información que necesitamos.

En los casos de la búsqueda de la información, esta puede resultar prolija; así, por ejemplo si deseamos ir al teatro y no sabemos el número de teléfono para llamar y reservar localidades, ni conocemos el horario de autobuses, para el caso de decidir ir, será preciso buscar información en algún centro de información, como puede ser una guía de teléfonos. Sin embargo, esta ausencia de información nos fuerza a pensar detenidamente como conseguirla. No podemos solucionar nuestro problema a menos que podamos decidir. Shave en 1975 enuncio los principios de la decisión:

a) ¿Qué deseamos conocer?
b) ¿Dónde se puede encontrar la información?
c) ¿Cómo obtenerla?

En nuestra situación actual (ir al teatro),

a) Deseamos conocer el número de teléfono
b) Este se encuentra en la guía telefónica (conocimiento de la memoria);

c) En la guía, los nombres de los abonados, están listados en orden alfabético y de este modo obtenemos el número deseado consultando el nombre del teatro en esa lista (en este caso, el método de acceso -alfabético- a la información se debe conocer).

Este ejemplo muestra que la información (el significado relacionado con los datos) puede implicar no solo un conjunto de valores de datos, sino también una disposición-estructura- que relaciona los valores de algún modo.

5.2.1 Manipulación de los datos

La clasificación alfabética por nombres de abonados de la guía telefónica es una referencia fácil, pero no la única. Existe otra clasificación –también alfabética- de nombres de calles o distritos, y dentro de cada calle o distrito listados alfabéticos de los suscriptores. Esta última guía o directorio se le llama dos niveles, debido a que necesitamos aplicar nuestro proceso de búsqueda dos veces: una ves para encontrar la calle o distrito y a continuación nuevamente para encontrar el abonado. Existe un tercer formato multinivel utilizado en las páginas amarillas. En esta guía las entradas se listan alfabéticamente por actividades comerciales o profesionales y a continuación alfabéticamente por abonado dentro de cada clase. Incluso se podría considerar un tercer nivel, con un índice alfabético, que guíe hacia la clasificación correcta.

Nombre y Apellidos	*Número de teléfono*
…	…
García Andrade, L.	*645 12 36*
García Antoranz, J.L.	*721 46 32*
García Botín, M.	*645 13 21*
García Bosano,N.	*645 12 16*
García Candil,J.	*645 30 92*
García Castro,P.	*645 40 27*
García Edil,A.J.	*646 14 60*
García Fernandez,R.	*645 21 80*
García Fortuna,O.	*645 17 21*
García García,V.	*647 04 06*

Calle o plaza	*Número de teléfono*
La cruz	
…	…
García Martinez,A.	*715 43 20*
Martínez Gil, R:	*715 14 16*
Martínez Hidalgo,J.R.	*717 04 06*
Navarrete Gil, B.I.	*821 50 14*
Peláez Garcia,J.L.	*715 00 40*
Romero García,N.	*715 10 13*
Romero Gil,C.	*717 40 21*
…	…

Automóviles	
…	…

SEAT	
...	...
Abengo ,S.A.	721 40 86
HISPANIDAD,S.L.	639 14 16
JUMAR,S.A,	714 51 64
ROMERO,S.L.	821 21 40
...	...

Figura 5.2. Estructuras de datos típicas

Por consiguiente se puede ver que la estructura alfabética de las guías telefónicas no es esencial, sino que ha sido impuesto en aras de la comunidad del usuario medio.

Existen casos en que el diseño de los datos tiene una estructura de árbol con un número de bifurcaciones, cada una de las cuales tiene a su vez una estructura de árbol. Imaginemos el caso de un árbol genealógico (figura 5.2.).

La clasificación alfabética de la guía telefónica permite que las entradas sean listadas en una secuencia lógica. Las entradas en un árbol no pueden ser listadas en secuencia adecuadamente –al menos sin destruir la estructura del árbol-. En este caso, un diagrama o dibujo es un mejor medio de presentación de datos. Un diagrama es también útil para los datos que tienen estructura rectangular o <<tabular>>.

Un ejemplo típico de una estructura tabular es el caso de los cuadros horarios de trenes. La siguiente tabla es de dos dimensiones y permite una fácil lectura de arriba-abajo y de izquierda a derecha, con incrementos de tiempo desde el punto de vista inicial.

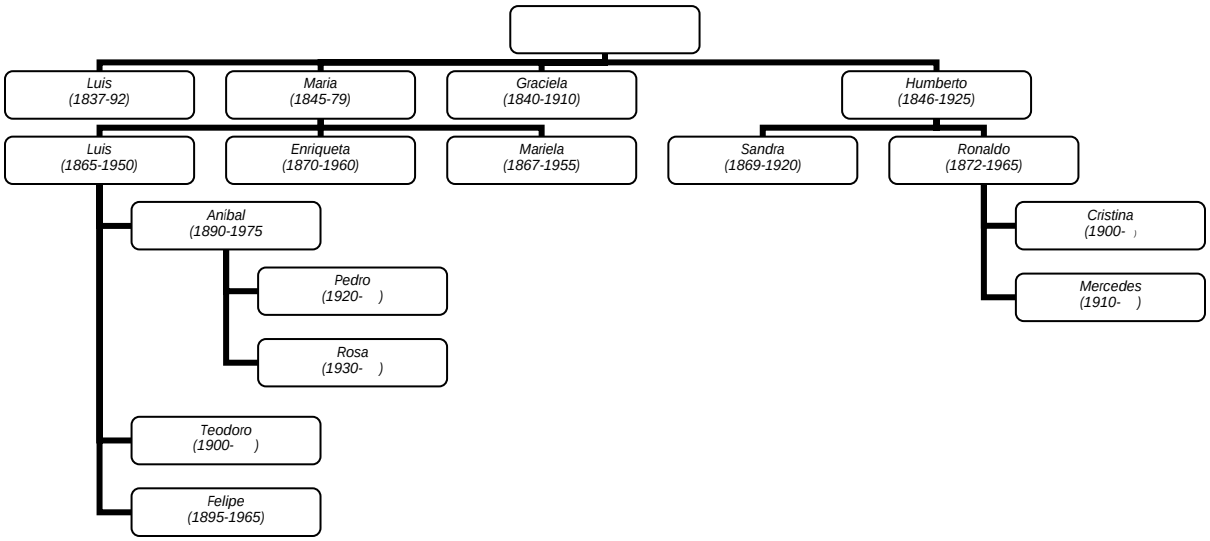


Figura 5.3. Estructura en árbol

	A.M.	A.M.	A.M.	P.M.	P.M.
Madrid	7:30	8:30	11:00	15:00	19:00
Las Rozas	7:45	8:45	11:15	15:15	-
Las Matas	7:58	8:58	11:28	15:25	-
Galapagar	8:15	9:17	12:45	15:40	19:30
Torrelodones	8:30	9:28	12:55	15:50	19:40
Collado-Villalba	8:35	9:34	13:00	-	-
Guadarrama	8:40	8:41	13:10	12:58	19:50
Los molinos	9:50	9:50	13:20	-	-
Cercedilla	8:50	9:58	13:28	-	-
San Rafael	9:05	10:04	13:32	16:10	20:03
Otero de Herreros	9:15	10:12	13:42	-	-
Madrona	9:20	10:15	13:45	-	-
Segovia	9:28	10:20	13:55	16:30	20:20

Figura 5.4. Tabla de dos dimensiones

Los cuadros o diagramas de distancias típicos de agendas, mapas, guías de carretera, etc., se pueden utilizar con tablas de 2 entradas

5.2.2 Estructura de datos

Una estructura de datos es una colección de elementos o ítems de datos. El medio en el que se relacionan unos elementos con otros determina el tipo de estructura de datos. El valor de la estructura se determina por:

a) *Los valores de los elementos*
b) *La disposición de los elementos*

En el caso de la guía telefónica, cada elemento de datos comprende un nombre, dirección y número de teléfono (tal elemento se suele denominar registro o átomo. Las estructuras de datos telefónicos son del mismo tipo, y se denominan ficheros secuenciales.

Un elemento o ítem de datos o simplemente **un dato** *es una cantidad que toma un único elemento que puede cambiar de cuando en cuando (se denomina variable o identificador). Los algoritmos de resolución de problemas deben incluir reglas de acceso a la estructura de datos. En el caso de la guía de teléfonos tal regla puede ser una descripción al principio de la guía, del modo para buscar un número; en el caso del cuadro horario de trenes una explicación de cómo utilizar el cuadro; en general como buscar un valor en una tabla matemática, estadística; etc.*

Normalmente la selección de una estructura de datos se hará en función de su contenido; tras la selección, se elegirá la mejor disposición de los datos y a continuación se colocan los valores en los datos. En consecuencia los valores de los datos pueden ser cambiados con relativa frecuencia, la disposición de ellos se cambia con menos frecuencia y la estructura de los datos se cambia raramente. Este es, de

hecho, el caso, por ejemplo, en una guía de teléfono donde las entradas cambian anualmente (cambios de direcciones de los abonados, altas, bajas, etc.) sin embargo, la disposición alfabética no suele e variar.

No obstante el ejemplo no siempre es significativo pues, en ocasiones, un cambio en los valores de los datos produce cambio en la disposición de los mismos. Consideremos la tabla de clasificación de la liga de fútbol española de primera división. Los resultados de los partidos semanales influirán en la clasificación, modificando no solo el valor de los datos (puntuación), sino el orden de los equipos.

Otro caso significativo, aunque no frecuente, se produce cuando se modifica el tipo de estructura de datos, permaneciendo constantes los valores de los datos; es ese el caso de la guía telefónica por direcciones.

La elección del método adecuado para expresar los datos del problema es esencial para conseguir una buena solución. Si se hace una elección incorrecta el algoritmo resultante puede ser muchas veces complicado y por consiguiente difícil de codificar. Por el contrario, una buena elección de estructuras de datos puede llevar a un algoritmo que sea más claro, más sencillo y más rápido.

Una vez que se elige una buena estructura de datos debemos intentar utilizar un lenguaje de programación en el que esta estructura de datos si pueda ser fácilmente descrita y con rapidez y cómodo acceso.

5.3 CLASIFICACIÓN DE LAS ESTRUCTURAS DE DATOS

Las estructuras de datos más usuales son: cadenas, matrices, listas, tablas, árboles, pilas, colas y ficheros. El estudio de las estructuras conduce a los algoritmos de manipulación de los datos. Supongamos que se desea escribir un programa en FORTRAN para almacenar los días de un mes y sus temperaturas medias; como la única estructura de datos es la matriz – un tipo determinado de lista - habrá que realizar los algoritmos de manipulación; por ello cuando se desee añadir nuevos nombres a la lista habrá que diseñar un algoritmo de inserción y cuando se desee buscar un nombre dado se deberá diseñar un algoritmo de búsqueda.

Las estructuras de datos se pueden definir como una forma práctica de organizar hileras, listas, árboles, etc., en la memoria interna o central de las computadoras

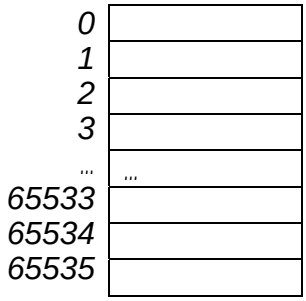
Una de las primeras estructuras utilizadas y conocidas por los programadores son las matriciales conocida por arrays o matrices (vectores – arrays unidimensionales-, tablas o variables con subíndices – arrays multidimensionales -).

Las matrices se almacenan en un bloque de posiciones de memoria consecutivas. Cada elemento de una matriz se puede recuperar o identificar por su dirección (valor numérico único).

Cada posición de una computadora se asocia con un número fijo o dirección. Así pues, la computadora ve la memoria como un bloque de posiciones consecutivas que identifica con una dirección o valor numérico.

Si nos imaginamos una relación de apellidos de personas, podría asociar a cada posición de memoria a un apellido.

El orden o disposición de los apellidos formaría una estructura de datos. En realidad la memoria no esta organizada así, porque normalmente una posición de memoria equivale a una palabra de 8 o 16 bits, y su equivalente es un carácter de alfabeto –en el caso de 8 bits-; sin embargo, a efectos didácticos preferimos dejar el ejemplo tal y como se a enunciado, por no ser relevante en este momento el contenido real de cada posición, sino su contenido relativo.



Dirección (valor numérico único)

Contenido de cada posición de memoria

Figura 5.5.

Una disposición como la relación de apellidos constituye también una estructura de datos conocida como lista. Todos sus elementos están en posiciones consecutivas de memoria. Para definir una lista será preciso conocer el punto inicial y final, es decir la longitud.

Otro tipo de estructura de datos son los ficheros – conjunto de registros que se almacenan en la memoria externa de una computadora- y los árboles – representación de los elementos en forma jerárquica-.

5.3.1 Matrices (arrays)

Una de las estructuras más simples consiste en una secuencia de elementos del mismo tipo, relacionados unos con otros por el orden de que están definidos.

Los matemáticos y científicos en general utilizan un índice o subíndice para representar la notación de tales estructuras. Así, una notación típica es:

$X_1, x_2, ..., x_{10}$

Y representa 10 elementos. La x es el nombre general por el que se conoce un elemento, mientras que el subíndice, 1,2,..., es la posición en la secuencia.

En programación una secuencia de este tipo se denomina matriz o <<array>>

Un matriz es una colección ordenada de objetos o elementos. La mayoría de los lenguajes de programación requieren que todos los elementos sean del mismo tipo (enteros, reales o caracteres).

Las matrices se clasifican en vectores (matrices unidimensionales) y tablas o matrices propiamente dichas (matrices multidimensionales).

5.3.1.1 Vectores

Un vector es un conjunto ordenado de elementos que contienen un número fijo de ellos. Los vectores son matrices unidimensionales, sus elementos deben ser del mismo tipo (enteros, reales o caracteres).

Un vector es: (2, 3, 4, 8, 9, 10, 15,17)

Y representa una lista de elementos.

La longitud de un vector es el número total de elementos
Cada elemento de un vector se representa por el nombre del vector y un índice o subíndice. En el ejemplo anterior si x es el nombre del vector, los elementos son:

$X_1=2$ $x_2=3$ $x_3=4$ $x_4=8$ $x_5=9$ $x_6=10$ $x_7=15$ $x_8=17$

En lenguajes de programación, el índice se escribe normalmente después del nombre del vector y enmarcado entre paréntesis (BASICO o FORTTRAN) o corchetes (en PASCAL o ALGOL). Los elementos x1 y x2anteriores se representan por:

X(1), x(2),	*o bien*	*x[1], x[2],*
BASIC, FORTRAN		*PASCAL*

Un ejemplo típico de un vector son las temperaturas diarias del mes de febrero de una ciudad.

(20 25 26... 17)

o bien,

$(X_1)\ (X_2)\ (X_3)...\ (X_{28})$

Los elementos del vector podrían ser caracteres

Un ejemplo de una aplicación de un vector, es el diagrama de flujo de la figura 5.7 que calcula la media (media aritmética), de estructuras de los alumnos de una clase, cuantos son mas altos que la media, cuantos mas bajos.

n es el número de estudiantes de una clase
a[1] a[2] … a[n] son sus estaturas.

5.3.1.2 Matrices

Una matriz con un solo índice se denomina unidimensional. Una matriz con 2 o más índices se llama multidimensional. El número de índices necesario para especificar un elemento de una matriz se denomina dimensión.

Las matrices más útiles son las de dos dimensiones, que se asemeja a una tabla o conjunto de filas (primer índice) y columnas (segundo índice) como se ve en la figura 5.6.

Los elementos de una matriz de 2 dimensiones se representan por 2 expresiones separadas por una coma y encerradas entre corchetes o paréntesis.

A[I,J] en PASCAL A(I,J) en BASIC.

Se refieren al elemento de la fila i y columna j.

En lenguajes de programación las matrices se suelen denominar tablas. En matemáticas se suele representar con las notaciones

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \quad o\ bien \quad \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

$$\begin{bmatrix} 3 & 4 & 5 \\ 1 & 2 & 3 \\ 4 & 0 & 7 \end{bmatrix} \quad o\ bien \quad \begin{pmatrix} 3 & 4 & 5 \\ 1 & 2 & 3 \\ 4 & 0 & 7 \end{pmatrix}$$

Fila 1	Columna 1	Columna 2	Columna 3		
Fila 2					A (2,3)
Fila 3					A (3,1)

Figura 5.6 Matriz de dos dimensiones

Una tabla o matriz de dimensiones se puede considerar un vector de vectores. Es por consiguiente un conjunto de elementos, todos del mismo tipo, en los que el orden de los elementos es significativo y en el que se necesitan 2 subíndices para definir un elemento.

Las matrices pueden tener más de dos dimensiones y se denomina multidimensional. `Por ejemplo, un sistema de reserva de plazas en una línea aérea, se podrían representar por una matriz de 3 dimensiones:

A[i,j,k]

donde,

i=1,2,...,5 representa el número de vuelo

j=1,2,...,60 representa la fila de avión

k=1,2,...,12 representa el asiento en la fila

Así,

asiento [i,j,k]=0 asiento libre

asiento [i,j,k]=1 asiento ocupado

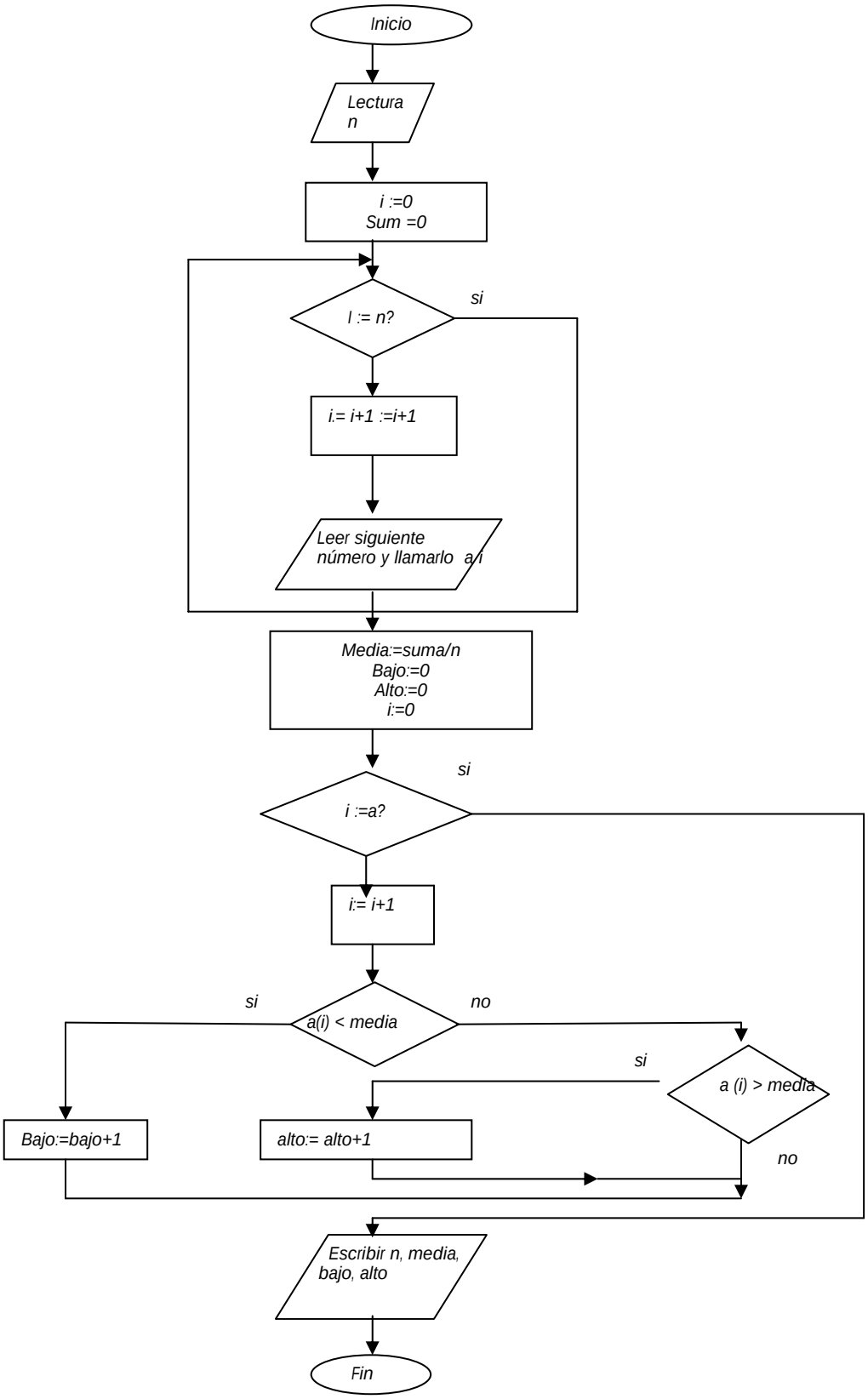


Figura 5.7. Diagrama de flujo para cálculo de la media de varios números

Almacenamiento de matrices.

Las matrices se almacenan en memoria de modo secuencial. Una matriz (i,j) equivale a un vector de i x j elementos. La figura 5.8 compara el almacenamiento de la matriz m (3,2) con el almacenamiento de un vector de sus elementos y muestra los 2 posibles métodos de almacenar las matrices en memoria: por filas o por columnas.

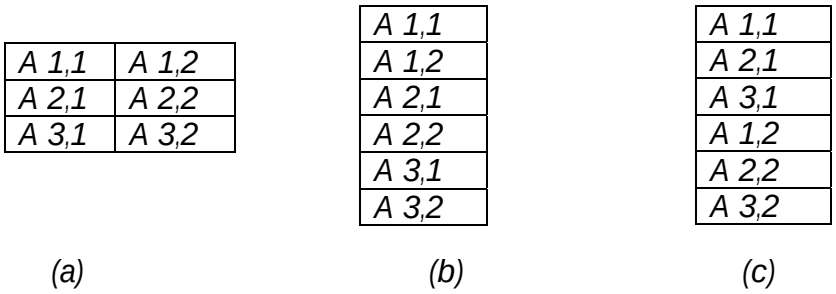


Figura 5.8 *(a) Matriz a 3,2: (b) Matriz por filas (c) Matriz por columnas*

FORTRAN almacena las matrices por columnas. PASCAL, PL/I, BASIC y COBOL almacenan las matrices por filas.

El compilador o intérprete correspondiente se encarga de almacenar en memoria adecuadamente las matrices en función de los subíndices y dimensiones de las mismas.

El compilador de un lenguaje de alto nivel genera instrucciones que se encargan de calcular la posición de cada variable subindicada y debe conocer si la matriz está almacenada por columnas o por filas, así como en los subíndices y las dimensiones máximas de la matriz. Por ejemplo, si la matriz esta almacenada por columnas, el elemento A [3,1] esta en la tercera posición, A [1,2] en la cuarta y así sucesivamente.

Las matrices o tablas pueden ser de 2,3, o 4 dimensiones.

5.4 LISTAS

Una lista es un conjunto o serie de electos (objetos del mismo tipo, denominados también átomos), en la que cada uno de ellos esta relacionado implícita o explícitamente con el anterior y/o con el siguiente. Las listas deben estar almacenadas en un soporte direccionable (memoria central o periférico de acceso seleccionable, cintas o discos). Para designar un elemento se utiliza un puntero.

Para hacer referencia a una lista se debe conocer donde comienza, donde termina o su longitud. El tratamiento o manipulación de lista exige poder:

- *Acceder al primer elemento (cabeza de la lista)*
- *Acceder a otros elementos*
- *Suprimir o añadir elementos*

La realización de las operaciones anteriores presenta diferentes dificultades. Por ejemplo, añadir un elemento a una lista de compras o un listado de clientes es fácil, ya que solo es necesario poner los elementos al final. Sin embargo si el listado de clientes ha de ser en orden alfabético, seria preciso intercalar el elemento en su lugar correspondiente. La eliminación presenta los mismos problemas. Las listas, se almacena en la memoria de una computadora.

Las listas se pueden clasificar en diferentes grupos:

- *Lineales*
- *Encadenadas*
- *Circulares*
- *Y doblemente encadenadas*

5.4.1 Listas lineales

Una lista lineal es aquella que tiene un primer y un ultimo elemento, existiendo una relación estructuras entre sus diferentes elementos; normalmente cada elemento esta relacionado con el anterior y el siguiente, excepto el primero –solo con el segundo- y el último –solo con el anterior-. Las listas lineales se llaman también densas debido a que los elementos de la lista se encuentran en posición de almacenamiento (memoria) físicamente contiguos.

En general se suelen expresar las listas como un conjunto lógico de átomos; un átomo es un conjunto de elementos relacionados llamados campos, siendo el campo la unidad básica de información. Un campo se puede dividir en subcampos. Si se trabaja con ficheros, el termino registro es sinónimo de átomo (el concepto átomo procede de los orígenes del lenguaje LISP en los años 60's).

Los elementos de una lista pueden ser de cualquier tipo y se caracterizan por la posición que ocupan.

Las operaciones fundamentales que se pueden realizar con los elementos de una lista son:

- *Calcular la longitud de una lista- número de elementos de la misma*
- *Consulta secuencial de cada uno de los elementos*
- *Acceso a un elemento n, examen y/o modificar su contenido*
- *Inserción de un nuevo elemento en la posición n de la lista*
- *Almacenar un elemento en la posición n de la lista*
- *Suprimir un elemento de la lista*
- *Reorganización de la lista: fusión de 2 o mas listas de una sola: división de una lista en varias*
- *Clasificación de los elementos según determinados criterios.*

Inserción de un elemento

La operación de inserción conserva todos los elementos de la lista, desplazando a aquellos cuyo número de orden es superior al que tratamos de insertar

e_1	e_2	...	e_n	e_{n+1}

elemento 1 elemento2 elemento n elemento n+1

Después de insertar el elemento ex en la posición n de la lista quedará la siguiente lista

e_1	e_2	...	e_x	e_{n+1}

elemento 1 elemento2 elemento n elemento n+1

Almacenamiento de un elemento

El almacenamiento de un elemento supone borrar el contenido que hubiera en la posición donde se desea almacenar. En el caso de almacenar ex en la posición 2 quedaría:

e_1	e_x	e_3	...	e_n	e_{n+1}	...

Supresión de un elemento

Excepto en la cabeza o final de la lista la supresión de un elemento exige un desplazamiento de los elementos posteriores. Otro método consiste en marcar los elementos considerándolos inactivos.

5.4.1.1 Colas.

Una cola es una subclase de lista lineal en la que se permiten las eliminaciones en el comienzo de la lista (el extremo llamado frente <front>) y las inserciones, se realizan al final de la lista (extremo opuesto <read>) los elementos de este tipo de lista se procesan en el mismo orden en que se reciben el primero en entrar es el primero en salir (FIFO-first-out).

La figura 5.9 representa una estructura tipo cola en la cual las inserciones se realizan al final y los elementos se eliminan en el frente de la estructura.

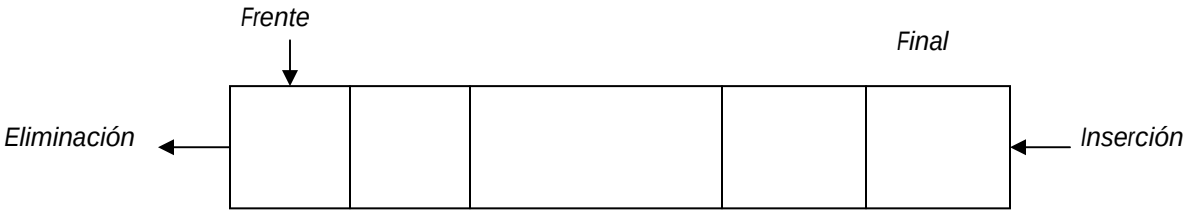
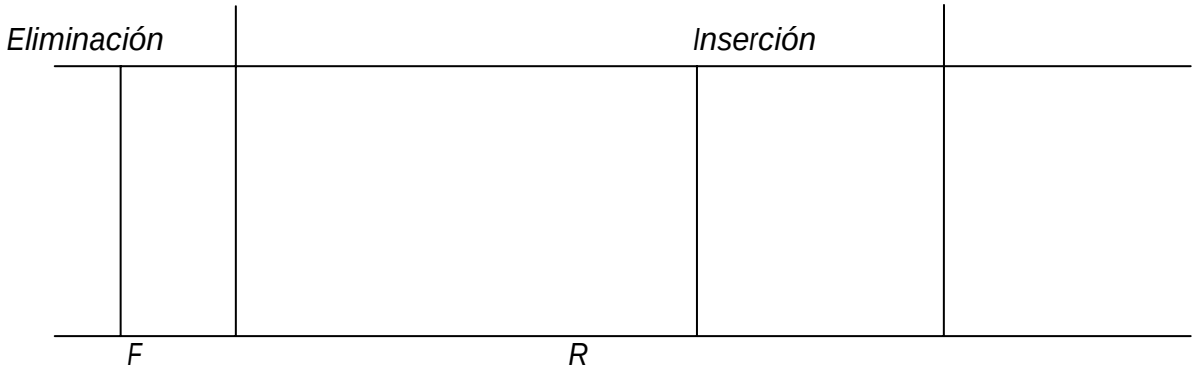


Figura 5.9. Estructura tipo cola

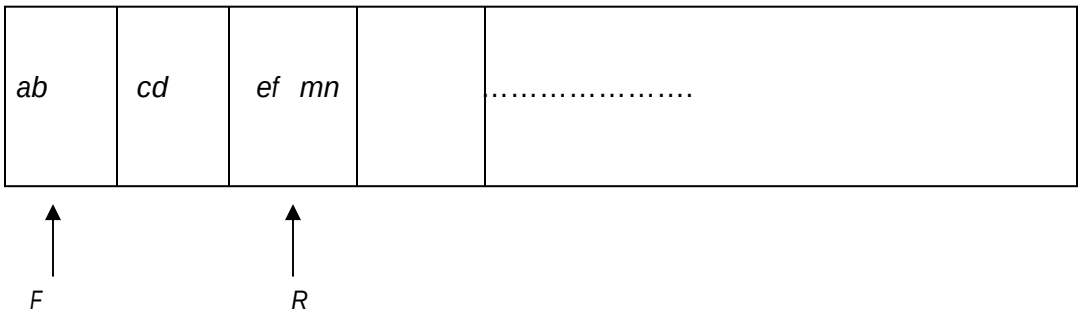
Un ejemplo típico se muestra en los sistemas de computadoras de tiempo compartido, en el cual muchos usuarios comparten el mismo recurso informativo. Con frecuencia, se comparte la unidad central de proceso y la memoria principal. Estos recursos se comparten permitiendo que el programa de un usuario se ejecute durante un pequeño espacio de tiempo, seguido de un segundo programa de otro usuario, y así sucesivamente. Se utiliza una cola para almacenar los programas de los usuarios que están esperando su turno de ejecución. Sin embargo, una cola de este tipo no siempre trabaja en un estricto orden, primero en entrar, primero en salir, sino que trabaja con un esquema de prioridades basada en varios factores: tiempo de ejecución requerido, número de líneas de salida, hora del día, etc.

Otro tipo de cola muy conocida en informática son las colas de impresión para permitir la impresión de diversos documentos procedentes de diversas fuentes, con una sola impresora.

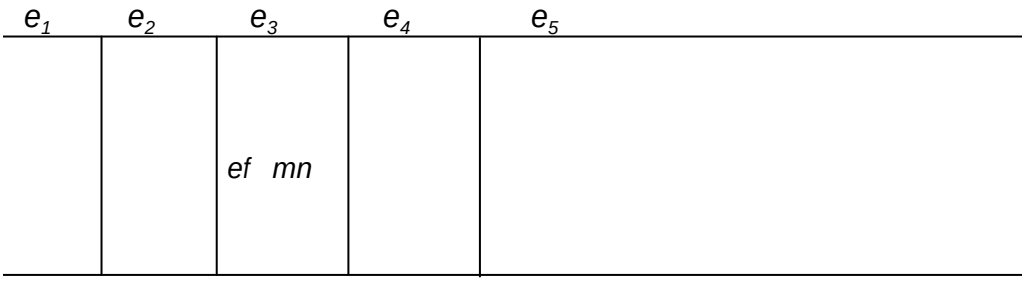
Las colas se representan con 2 punteros F (frente) y R (Final) para denotar las posiciones frontal y final de los elementos, respectivamente.



La operación de inserción (elemento m,n) en una cola se muestra a continuación



La operación de supresión de un elemento se puede realizar indicando el número de orden que se le asignó cuando ese mismo elemento fue insertado, de modo que el extremo frente se desplaza con cada operación de supresión



La estructura cola es frecuente en la vida diaria, y ejemplos típicos son: la cola de un cine, la hilera de automóviles parada frente a un semáforo, etc.

5.4.1.2 Pilas

Una pila es una subclase de lista lineal en la que las inserciones y eliminaciones se realizan por un solo extremo y solamente el ultimo elemento resulta accesible –el que esta arriba de la pila, la cima o cabecera de la pila <top>-. La adición de un elemento a la pila se produce siempre por la parte superior de la pila. El único elemento accesible de la pila es el elemento cima y el menos accesible el situado en el fondo. Esta estructura se conoce por LIFO, last-in, first-out (ultimo en entrar, primero en salir).

La noción de pila nos es familiar en la vida diaria: pila de libros, pila de platos, etc. Para añadir un plato se coloca arriba de la pila, para tomarlo se coge de la parte superior de la pila.

En una pila, las inserciones y las supresiones se efectúan en la misma punta de la pila, llamada cabecera de pila. En una cola, los elementos se insertan por un extremo y van desplazándose por el interior de la cola hasta llegar al otro extremo, donde son borradas.

El diagrama de una pila es:

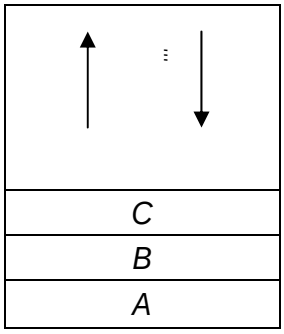
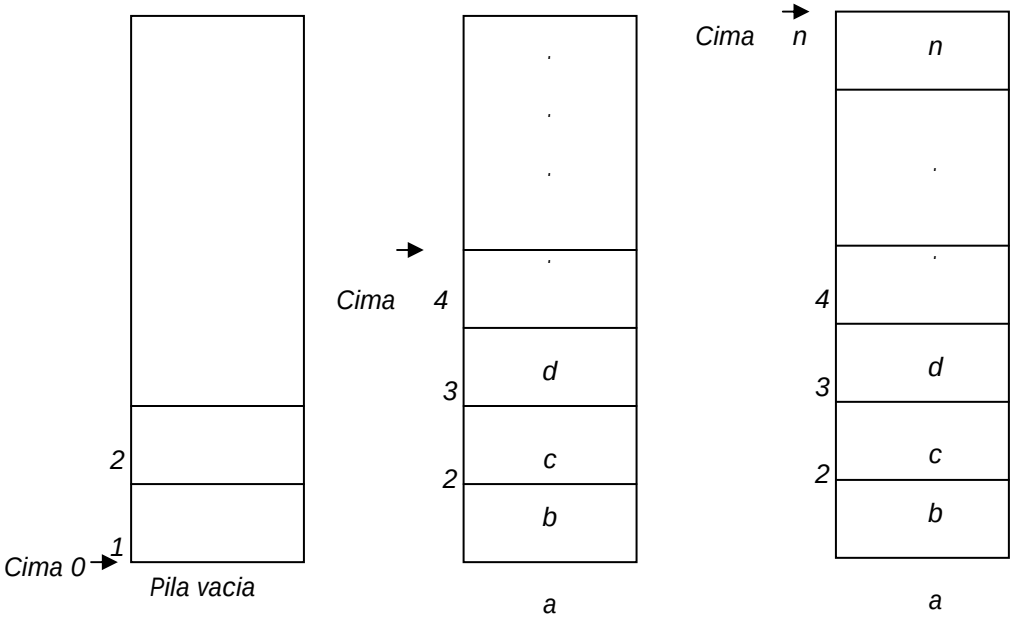
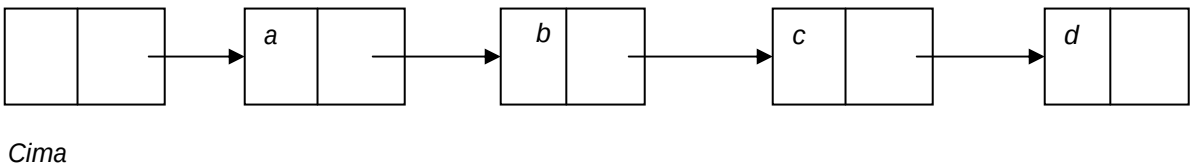


Figura 5.10. Diagrama de una pila

Representación física de una pila



Esta representación puede realizarse con una lista simple encadenada.



En los sistemas de computadora las pilas se utilizan para almacenar información de modo que puede volver a recuperarse cuando se desee. El ejemplo más típico son las llamadas a subrutinas o funciones, en el que la dirección del programa en curso se almacena en la pila, de modo que tras la ejecución de las subrutinas o funciones, el control del programa retorna a la dirección donde debe reanudarse la ejecución del programa.

Supongamos un programa PRINCIPAL que llama a la subrutina DEMO y luego a PRUEBA, y a su vez PRUEBA llama a la subrutina BLANCO y luego a la subrutina ROJO (figura 11).

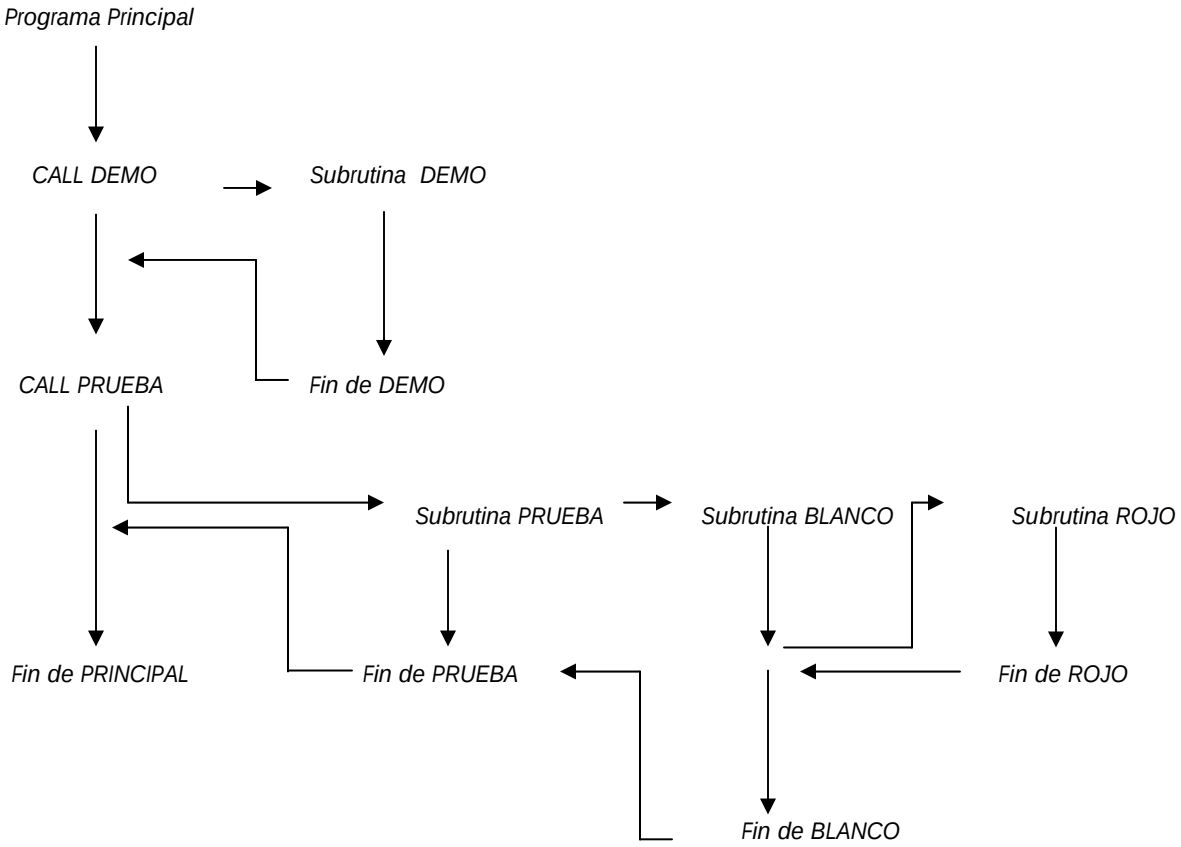


Figura 5.11. Llamadas sucesivas a subrutinas

La pila de ejecución del programa PRINCIPAL se muestra en la figura 5.12

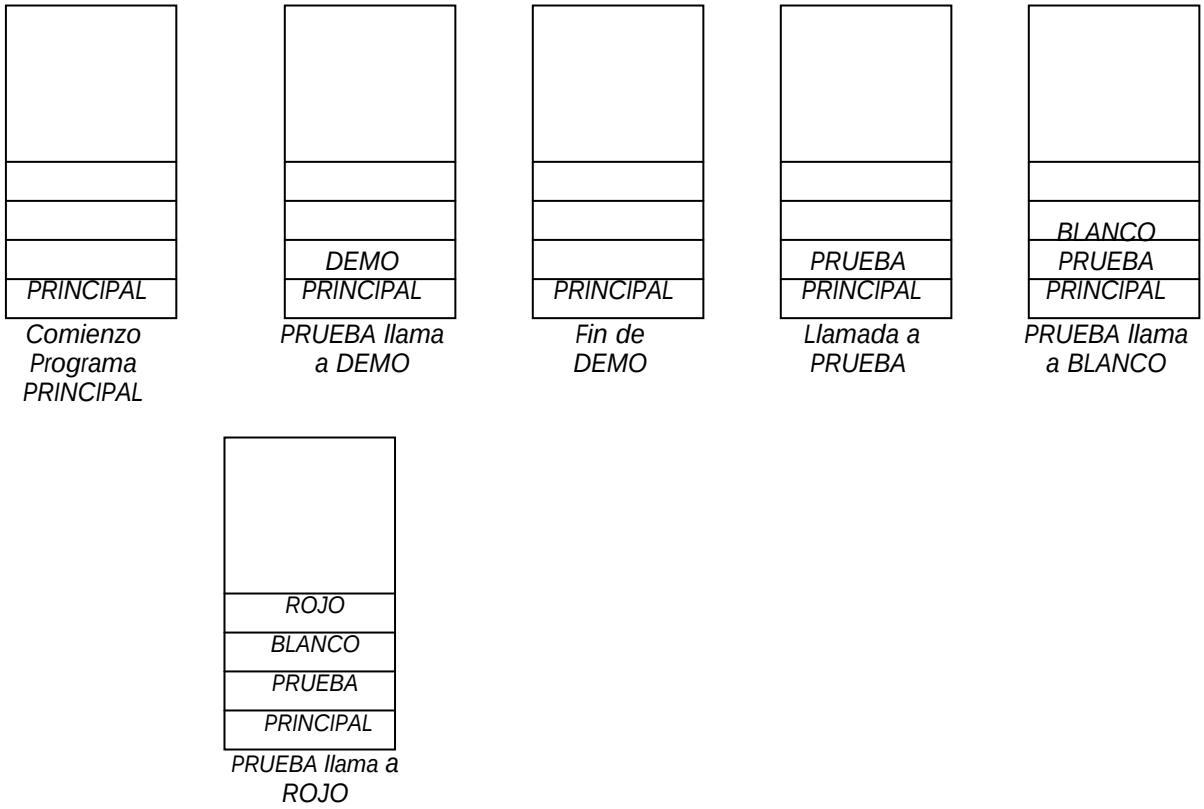


Figura 5.12. Programa principal y la pila

Las operaciones de situar o insertar un elemento en una pila se conoce por empujar (push) y la operación de extracción o borrado por expulsar (pop).

5.4.2 Listas encadenadas

Aunque como se ha visto en el párrafo anterior las listas densas tienen una estructura simple, presentan algunas desventajas: dificultades en las inserciones o supresiones –sobre todo si se ha de conservar ordenada-, modificación del tamaño o formato de la lista, etc. La lista esta sometida a una continúa reorganización que es aconsejable.

Las listas encadenadas son aquellas en las que cada elemento de la lista se incluye un campo especial llamado puntero (pointer) o apuntador que sirve para enlazar dicho elemento con el siguiente de la lista, evitando así la ocupación de posiciones contiguas de memoria. La mayor ventaja de las listas encadenadas es la utilización eficaz de la memoria, ya que se puede utilizar cualquier espacio vacío por no elegir posiciones secuenciales de memoria. Por el contrario, su mayor inconveniente es la lentitud del proceso, ya que es necesario ir recorriendo elemento a elemento, a través de los punteros para poder acceder a uno especifico.

Así una organización secuencial seria:

$X_{(1)}$	$X_{(2)}$	$X_{(3)}$	...	$X_{(n-1)}$	X_n

En la organización encadenada cada elemento de la lista tiene 2 partes: la primera contiene la información perteneciente a este elemento, y la segunda parte contiene la dirección del siguiente elemento de la lista. Nótese que el último elemento de la lista no tiene sucesor; el campo del puntero no contiene una dirección real.

La figura 5.13 representa una lista encadenada con el puntero PRIMERO conteniendo la información de un elemento y la segunda parte la dirección del siguiente elemento

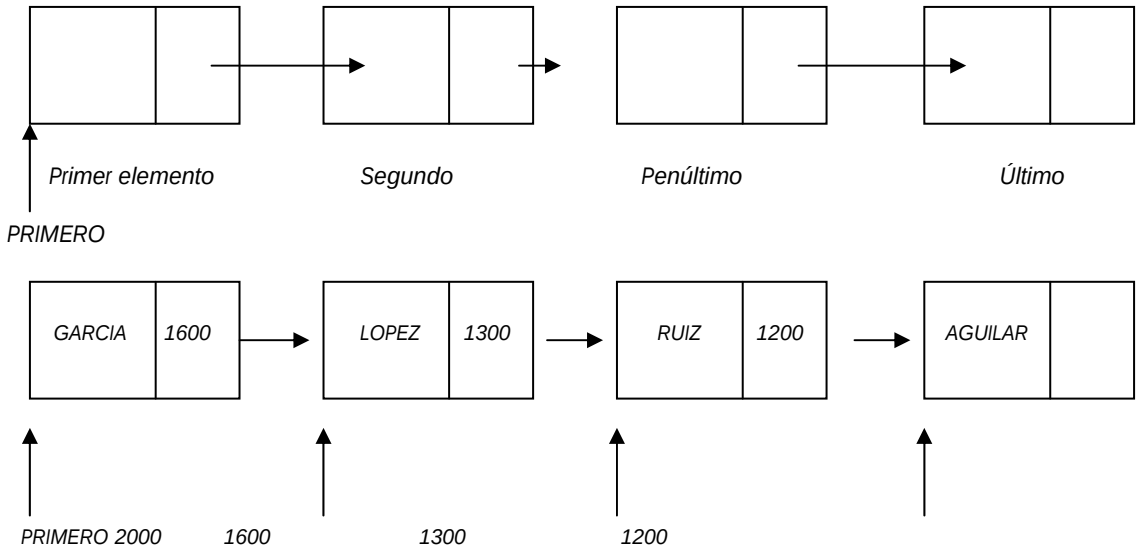


Figura 5.13. *Lista encadenada*

El último elemento contiene una dirección vacía, NULO (una diagonal).

La operación de inserción en una lista encadenada es muy fácil ya que solo implica un intercambio de punteros. Por ejemplo, la figura 5.13 muestra la inserción del elemento <DELGADO> entre el segundo y el tercero de la lista compuesto de cuatro elementos suponiendo que esta almacenado en la posición 1100. El campo del encadenamiento del segundo elemento se cambia de 1500 a 1100; se le asigna también el puntero del nuevo elemento el valor 1400.

La operación de borrado demuestra la eliminación del tercer elemento <RUIZ> de la lista original. El valor del puntero del segundo elemento se borra de 1300 a 1200.

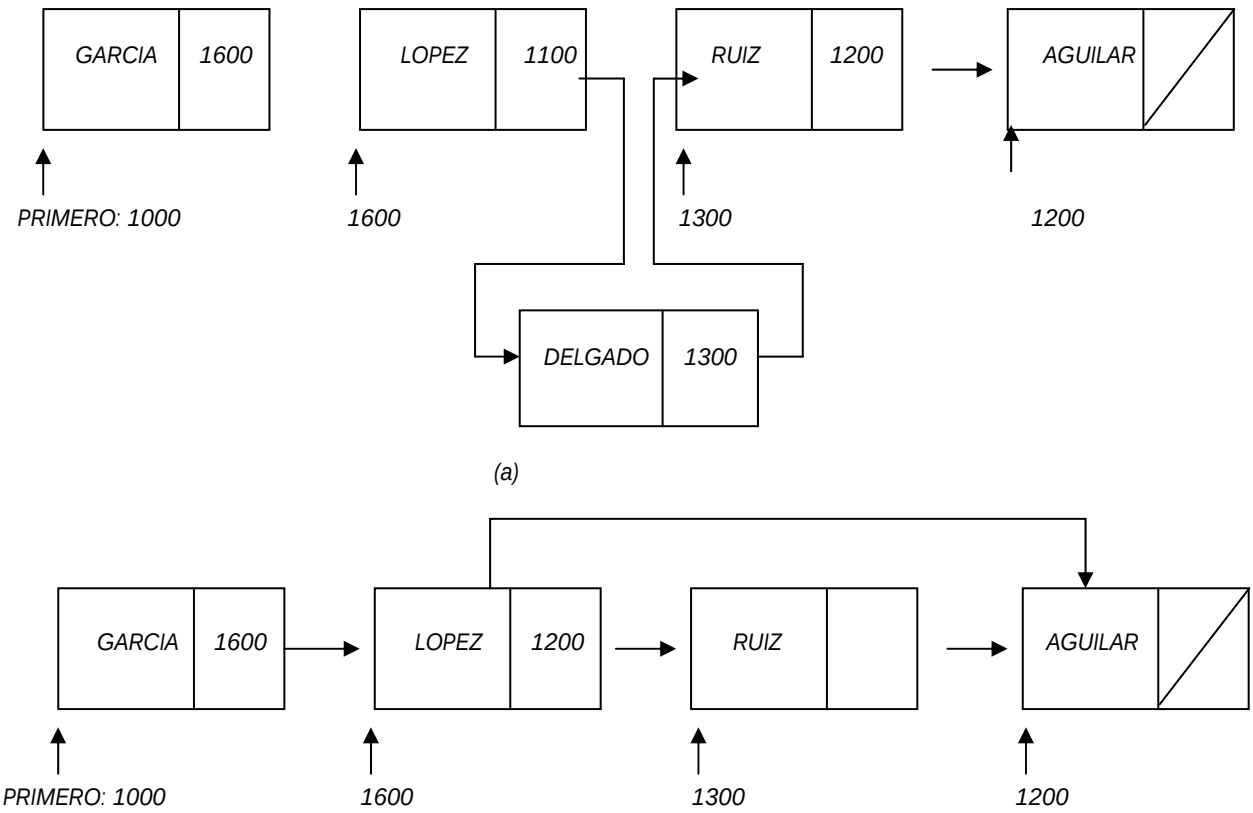


Figura 5.14. Inserción/borrado en una lista encadenada

La lista encadenada, que necesita un enlace o puntero entre 2 elementos consecutivos de la lista se utiliza en el caso de que el número de eliminaciones e inserciones de elementos sea muy grande, ya que basta modificar el puntero del elemento anterior en el caso de eliminaciones y añadir un nuevo puntero en el caso de una inserción.

La manipulación de listas se puede hacer en cualquier lenguaje de programación, aunque existan lenguajes especiales para su manipulación, como son: LIST, SNOBOL, etc.

5.4.3 Listas circulares

Esta estructura conocida también como anillo es una subclase de una lista lineal consistente en enlazar el último elemento de la lista (con el puntero) con el primero.

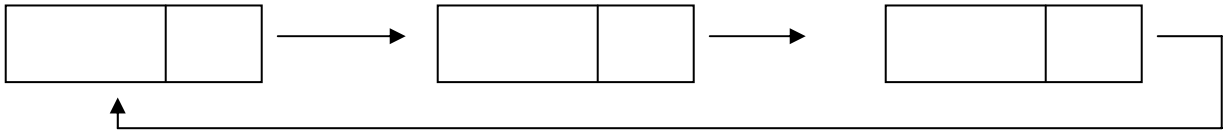


Figura 5.15.

Este tipo de lista es cerrada. Puede recorrerse circularmente, en el sentido indicado por los punteros. Las listas circulares presentan el inconveniente de los bucles o lazos infinitos que se presenta cuando no tienen especial cuidado en detectar el final de la lista.

5.5 LISTAS DOBLEMENTE ENCADENADAS

Hasta este momento las listas lineales se han recorrido de izquierda a derecha. En números ocasiones se necesita recorrerlas en ambas direcciones. La explotación en los dos sentidos puede realizarse con dos punteros, en lugar de uno. De este modo se puede utilizar un puntero para localizar el registro procedente y otro para el siguiente.

Las listas lineales son doblemente encadenadas cuando su estructura contiene 2 campos o punteros encadenados: uno que señale el elemento siguiente, el que apunta al elemento precedente

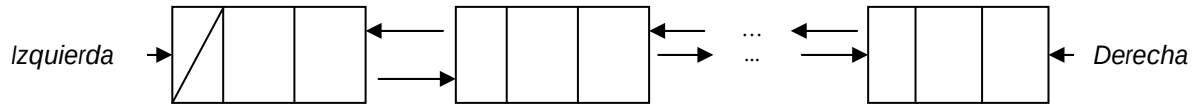


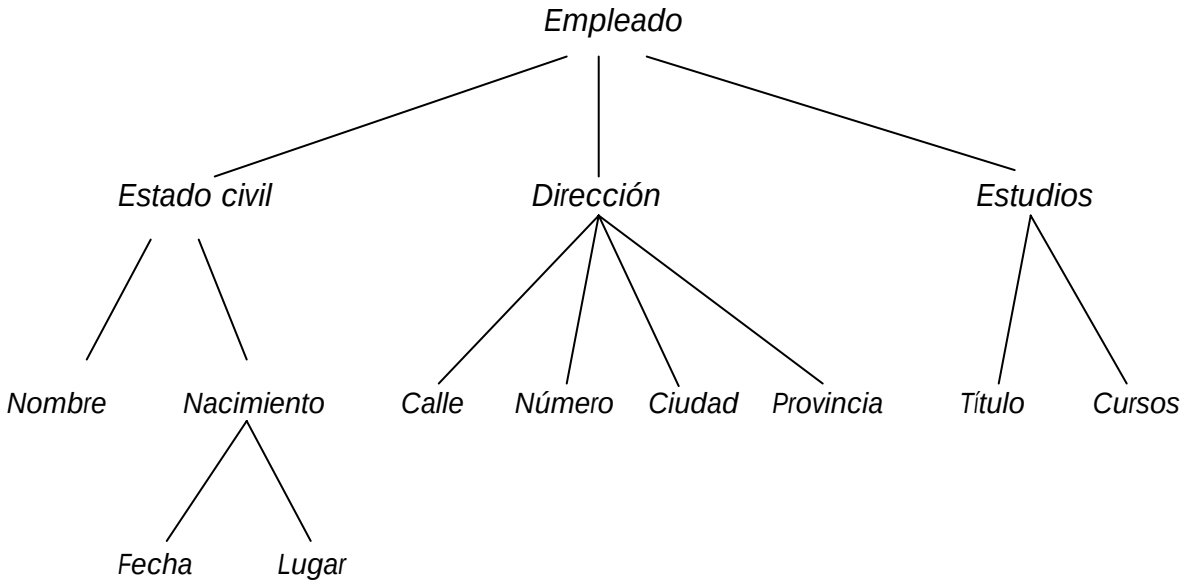
Figura 5.16.

Un uso de listas de varios punteros es llevar la cuenta en un orden diferente al ordinario.

5.6 ÁRBOLES

Las estructuras de datos estudiados anteriormente no permiten representar informaciones jerárquicas. Sin embargo en informática existen múltiples aplicaciones donde se utilizan este tipo de organizaciones.

Algunos ejemplos de representaciones jerárquicas de estructuras de objetos son:

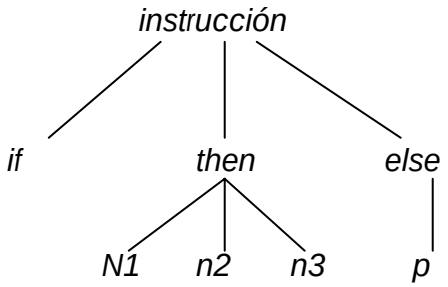


Esta estructura puesta en forma de variable compuesta declarada en COBOL seria:

```
01 Empleado
   02 Estado civil
      03 Nombre
      03 Nacimiento
            04 Fecha
            04 Lugar
   02 Dirección
         03 Calle
         03 Número
         03 Ciudad
         03 Provincia
  02 Estudios
        03 Título
        03 Cursos
```

Las instrucciones pueden representarse en un lenguaje de programación como una **estructura arborescente**.

```
If m then
      begin
      n1;
      n2;
      n3;
      End
      Else p
```



Estructura en árbol

5.6.1 Arborescente (definición).

Un árbol, estructura arborescente o arborescencia es una estructura no lineal en la que cada elemento esta relacionado con dos o mas elementos. Un árbol esta conformado de un conjunto de nodos (unidades de información) en las que además de la propia información contiene información de otros nodos de menor importancia o jerarquía, y cumple con las siguientes condiciones:

a) *Existe un nodo raíz.*
b) *El resto de los nodos se distribuye en un número n de subconjuntos indistintos.*
c) *Cada uno de esos subconjuntos es un subárbol del nodo raíz.*

En el ejemplo de estructura EMPLEADO, existe un árbol raíz (EMPLEADO) y por los tres subárboles: ESTADO CIVIL, DIRECCION Y ESTUDIOS. Los elementos de una estructura se denominan nodos. El nodo A es el nodo raíz del árbol

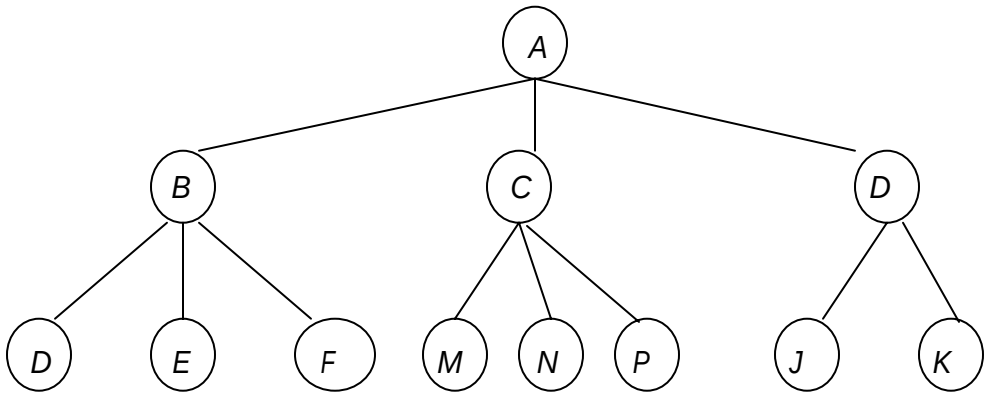


Figura 5.17.

El nodo B es nodo del subárbol, el nodo C es raíz del subárbol 2 y el D es raíz del subárbol 3.

Grado de un nodo *es el número de subárboles de ese nodo. El grado de nodo A es 3 y el del nodo D es 2.*

Los nodos de grados terminales también se les llama hojas.

Las líneas que unen dos nodos se llaman lados o aristas o ramas.

Camino de un nodo *es el conjunto de aristas a través de las cuales se pasan desde el nodo raíz a ese nodo.*

A cada nodo se le asocian uno o varios subárboles llamados descendientes o hijos.

Los nodos B, C y D son hijos del nodo A.

Los nodos hermanos son los sucesores directos de un mismo nodo (hijos de un mismo padre). D, E y F son todos hermanos. F y P no son nodos hermanos.

Si el nivel del nodo raíz es uno, el nivel de un nodo es el de su padre mas uno.

Se dice que un árbol es n-ario (n, constante entera). Un árbol binario es un árbol en el cual cada nodo tiene, y como máximo, dos hijos. Todo árbol n-ario se puede transformar en un árbol binario equivalente. El árbol EMPLEADO es un árbol ternario.

Un árbol no puede estar vacío: contiene como mínimo un elemento, la raíz.

Un ejemplo gráfico completo seria:

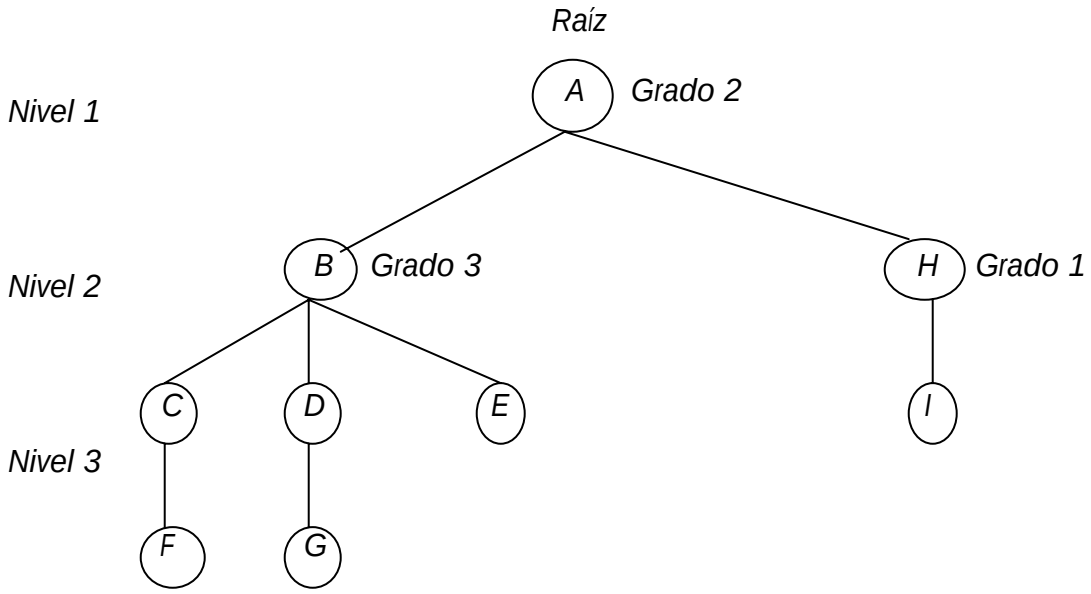


Figura 5.18.

Otras formas de representar un árbol son:

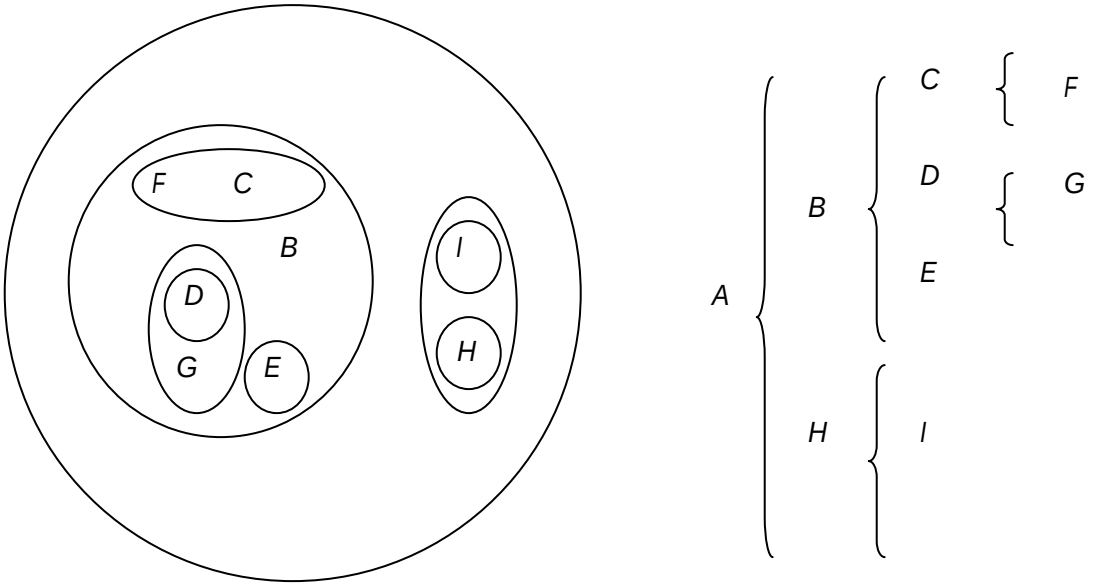


Figura 5.19.

5.6.2 Árboles binarios

Una de las estructuras más comunes de los árboles, es aquella en que cada nodo tiene como máximo dos subárboles (grado 2), que se conocen como subárbol izquierdo y derecho. Dos árboles binarios contiguos son distintos aunque esas representaciones sean idénticas (igual árbol en el caso de árboles no binarios).

Son árboles iguales, pero árboles binarios distintos.

Figura 5.20.

La altura o profundidad de un árbol binario es el número de nodos que constituyen el camino más largo desde la raíz a una hoja.

Un problema típico consiste en la conversión de un árbol ordinario en un árbol binario equivalente.

Un árbol binario se dice que esta completo o lleno cuando tiene todos los nodos de grado 2 excepto los nodos terminales.

Ejemplo:

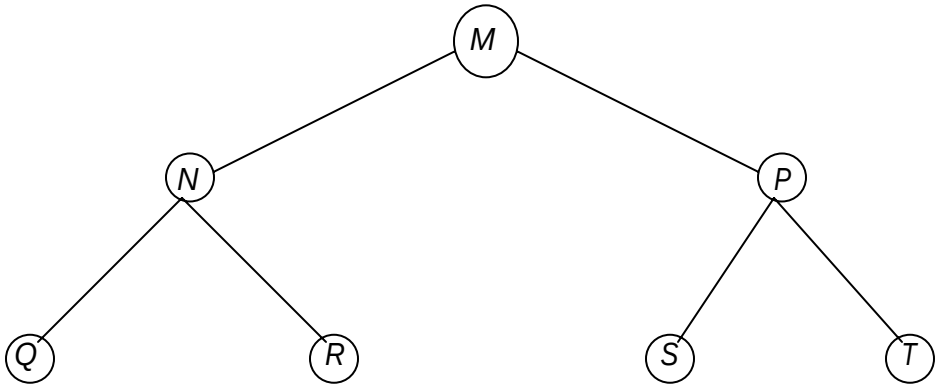


Figura 5.21.

El problema de convertir un árbol en cualquier árbol binario se representa en la figura 5.22.

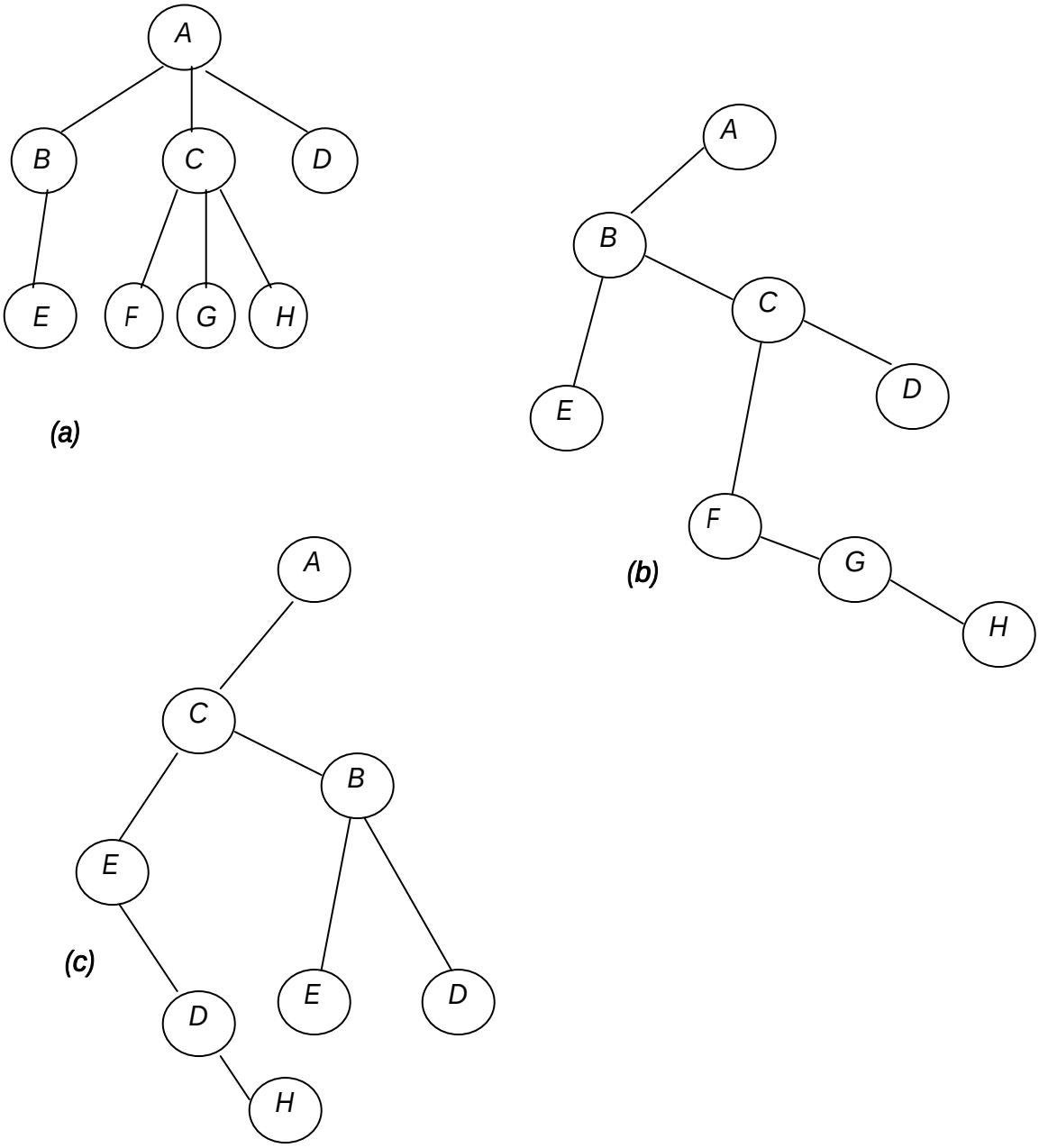


Figura 5.22. Conversión de un árbol cualquiera en árbol binario.

5.6.3 Recorrido de un árbol binario

Existen diferentes maneras de recorrer un árbol para tratar los diferentes nodos que le constituyen. Supongamos el árbol binario:.

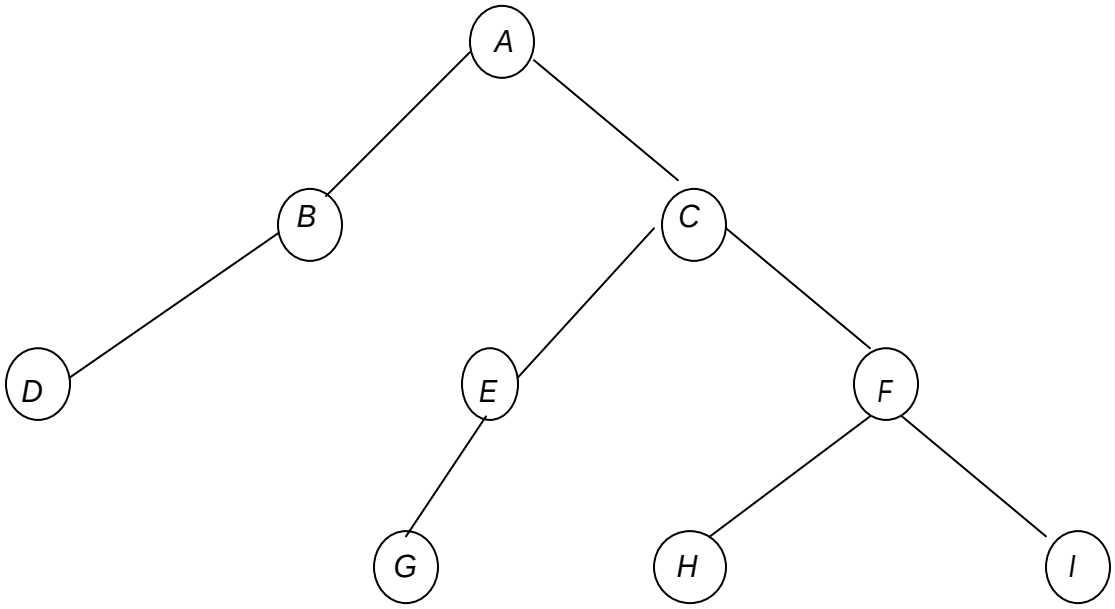


Figura 5.23.

Las acciones a considerar son tres y según el orden en que se efectúen se tendrá el recorrido en un sistema o en otro

- *Posicionarse en la raíz*
- *Recorrer el subárbol izquierdo*
- *Recorrer el subárbol derecho*

Los tres tipos de recorridos mas frecuentes y su aplicación concreta al árbol anterior son:

Recorrido fijo o preorden

- *Posicionarse en la raíz*
- *Recorrer el subárbol izquierdo*
- *Recorrer el subárbol derecho*

Recorrido postfijo o postorden

- *Recorrer el subárbol izquierdo*
- *Recorrer el subárbol derecho*
- *Posicionarse en la raíz*
  D B G E H I F C A

Recorrido en orden o simétrico

- *Recorrer el subárbol izquierdo*
- *Posicionarse en la raíz*
- *Recorrer el subárbol derecho*
  D B A G E C H F I

APENDICE A

Breve curso de programación en C

CONCEPTOS PREELIMINARES:

SOFTWARE Y SU CLASIFICACION:

Todos los programas de cómputo pueden clasificarse principalmente en las siguientes áreas:

* DE SISTEMAS

Abarca en general a los sistemas operativos como el antiguo MS-DOS, y las versiones actuales de Windows y todos los programas que se encargan de la comunicación de computadora a computadora para conformar redes, entre ellas INTERNET. Los programas enfocados a sistemas también se emplean para permitir la comunicación de la computadora con sus distintos periféricos (impresora, escáner, módem etc.). Este tipo de programas regularmente son desarrollados por ingenieros en computación, pues se orientan fuertemente al manejo del Hardware.

Recuérdese que el sistema operativo es un conjunto de programas que permiten a un usuario la explotación y administración de todos los recursos de un sistema de cómputo. Sin sistema operativo NO es posible el uso de una computadora ni tampoco el uso de ningún programa. Los sistemas operativos pueden clasificarse en los que se basen en comandos, como el MS DOS, en el cual cada tarea tal como visualizar el contenido de un disco, cambiar el nombre de un archivo o borrarlo; está asociada a un comando u orden similar al inglés. Estos comandos pueden estar acompañados de parámetros que pueden modificar el comportamiento de un comando de acuerdo a las necesidades del usuario. Su relativa desventaja es que para cada tarea que se desee realizar se debe conocer el nombre del comando y sus parámetros. No obstante, éste tipo de sistemas operativos aparentemente rústicos y poco estéticos son muy estables, raramente fallan y cuando surgen errores es relativamente fácil restaurarlos y recuperarse de ellos.

En el otro extremo de los sistemas operativos a base de comandos, se encuentran los sistemas operativos visuales, tales como Windows, dónde todas las tareas están representadas por íconos , "ventanas" y "botones". No requieren conocer de memoria una gran cantidad de comandos, Su uso es fácil e intuitivo. Sin embargo este tipo de sistemas operativos tan "amigables para el usuario" y tan estéticos son bastante inestables y generalmente al presentar fallos vienen problemas como la pérdida irreversible de información. Es muy difícil y a veces casi imposible restaurar el sistema operativo después de un error grave y generalmente tendrá que procederse a dar formato al disco y volver a instalar Windows. Según prometió el controvertido "Bill" Gates, su nuevo Windows XP es muy estable y prometió una reducción considerable de fallas graves. (¿?)

Otro tipo de sistemas operativos de gran auge y que cada vez cobran mayor importancia son el UNIX y LINUX, que por lo regular se emplean en redes de computadoras. Son sistemas muy confiables y muy poderosos, por lo que son empleados generalmente en grandes compañías y centros de investigación.

* DE APLICACIÓN GENERAL

Regularmente se concentran en procesadores de textos (Como Word), hojas de cálculo (como Excel), bases de datos (como Access) y en programas para todo tipo de cálculos y tareas, incluyendo el diseño gráfico, dibujo, y entretenimiento entre otros.

* LENGUAJES DE PROGRAMACION

Son programas muy parecidos a cualquier lenguaje tal como el español, en el sentido de que se rigen bajo un conjunto de reglas gramaticales de sintaxis y semántica. Un lenguaje de programación permite traducir un algoritmo a un lenguaje y este a su vez se convierte a lenguaje máquina para ser ejecutado y realizar una tarea o tareas.

Los lenguajes de programación pueden clasificarse de acuerdo a su potencia y complejidad:

LENGUAJES DE ALTO NIVEL: *La sintaxis (reglas de escritura) es muy similar al inglés, por lo que son fáciles de aprender. Se emplean para resolver una amplia gama de problemas. Sin embargo no permiten fácilmente la manipulación del hardware. Su potencia y velocidad es relativamente limitada, sobre todo en programas grandes. Ejemplos de lenguajes: BASIC, FORTRAN, PASCAL, COBOL, ADA, ALGOL.*

LENGUAJES DE BAJO NIVEL: *Su sintaxis es similar al código máquina (sistema binario ó hexadecimal) por lo que son muy difíciles de aprender. Permiten la manipulación del hardware. Son muy potentes y veloces, aún en programas grandes, Regularmente se usan para programar sistemas operativos, virus, vacunas, juegos y aplicaciones tales como ¡Windows!. Un ejemplo de éste tipo de lenguajes es el Ensamblador. Desafortunadamente, no existe un solo lenguaje ensamblador, pues para cada familia de microprocesadores (Pentium, Celeron, AMD etc.) existe su propio lenguaje.*

LENGUAJES DE NIVEL MEDIO: *Son aquellos que combinan la facilidad de un lenguaje de alto nivel con la potencia y velocidad de uno de bajo nivel. Ejemplo: Lenguaje C. Este lenguaje es una magnífica alternativa al uso del complicado ensamblador. El famoso sistema operativo UNIX está programado en lenguaje C.*

Además, existe otra clasificación de los lenguajes de programación, a saber:

LENGUAJES PROCEDURALES (O ESTRUCTURADOS): *Anteriormente, los programas de computadora resolvían problemas de manera secuencial (una instrucción detrás de otra). Al final de los años 70 e inicio de los 80 surge un nuevo estilo de programación, el de la programación estructurada. Su idea fundamental consiste en "descomponer" un problema dado en "pequeños" problemas más fáciles de atacar. Resolver cada pequeña parte por separado y al final juntar todos los resultados en uno sólo. Este estilo es muy eficiente y se empleó durante toda la década de los 80. Ejemplos de estos lenguajes son: Turbo Basic, Turbo Pascal y Turbo C.*

LENGUAJES ORIENTADOS A OBJETOS: A finales de los 80´s e inicio de los 90´s, la programación estructurada o procedural evoluciona a un nivel superior: la programación orientada a objetos. Este estilo mantiene la misma idea de dividir un problema grande en pequeñas partes, sin embargo va más allá. A dichas partes las clasifica en conjuntos denominados "objetos" a los cuales asocia "propiedades" (o cualidades) y "métodos" (acciones o tareas que el objeto puede realizar). Las propiedades y métodos de un objeto pueden heredarse a otro. En pocas palabras, podemos desarrollar un programa que resuelva cierta tarea mediante la creación de objetos y emplear sólo ciertas características que nos sean útiles de este objeto heredándoselas a otro para que resuelva un problema parecido al original, Este tipo de técnicas de programación se consideran formales y de nivel avanzado y permiten ahorrar trabajo reescribiendo tareas que ya estén elaboradas. Aunque no es estrictamente necesario conocer la programación estructurada para poder comprender la programación orientada a objetos, si proporciona las bases teóricas y facilita en gran medida su entendimiento. Ejemplo el lenguaje C++.

LENGUAJES BASADOS EN OBJETOS: A este tipo de lenguajes también se les conoce como lenguajes VISUALES u orientados a eventos. Ejemplos de este tipo de lenguajes: Visual Basic, Visual C, Visual C++, Visual Java (también conocido como Visual J), Visual Fox y otros. Es importante señalar que un lenguaje basado en objetos NO ES LO MISMO QUE UN LENGUAJE ORIENTADO A OBJETOS. El trabajo con lenguajes basados en objetos consiste en dos etapas:

El diseño de la interfaz (una interfaz es la manera en como un usuario se comunica con un programa, por ejemplo Windows y sus aplicaciones son una interfaz gráfica en la que el usuario se comunica con el programa a través de ventanas, botones y cuadros de diálogo) y el diseño del código. El diseño de la interfaz no requiere conocimientos de programación, pues únicamente consiste en "arrastrar" elementos como ventanas, botones y cuadros de diálogo y colocarlos sobre un área de trabajo denominada FORMA.

A dichos elementos se les conoce como OBJETOS (de ahí la confusión de que a lenguajes como Visual Basic, Visual C y otros mencionados arriba se les clasifique erróneamente como orientados a objetos.) Ejemplo de un lenguaje visual o basado en objetos (u orientado a eventos), en este caso Visual Basic:

Sin embargo, el diseño visual solamente es la "forma" o "apariencia" que tendrá el programa, pero TODAVÍA NO REALIZA UNA TAREA UTIL. Aquí viene la segunda parte: El diseño del Código. Esta etapa consiste en escribir en lenguaje Basic o Lenguaje C, las tareas que realizará cada uno de los objetos colocados durante la fase del diseño visual. En esta fase SI REQUIERE CONOCIMIENTOS DE PROGRAMACIÓN,

Una vez realizadas las dos etapas se puede ejecutar el programa, el cual responderá cuando suceda un EVENTO. (Un evento es una acción que activa un objeto para que realice una tarea, por ejemplo: escribir datos como una dirección electrónica en un cuadro de diálogo y enviarlo haciendo "click" sobre el botón enviar, aquí el cuadro de diálogo y el botón de enviar son objetos, el evento asociado al objeto botón es hacer "click" y la acción asociada a este evento es enviar un mensaje).

CARACTERÍSTICAS DEL LENGUAJE C

El lenguaje C se considera de nivel medio, que originalmente fue concebido para la programación de sistemas, la cual comprende principalmente las siguientes tareas:

- Sistemas operativos
- Ensambladores
- Editores de textos
- Bases de datos
- Compiladores
- Controladores de red

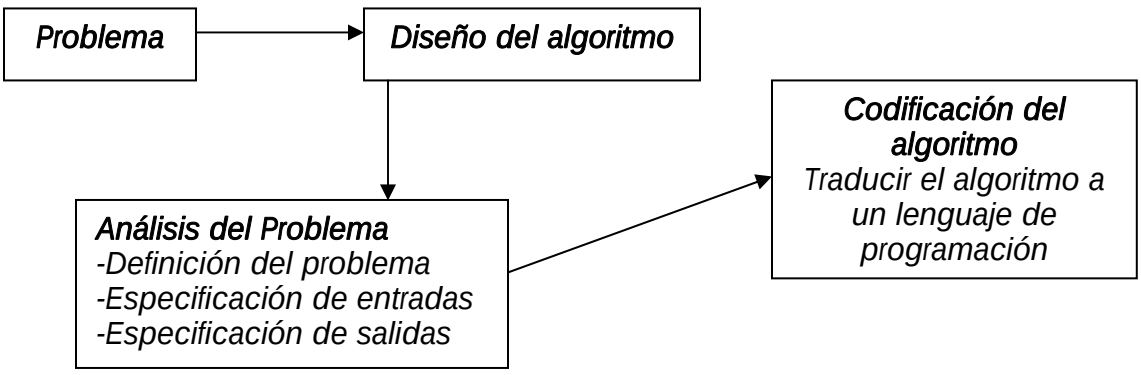
El lenguaje C nace en 1971 y fue diseñado para trabajar inicialmente en sistema operativo UNIX (de uso frecuente en la actualidad). Las primeras versiones de C fueron desarrolladas bajo la filosofía de la programación estructurada, sin embargo, el lenguaje evolucionó hasta el enfoque de objetos. El C es un lenguaje compacto, en el sentido de que tiene pocas instrucciones (28 en total); la mayoría de las tareas comunes como son la lectura y escritura de datos y muchas otras NO FORMAN PARTE DEL LENGUAJE C, por lo que el usuario debe construirlas el mismo. Por esta razón C se considera un lenguaje para programadores.

La sintaxis (forma de escribir) de C es formal y rígida, de tal manera que inculca en el usuario buenos principios y disciplina de programación como son orden y metodología.

El actual lenguaje Java que se emplea actualmente en ambiente de Internet es un producto derivado del lenguaje C. Una de las más importantes características de C consiste en que es un lenguaje TRANSPORTABLE (estándar), es decir funciona en cualquier computadora

PASOS EN LA RESOLUCIÓN DE UN PROBLEMA POR COMPUTADORA

Recuérdese los pasos para resolver un problema por computadora



La última etapa del diagrama anterior (Programar el algoritmo) consiste de los siguientes pasos:

- *Escribir el programa en código fuente*
- *Compilarlo (Traducirlo a código objeto)*
- *Encadenarlo (Incluir las funciones necesarias tomándolas de las librerías adecuadas)*
- *Ejecutarlo (Poner en funcionamiento el programa)*

DEFINICIONES

Código Fuente.- *Es un programa (conjunto de instrucciones) escrito en lenguaje comprensible para el programador (C, BASIC, Pascal, Fortran...). La computadora no lo entiende, por lo que no puede ejecutarlo.*

Código Objeto.- *Es un programa ya traducido a lenguaje de máquina (código binario ó código hexadecimal), por lo que no es legible para el programador, pero si para la computadora, por lo que ya puede ejecutarse y realizar tareas útiles.*

Compilador.- *Es una herramienta que traduce código fuente a código objeto. Existen compiladores de diversos lenguajes tales como: C, Pascal, Fortran, etc.*

Librerías.- *Las tareas que no forman parte del C, tales como las instrucciones de lectura y escritura de datos, funciones matemáticas y varias más, han sido desarrolladas por los fabricantes de C y agrupadas por categorías en archivos llamadas librerías. De ahí pueden ser tomadas para incluirse en un programa de C y ya no es necesario construirlas uno mismo.*

Encadenador (Linker).- *El compilador de C está acompañado de una herramienta llamada encadenador, la cual inserta en el código original las funciones que son tomadas de una librería.*

Editor.- *El compilador de C incluye una herramienta similar a un procesador de textos, en la cual puede escribirse código fuente de un programa para su posterior compilación, encadenamiento y ejecución.*

Archivo Ejecutable.- *Es un programa compilado que ya es autónomo y puede ejecutarse en cualquier computadora sin necesidad de tener instalado el compilador. Tienen la extensión .EXE.*

ESTRUCTURA GENERAL DE UN PROGRAMA EN LENGUAJE C

La estructura general más simple de un programa en lenguaje C se ilustra a continuación:

#include <*nombre del archivo de cabecera o librería*>.....*Directivas de inclusión de librerías*
main().....*Inicio de programa*
{
declaración de variables.....*Cuerpo del programa*
instrucciones (sentencias de entrada, expresiones, sentencias iterativas, sentencias de bifurcación).....*Cuerpo del programa*
} .....*Fin del programa*

PARTES DEL LENGUAJE C

Nota: *Al C estándar se le denota por lenguaje C y al C orientado a objetos por C++. Su estructura es la misma, sin embargo el C++ es más amplio que el C común. El compilador de C++ de Borland (que es el que emplearemos en este curso), permite combinar sin ningún problema a ambos.*

Recuérdese que las instrucciones del C deben de ser escrita en minúsculas.

En el C estándar a los archivos de cabecera se les denomina **librerías** *y a las tareas que realizan se les llama* **funciones**. *En términos generales, una sentencia y una instrucción es lo mismo.*

Ejemplo:

// Conversión de pesos a dólares.....*Comentario*
#include <stdio.h>.....*Directiva de inclusión de librería*
#include <conio.h>>.....*Directiva de inclusión de librería*
main().....*Inicio de programa*
{
float pesos,dolares;.....*Declaración de variables*
clrscr().....*Función para borrar pantalla*
puts("Dar los pesos");.....*Función de escritura*
scanf("%f" , &pesos);.....*Función de lectura de datos*
dolares = pesos/11;.....*Expresión aritmética*
printf("%s%f","El resultado es: ",dolares);.....*Función de escritura de datos*
getch();.....*Lee un caracter de teclado*
}.....*Fin de programa*

TIPOS DE DATOS:

Tipo	*Longitud*	*Rango*
unsigned char	*8 bits*	*0 a 255*
Char	*8 bits*	*-128 a 127*
Enum	*16 bits*	*-32 768 a 32 767*
unsigned int	*16 bits*	*0 a 65 535*
short int	*16 bits*	*-32 768 a 32 767*
Int	*16 bits*	*-32 768 a 32 767*
unsigned long	*32 bits*	*0 a 4,249,967,295*
Long	*32 bits*	*- 2,147,483,648 a 2,147,483,647*
Flota	*32 bits*	$\pm$ *3.4x10^{-38} a* $\pm$ *3.4x10^{+38}*
Double	*64 bits*	$\pm$ *1.7x10^{-308} a* $\pm$ *1.7x10^{+308}*
long double	*80 bits*	$\pm$ *3.4x10^{-4932} a* $\pm$ *1.1x10^{+4932}*

SENTENCIAS DE ENTRADA Y SALIDA

Archivo de cabecera o librería: ***<stdio.h>****. (Estándar input output).*

Sentencias de entrada*: (Lectura)*

- ***Función gets()****. Lee una cadena desde el teclado. Por ejemplo: gets(nombre);, lee una cadena desde el teclado y la almacena en la variable "nombre".*

- ***Función scanf("Cadena de control",lista de variables)****. Lee cualquier tipo de datos desde el teclado siempre y cuando se especifique el tipo de estos en la cadena de control. Por ejemplo: scanf("%d",&edad) lee una cantidad entera desde el teclado y la almacena en la variable "edad".*

Cadena de control	*Tipo de dato leído*
%d	*Entero*
%i	*Entero*
%c	*Caracter*
%s	*Cadena*
%f	*Real, permitiendo la notación exponencial*

Archivo de cabecera ***<conio.h>****. (Console Input Output)*

- ***Función getche()****. Lee un caracter desde el teclado. Produce eco (se visualiza en la pantalla) y no requiere retorno de carro (no se debe presionar la tecla Return). Ejemplo: a = getche(), lee un carácter de teclado y lo almacena en la variable "a".*
- ***Función getch()****. Lee un caracter desde el teclado. No produce eco (no se visualiza en pantalla) y no requiere retorno de carro (No se debe presionar la tecla Return).. Generalmente sirve para introducir una pausa en los programas.*

Sentencias de salida: *(Escritura)*
Archivo de cabecera o librería: **<stdio.h>**. *(Estándar input output).*

- **Función puts()**. *Escribe una cadena en la pantalla. Por ejemplo: puts("Hola amigos") visualiza el mensaje Hola amigos en el monitor.*
- **Función printf("Cadena de control",lista de argumentos)**. *Escribe cualquier tipo de datos en pantalla siempre y cuando se especifique en la cadena de control a que tipo de dato pertenece. Por ejemplo: printf("%s%f","La suma es",suma); muestra el mensaje La suma es seguido del valor de variable "suma".*

Cadena de control	Tipo de dato escrito
%d	Entero
%i	Entero
%c	Caracter
%s	Cadena de caracteres
%f	Real Puede emplearse la notación exponencial
\n	Nueva línea
\r	Retorno de carro
\t	Tabulador horizontal
\´	Comilla simple
\"	Comilla doble
\a	Alerta. Emite un sonido

Para finalizar la parte de entrada/salida de datos, se mencionan algunas funciones útiles, sobre todo en lo que respecta a la apariencia del programa:

Archivo de cabecera **<conio.h>**
- **clrscr()**.....*Borra la pantalla*
- **textcolor(BLUE)**.....*Cambia el color del texto*
- **textbackground**.....*Cambia el color del fondo*
- **gotoxy(x,y)**.....*Situa el cursor en la fila x, columna y*

Archivo de cabecera **<dos.h>**
- **sound(300)**.....*Emite un sonido con la frecuencia indicada*
- **delay(200)**.....*Determina la duración del sonido*
- **nosound()**.....*Apaga el sonido.*

OPERADORES

Los siguientes operadores forman parte de C (en consecuencia del C++). Por lo que **no es necesario tomarlos de ninguna librería:**

Operadores Aritméticos:
(Clasificados por jerarquía de ejecución ó precedencia, de mayor a menor; continúa en la siguiente página).

Operador	*Significado*
++, - -	*Operador incremento y decremento respectivamente*
-	*Menos unario*
*, /, %	*Producto, división convencional y división módulo respectivamente*
+ , -	*Adición y sustracción*
=	*Asignación*

Operadores Relacionales

(En este caso no se clasifican por jerarquías, pues no es la misma entre los diversos lenguajes de programación).

Operador	*Significado*
>	*Mayor que*
>=	*Mayor o igual que*
<	*Menor que*
<=	*Menor o igual que*
= =	*Comparación*
!=	*Diferente de*

Operadores Lógicos. *(Tampoco se clasifican por jerarquías, por la misma razón que el caso anterior).*

Operador	*Significado*
&&	*Y (Conjunción)*
||	*O (Disyunción)*
!	*NO (Negación)*

Operadores matemáticos

Este tipo de operadores no forman parte del lenguaje C. Por esta razón, dichos operadores deberán ser tomados del archivo de cabecera (librería) ***<math.h>.***

Operador	*Significado*
sin(x)	*sen x (seno de x)*
cos(x)	*cos x (coseno de x)*
tan(x)	*tan x (tangente de x)*
asin(x)	*arcsen x (No confundir con el recíproco del seno)*
acos(x)	*arccos x (No confundir con el recíproco del coseno)*
atan(x)	*arctan x (No confundir con el recíprocode la tangente)*
sinh(x)	*senh x (seno hiperbólico de x)*
cosh(x)	*cosh x (coseno hiperbólico de x)*
tanh(x)	*tanh x (tangente hiperbólica de x)*
exp(x)	e^x *(Número de Euler, base de los logaritmos naturales)*
log(x)	*ln x (Logaritmo natural de x)*
log10(x)	$\log_{10}x$ *(Logaritmo en base 10 de x)*
pow(x,y)	x^y *(Potenciación)*
fabs(x)	$\|x\|$ *(Valor absoluto de x)*

SENTENCIAS DE BIFURCACIÓN (DECISIÓN)

Estas instrucciones, si forman parte del lenguaje C, por lo que no requieren de ninguna librería. Este tipo de sentencias permiten la toma de decisiones dentro de un programa y dependen de que su expresión lógico-relacional asociada sea verdadera o falsa.

Decisión simple:

- *if(expresión lógico-relacional) <acción>*

- *if(expresión lógico-relacional)*
  {
  <conjunto de acciones>
  }

Decisión múltiple:

- *if (expresión lógico-relacional 1)*
  {
  <conjunto de acciones 1>
  }
  else if (expresión lógico- relacional 2)
  {
  <conjunto de acciones 2)
  }
  …
  else
  {
  <conjunto de acciones n}
  }

- *switch(valor)*
  {
  case ´valor 1`:
  <conjunto de acciones 1>
  break;
  case ´valor 2`:
  <conjunto de acciones 2>
  break;
  …
  default:
  <conjunto de acciones n>
  }

Ejemplo de decisión simple:

```
//Programa que calcula área de un círculo
//evita la entrada de datos negativos
```

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
main()
{
float radio, pi=3.14159265, area;
etiqueta1:
clrscr();
puts("Dar el valor del radio");
scanf("%f",&radio);
if (radio<=0)
  {
    puts ("El radio debe ser mayor que cero");
   goto etiqueta1;
}
area=pi*pow(radio,2);
printf("%s%f","El área es: ",area);
getch();
} // Fín del programa
```

SENTENCIAS ITERATIVAS

Estas instrucciones, si forman parte del lenguaje C, por lo que no requieren de ninguna librería. Este tipo de instrucciones permiten repetir una acción ó acciones un determinado número de veces. Las sentencias iterativas pueden ser de dos tipos, a) De duración definida.- Se establece explícitamente el número de veces que se realizará una acción ó conjunto de acciones y b) Duración indefinida. No se conoce el número de veces que se repetirá la acción ó conjunto de acciones, más bien se depende del cumplimiento ó no de una condición-lógico-relacional.

Sentencias iterativas de duración definida:

- *for(inicio;fin;incremento)*

```
{
   <conjunto de acciones>;
}
```

Ejemplo: *Sumar los números del 1 al 100:*

```
#include <stdio.h>
#include <conio.h>
main()
  {
   int suma=0,i;
   clrscr();
   for(i=1;i<=100;i++)
      {
       suma = suma + i;
       }
   printf("%s%d","La suma es: ",suma);
  getch();
}// Fín del programa.
```

Sentencias iterativas de duración indefinida:

- ***while(condición lógico-relacional)***
  ```
  {
    <conjunto de acciones>;
  }
  ```

Ejemplo:

```
//Programa que lee números desde el teclado, los multiplica por dos, los muestra en pantalla y termina cuando se
//introduce un número negativo
#include <stdio.h>
#include <conio.h>
main()
  {
    int numero;
    while(numero>=0)
          {
           puts("Dar un número");
           scanf("%d",&numero);
           numero=numero*2;
          printf("%d",numero);
```

```
        }
}// Fin del programa.
```

LECTURA DE CADENAS EN LENGUAJE C

Como ya se ha revisado anteriormente, en el lenguaje C ***NO EXISTEN LAS CADENAS*** *como tales, si es necesario hacer uso de ellas, el programador debe construirlas a partir de un vector de caracteres, por ejemplo: char nombre[15]; declara un vector cuyas componentes son caracteres, pero para fines prácticos, este vector puede considerarse como* ***una cadena cuya extensión es de 15 caracteres máximo****. Se puede asignar valor a esta cadena a través de la* ***función*** *scanf( ) que se encuentra en la* ***librería*** *stdio.h o mediante el* ***método*** *cin que se encuentra en la* ***clase*** *iostream.h.*

No obstante que ambas sentencias permiten la lectura de cualquier tipo de dato a través del teclado, presentan un inconveniente al leer cadenas, ya que ninguna de las dos permite que la cadena lleve intercalados espacios en blanco. Por ejemplo, si se intenta asignar el Valor "Diana Almeida" a la cadena nombre definida anteriormente como char nombre[15], se observa que se tiene una longitud de 13 caracteres por que el espacio entre Diana y Almeida también se cuenta como un caracter. El problema radica que al escribir el texto Diana y después pulsar la barra espaciadora, el lenguaje C interpreta a los espacios en blanco (Ya sea que se introduzcan con la barra espaciadora o con la tecla de tabulación Tab Space) como ***terminadores de cadena****, por lo que en ese momento da por terminada la cadena y el apellido Almeida ya* ***no alcanza a ser almacenado en la variable nombre****, es decir, el valor final de la variable nombre será "Diana" y no "Diana Almeida". Esto implica evidentemente una pérdida de información en los datos que son variables de cadena y que llevan intercalados espacios en blanco. Por este motivo, debe desarrollarse un fragmento de código que permita leer cadenas de texto que lleven intercalados espacios en blanco y que estos* ***no se interpreten como terminadores de cadena****.*

El siguiente programa incluye las instrucciones necesarias para leer una cadena que incluya espacios en blanco y almacenarla en una variable ***sin pérdida de información****. Es un ejemplo simple que consiste en leer de teclado un nombre con apellido y se almacena en una cadena. Este ejemplo puede aplicarse en programas que requieran leer cadenas que contengan espacios en blanco desde el teclado y que requieran ser almacenadas en variables.*

```
// Implementar código para leer cadenas que incluyan espacios en blanco
#include <stdio.h>
#include <iostream.h>
#include <conio.h>
#include <string.h>
main()
```

```
{
 int c,i;
 char  nombre[40];
 clrscr();
  //Leer cadena permitiendo espacios en blanco
puts("Escribe tu nombre y apellido")
 for(i=0 ; (c=getchar()) != EOF && c! = '\n' ; i++)
             nombre[i]=c;
            if (c=='\n')
              {
               nombre[i]=c;
              }
            nombre[i]='\0';
    // fin de lectura de cadena
// Visualizar nombre
clrscr();
printf("%s%s","Tus datos son :",nombre);
}// Fin de main
```

NOTA: La función getchar() se encuentra en la librería string.h, la cual contiene funciones para la manipulación de cadenas. En este caso, la instrucción getchar permite la entrada de caracteres desde el teclado. Aquí se usa para ir recibiendo desde el teclado cada uno de los caracteres del nombre y apellido, incluyendo los espacios en blanco, para uno por uno irlos almacenando en la variable nombre.

ALGUNAS APLICACIONES A LA CRIPTOGRAFÍA

Desde la antigüedad, al ser humano le ha interesado por diversos motivos mantener en secreto ciertos asuntos e información. Desde la época de los grandes imperios, resultaba indispensable mantener en secreto los mensajes que se mandaban entre sí los Jerarcas. No es necesario explicar las consecuencias que conlleva la intercepción del mensaje por parte de manos enemigas.

Por este motivo, el hombre ha inventado desde esos días técnicas que permitan ***cifrar*** *(hacer no entendible) un mensaje, para que en caso de ser interceptado no pueda ser entendido por el enemigo. A la ciencia (y arte) de cifrar mensajes se le denomina* ***Criptografía****. Por otro lado, también el ingenio de la humanidad se ha preocupado por desarrollar técnicas que permitan* ***decodificar*** *o* ***descifrar*** *un mensaje. A esta ciencia se le*

conoce como **Criptoanálisis**. *En conjunto, la Criptografía y el Criptoanálisis constituyen a la ciencia de la* **Criptología**.

Esta ciencia ha cobrado gran importancia, por ejemplo: durante la Segunda Guerra Mundial cuando matemáticos de los países aliados descifraron los códigos JM25 de Japón y ENIGMA de Alemania. Este hecho inclinó significativamente la balanza a favor de los países aliados.

En la actualidad debido al gran impacto de las telecomunicaciones, incluyendo todas las transacciones por Internet y el servicio de correo electrónico se requiere de técnicas criptográficas para garantizar la confidencialidad de las comunicaciones. Existen diversas técnicas de criptografía no obstante mediante sofisticadas técnicas estadísticas pueden ser descifradas. Actualmente las técnicas más seguras son los algoritmos de **Clave Pública** *que se basan en las propiedades de números primos de gran magnitud.*

El siguiente ejemplo en lenguaje C ilustra un método sencillo de criptografía que se conoce como método de transposición. Consiste en sustituir cada letra del alfabeto original por otra del mismo alfabeto pero desplazándose n unidades. Por ejemplo para un desplazamiento de tres se tiene la relación:

ALFABETO ORIGINAL A B C.....Z
ALFABETO PARA CIFRADO D E F.....A

Es decir la A se escribe como D, la B se escribe como E y así sucesivamente. La palabra LA UNIVERSIDAD se cifraría como OD X QLYHUVLGDG, que no sería entendible para quien interceptara el mensaje.

El siguiente programa emplea el procedimiento de transposición para cifrar datos leídos desde el teclado y enviarlos a un archivo en disco. En caso de que el archivo en disco sea revisado por personal no autorizado, no podrá entender su contenido.

Los datos se capturan hasta que se presione la tecla ESC (Escape)

```
// Encriptar Datos
#include <fstream.h>
#include <stdio.h>
#include <iostream.h>
#include <conio.h>
#include <string.h>
main()
{
 int c,i,nreg=0,
 char tecla,  nombre[40];
 char pais[20];
 ofstream datos_salida;
```

```
 ofstream numero_registros;
 datos_salida.open("info.dat");
 numero_registros.open("nreg.dat");
 clrscr();
 while (tecla!=27) // Capturar datos mientras no se presione ESC
  {
   clrscr();
   cout<<"Dato Numero : "<<nreg<<endl;
   cout<<"Nombre  ";
   //Leer cadena permitiendo espacios en blanco
   for(i=0;(c=getchar())!=EOF && c!='\n';i++)
            nombre[i]=c;
           if (c=='\n')
             {
              nombre[i]=c;
             }
           nombre[i]='\0';
   // fin de lectura de cadena

   // Encriptar datos empleando un desplazamiento de 1 unidad.
   for(i=0;i<strlen(nombre);i++)
     {
          nombre[i]++;
     }
      datos_salida<<nombre<<endl;
   nreg++;
   tecla=getch();
  }
 numero_registros<<nreg;
 datos_salida.close();
 numero_registros.close();
}// Fin
```

Enseguida se presenta el programa que accede al archivo de datos cifrados para proceder a su decodificación y presentar los datos en pantalla.

```
#include <fstream.h>
#include <stdio.h>
#include <iostream.h>
#include <conio.h>
#include <string.h>
main()
{
 int c,i,j,nreg,;
 char tecla,nombre[40];
 ifstream datos_entrada;
 ifstream numero_registros;
 datos_entrada.open("info.dat");
 numero_registros.open("nreg.dat");
 clrscr();
 numero_registros>>nreg;
 // Leer datos desde disco:
 for(i=1;i<=nreg;i++)
   {
    datos_entrada>>nombre;
    //Decodificar datos:
    for(j=0;j<strlen(nombre);j++)
          {
          nombre[j]--;
          }
     cout<<"Nombre : "<<nombre<<endl;
    }
 datos_entrada.close();
 numero_registros.close();
 getch();
}// Fin
```

***La Criptología es un área de gran interés para el actuario**, pues se fundamenta en Teoría de Números, Técnicas Estadísticas, Teoría de la Información y Sistemas Computacionales entre otras ciencias. Para profundizar en el tema se recomienda la lectura de las novelas:*

CRIPTONOMICÓN I, El Código Enigma
CRIPTONOMICÓN II, El Código Pontifex
CRIPTONOMICÓN III, El Código Aretusa de Neal Stephenson. Ed Nova. España.

Esta novela de tres tomos relata de forma extraordinaria la historia de la Criptología durante la Segunda Guerra Mundial al tiempo que paralelamente describe el mundo actual de los Hackers. Es un buen material pues se documenta en hechos reales y profundiza en la vida del matemático Alan Turing, uno de los padres de la Teoría de la Computación.

ESTRUCTURAS

En todos los lenguajes de programación se definen ***datos primitivos*** *y con ellos se construyen* ***estructuras de datos****. Los datos primitivos son las unidades mínimas de construcción en un lenguaje y su característica principal es que son* ***indivisibles****.*

Metafóricamente hablando, corresponden a ladrillos con los que puede construirse una pared. El ladrillo es la unidad mínima de construcción, una pared es una colección de estas unidades y con esta estructura construida a partir de unidades primitivas o fundamentales, pueden desarrollarse otras estructuras más complejas, es decir, con un conjunto de paredes se puede obtener una habitación. La habitación es una estructura compuesta de otras estructuras más simples (paredes) compuestas a su vez de unidades básicas (ladrillos). Asimismo, un conjunto de habitaciones puede estructurarse para construir un edificio, que es una estructura compleja compuesta de estructuras más simples. Este concepto se extiende arbitrariamente.

Respecto a un lenguaje de programación los datos primitivos son los tipos de datos: ***char****,* ***int****,* ***float****,* ***double****,* ***etc****, pues son las mínimas unidades de construcción. Éstas pueden organizarse para conformar estructuras, tales como un* ***vector o arreglo*** *(que como se vio anteriormente son una colección de variables referenciadas por un subíndice) y las* ***matrices****. Esas estructuras de datos se construyen a partir de los datos primitivos. Con ellas pueden construirse otras* ***estructuras de datos*** *más complejas tal y como sucede con el caso de los ladrillos y el edificio, es decir, las estructuras de datos pueden combinarse entre sí para generar estructuras más complejas.*

Una estructura de datos es un conjunto de datos organizado y estructurado de una forma en particular y sus principales objetivos son el almacenamiento de datos así como permitir el acceso a ellos a través de estrategias adecuadas. Una estructura de datos permite además el ordenamiento de la información que almacena.

Algunos tipos de estructuras de datos son ***pila (stack)****,* ***cola (queue)****,* ***lista circular y archivo****. Generalmente sobre ellas se definen las operaciones de escritura (introducir datos en ellas), lectura (obtener datos de ellas), búsqueda (buscar específicamente algún dato almacenado en la estructura) y ordenación de sus elementos.*

En lenguaje C se define la estructura ***struct****, como una colección de elementos que tienen características que los describen. En cierta forma los elementos de una estructura* ***struct*** *pueden considerarse como* ***un tipo de variable que tiene varias dimensiones****.*

Por ejemplo una persona tiene nombre, apellido y edad, que son atributos que la describen. Entonces una persona puede considerarse como una variable que puede describirse por los atributos nombre, apellido, edad.

Para utilizar en lenguaje C una estructura del tipo **struct***, primero ha definirse el nombre de la estructura y dentro de ella los atributos o características que describen a los elementos que pertenecen a dicha estructura.*

Una vez hecho lo anterior, ***pueden asignarse valores a los descriptores a través del operador punto (.)****. El operador punto vincula a una variable con su respectivo atributo. El siguiente ejemplo define una estructura empleado con tres atributos nombre, número de trabajador y salario. Se definen dos elementos que pertenecen a esta estructura y se asignan valores a sus respectivos atributos mediante el operador punto.*

```
// Ejemplo de la estructura struct
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
main()
{
 // Aquí se define la estructura empleado y sus tres atributos
struct empleado{
                    char nombre[20]; // Primer atributo, es una cadena de longitud máxima 20 caracteres
                    int numero_trabajador; //Segundo atributo, es un número entero
                    float salario;// Tercer atributo, es un número real
               };
 struct empleado trabajador1,trabajador2;// Declara dos variables del tipo struct, ambas variables tienen tres
// dimensiones o atributos: nombre, número de trabajador y salario
// A continuación se piden sus datos desde el teclado y se asigna valores mediante el operador punto.
// Datos del trabajador 1
cout<<"Nombre del trabajador 1";
cin>>trabajador1.nombre;
cout<<"Número del trabajador 1";
cin>>trabajador1.numero_trabajador;
cout<<"Salario del trabajador 1";
cin>>trabajador1.salario;
// Datos del trabajador 2
cout<<"Nombre del trabajador 2";
cin>>trabajador2.nombre;
```

```
cout<<"Número del trabajador 2";
cin>>trabajador2.numero_trabajador;
cout<<"Salario del trabajador 2";
cin>>trabajador2.salario;
// Visualizar datos del trabajador 1
clrscr();
cout<<"Nombre del trabajador 1"<<trabajador1.nombre<<endl;
cout<<"Número del trabajador 1"<<trabajador1.numero_trabajador<<endl;
cout<<"Salario del trabajador 1  "<<trabajador1.salario;
// Visualizar datos del trabajador 2
cout<<"Nombre del trabajador 2"<<trabajador2.nombre<<endl;
cout<<"Número del trabajador 2"<<trabajador2.numero_trabajador<<endl;
cout<<"Salario del trabajador 2  "<<trabajador2.salario;
getch();
} // Fin de programa
```

Este programa sólo permite manejar 2 trabajadores ¿Que pasaría si se necesitaran los datos de 100 trabajadores? Sería un tanto infructuoso definir 100 variables del tipo ***struct****. Una alternativa adecuada es construir a partir de una estructura, otra mas compleja.*

En este caso, puede definirse un ***vector o arreglo, cuyas componentes son estructuras, es decir cada componente del vector corresponde a un trabajador, pero como cada trabajador posee tres atributos, se tiene que cada componente del vector almacenará tres atributos de un trabajador****. Considere la segunda versión del programa anterior:*

```
// Ejemplo de un vector de estructuras
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
main()
{
 // Aquí se define la estructura empleado y sus tres atributos
struct empleado{
                char nombre[20]; // Primer atributo, es una cadena de longitud máxima 20 caracteres
                int numero_trabajador; //Segundo atributo, es un número entero
                float salario;// Tercer atributo, es un número real
        }
```

```
 struct empleado trabajador[100]; // Declara un vector  struct que almacenara 100 trabajadores, cada uno con
// tres características
int i;
// Leer datos de 100 trabajadores
for(i = 0; i < = 99 , i++)
{
 cout<<"Nombre del trabajador "<<i;
 cin>>trabajador[i].nombre;
 cout<<"Número del trabajador "<<i;
 cin>>trabajador[i].numero_trabajador;
 cout<<"Salario del trabajador "<< i;
 cin>>trabajador[i].salario;
}
// Visualizar datos de los 100 trabajadores
for(i = 0; i < = 99 , i++)
{
 cout<<"Nombre del trabajador "<<i<<trabajador[i].nombre<<endl;
 cout<<"Número del trabajador "<<i<<trabajador[i].numero_trabajador<<endl;
 cout<<"Salario del trabajador "<< i<<trabajador[i].salario<<endl;
}
getch();
} // Fin de programa
```

ARCHIVOS EN DISCO

Hasta el momento, todos los datos que se han manejado en los programas, se almacenado en variables o en estructuras de datos las cuales se alojan en la memoria RAM de la computadora, de tal suerte que cuando se termina la ejecución del programa, los datos se pierden. Una alternativa consiste en almacenar los datos en disco, para preservarlos y posteriormente volver a utilizarlos. Esto es aconsejable especialmente cuando se manejan grandes cantidades de datos.

TERMINOS EMPLEADOS EN EL MANEJO DE ARCHIVOS:

Un archivo es una estructura de datos compuesta por estructuras menores, por ejemplo, suponga que se desea tener un registro de los alumnos de Actuaría, se puede tener una ficha por cada alumno y que contenga sus respectivos datos como nombre, domicilio,

fecha de nacimiento, número de cuenta etc. A los datos de cada ficha (Nombre del alumno, fecha de nacimiento, número de cuenta) se les llama ***campos****. A todo el conjunto de campos (o datos) de una ficha se le denomina* ***registro*** *(se tiene un registro por cada alumno) y al conjunto de registros se le llama* ***archivo****.*

El lenguaje C permite almacenar archivos en disco mediante el uso de ***métodos*** *contenidos en la clase* ***fstream.h****. Un programa en C puede enviar datos hacia el disco para guardarlos, o puede leer datos desde un disco para utilizarlos. En el primer caso se habla de datos de salida y en el segundo caso de datos de entrada o insumo de datos. Los datos deben almacenarse en un disco con un cierto nombre y extensión, en este caso se recomienda usar la extensión .DAT, pues permite leer los datos desde el Block de Notas o el programa WordPad.*

El medio (ruta o vía) a través del cual fluyen los datos hacia o desde el disco se denomina ***corriente****. Una corriente de salida permite enviar datos a través de ella hacia el disco. Una corriente de entrada de datos o de insumo de datos, permite transportar datos a través de ella desde el disco hacia el programa principal. Una corriente es la ruta por donde fluyen los datos y debe asignársele un nombre. Un archivo en disco es el lugar en donde se almacenan los datos y debe también asignarle un nombre.*

El siguiente ejemplo maneja los datos de una tienda de discos, captura el nombre del artista, cuantos discos tiene en la tienda y el título del disco. El proceso de captura termina cuando se presiona la tecla ESC (Escape), al término de este proceso los datos son transportados a través de una corriente denominada en este caso datos_salida y se almacenan en disco en el archivo info.dat.

La variable nreg también se almacena en disco y su función es contar el número de datos capturados para que en otro programa pueda saberse cuantos elementos deben leerse desde el disco y así poder acceder a ellos y visualizarlos. Este programa emplea además un vector de estructuras para almacenar temporalmente los datos y posteriormente enviarlos a disco.

```
// Creación de un archivo de datos
#include <fstream.h> // Clase para manejo de archivos
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
main()
{
 int nreg=0;
 char tecla;
// Define estructura
struct musica{
```

```
                    char nombre[50];
                    int discos;
                    char titulo[20];
                };
 struct musica artista[50];
 textcolor(YELLOW);
 textbackground(BLUE);
 clrscr();
 ofstream datos_salida; // Define una corriente de salida
 ofstream numero_registros;//Define una corriente de salida
 datos_salida.open("info.dat");// Define el nombre de archivo a disco y lo vincula con la corriente de salida
 numero_registros.open("nreg.dat");Define el nombre de archivo a disco y lo vincula con la corriente de
 // salida
// Inicia la captura y se termina al presionar la tecla ESC
 while (tecla!=27)
   {
    gotoxy (32,10);cout<<"Dato No. "<<nreg;
    gotoxy (32,11);cout<<"Nombre del artista ";
    cin>>artista[nreg].nombre;
    gotoxy(32,12);cout<<"No. de discos ";
    cin>>artista[nreg].discos;
    gotoxy(32,13);cout<<"Título del CD";
    cin>>artista[nreg].pais;
// A continuación manda los datos a disco:
datos_salida<<artista[nreg].nombre<<endl<<artista[nreg].discos<<endl<<artista[nreg].pais<<endl<<endl;
    nreg++;
    gotoxy(32,15);
    cout<<"Presione una tecla...";
    tecla=getch();
    sound(900);delay(100);nosound();
    clrscr();
   }
  numero_registros<<nreg;
  // Se cierran las corrientes para evitar pérdida de información
```

```
  datos_salida.close();
  numero_registros.close();
  getch();
}// Fin de main
```

El siguiente programa accede al disco para leer los datos, transportarlos al programa principal y visualizarlos.

```
// Acceso a un archivo de datos
#include <fstream.h> // Clase para manejo de archivos
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
main()
{
 int nreg;
 int i;
// Define estructura
struct musica{
                    char nombre[50];
                    int discos;
                    char titulo[20];
                };
 struct musica artista[50];
 textcolor(YELLOW);
 textbackground(BLUE);
 clrscr();
 ifstream datos_entrada; // Define una corriente de entrada
 ifstream numero_registros;//Define una corriente de entrada
 datos_entrada.open("info.dat");// Define el nombre de archivo a disco y lo vincula con la corriente de entrada
 numero_registros.open("nreg.dat");Define el nombre de archivo a disco y lo vincula con la corriente de
 // entrada
```

```
// Recupera el número de datos almacenados en disco
numero_registros>>nreg;
//Accede al disco y recupera datos en una estructura
 for (i = 0 ; i < n = nreg-1 ; i++)
   {
     datos_entrada>>artista[i].nombre>>artista[i].discos>>artista[i].titulo;
     }
clrscr();
// Visualiza datos de la estructura
 for (i = 0 ; i < n = nreg-1 ; i++)
   {
    gotoxy (32,10);cout<<"Dato No. "<<nreg;
    gotoxy (32,11);cout<<"Nombre del artista "<<artista[i].nombre;
    gotoxy(32,12);cout<<"No. de discos "<<artista[i].discos;
    gotoxy(32,13);cout<<"Titulo  "<<artista[i].pais;
    }
    // Se cierran las corrientes para evitar pérdida de información
  datos_entrada.close();
  numero_registros.close();
  getch();
 }// Fin de main
```

MANEJO DE VECTORES (ARREGLOS O ARRAYS) Y MATRICES.

En términos computacionales, los vectores (arreglos o arrays) son una colección de variables (no necesariamente ordenadas) subíndizadas de elementos de una misma especie: por ejemplo:

***x** = (-3, 4, 7, 0, 8, 9, 10, -3).*

*Nótese que pueden existir elementos repetidos a cada elemento se le denomina **componente**. Cada componente puede referenciarse mediante su respectivo índice. Por ejemplo $x_1 = -3$, $x_2 = 4$, $x_3 = 7$, $x_4 = 0$, $x_5 = 8$, $x_6 = 9$, $x7 = 10$, $x_8 = -3$. Al número de componentes de un vector se le denomina **orden**.*

Ejemplo de declaración de vectores en lenguaje C:

int a[15];	*Declara un vector llamado "a" de 15 componente enteras*
int b[10], c[20];	*Declara un vector "b" de 15 componentes enteros y un vector "c" de 20 componentes reales.*
float v[25];	*Declara un vector llamado "v" de 25 componentes reales*
char nombre[25];	*Declara un vector llamado "nombre" de 25 componentes de tipo caracter . Recuérdese que en lenguaje C, NO existen las cadenas, por lo que para manejar dichos elementos es necesario declarar un vector de caracteres. En este ejemplo, el vector de caracteres llamado "nombre" puede almacenar un mensaje de hasta 25 caracteres.*

Sobre los vectores o arreglos pueden definirse principalmente dos operaciones computacionales:

a) Escritura (Se considera una operación destructiva)

La escritura en un vector consiste en la asignación de valores a sus componentes:
Por ejemplo, considere el siguiente fragmento de código en lenguaje C:

```
int a[5];
a[0]=7, a[1]=0, a[2]=9, a[3]= -1, a[4]= 3;
printf("%d",a[3]);.....(Se visualiza –1)
a[3] = 0;.....(Sobrescribir el valor de a[3], es decir el –1 se pierde)
printf("%d",a[3]);.....(Se visualiza 0, que es el nuevo valor de a[3] )
```

b) Lectura (Se considera una operación no destructiva)

```
int b[5], num;
 b[0]= - 5, b[1]= 8, b[2]= 12, b[3]= 0, b[4]= 6;
num = b[1];.....(se lee el valor de b[1] y se asigna en la variable num. el valor de b[1] no se pierde)
printf("%d",num);.....(se visualiza 8, que es el valor de la variable num)
num = b[3] + b[4] ;...(se leen los valores de b[3] y b[4] para sumarse y asignarse a num. Los valores de b[3] y b[4] no se pierden)
printf("%d",num); .....(se visualiza el 6 que es el valor de la variable num).
```

Nota importante: *En lenguaje C, las componentes se enumeran desde el* ***cero*** *hasta la* ***n-1****, teniendo un total de* ***n*** *elementos.*

Ejemplo de aplicación a operaciones de Álgebra Lineal: Adición Vectorial:

```
//suma  de vectores de 3 componentes
#include <stdio.h>
#include <conio.h>
#include <stdio.h>
main()
```

```
{
 int i,a[3],b[3], c[3];
 clrscr();
 //Lectura de vectores
 for (i=0;i<=2;i++)
    {
     printf("%s%d","Introducir elemento a",i);
     scanf("%d",&a[i]);
     }
 clrscr();
  for (i=0;i<=2;i++)
    {

      printf("%s%d","Introducir elemento b",i);
      scanf("%d",&b[i]);
    }
// Suma y visualizacion de resultados
for(i=0;i<=2;i++)
    {
     c[i] = a[i]  + b[i];
     printf("%s%d%s%d","El elemento c",i, " es 2, c[i] );
    }
 getch();
 }// Fin de main
```

Extendiendo la idea de vector, se llega al concepto de matriz. Una matriz es un arreglo (colección) bidimensional de variables organizadas en ***m*** *filas y* ***n*** *columnas. A la expresión m x n se le denomina* ***orden*** *de la matriz. Si m = n se dice que la matriz es* ***cuadrada****. Dos matrices son conformables si son del mismo orden.*

Ejemplos de declaración de matrices:

int A[4][5];.....Declara una matriz de componentes reales de 4 filas y 5 columnas.

float B[3][4];..Declara una matriz de componentes reales de 3 filas y 4 columnas.

Sobre la matriz se definen las mismas operaciones computacionales que en el vector, salvo que para referenciar a una componente específica se necesitan dos subíndices ***i,j,*** *tal que* a_{ij} *denota la componente ij-ésima de la matriz A. (i = número de fila y j = número de columna).*

Ejemplo de aplicación al Álgebra Lineal:

```
//Adición matricial
#include <conio.h>
#include <stdio.h>
main()
{
 int i,j,k,A[3][3],B[3][3],C[3][3];
 clrscr();
 //Lectura de matriz A
 for (i=0;i<=2;i++)
   {

   for(j=0;j<=2;j++)
    {
     printf(%s%d%d",''Introducir elemento A",i,j);
     scanf("%d",&a[i][j] );
    }//fin de i
   }//fin de j
 clrscr();
 //Lectura de matriz B
 for (i=0;i<=2;i++)
   {
   for(j=0;j<=2;j++)
    {
     printf("%s%d%d",i,j);
     scanf("%d",&b[i][j] );
    }//fin de i
   }//fin de j
 clrscr();
 //Calculo de suma y visualización de resultados
 for(i=0;i<=2;i++)
```

```
   {
    for(j=0;j<=2;j++)
          {
           C[i][j] = A[i][j] +  B[i][j];
            printf ("%s%d%d%d", "La componente C",i,j," es ", C[i][j] );
          } //fin de j
   } //fin de i
 getch();
}//fin de main
```

NÚMEROS ALEATORIOS (PSEUDO-ALEATORIOS).

En la librería **<stdlib.h>** *se encuentran funciones para tareas muy diversas. Entre ellas se encuentran las correspondientes a la generación de números aleatorios, o, mas correctamente, pseudo-aleatorios. La razón de que se les califique como pseudo-aleatorios es que son generados por métodos deterministas que "imitan" razonablemente bien al azar.*

Estos procedimientos generan números que responden de manera satisfactoria a las pruebas estadísticas de aleatoriedad. Las principales funciones para generar números aleatorios en lenguaje C son:

randomize( );	*Activa el generador de números aleatorios tomando como semilla la hora actual del sistema:*
rand( )% n;	*Genera un número aleatorio en ele intervalo cerrado [0,n-1]*
random(n);	*Genera un número aleatorio en ele intervalo cerrado [0,n-1]*

Ejemplos:

Considérese el siguiente fragmento de código en lenguaje C:

```
int numero;
randomize( );
numero = random(5);      Genera un número aleatorio entre 0 y 4
printf ("%d",numero);    Visualiza el valor del número:
```

Se tiene el código equivalente:

```
int numero;
randomize( );
numero = rand%5;         Genera un número aleatorio entre 0 y 4
printf ("%d",numero);    Visualiza el valor del número:
```

Si se quiere generar un número entre m y n se utilizan las expresiones equivalentes

```
numero = random(n – m + 1)  + m;
numero = rand % (n - m +1) + m;
```

***Ejemplo**: Generar un número entre 5 y 15*

```
numero = random(11) + 5
```

Ejemplos de aplicación:

```
//Simulación de un dado
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
main()
{
 int n;
 randomize(); // Activar generador de números aleatorios
 clrscr();
// Dibujar con código Ascii un cuadrado para simular un dado
 gotoxy(35,10);
 printfs(""%c%c%c%c%c",218,196,196,196,191);
gotoxy(35,11);
 printf("%c%s%c",179,"   ",179);
 gotoxy(35,12);
 printf("%c%c%c%c%c",192,196,196,196,217);
 gotoxy(37,11);
 n=rand()%6+1;
 printf("%d",n);
 getch();
 }
```

Ejemplo 2: simulación de lanzamientos de una moneda

```
//Simulación de una moneda
#include <stdio.h>
#include <conio.h>
```

```
#include <stdlib.h>
main()
{
 int i,n;
 float a,s;
 randomize(); // Activar generador de numeros aleatorios
 clrscr();
 for(i=1;i<=100;i++)
   {
   n=random(10);
   if (n<5)
     {
          puts("Aguila");
          a++;
     }
   else
     {
          puts("Sol");
          s++;
     }
   }
 printf("%1.0f%s%1.0f%s",a," Aguilas :",s," Soles");
 getch();
} // Fin de main
```

MANEJO DE GRÁFICAS EN C.

Las funciones gráficas se encuentran en la librería <graphics.h>

Resumen de funciones:

initgraph( ).....*Inicializa el modo gráfico.*
setbkcolor( ).....*Cambia el color de fondo de la pantalla gráfica*
cleardevice( ).....*Borra la pantalla gráfica*
setcolor( ).....*Cambia el color de los píxeles*
getmaxx( ).....*Obtiene el máximo número de píxeles de base del monitor*

getmaxy( ).....*Obtiene el máximo número de píxeles de altura del monitor*
putpixel ( ).....*Dibuja un píxel*
line( ).....*Dibuja una línea*
rectangle( ).....*Dibuja un rectángulo*
bar( ).....*Dibuja un rectángulo sólido*
bar3d( ).....*Dibuja un rectángulo en tres dimensiones*
circle( ).....*Dibuja un círculo*
ellipse().....*Dibuja una elipse*

Ejemplo:

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
main()
{
  // Autodetectar manejador gráfico y tipo de monitor
  int gdriver = DETECT, gmode, errorcode;
  int i,c,x,y;
  // Activar el modo gráfico
  initgraph(&gdriver, &gmode, "");
  // Leer el resultado de la activacion
  errorcode = graphresult();
  if (errorcode != grOk)  // Ocurrencia de error
  {
    printf("%s%d","Error de gráficos:",errorcode);
    puts("Presione una tecla para continuar:");
    getch();
    exit(1);            //Regresar con codigo de error
  }
  randomize(); // Activar el generador de números aleatorios
  setbkcolor(BLUE);// Cambiar color de fondo
  puts("Hola cuates...");
  getch();
  cleardevice(); // Borrar pantalla gráfica
  setbkcolor(RED);
  puts("Hola cuates...");
```

```
getch();
setcolor(YELLOW); // Cambia color de los pixeles
cleardevice();
setbkcolor(BLACK);
// Dibujar 1000 puntos en posiciones y color aleatorios
for (i=0;i<=9999;i++)
   {
        x=random(getmaxx()); // getmaxx() obtiene el máximo valor de x
        y=random(getmaxy());  // getmaxy() obtiene el máximo valor de y
        c=random(10);
        putpixel(x,y,c);
        gotoxy(10,10);
        cout<<i;
   }
getch();
// Dibuja una línea
line(0, 0, getmaxx(), getmaxy());
getch();
cleardevice();
setbkcolor(YELLOW);
setcolor(RED);
line(0, 0, getmaxx(), getmaxy());
getch();
cleardevice();
setbkcolor(BLUE);
setcolor(YELLOW);
// Trazar un rectangulo
rectangle(100,50,400,100); // rectangle (x1,y1,x2,y2)
getch();
// Dibujar un rectangulo solido
bar(100,150,400,200);
// Dibujar un rectangulo tridimensional
getch();
bar3d(100,250,400,300,20,1); // bar3d(x1,y1,x2,y2,profundidad,tapa)
```

```
  getch();
  cleardevice();
  setbkcolor(BLUE);
  setcolor(YELLOW);
  // Dibuja un arco/
  arc(100,100,0,90,50); // arc(x,y,anguloinicio,angulofin,radio)
  // El angulo esta en grados
  getch();
  // Dibujar un circulo //
  circle(200,200,50); // circle(x,y,radio)
  getch();
  // Dibujar un arco de elipse
  ellipse(50,50,0,180,30,50); // ellipse(x,y,anguloinicio,angulofin,radiox,radioy)
  getch();
  ellipse(50,50,0,360,30,50);
  getch();
  // Rellenar una figura cerrada
 floodfill(50,50,YELLOW);
 getch();
 floodfill(200,200,YELLOW);
 getch();
 closegraph();
}// Fin de main
```

TIPO DE TEXTO EN MODO GRÁFICO:

Resumen de instrucciones:

settextjustify( )*.....Cambia la justificación del texto gráfico*
settextstyle( )*.....Cambia el tipo de texto*
outextxy( )*.....Visualiza texto en modo gráfico*

Ejemplo:

```
#include <graphics.h>
#include <stdlib.h>
```

```
#include <stdio.h>
#include <conio.h>
/* the names of the text styles supported */
char *fname[] = { "DEFAULT font",
                  "TRIPLEX font",
                  "SMALL font",
                  "SANS SERIF font",
                  "GOTHIC font"
                };

 main()
{
   int gdriver = DETECT, gmode, errorcode;
   int style, midx, midy;
   int size = 1;
   initgraph(&gdriver, &gmode, "");
   cleardevice();
   // Escribir vertical y horizontalmente un mensaje con la fuente default en tamaño 4
   size = 4;
 settextstyle(DEFAULT_FONT, VERT_DIR, size);
 outext(100,100, “Hola”);
 getch( );cleardevice( );
 settextstyle(DEFAULT_FONT, HORIZ_DIR, size);
 getch( );cleardevice( );

 // Escribir vertical y horizontalmente un mensaje con la fuente triplex  en tamaño 4
 size = 4;
 settextstyle(TRIPLEX_FONT, VERT_DIR, size);
 outext(100,100, “Hola”);
 getch( );cleardevice( );
 settextstyle(TRIPLEX_FONT, HORIZ_DIR, size);
 getch( );cleardevice( );

 // Escribir vertical y horizontalmente un mensaje con la fuente small en tamaño 4
```

```
 size = 4;
 settextstyle(SMALL_FONT, VERT_DIR, size);
 outext(100,100, “Hola”);
 getch( );cleardevice( );
 settextstyle(SMALL_FONT, HORIZ_DIR, size);
 getch( );cleardevice( );

 // Escribir vertical y horizontalmente un mensaje con la fuente sans serif  en tamaño 4
 size = 4;
 settextstyle(SANS_SERIF_FONT, VERT_DIR, size);
 outext(100,100, “Hola”);
 getch( );cleardevice( );
 settextstyle(SANS_SERIF_FONT, HORIZ_DIR, size);
 getch( );cleardevice( );

 // Escribir vertical y horizontalmente un mensaje con la fuente gothic en tamaño 4
 size = 4;
 settextstyle(GOTHIC_FONT, VERT_DIR, size);
 outext(100,100, “Hola”);
 getch( );cleardevice( );
settextstyle(GOTHIC_FONT, HORIZ_DIR, size);
getch( );cleardevice( );
closegraph();
} // Fin de main
```

PROGRAMACIÓN ESTRUCTURADA

La programación estructura, también conocida como ***programación modular*** *se basa en la idea de* ***"Divide y vencerás"****. Esto quiere decir que en lugar de desarrollar programas largos, es conveniente descomponer un problema grande en partes pequeñas. Cada una de las partes pequeñas se resuelve por separado y al final todas las soluciones se integran en una sola.*

Entonces al resolver un problema mediante computadora, el problema debe descomponerse en partes mas pequeñas, debe escribirse para cada subproblema un pequeño programa para resolverlo, y al final se concentran todas las soluciones en una

sola. A estos pequeños programas que resuelven tareas pequeñas o muy específicas se les denomina **funciones**.

En este sentido, un programa en lenguaje C es una colección de funciones, donde cada una de ellas realiza sólo una tarea en particular de un problema.

Formato de un programa en C bajo el enfoque de programación estructurada.

```
main()
{
  funcion1()
  funcion2()
     ...
  funcionN()
}
```

MAIN() se interpreta ahora como un **coordinador de funciones**, *pues se encarga ahora de invocar o llamar en el orden adecuado a cada una de las funciones para que realicen su tarea en particular.*

Antes de descomponer un problema en pequeñas partes debe definirse el **prototipo de una función**. *Un prototipo de función es una descripción de una función y principalmente da la siguiente información:* **Nombre la función, que argumentos o datos requiere y que resultado o resultados arroja o genera**.

Ejemplos de prototipo:

float pendiente (int, int, int, int); *La función se llama pendiente, requiere 4 argumentos enteros y devuelve o arroja un resultado real (float).*

float suma(float, float); *La función se llama suma, requiere dos datos o argumentos reales y regresa o genera un resultado real.*

void factor(int, float, int); *La función se llama factor, requiere en orden, un valor entero, un real y uno entero y no regresa resultado al programa principal.*

void proceso(void); *La función se llama proceso y no requiere argumentos ni devuelve resultados*

void cilindro(float, float, float&, float&); *La función se llama cilindro, en orden requiere dos reales y regresará dos resultados (los que están marcados como float&).*

El prototipo de la función como ya se mencionó sólo es una descripción de la misma y debe situarse al principio del programa, después de las librerías y antes de main. Su código debe escribirse al final del programa. El código de las funciones maneja sus propias variables que se denominan **parámetros formales y estos toman sus valores de los argumentos de la función, por lo que una función opera sobre parámetros formales y no directamente sobre los argumentos de la función**.

Ejemplo:

```
#include<conio.h>
#include<stdio.h>
#include<dos.h>
//Prototipos de función
void pantalla(void);
float suma(float a,float b);
float resta(float a,float b);
float producto(float a,float b);
// El programa presenta un menú principal para realizar operaciones básicas
// cada operación se realiza mediante una función.
  main()
  {
    float x,y,z;
    int opcion;
    inicio:
    pantalla();
    gotoxy(30,14);
    scanf("%d",&opcion);
    switch(opcion){
     case 1:
              puts("Dar valor de x");
              scanf("%f",&x);
              puts("Dar valor de y");
              scanf("%f",&y);
              z=suma(x,y);
              break;
     case 2:
              puts("Dar valor de x");
              scanf("%f",&x);
              puts("Dar valor de y");
              scanf("%f",&y);
              z=resta(x,y);
              break;
```

```
    case 3:
            puts("Dar valor de x");
            scanf("%f",&x);
            puts("Dar valor de y");
            scanf("%f",&y);
            z=producto(x,y);
            break;
    case 4:
            goto fin;
            break;
    default:
            sound(1000);
            delay(100);
            nosound();
            goto inicio;
            break;
   } // Fin de switch
   printf("%s%f","El resultado es ",z);
   getch();
   goto inicio;
   fin:
  }// Fin programa

// Implementar o desarrollar las funciones
void pantalla(void)
{
 textcolor(YELLOW);
 textbackground(BLUE);
 clrscr();
 gotoxy(30,10);puts("(1) Suma ");
 gotoxy(30,11);puts("(2) Resta");
 gotoxy(30,12);puts("(3) Producto");
 gotoxy(30,13);puts("(4) Salir");
}
```

// En las siguientes funciones se opera o trabaja sobre los parámetros formales a y b que toman sus valores respectivamente de los argumentos x,y del programa principal.

```
float suma(float a,float b)
{
 return(a+b);
}
float resta(float a,float b)
{
 return(a-b);
}
float producto(float a,float b)
{
 return(a*b);
} // Fin de programa
```

VECTORES COMO ARGUMENTOS

En el ejemplo anterior se manejaron funciones con argumentos numéricos. No obstante una función puede tener a un vector como argumento.
El siguiente programa lee desde teclado un vector de componentes reales y éste es pasado ***completo*** *como argumento de una función para calcular su norma*

```
//Cálculo de la norma de un vector
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <math.h>
void norma( int[]  );//Prototipo de función que tiene como argumento un vector
main()
{
 int i,a[3];
 clrscr();
 //Lectura de vectores
 for (i=0;i<3;i++)
    {
```

```
    cout<<"Introducir elemento["<<i<<"] del vector A"<<endl;
    cin>>a[i];
    }
 clrscr();
 norma(a);//llamado a funcion usando al vectores a como argumentos
 getch();
}// Fin de programa
// Implementación o construcción de la función norma
void norma(int x[]) // La función opera sobre el parámetro formal que en este caso es un vector
{
 int i;
 float s=0;
 for(i=0;i<3;i++)
   {
    s = s + pow(x[i],2);
    s = pow(s,0.5);
    }
 cout<<"La norma del vector es:"<<s;
 }
```

VARIABLES LOCALES Y GLOBALES

Todas las variables definidas dentro de una función se conocen como ***variables locales****, es decir sólo son conocidas dentro del ámbito en dónde fueron declaradas. Cualquier otra función del programa NO TIENE CONOCIMIENTO NI ACCESO A ELLA. Estas variables, literalmente hablando SE CREAN al ser invocada su respectiva función y se DESTRUYEN cuando la función ha realizado su tarea y devuelve el control al programa principal. Por otro lado, una variable que ha sido declarada antes de la sentencia* ***main()*** *se le denomina* ***variable global*** *y es conocida y puede ser utilizada por CUALQUIER PARTE Y FUNCIÓN DEL PROGRAMA. Empero NO PUEDE SER MODIFICADA (en primera instancia) desde alguna de las funciones que hacen referencia a ella o la usan. Es importante que un programa puede tener variables globales y locales que tengan el mismo nombre, sin embargo, al referirse dentro de una función a dicha variable,* ***SE CONSIDERA O DA PREFERENCIA A LA VARIABLE LOCAL****.*

Ejemplo:

```
//Variables locales y globales//Ejemplo con variables globales y locales
#include <stdio.h>
```

```
#include <conio.h>
int x=5; // Variable Global
void funcion1(void);
void funcion2(void);
void funcion3(void);
main()
{
 int x=6; // Variable local en función main
 clrscr();
 printf("%d\n",x);// variable local
 funcion1();
 printf("%d\n",x);//variable local
 funcion2();
 printf("%d\n",x);//variable local
 funcion3();
 printf("%d\n",x);//variable local
 getch();
}//Fin de programa
void funcion1(void)
{
  printf("%d\n",x); // Variable global
}
void funcion2(void)
{
 int x=7; // Variable local
 printf("%d\n",x);
}
void funcion3(void)
{
x=9;
  x=x*2;   // variable global
  printf("%d\n",x);
}
```

La salida que arrojará el programa es: ***6, 5, 6, 7, 6.***

OPERADOR DE RESOLUCIÓN GLOBAL

Ya se mencionó que en primera instancia no puede modificarse el valor de una variable global desde dentro de una función. A través del operador de ***resolución global*** *:: , se puede modificar el valor de una variable global desde dentro de una función :*

```
//Ejemplo 2 con variables globales y locales
#include <stdio.h>
#include <conio.h>
int x=5; // Variable Global
void funcion1(void);
void funcion2(void);
void funcion3(void);
main()
{
 int x=6; // Variable local en función main
 clrscr();
 printf("%d\n",x);// variable local
 printf("%d\n",::x);// variable global
 funcion1();
 printf("%d\n",x);//variable local
 funcion2();
 printf("%d\n",x);//variable local
 printf("%d\n",::x);//variable global
 funcion3();
 printf("%d\n",::x);//variable global
 getch();
}//Fin de programa
void funcion1(void)
{
  printf("%d\n",x); // Variable global
}
void funcion2(void)
{
 int x=7; // Variable local
 printf("%d\n",x);
```

```
}
void funcion3(void)
{
  int x=8;
  printf("%d\n",x);
  ::x=::x*2;   // variable global
  printf("%d\n",x);
  printf("%d\n",::x);
}
```

El programa arrojará la siguiente secuencia numérica: ***6,5,5,6,7,6,5,8,8,10,10.***

CONSTRUCCIÓN DE LIBRERÍAS PROPIAS.

Como ya se ha mencionado anteriormente, el C es un poderoso lenguaje de programación de un tamaño compacto. El C estándar consta de únicamente 28 palabras reservadas. Debido a sus características, el C se considera un lenguaje formal en el sentido de que por su rigidez inculca buenos principios de programación en sus usuarios.

Es por eso que al C se le conoce como un lenguaje para los programadores, pues no incluye instrucciones para la mayoría de las tareas, lo que implica que el usuario mismo es quien tiene que desarrollar sus propias funciones. Tareas tan fundamentales como las instrucciones para entrada y salida de datos ***no forman parte del c****, si no que son "tomadas" de la librería stdio.h y de la clase iostream.h.*

Una librería es una colección de tareas comunes agrupadas bajo un nombre, por ejemplo la librería para manipular operadores matemáticos es math.h. Otras librerías importantes son conio.h, string.h, dos.h, graphics.h, dos.h, stdlib.h etc. Estas librerías fueron desarrolladas por los fabricantes de C y se incluyen junto con el compilador.

En ciertos casos puede ser de utilidad desarrollar librerías propias para realizar ciertas tareas específicas. Es importante mencionar que las librerías stdio.h, math.h y todas las demás se encuentran en la carpeta INCLUDE. La ventaja de crear librerías propias radica en que estas herramientas quedan listas para usarse varias veces en diversos programas y no hay que volver a escribir su código, es decir son reutilizables. Como se ha observado los archivos asociados a las librerías tienen la ***extensión (*.h)*** *Un archivo de este tipo es un programa en C que consiste en una colección de funciones para realizar ciertas tareas en común. A continuación se explican los pasos para crear una librería para cálculos financieros*

Primero debe escribirse desde el editor de C el archivo y guardarlo con la extensión (*.h) en la carpeta INCLUDE

*En este **Ejemplo**, el archivo se llama **matefin.h** se guarda en la carpeta INCLUDE*

```
//LIBRERIA MATEFINANCIERAS
// ESTA LIBRERÍA CONSISTE EN FUNCIONES PARA CÁLCULOS FINANCIEROS COMO EL CÁLCULO
// DE INTERÉS SIMPLE, INTERÉS COMPUESTO, INTERÉS EXACTOM, MONTO Y VALOR PRESENTE
// PODRÁN SER USADAS DESDE CUALQUIER PROGRAMA EN C COMO SE HACE CON CUALQUIERA
// DE LAS OTRAS LIBRERÍAS.
#include<math.h>
#define Int_sim(C,r,t) (C*(r/100)*(t/12));
#define Int_exac(C,r,d) (C*(r/100)*(d/365));
#define Int_comp(C,i,n) (C*pow((1+(i/100),n)));
#define Monto(C,r,t) (C*(1+((r/100)*(t/12))));
#define Val_pres(S,r,t) (S/(1+(r/100)*(t/12)));
// Fin de archivo
```

Luego de manera normal se realiza un programa en C en el cual se incluye la librería matefin.h PREVIAMENTE DESARROLLADA

El programa presenta un menú de opciones y los cálculos financieros son realizados mediante las funciones tomadas de la librería matefin.h

```
//MENU DE MATEFINANCIERAS
#include<iostream.h>
#include<math.h>
#include<conio.h>
#include<stdlib.h>
#include<dos.h>
#include "<matefin.h>
void intcomp (void);
void intex (void);
void intsim (void);
void  monto (void);
void valpre (void);
main()
    {
    clrscr();
    textcolor(14);
```

```
textbackground(25);
int opcion;char tecla;
caldin:
clrscr();
gotoxy(10,10);cout<<"    [1] INTERES COMPUESTO   "<<endl;
gotoxy(11,11);cout<<"    [2] INTERES EXACTO   "<<endl;
gotoxy(12,12);cout<<"    [3] INTERES SIMPLE   "<<endl;
gotoxy(13,13);cout<<"    [4] MONTO   "<<endl;
gotoxy(14,14);cout<<"    [5] VALOR PRESENTE  "<<endl;
gotoxy(15,15);cout<<"    [6] SALIR   "<<endl;
cin>>opcion;
switch (opcion)
 {
 case 1:
 {
 intcomp();
 }
 break;
  case 2:
 {
 intex();
 }
 break;
 case 3:
 {
 intsim();
 }
 break;
 case 4:
 {
 monto();
 }
 break;
 case 5:
```

```
    {
    valpre();
    }
    break;
    case 6:
    {
    goto fin;
    }
    break;
    }
    fin:
    getch();
    clrscr();
gotoxy(10,10);cout<<"    ¨DESEAS REALIZAR OTRO CALCULO DEL MENU PRINCIPAL [S/N]?   ";
tecla=getch();
if ((tecla=='S') || (tecla=='s'))
{
goto caldin;
}
    }
void intcomp (void)
 {
   float C,i,n,Int_comp;char tecla;
    //Realizar otra operacion
    matfi:
    clrscr();
    gotoxy(7,7);cout<<" Introduzca el capital ";
    gotoxy(8,8);cin>>C;
    gotoxy(9,9);cout<<" Introduzca la taza de interés ";
    gotoxy(10,10);cin>>i;
    gotoxy(11,11);cout<<" Introduzca el tiempo ";
    gotoxy(12,12);cin>>n;
    clrscr();
    gotoxy(10,10);cout<< "El interés compuesto es "<<Int_comp (C,i,n);
```

```
      gotoxy(18,18);cout<<" ¨Desea realizar otra operación [s/n]? ";
      tecla=getch();
      if ((tecla=='S') || (tecla=='s'))
      {
      goto matfi;
      }
      getch();
     }//fin
void intex (void)
{
 float C,r,d,Int_exac;char tecla;
      //Realizar otra operacion
      matfi:
      clrscr();
      gotoxy(8,8);cout<<" Introduzca el capital ";
      gotoxy(9,9);cin>>C;
      gotoxy(10,10);cout<<" Introduzca la taza de interés ";
      gotoxy(11,11);cin>>r;
      gotoxy(12,12);cout<<" Introduzca los días ";
      gotoxy(13,13);cin>>d;
      clrscr();
      gotoxy(10,10);cout<<" El interés exacto es " <<Int_exac (C,r,d);
      gotoxy(18,18);cout<<" ¨Desea realizar otra operación [S/N]? ";
      tecla=getch();
      if ((tecla=='S') || (tecla=='s'))
      {
      goto matfi;
      }
      getch();
     }//fin
void intsim (void)
  {
    float C,r,t,Int_sim;char tecla;
      //Realizar otra operación
```

```
    matfi:
    clrscr();
    gotoxy(8,8);cout<<" Introduzca el capital ";
    gotoxy(9,9);cin>>C;
    gotoxy(10,10);cout<<" Introduzca la taza de interés ";
    gotoxy(11,11);cin>>r;
    gotoxy(12,12);cout<<" Introduzca el tiempo ";
    gotoxy(13,13);cin>>t;
    clrscr();
    gotoxy(10,10);cout<< "El interés simple es "<<Int_sim (C,r,t);
    gotoxy(18,18);cout<<" ¨Desea realizar otra operación [s/n]? ";
    tecla=getch();
    if ((tecla=='S') || (tecla=='s'))
    {
    goto matfi;
    }
    getch();
    }//fin
void monto (void)
  {
    float C,r,t,Monto; char tecla;
    //Realizar otra operacion
    matfi:
    clrscr();
    gotoxy(8,8);cout<<" Introduzca el capital ";
    gotoxy(9,9);cin>>C;
    gotoxy(10,10);cout<<" Introduzca la taza de interés ";
    gotoxy(11,11);cin>>r;
    gotoxy(12,12);cout<<" Introduzca el tiempo ";
    gotoxy(13,13);cin>>t;
    clrscr();
    gotoxy(10,10);cout<<" El monto es "<<Monto (C,r,t);
    gotoxy(18,18);cout<<" ¨Desea realizar otra operación [s/n]? ";
    tecla=getch();
```

```
    if ((tecla=='S') || (tecla=='s'))
    {
    goto matfi;
    }
    getch();
    }//fin
void valpre (void)
 {
 float S,r,t,Val_pres;char tecla;
      //realizar otro calculo
      matfi:
      clrscr();
      gotoxy(8,8);cout<<" Introducir el monto ";
      gotoxy(9,9);cin>>S;
      gotoxy(10,10);cout<<" Introducir la taza de interés ";
      gotoxy(11,11);cin>>r;
      gotoxy(12,12);cout<<" Introducir el tiempo ";
      gotoxy(13,13);cin>>t;
      clrscr();
      gotoxy(10,10);cout<<" El valor presente es "<<Val_pres (S,r,t);
      gotoxy(18,18);cout<<" ¨Desea realizar otra operación [s/n]?   ";
      tecla=getch();
      if ((tecla=='S') || (tecla=='s'))
      {
      goto matfi;
      }
      getch();
      }//fin
```

PROGRAMACIÓN ORIENTADA A OBJETOS

La Programación estructurada tuvo su auge durante la década de los ochenta. A finales de los ochenta y principios de los noventa surge el concepto de la programación orientada a objetos (POO) o POO (Programming Objects Oriented). La POO, se considera

la evolución natural de la programación estructurada, pues conserva el enfoque de dividir y vencer pero lo extiende de manera abstracta al concepto de CLASE. La POO maneja elementos similares a los conjuntos.

Una ***clase*** *es un conjunto abstracto de elementos, por ejemplo la clase árbol es un conjunto en el cual se incluyen a los pinos, los robles, los arces etc. Árbol es el* ***nombre genérico o abstracto*** *del conjunto. Entonces árbol es una CLASE.*

Los ***miembros de datos*** *son las características, atributos o descriptores que deben cumplir los elementos que pertenezcan a una clase dada. Por ejemplo para la clase árbol, el tamaño y edad son características que describen a los elementos de la clase árbol, por ello se les considera MIEMBROS DE DATOS.*

Las ***funciones miembro o métodos,*** *son las tareas o acciones que los elementos de una clase pueden realizar. Por ejemplo para la clase árbol, sus elementos pueden retoñar y pueden crecer. Entonces estas tareas que pueden realizar los elementos que pertenecen a la clase árbol se denominan FUNCIONES MIEMBRO O MÉTODOS.*

Cuando se especifican elementos que pertenecen a una cierta clase se denominan a éstos ***objetos o instancias****. Por ejemplo El pino, el Arce y el Oyamel, son tres OBJETOS O INSTANCIAS que pertenecen a la clase árbol, pues responden a las características definidas sobre dicha clase.*

Ejemplo 2:

Se define una clase y sus atributos y sus métodos

```
clase BEBE {
        Apellido Paterno
        Edad
        Peso
        Llorar ( )
        Dormir ( )
        Comer ( )
    }
```

A continuación se especifican objetos o instancias que pertenecen a la clase BEBE:

BEBE Pedro, Jimena (Pedro y Jimena son, en sentido literal, dos objetos o instancias que pertenecen a la clase BEBE previamente definida y cumplen con los atributos (miembros de datos) y métodos (tareas) definidas en dicha clase.

La clase se denomina BEBE, los miembros de datos (atributos o características que deben cumplirse) son Apellido Paterno, Edad y Peso. Las tareas que pueden realizar (funciones miembro o métodos) son Dormir( ) y Comer( ).

Entonces a través del ***operador punto*** *se puede acceder a los miembros de datos y métodos de cada objeto:*

```
Pedro.ApellidoPaterno = Hernández
Pedro.Peso = 5
Pedro.Edad = 3
Pedro.Llorar( )

Jimena.ApellidoPaterno = Suárez
Jimena.Peso = 4
Jimena.Edad = 3
Jimena.Dormir( )
```

Ejemplo 1 de Programación Orientada a Objetos:

```
// Creacion de clases y objetos
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>
#include <stdio.h>
class empleado {
    public:   // Los datos públicos son accesibles desde el programa principal a través del operador punto.
          char nombre[20]; //Miembros de datos
          int id_empleado;
          float salario;
          void mostrar_empleado(void)
           {
            cout<<"Nombre  : "<<nombre<<endl;
            cout<<"Salario : "<<salario<<endl;
            cout<<"Numero  : "<<id_empleado<<endl;
           };// Fin de funcion miembro
              };//Fin de Clase
main()
{
 empleado trabajador, jefe ; //Crear 2 objetos o instancias  "trabajador" y "jefe"
 clrscr();
// Acceder a los miembros de datos del objeto trabajador desde el programa principal
cout<<"Nombre del trabajador: "<<endl;
 cin>>trabajador.nombre;
 cout<<"Salario del trabajador : "<<endl;
 cin>>trabajador.salario;
```

```
 cout<<"Numero del trabajador: "<<endl;
 cin>>trabajador.id_empleado;
 clrscr();
// Acceder a los miembros de datos del objeto jefe desde el programa principaL
cout<<"Nombre del jefe"<<endl;
 cin>>jefe.nombre;
 cout<<"Salario del jefe : "<<endl;
 cin>>jefe.salario;
 cout<<"Numero del jefe "<<endl;
 cin>>jefe.id_empleado;
//Acceder a los método del objeto trabajador y objeto jefe
trabajador.mostrar_empleado( );
jefe.mostrar_empleado( );
 getch();
}// Fin de Programa
```

Nótese que en el ejemplo anterior, el código de la función miembro o método se desarrollo dentro de la definición de la clase. Sin embargo se considera más adecuado desarrollar su código afuera de la definición de la clase y dentro de ella sólo dejar el prototipo de la función, tal y como se ilustra en la segunda versión del mismo programa:

```
// Creacion de clases y objetos versión 2
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>
#include <stdio.h>
class empleado {
    public:   // Los datos públicos son accesibles desde el programa principal a través del operador punto.
          char nombre[20]; //Miembros de datos
          int id_empleado;
          float salario;
          void mostrar_empleado(void) // Aquí sólo va el prototipo de la función  miembro o método.
              };//Fin de Clase
// A continuación se implementa o desarrolla el código de la función miembro o método. Nótese que se hace uso
// del operador de resolución global :: para indicar que la función mostrar_empleado opera SOBRE LA CLASE
// empleado
```

```
void empleado::mostrar_empleado(void)
            {
             cout<<"Nombre  : "<<nombre<<endl;
             cout<<"Salario : "<<salario<<endl;
             cout<<"Numero  : "<<id_empleado<<endl;
            };// Fin de funcion miembro
main()
{
 empleado trabajador, jefe ; //Crear 2 objetos o instancias  "trabajador" y "jefe"
 clrscr();
// Acceder a los miembros de datos del objeto trabajador desde el programa principal
cout<<"Nombre del trabajador: "<<endl;
 cin>>trabajador.nombre;
 cout<<"Salario del trabajador : "<<endl;
 cin>>trabajador.salario;
 cout<<"Numero del trabajador: "<<endl;
 cin>>trabajador.id_empleado;
 clrscr();
// Acceder a los miembros de datos del objeto jefe desde el programa principal
cout<<"Nombre del jefe"<<endl;
 cin>>jefe.nombre;
 cout<<"Salario del jefe : "<<endl;
 cin>>jefe.salario;
 cout<<"Número del jefe "<<endl;
 cin>>jefe.id_empleado;
//Acceder a los método del objeto trabajador y objeto jefe
trabajador.mostrar_empleado( );
jefe.mostrar_empleado( );
 getch();
}// Fin de Programa
```

Los dos programas anteriores realizan exactamente la misma tarea, aunque tengan forma diferente.

INTRODUCCION A LOS DATOS PRIVADOS

En determinadas circunstancias, no es recomendable que un usuario tenga acceso directo a los miembros de datos de alguna clase desde el programa principal, por ejemplo, algunos datos introducidos por un usuario a un programa deben ser validados o verificados mediante algún mecanismo antes de ser asignados a los miembros de datos de una clase. A este mecanismo se le denomina **función interfaz***, y su objetivo es verificar que los datos introducidos por un usuario a un programa cumplan ciertos requisitos o se encuentren en un cierto rango permitido antes de ser asignados a los miembros de datos de una clase. Cuando los miembros de datos de una clase son declarados como* **private***, EL ACCESO A ELLOS DESDE EL PROGRAMA PRINCIPAL SÓLO ES POSIBLE MEDIANTE EL USO DE UNA FUNCIÓN INTERFAZ. DICHO ACCESO YA NO ES EMPLEANDO EL OPERADOR PUNTO,* **SINO MEDIANTE EL OPERADOR ASIGNACIÓN (=).**

A continuación se muestran dos programas que realizan las mismas tareas. Sin embargo utiliza miembros de datos públicos, por lo que el acceso a ellos desde el programa principal no está restringido y puede llevarse a cabo a través del operador punto. El segundo programa considera miembros de datos privados, lo que restringe el acceso a los datos mediante el empleo de una función interfaz, la cual se encarga de verificar que los valores introducidos por el usuario se encuentren en un cierto rango antes de asignarlos a los miembros de datos con el operador asignación:

```
// Ejemplo de miembros de datos públicos
#include <iostream.h>
#include <conio.h>
// Establecer una clase con datos públicos:
class rectangulo {
                    public:
                    //Miembros de Datos publicos:
                    float base,altura;
                    //Prototipos de Funciones Miembro: (o sea  métodos)
                    void calcular_perimetro(void);
                    void calcular_area(void);
                   };
//Implementacion de funciones miembro (Métodos de la clase)
void rectangulo::calcular_perimetro(void)
        {
         cout<<"El perimetro es :"<< 2*(base+altura);
        };//Fin de Metodo
```

```
void rectangulo::calcular_area(void)
        {
         cout<<"El área es :"<<base*altura;
        };//fin de método

//Programa principal
main()
{
 //Crear un objeto o instancia de la clase "rectángulo"
 rectangulo figura;
 clrscr();
 //Asignar valores a los miembros de datos
 cout<<"Introducir Base ";
 cin>>figura.base;
 cout<<"Introducir Altura ";
 cin>>figura.altura;
 //Invocar a los métodos de la clase
 figura.calcular_perimetro();
 cout<<endl;
 figura.calcular_area();
 getch();
}// Fin de programa
```

//Ejemplo de miembros de datos privados

```
#include <iostream.h>
#include <conio.h>
// Establecer una clase con datos privados:
class rectangulo {
                public:
                //Prototipo de Función Interfaz
                int asignar_valores(float,float);
                //Prototipo de Funciones miembro o métodos
                void calcular_perimetro(void);
```

```
                    void calcular_area(void);
                    //Miembros de Datos privados:
                    float base,altura;
                   };
//Implementación de función INTERFAZ para acceso a datos privados
int rectangulo::asignar_valores(float b,float h);
          {
           //Permitir el acceso a datos privados solo si la base y la altura
           //son positivos:
          if( (b>0)&&(h>0) )
            {
             base=b;
             altura=h;
             return(0);
            }
          else
            return(-1);
          }//Fin de Funcion Interfaz

//Implementar funciones miembro o métodos de clase
void rectangulo::calcular_perimetro(void);
          {
           cout<<"El perímetro es :"<<2*(base+altura);
          };//fin de método
void rectangulo::calcular_area(void)
          {
           cout<<"El área es :"<<base*altura;
          };//fin de método

//Programa principal
main()
{
 //Crear un objeto o instancia de la clase "rectángulo"
 rectangulo figura;
```

```
 float bas,alt;
 clrscr();
 //Leer valores
 cout<<"Introducir Base ";
 cin>>bas;
 cout<<"Introducir Altura ";
 cin>>alt;
 //Invocar a función inter para intentar acceso a datos privados
 if( (figura.asignar_valores(bas,alt))==0)
   {
    cout<<"Se permitió la asignación de datos"<<endl;
    figura.calcular_perimetro();
    cout<<endl;
    figura.calcular_area();
   }
 else
   cout<<"Datos Incorrectos. No es posible calcular";
 getch();
}// Fin de Programa
```

FUNCIONES CONSTRUCTORAS Y DESTRUCTORAS

Una ***función constructora*** *es una función interfaz y* ***su objetivo es asignar valores iniciales a los miembros de datos de una clase dada****. Una vez hecho lo anterior, debe emplearse una* ***función destructora*** *para liberar la memoria ocupada con los valores iniciales generados por la constructora. Una función destructora debe tener el mismo nombre que la constructora, con la diferencia de que su nombre va antecedido por el* ***operador tilde*** *~.*

```
// Ejemplo de clase con miembros de datos privados y asignación de valores iniciales mediante una
// función constructora
#include <iostream.h>
#include <conio.h>
// Establecer una clase con datos privados:
class rectangulo {
                        public:
```

```
                    //Prototipo de Función CONSTRUCTORA
                    rectangulo(float,float);
                    //Prototipo de Función DESTRUCTORA
                    ~rectangulo(void);
                    //Prototipo de Funciones miembro o métodos
                    void calcular_perimetro(void);
                    void calcular_area(void);
                    //Miembros de Datos privados:
               private:
                    float base,altura;
                   };
//Implementación de función CONSTRUCTORA para inicializar valores
rectangulo::rectangulo(float b,float h)
          {
           //Permitir el acceso a datos privados solo si la base y la altura
           //son positivos:
          if( (b>0)&&(h>0) )
            {
             base=b;
             altura=h;
            }
          else
            {
             //Si se dan valores negativos, se asignan "arbitrariamente"
             //Los siguientes valores:
             base=1;
             altura=2;
            }
          };//Fin de Función Constructora
//Implementar Función Destructora
void rectangulo::~rectangulo(void);
          {
          cout<<"Destruyendo Objeto...";
       getch();
```

```
        }
//Implementar funciones miembro o métodos de clase
void rectangulo::calcular_perimetro(void);
        {
         cout<<"El perímetro es :"<<2*(base+altura);
        };//fin de metodo
void rectangulo::calcular_area(void)
        {
         cout<<"El área es :"<<base*altura;
        };//fin de método

//Programa principal

main()
{
 //Crear un objeto o instancia de la clase "rectángulo" y dar valores iniciales
 //mediante función constructora
 rectangulo figura(3,4);
 float bas,alt;
 clrscr();
 //Realizar cálculos con los valores iniciales
 figura.calcular_perimetro();
 cout<<endl;
 figura.calcular_area();
 cout<<endl;
 getch();
}
```

HERENCIA

Como ha podido observarse, un lenguaje orientado a objetos presenta una ESTRUCTURA ABSTRACTA o formal y maneja elementos que no existen en los lenguajes que no son orientados a objetos. Estos elementos que caracterizan a la POO son:

**** Encapsulamiento**. En un lenguaje para POO pueden encapsularse o agruparse los miembros de datos y métodos en una entidad denominada clase y mantenerse como un organismo autónomo, bien definido, pero con capacidad de interactuar con diversos programas y otras clases.*

**** Herencia**. Una vez definidas clases, pueden construirse otras similares a partir de las primeras pero tomando de ellas sólo las características que se necesiten. A este proceso se le llama herencia.*

**** Polimorfismo**. El polimorfismo significa que una cierta clase puede adaptarse a las necesidades de diferentes programas y trabajar con ellos dependiendo de las características de los mismos, es decir, una clase puede comportarse de diferentes formas, según las exigencias de cada programa.*

```
// Ejemplo de Herencia
#include <iostream.h>
#include <conio.h>
// Ejemplo de herencia:
class rectangulo {
                    public:
                    //Prototipo de Función CONSTRUCTORA
                    rectangulo(float,float);
                    //Prototipo de Funciones miembro o métodos
                    void calcular_perimetro(void);
                    void calcular_area(void);
                    float base,altura;
                   };
//Implementación de función CONSTRUCTORA para inicializar valores
rectangulo::rectangulo(float b,float h);
          {
           //Permitir el acceso a datos privados solo si la base y la altura
           //son positivos:
          if( (b>0)&&(h>0) )
           {
            base=b;
            altura=h;
           }
          Else
```

```
            {
             //Si se dan valores negativos, se asignan "arbitrariamente"
             //Los siguientes valores:
             base=1;
             altura=2;
            }
           };//Fin de Función Constructora
//Implementar funciones miembro o métodos de clase
void rectangulo::calcular_perimetro(void)
           {
            cout<<"El perímetro es :"<<2*(base+altura);
           };//fin de método
void rectangulo::calcular_area(void)
           {
            cout<<"El área es :"<<base*altura;
           };//fin de método
//Crear una nueva clase que hereda características de la clase rectángulo
class prisma:public rectangulo{
    public:
    //Función constructora
    prisma(float,float,float);
    //Prototipo de función miembro
    void calcular_volumen(void);
    //Datos privados
    private:
    float profundidad;
    };
//Implementar función constructora
prisma::prisma(float base, float altura, float profundidad):rectangulo(base,altura);
{
 prisma::profundidad=profundidad;
}
//Implementar función miembro de clase derivada
void prisma::calcular_volumen(void)
```

```
{
 cout<<"El volumen es..."<<base*altura*profundidad;
}
 //Programa principal
main()
{
 //Crear un objeto o instancia de la clase "rectángulo" y dar valores iniciales
 //mediante función constructora
 rectangulo figura1(3,4);
 prisma figura2(3,4,2);
 float bas,alt;
 clrscr();
 //Realizar cálculos con los valores iniciales
 figura1.calcular_perimetro();
 cout<<endl;
 figura1.calcular_area();
 cout<<endl;
 figura2.calcular_volumen();
 getch();
} // Fín de programa
```