FACULTAD DE INGENIERIA

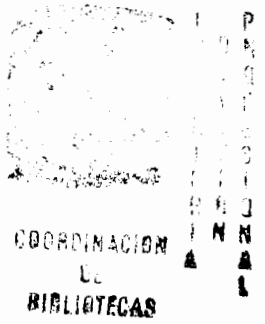
UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

APUNTE 195 FACULTAD DE INGENIERIA UNAM.

RAYMUNDO H. RANGEL GUTIERREZ

APUNTES DE PROGRAMACION ESTRUCTURADA

DIVISION DE INGENIERIA MECANICA Y ELECTRICA
DEPARTAMENTO DE COMPUTACION



APUNTES DE PROGRAMACION ESTRUCTURADA

PROLOGO

Debido al avance progresivo de la computación y en particular en el área de software, el desarrollo de programas es de gran importancia, ya que existen varias metodologías para la programación de computadoras digitales.

Por esto es necesario estudiar los principios metodológicos, a fin de unificar los diferentes estilos individuales que permitan la comunicación, modificación, transportabilidad y mantenimiento de los sistemas de programación. Una forma de aplicar estos principios metodológicos es la programación estructurada, la cual es una disciplina eminentemente práctica y por ello se ha escogido el lenguaje *Pascal*, ya que en particular facilita la construcción de programas estructurados.

El contenido temático de esta obra consta de tres partes. La primera hace énfasis en la solución de problemas independientemente del lenguaje de programación en que se va a implantar la solución. En la segunda parte se estudia el lenguaje de programación Pascal. Estas dos partes están íntimamente relacionadas; sin embargo, se han tratado de forma separada para enfatizar las dos etapas del desarrollo de un programa: la resolución del problema a programar y su codificación, siendo la primera un proceso creativo y la segunda un proceso mecánico.

En la tercera parte se estudia el análisis y diseño estructurado, y se proporcionan algunos elementos de éstos para ubicar a la programación dentro de un contexto de un proceso mayor: El desarrollo de sistemas de información.

Aun cuando en la obra se estudian todos los aspectos de Pascal, su contenido se mantiene a un nivel introductorio.

Al final de la obra aparece una bibliografía clasificada por los temas tratados en ella. Se recomienda consultarla para ampliar y profundizar el contenido del curso de *Programación Estructurada*.

El mejoramiento de esta obra podrá lograrse con las observaciones y sugerencias de profesores y alumnos, por lo que se agradecerán las aportaciones que se hagan llegar al Departamento de Computación de la División de Ingeniería Mecánica y Eléctrica, a fin de mejorar futuras ediciones.

Se agradece a la Lic. María Cuairán Ruidíaz por su colaboración en la adaptación pedagógica de esta obra, así como a Edmundo Rosales Valderrábano, Bernardo Mendieta Hernández y R. Angel Castro Flores por su intervención en la revisión técnica de la misma.

RAYMUNDO HUGO RANGEL GUTIERREZ.

C O N T E N I D O

PARTE I INTRODUCCION A LA PROGRAMACION ESTRUCTURADA

CAPITULO 1 METODO Y HERRAMIENTAS DE LA PROGRAMACION ESTRUCTURADA

GENERALIDADES

En sus inicios, la programación de los sistemas de cómputo estuvo determinada por el trabajo individual de los programadores; es decir no había un grupo que se coordinara en la tarea del desarrollo de los sistemas, ni una forma sistemática de estudiar la complejidad de los mismos.

Como consecuencia de ello, el usuario, al hacer uso del sistema tenía una serie de dificultades que, en gran medida, se podrían atribuir a que la programación del sistema no cumplía con los requerimientos.

Por otra parte, cuando un programador codificaba individualmente sin atenerse a un método específico, ocasionaba que otros programadores tuvieran dificultad para implementar cambios sustanciales al sistema, tales como reprogramar módulos incorrectos, integrar otras funciones al sistema y cualquier otra especificación para que éste fuera actualizado.

Esto, naturalmente incrementaba los costos de mantenimiento, además de que un sistema de estas características ocasionaba grandes retrasos. Asimismo, los programadores no aplicaban una metodología en el desarrollo de la programación de los sistemas que permitiera obtener un producto que cumpliera con ciertas normas de calidad. Esto se debía a que el programador aplicaba la mayor parte de su esfuerzo a la programación y prueba del sistema y desatendía por completo las fases de análisis y diseño, además de que olvidaba que un programa también es un documento para comunicar la solución de un problema.

Resultaba así, que mientras el equipo de cómputo progresaba grandemente y su costo disminuía, la eficiencia en la programación se incrementaba con la complejidad del sistema y, como consecuencia, sus costos se incrementaban. Esta situación llegó a agravarse tanto que dio origen a lo que se llamó crisis del software o de la programación; pero esto motivó la búsqueda de una solución a los problemas, con objeto de:

- abatir costos de mantenimiento
- obtener alta confiabilidad
- entregar a tiempo
- lograr una administración efectiva
- otros

Con este fin nació la programación estructurada o sistemática, siendo la primera metodología que surgió para solucionar los problemas de programación en los sistemas de cómputo. Se considera a *Edsger W. Dijkstra* como el padre de la programación estructurada, ya que fue quien propuso un método que los resolviera. *Dijkstra* hace mención de conceptos que son clásicos de esta metodología, como son:

- el diseño de arriba hacia abajo (top-down)
- la programación sin go to's
- otros

Sin embargo, la programación estructurada no resolvía todos los problemas que se presentaban, por ejemplo no daba una visión general del sistema en las primeras fases del desarrollo del mismo.

Diversos autores propusieron métodos para solucionar este problema. Uno de los métodos que ha sido aceptado en la actualidad es el diseño estructurado; pero éste aún no resuelve el problema de la especificación de los requerimientos que el usuario pide del sistema, lo cual ha llevado a la búsqueda de conceptos fundamentales, incluso matemáticos para resolver este problema.

Un método que en la actualidad también es ampliamente aceptado para la especificación de los requerimientos del usuario es el análisis estructurado.

El análisis, el diseño y la programación estructurada en realidad no se excluyen sino que se complementan y, en la actualidad, es común usarlos en las fases de análisis, diseño y programación de sistemas respectivamente.

En esta obra se utiliza el lenguaje Pascal, el cual presenta facilidades para el estudio de la programación estructurada.

1.1 FASES EN EL DESARROLLO DE PROGRAMAS

Todo programa, como producto final de un proceso de manufacturas, emerge como consecuencia de una serie de pasos o fases que deben seguirse rigurosamente. Se identifican las cuatro fases siguientes:

1.- DISEÑO DEL PROGRAMA

En esta fase se lleva a cabo el análisis del problema y el desarrollo de la solución. Estas dos actividades forman una estrecha simbiosis. Durante esta fase se hace un análisis cuidadoso del problema propuesto con el objeto de lograr una representación mental (abstracción) de los elementos esenciales del mismo. Una vez logrado este primer nivel de abstracción se puede continuar subdividiendo estos elementos, siempre y cuando sea posible, en partes más simples de la misma forma y continuar de este modo hasta llegar a un nivel de gran detalle.

Es evidente que se hace necesario contar con una notación conveniente para ir concretando la solución, ya que mentalmente no se podría retener la gran cantidad de detalles que se originan conforme se refina la solución. A todo este proceso se le conoce como refinamiento a pasos y es central a la programación estructurada. Esto no es más que un proceso de ensa

yo y error. Obsérvese que el problema comienza con un enunciado que dice lo "que" se intenta resolver y se termina describiendo "cómo" se intenta resolver lo.

2.- CODIFICACION.

De la fase anterior surge un programa abstracto, el cual, en esta segunda fase, se traduce o se implementa en un lenguaje de programación determinado. El proceso de codificar un programa en un lenguaje de programación es totalmente directo.

3.- PRUEBA

Después de la implementación de la solución presentada, es necesario que ésta sea probada, para así asegurar que resolverá correctamente el problema original.

La depuración del programa y la eliminación de errores en la codificación o en la carga a la computadora (perforación de tarjetas, edición vía terminal, etc.), se realiza mediante la "compilación" del programa.

Para la prueba del programa se deberán diseñar conjuntos de datos, en los que se puedan contemplar todos los casos o variaciones posibles en la presentación del problema, principalmente los casos extremos.

4.- MANTENIMIENTO

Durante la fase de mantenimiento del programa se realizan las modificaciones que son necesarias cuando cambian algunas características del problema que no implican un cambio en el diseño original del programa. La "vida útil" es el tiempo durante el cual el programa funciona sin que sea modificado sustancialmente; es decir, el tiempo en que es útil para la solución del problema original.

La fase de mantenimiento se mantendrá presente durante

toda la vida útil del programa, y es en la que se determina cuándo el programa ha dejado de ser útil.

1.2 CALIDAD DE PROGRAMAS

La forma tradicional de diseño condujo a programas difíciles para darles mantenimiento (escasa calidad), debido, por una parte a su casi nula estructuración y escasa claridad, y por otra a artificios "inteligentes" de programadores "abusados", artificios que por lo general sólo servían para realzar el ego del programador y no contribuían en nada a hacer del programa un vehículo para comunicar ideas.

Debido a esta situación se hizo patente la necesidad de realizar el diseño de programas de una manera sistemática, logrando con ello programas de mayor calidad. Lograr programas de mayor calidad mediante un diseño sistemático es el objetivo fundamental de la programación estructurada.

No hay dos personas que estén de acuerdo sobre lo que es la calidad de programas; sin embargo, algunos criterios que nos dan una idea de lo que se entiende por calidad de programas son los siguientes:

1.- QUE EL PROGRAMA FUNCIONE

Esto significa que el programa debe cumplir con los requerimientos especificados. Es importante que las especificaciones sean revisadas continuamente a través de todo el proceso de diseño e implementación del programa. Un programa que no hace o hace parcialmente lo que se supone debe hacer, refleja que el diseñador tiene una escasa comprensión de las especificaciones requeridas del programa.

2.- QUE EL PROGRAMA ESTE LIBRE DE ERRORES

Generalmente los errores de un programa son debidos a que el programador es poco cuidadoso; tiene un entendimiento poco claro de las especificaciones o de los

rasgos del lenguaje en que codifica o falla al anticipar una situación particular.

3.- QUE EL PROGRAMA ESTE BIEN DOCUMENTADO

Tradicionalmente, la eficiencia de un programa se medía en términos de la cantidad de memoria que ocupaba y de la velocidad de ejecución del mismo. Esto se justificaba cuando las máquinas eran lentas y de poca capacidad de memoria y por consiguiente el programador dedicaba tiempo a reducir el requerimiento que demandaba su programa de estos recursos.

En la actualidad, estos criterios de eficiencia ya no son válidos, excepto en casos muy particulares, debido a que los costos del hardware han descendido y los costos humanos se han incrementado. De modo que si un programador intenta hacer eficiente un programa bajo el criterio de espacio/velocidad a costa de producir un programa difícil de darle mantenimiento, es claro que habrá logrado un producto de baja calidad.

1.3 EJEMPLO DE UN DISEÑO

Un ejemplo sobre cómo resolver un problema dará una idea del proceso de diseño. Aun cuando no se cuenten con los elementos necesarios para hacer un diseño sistemático, este ejemplo fundamentalmente está enfocado a ir derivando en la forma más natural posible estos elementos, al tiempo que se logra un diseño basado en los mismos.

Supóngase que se plantea el siguiente problema:

De un conjunto de 'n' números desordenados, obtener el que tenga el menor valor, con la restricción de que únicamente, se pueden 'ver' dos valores numéricos a la vez.

La siguiente figura será de gran ayuda para visualizar el problema, es decir, hacer el análisis.

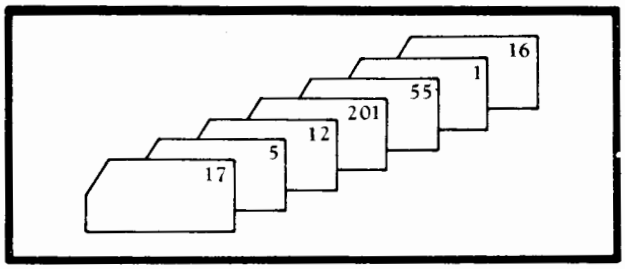


Figura 1.1 Conjunto de tarjetas

En el diagrama anterior se ve que la tarjeta con el valor 1 (sexta tarjeta) tiene el valor mínimo, pero recuérdese que se tiene una restricción: no se pueden ver todas las tarjetas a la vez, sólo se pueden ver dos. Para hacer énfasis en esto, supóngase que se tienen apiladas las tarjetas como lo muestra la siguiente figura.

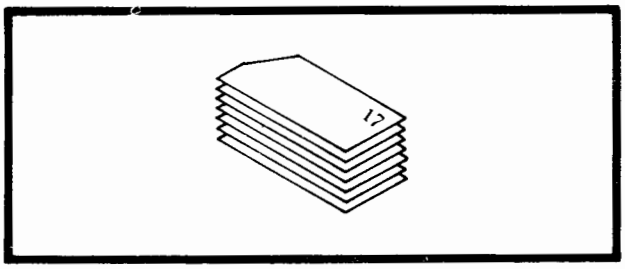


Figura 1.2 Pila de tarjetas

Se anota el valor de la primera tarjeta en otra que se llamará tarjeta temporal, como lo muestra la siguiente figura:

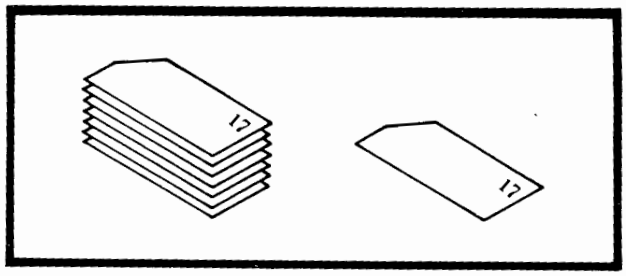


Figura 1.3

Ahora, el valor de la tarjeta temporal se puede comparar con cada uno de los valores de las tarjetas de la pila, y cada vez que en la pila se encuentre un valor menor al valor de la tarjeta temporal, se anota ese valor (menor) en la tarjeta temporal. Por supuesto que el valor anterior de la tarjeta temporal se borra. Es claro que al final se tendrá anotado el valor menor (1) en la tarjeta temporal.

A continuación se da una descripción detallada de la solución anterior:

1.- Se anota el valor de la primera tarjeta de la pila en la tarjeta temporal y luego se mueve de la pila esa tarjeta.

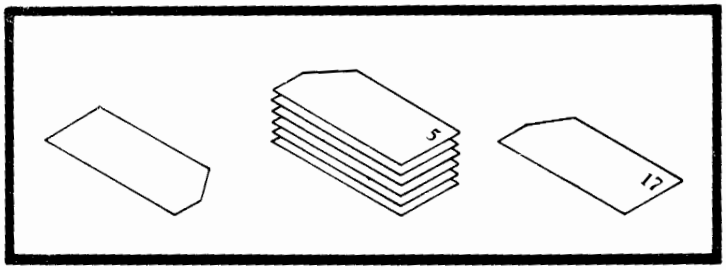


Figura 1.4

2.- Se compara la primera tarjeta de la pila con la tarjeta temporal. Si el valor de la siguiente tarjeta de la pila es menor que el de la tarjeta temporal, se anota su valor en la tarjeta temporal, y se remueve de la pila (éste es el caso del ejemplo).

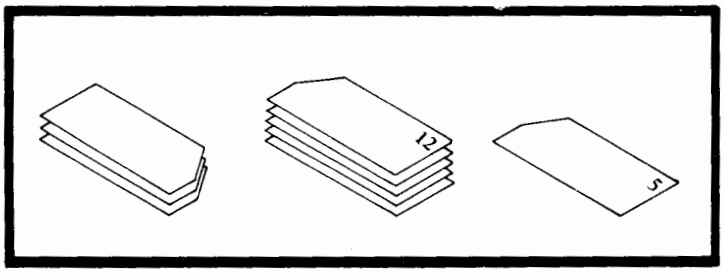


Figura 1.5

En caso contrario, sólo se remueve la tarjeta de la pila para hacer la siguiente comparación.

3.- Se repite el proceso hasta agotar las tarjetas de la pila.

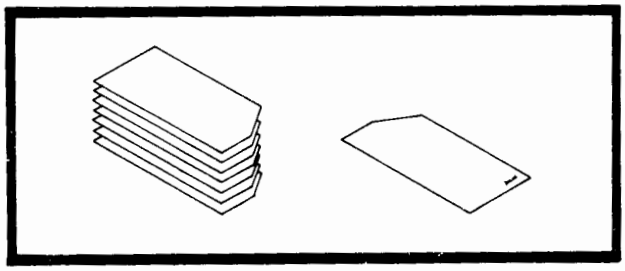


Figura 1.6

Obsérvese que en esta última descripción de la solución se han realizado esencialmente tres acciones básicas, las cuales se enuncian a continuación:

1.- Una acción de asignación, es decir, se anota un valor en la tarjeta temporal.

2.- Una acción de decisión con base en la comparación de un par de valores.

3.- Una acción repetitiva de la decisión.

Es evidente que diferentes personas describirían las acciones de este proceso de diferentes maneras; por lo cual, es necesario establecer un convenio sobre la descripción de estas acciones, de tal manera que sean legibles (entendibles) para todos.

Se asume el siguiente formato para la acción de decisión:

```
si (condición) luego
   acciones 1 a tomar
o bien
   acciones 2 a tomar
fin
```

Esta acción de decisión se realiza como sigue: si la condición es verdadera se realizan las acciones 1, de lo contrario se realizan las acciones 2.

La acción de decisión del ejemplo quedaría como sigue:

```
si ((valor (tarjeta-siguiente)) < (valor (tarjeta-temporal)))
    luego anotar valor (tarjeta-siguiente) en tarjeta-temporal

o bien

    pasar a la siguiente tarjeta de la pila

fin
```

Obsérvese que se ha usado una notación taquigráfica; esto es válido siempre y cuando sea evidente lo que se quiere dar a entender.

La notación para la asignación se denota con el siguiente símbolo <---, por consiguiente, la acción de anotar queda como sigue:

```
tarjeta-temporal <--- valor (tarjeta-siguiente)
```

Es decir el valor de la tarjeta siguiente se asigna a la tarjeta temporal.

Se tiene que repetir la acción de decisión hasta agotar las tarjetas de la pila. Se asume ahora el siguiente formato para repetir una y otra vez una acción determinada:

```
repetir
     acciones a tomar
hasta (condición)
```

Esta acción iterativa se lleva a cabo como sigue: las acciones a tomar se realizan una y otra vez hasta que la condición se cumple, es decir, hasta que ésta sea verdadera.

Esta acción repetitiva en el contexto del ejemplo queda como sigue:

```
repetir
  si (valor (tarjeta-siguiente) < valor (tarjeta-tem
      poral)) luego tarjeta-temporal <--- valor (tar
      jeta-siguiente)
  o bien
     pasar a la siguiente tarjeta
  fin
hasta (agotar tarjetas en pila)
```

A las acciones de decisión y repetición, debido a su importancia, se les da un nombre especial: *Estructuras de Control*. Obsérvese que ahora se tiene una descripción mucho más compacta y precisa del proceso (solución). Las acciones a tomar dentro de una estructura de control se conocen como alcance de las estructuras de control.

Obsérvese que las acciones dentro de la estructura de control tienen sangría con respecto a la estructura, lo cual tiene por objeto delimitar visualmente este alcance; esto que parece trivial, es de gran importancia ya que hace más legible la lectura del proceso.

Volviendo al ejemplo anterior, se observa que en la acción de la decisión se está asumiendo qué valor (tarjeta-temporal) tiene ya un valor asignado, esto, por consiguiente, no se ha hecho explícito:

1.- Tarjeta-temporal <--- valor (tarjeta en tope de pila)

2.- Repetir

si (valor (tarjeta-siguiente)< valor(tarjeta-temporal)) luego tarjeta-temporal <--- valor(tarjeta-siguiente)

o bien

pasar a la siguiente tarjeta

fin

hasta (agotar tarjetas en pila)

A continuación se da un listado de las estructuras básicas de control. Aquí predicado es sinónimo de condición, siendo predicado o condición una afirmación que puede ser falsa o verdadera.

1.- SECUENCIA

```
        .
        .
        .
    instrucción 1
    instrucción 2
        .
        .
        .
```

2.- DECISION

A) Si (predicado 1) luego
   seudocódigo 1

   o si (predicado 2) luego
   seudocódigo 2
        .
        .
        .
   o bien
   seudocódigo n
   fin

Esta estructura evalúa el predicado 1, si es verdadero realiza el seudocódigo 1, si no se evalúa el predicado 2 y así sucesivamente. Si ninguno de los predicados es verdadero se realiza el seudocódigo n. Esta es una estructura de decisión múltiple.

B) si (predicado) luego
   seudocódigo 1

   o bien seudocódigo 2
   fin

Esta estructura es un caso particular de la anterior. Es una estructura de decisión binaria, es decir sólo toma dos decisiones. Si el predicado es verdadero se realiza el seudocódigo 1 en caso contrario, el seudocódigo 2.

C) si (predicado) luego
   seudocódigo
   fin

Esta estructura es también un caso particular de A); toma una sola decisión. Si el predicado es verdadero se realiza el seudocódigo.

3. ITERATIVAS

A) En tanto (predicado) repetir
    seudocódigo
   fin

Esta estructura iterativa realiza el seudocódigo que se encuentra dentro de su alcance, en tanto el predicado sea verdadero.

B) Repetir
    seudocódigo
   Hasta (predicado)

Esta estructura realiza el seudocódigo que está dentro de su alcance, mientras el predicado sea falso y deja de hacerlo hasta que el predicado sea verdadero.

C) Desde (i <-- expa 1 hasta expa 2) repetir
    seudocódigo
   fin

Esta estructura realiza el seudocódigo que se encuentra dentro de su alcance, en tanto el valor de la variable de control i no sobrepase el valor de la expresión aritmética 2 (exp 2). La variable i toma como valor inicial el de la expresión aritmética 1 (exp 1).

PROBLEMAS PROPUESTOS

1. Hacer el seudocódigo de un programa que lea dos números y los sume.

2. Hacer el seudocódigo de un programa que lea dos vectores A y B de cinco elementos cada uno. Realizar la suma elemento a elemento y guardar el resultado en un vector C.

3. Escribir el seudocódigo de un programa que lea 25 datos y obtenga la media.

4. Hacer el seudocódigo de un programa que lea dos números reales A y B y un número entero N tal que $1 <= N <= 5$, y dependiendo del valor de N, que efectúe las siguientes operaciones:

Si	N=1	A+B
"	N=2	A-B
"	N=3	(A)(B)
"	N=4	A/B
"	N=5	(A)(A) + (B)(B)

5. Escribir el seudocódigo de un programa que realice la siguiente sumatoria:

$$X = \sum_{i=n-1}^{1} (n-i) - 1$$

donde n=25

PARTE II PASCAL ESTRUCTURADO

CAPITULO 2 ELEMENTOS BASICOS DE PASCAL

GENERALIDADES

Pascal es un lenguaje de alto nivel y su propósito es general.

Fue diseñado por *Niklaus Wirth* en 1970 siguiendo las ideas de *A.R. Hoare*. Su objetivo era desarrollar un lenguaje que facilitara la programación, lo cual convirtió a Pascal, en el lenguaje más adecuado para la enseñanza de los conceptos de la programación estructurada.

Como consecuencia de esto, en la actualidad diversas universidades utilizan Pascal como el lenguaje principal en la enseñanza de la programación.

Aunque Pascal fue ideado para ser utilizado en un ambiente académico, actualmente ha rebasado este ámbito y ha empezado a usarse ampliamente en aplicaciones comerciales.

Entre las características que hacen de Pascal un lenguaje ideal para la enseñanza de la programación estructurada se tienen las siguientes:

a) Los datos están fuertemente tipificados; cada dato debe declararse con un tipo específico, es decir entero, real, lógico, caracter, etcétera.

b) Presenta un conjunto de estructuras de control bastante variado.

c) Es autodocumentable, se asemeja bastante al seudocódigo (en relación con el usuario, al menos en inglés).

d) Es fácil de aprender.

e) Es modular, es decir, un programa se puede descomponer en partes, donde cada parte tiene funciones muy específicas.

f) Es un traductor que toma un programa en Pascal y lo traduce a un lenguaje ejecutable por la máquina.

2.1 ESTRUCTURA GENERAL DE UN PROGRAMA EN PASCAL

La estructura de un programa en Pascal es la siguiente:

```
PROGRAM  nombre(INPUT,OUTPUT);
- declaraciones   -
BEGIN
- proposiciones   -
END.
```

Donde PROGRAM, INPUT, OUTPUT, BEGIN y END son palabras reservadas, es decir, son palabras propias del lenguaje Pascal y no tienen otro uso más que el indicado.

PROGRAM indica el inicio del programa.

El nombre del programa lo elige el programador y consta de uno a ocho caracteres, siendo el primero una letra y los restantes, si los hay, letras y/o dígitos.

INPUT es el nombre del archivo de entrada (lectura de datos).

OUTPUT es el nombre del archivo de salida (impresión de datos).

Las declaraciones son instrucciones no ejecutables y simplemente le indican al compilador el tipo de las variables que van a ser usadas en el programa. Siempre estarán en mayúsculas.

Las proposiciones son instrucciones ejecutables que realizan el proceso que pretende el programa.

La parte de las declaraciones consta de cinco secciones que deben aparecer en la secuencia que se muestra en seguida:

```
LABEL
        -declaraciones de etiquetas      -
CONST
        -declaraciones de constantes     -
TYPE
        -declaraciones de escalares      -
VAR
        -declaraciones de variables      -
-declaraciones de subprogramas -
```

Siendo LABEL,CONST,TYPE y VAR palabras reservadas.

Pueden omitirse aquellas secciones que no sean necesarias en un programa, pero conservando siempre el orden indicado.

Las proposiciones pueden ser: lectura e impresión de datos, estructuras de control, llamadas a subprogramas y asignaciones.

Como un primer ejemplo de programa en Pascal, se tiene el siguiente:

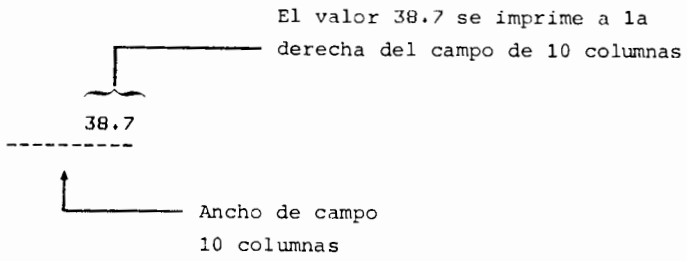
```
PROGRAM ejemplo;
BEGIN
WRITE(38.7);
END.
```

Donde (;) significa fin de declaraciones e instrucciones.

Obsérvese que se han omitido las declaraciones de los archivos de entrada y salida, debido a que el compilador asume, por omisión, que INPUT y OUTPUT son los archivos de entrada y salida respectivamente.

Este programa no tiene declaraciones y sólo tiene una instrucción ejecutable (tercera línea). El propósito de

este programa es imprimir el valor constante 38.7. Por omisión el ancho de campo en que se imprime el valor de 38.7 es de 10 columnas (en el sistema B7800), cargando el valor a la derecha de este campo, por ejemplo:

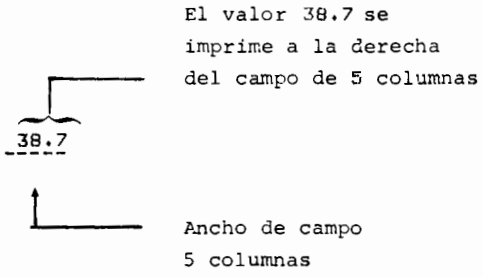


Se puede modificar el ancho de campo que se da por omisión, indicando explícitamente el ancho de campo que se quiere para la impresión del dato.

Por ejemplo:

```
PROGRAM ejemplo;
BEGIN
WRITE(38.7:5);
END.
```

Ahora se indica que el ancho de campo en que se imprime el valor constante 38.7 es de cinco columnas. Los dos puntos (:) indican que el valor entero que le sigue es el ancho de campo en que se imprime el valor numérico que le antecede. La impresión quedaría:



El ejemplo siguiente incluye la declaración de tres variables de tipo entero, así como la lectura de tres datos (valores) enteros que se asignan a estas variables en la lectura.

En este ejemplo, se han declarado tres variables de tipo entero en la sección VAR (segunda y tercera línea).

Obsérvese que la declaración de las variables (tercera línea) consta de dos partes delimitadas por el caracter (:). A la izquierda de éste aparece la lista de los nombres dados a las variables separadas por comas y, a la derecha, una palabra reservada que indica el tipo de las variables. En este caso es entero (INTEGER). Los nombres de las variables siguen la misma regla que el nombre del programa.

```
PROGRAM ejemplo;
VAR
       a,alfa,x13:INTEGER;
READ(a,x13,alfa);
WRITE(a,alfa,x13);
END.
```

Los valores que se asignan durante la lectura (cuarta línea), a las variables a,alfa y x13, deben estar separados por lo menos por un blanco. Ejemplo:

37 428 1

Los valores 37,428 y 1 se asignarán a las variables a,x13 y alfa respectivamente. Obsérvese que en la lista de variables que aparecen en la lectura (READ) éstas deben estar separadas por comas.

La segunda y tercera línea pueden sustituirse por la siguiente:

```
VAR a,alfa,x13:INTEGER;
```

Debe haber un espacio blanco por lo menos

Sin embargo se prefiere la primera forma, ya que se localiza con más claridad la sección VAR.

Cuando en una salida (impresión) aparezca la siguiente notación (----) y no exista ningún caracter, significa vacíos o espacios en blanco.

Además de la instrucción de lectura READ se tiene la instrucción de lectura READLN, que contrariamente al READ, una vez que se ejecuta pasa el control de lectura al inicio de la siguiente línea de datos. El ejemplo siguiente aclara esto. Supóngase que se tiene la siguiente secuencia de lecturas:

```
        *
        *
        *
READ(a,b);
READ(c);
READLN(d,e);
READLN(f);
        *
        *
        *
```

y los datos siguientes:

```
37.08 47.03 48.3 378 423.01
283.0473
```

Los valores 37.08, 47.03 y 48.3 se asignarán a las variables a,b y c respectivamente. Obsérvese que para esta forma de lectura (READ), el control de lectura no pasa al inicio de la siguiente línea sino que permanece en el mismo renglón.

Los valores 378, 423.01 y 283.0473 se asignarán a las variables d,e y f respectivamente, con esta forma de lectura (READLN), una vez que se leen los valores, el control de lectura pasa al inicio de la siguiente línea.

USO DE LA PROPOSICION DE ASIGNACION ARITMETICA

El ejemplo siguiente ilustra el uso de la proposición de asignación aritmética.

```
PROGRAM ejemplo;
VAR
    x,y:REAL;
BEGIN
x:=37.0;
y:=0.213;
WRITE(x:5,y:7);
END.
```

En la sección VAR, las variables x y y han sido declaradas de tipo REAL. Obsérvese que a las variables reales se les asigna una constante real (líneas 5 y 6). Toda constante explícita en Pascal debe tener un dígito por lo menos a ambos lados del punto decimal.

Por ejemplo, las siguientes asignaciones son incorrectas:

```
x:=37.;
y:=.213;
```

Lo correcto sería:

```
x:=37.0;
y:=0.213;
```

Obsérvese además que el valor de la variable x se imprimirá en un ancho de campo de cinco columnas y el valor de la variable y en un ancho de campo de siete columnas.

En general una proposición de asignación aritmética tendrá la forma siguiente:

```
nombre de variable:=expresion aritmetica;
```

Donde una expresión aritmética puede ser:

a) Una constante (real o entera).

b) Una variable (real o entera).

c) Cualquier combinación de a) y b) mediante operadores aritméticos.

+ suma

- resta

* producto

/ división

Son ejemplos de expresiones aritméticas válidas los siguientes.

38.0

238147

a+b*48.0-x13

a/d+omega

alfa-beta/x

gamma

w123

max+15

En una expresión aritmética primero se evalúan los productos y divisiones de izquierda a derecha (igual jerarquía entre sí) y luego las sumas y restas de izquierda a derecha (menor jerarquía que los anteriores, pero igual entre sí).

Esta relación jerárquica entre los operadores aritméticos puede modificarse haciendo uso de paréntesis, por ejemplo en la expresión:

a*(b+c)

donde: a=2 b=3 y c=4

Primero se evaluará la subexpresión, b+c (3+4=7) y luego el producto; a*7 (5*7=35). Toda subexpresión que se encuentre entre paréntesis tendrá la mayor jerarquía de evaluación.

En el caso de que en una expresión aritmética aparezcan paréntesis anidados, la evaluación será desde el parénte-

sis más interno hacia el más externo, por ejemplo, en la expresión aritmética:

```
a((a-c)+b*(c+d))
```

Donde a=2, b=3, c=1, d=5

Se evaluarán primero las subexpresiones a-c (2-1=1) y c+d (1+5=6), luego la subexpresión b*6 (3*6=18), después la subexpresión (a-c)+b*(c-d) (1+18=19) y finalmente a*((a-c)+b*(c+d)) (2*19=38).

La utilidad de la impresión de letreros se ilustra en el siguiente ejemplo:

```
PROGRAM ejemplo;
VAR
    x,y,z:REAL;
BEGIN
READ(x,z,y);
z:=(x+y)*z-x;
WRITE('z=',z:8);
END.
```

Donde a x, y y z se les asignarán por lectura los valores 5.1, 6.3 y 5.0 respectivamente.

En la séptima línea además de imprimirse el valor de la variable z (en un campo de 8 espacios se imprimen los símbolos z=). La impresión quedará como sigue:

```
z=  ____51.9
```

Ejemplos válidos de impresión de letreros:

```
WRITE('el resultado es =',x:12);
WRITE('este es un mensaje');
```

Obsérvese que todo letrero que se imprime está entre comillas sencillas.

Otra instrucción de impresión además del WRITE, es el WRITELN. La diferencia entre ambos quedará mejor ilustrada mediante el siguiente ejemplo:

```
WRITELN('El valor es',-30);
WRITE('x=',23);
WRITELN(' ':5,'Hola');
WRITE('este es un mensaje');
```

La impresión será:

```
El valor es       -30
--------------------
x=        23     Hola
--------------------
Este es un mensaje
```

Obsérvese que un WRITELN, contrariamente al WRITE, imprime sobre una línea y luego pasa el control de impresión al inicio de la siguiente línea. Por ejemplo, en una serie de tres WRITELN se imprimirá sobre tres líneas.

```
WRITELN(1);
WRITELN(2);
WRITELN(3);
```

La impresión será:

```
          1
----------
          2
----------
          3
----------
```

Una serie de tres WRITE imprimirá sobre una misma línea:

```
WRITE(1);
WRITE(2);
WRITE(3);
```

La impresión será:

```
          1         2         3
------------------------------
```

La instrucción:

```
WRITELN;
```

deja una línea en blanco, por ejemplo:

```
WRITELN('LINEA 1');
WRITELN;
WRITELN('LINEA 3');
```

se imprimirá:

```
LINEA 1
-------

-------
LINEA 3
-------
```


NOMBRES DE VARIABLES

Es deseable que el nombre de las variables escogidas por el programador refleje el propósito para el cual son utilizadas, ya que esto hace mucho más legibles los programas, por ejemplo la asignación:

```
x:=s*t;
```

no dice gran cosa acerca de lo que se intenta hacer, sin embargo la asignación:

```
longcir:=pi*radio;
```

da suficiente información acerca de lo que se intenta hacer: calcular la longitud de una circunferencia de radio dado.

El siguiente ejemplo ilustra esta idea:

```
PROGRAM calculo;

CONST
      pi=3.1416;
VAR
      longitud,area,volumen,radio :  REAL;
BEGIN
READ(radio);
longitud:=pi*radio;
area:=pi*radio*radio;
volumen:=pi*radio*radio*radio;
WRITELN('Radio = ',radio,'Longitud de circunferencia = ',
         longitud,'Area de circulo = ',area,
         'Volumen de esfera = ',volumen);
END.
```

En este ejemplo se tiene la sección CONST, en la cual se ha declarado e iniciado con un valor determinado a Pi. El valor de Pi no puede modificarse en el programa, es decir, no se puede asignarle otro valor, por ejemplo:

```
Pi:=4823.01;
```

no es válido. En la sección CONST sólo se inician identificadores cuyo valor no se va a modificar en el bloque de ejecución del programa, su valor permanece constante a lo largo de la ejecución del mismo.

COMENTARIOS

En el programa anterior, los nombres de las variables dan una idea clara de la intención del programa (calcular la longitud de una circunferencia, el área de un círculo y el volumen de una esfera). Se puede aún realzar más esta intención haciendo uso de comentarios dentro del programa. En un programa tan simple como el del ejemplo, esto puede parecer innecesario, pero la intención es mostrar la utilidad de los comentarios.

Se ilustrará el uso de comentarios con el ejemplo anterior.

```
PROGRAM calculo;

{ Este Programa calcula la longitud de una
  circunferencia, el area de un circulo y
  el volumen de una esfera }

CONST
      Pi=3.1416;
VAR
      longitud,     (* longitud de la circunferencia        *)
      area,         (* area del circulo                     *)
      volumen,      (* volumen de la esfera                 *)
      radio :       (* radio de la circunferencia, circulo
                    y esfera                                *)

      REAL;

BEGIN
(* lectura del radio *)
READ(radio);
(* calculo de la longitud de la circunferencia *)
longitud:=Pi*radio;
(* calculo del area del circulo *)
area:=Pi*radio*radio;
(* calculo del volumen de la esfera *)
volumen:=Pi*radio*radio*radio;
```

```
(* impresion de resultados *)
WRITELN('Radio=',radio,'longitud de circunferencia=',
        longitud,'Area de circulo=',area,'Volumen
        de esfera=',volumen);
END.
```

Obsérvese que los comentarios van entre paréntesis como sigue:

{comentarios}

(*comentarios*)

Es indiferente qué paréntesis se use.

En el programa del ejemplo anterior, el primer comentario especifica la función del mismo; los que se encuentran en la parte de declaraciones especifican el propósito de cada una de las variables, asimismo los que se encuentran en la parte de las instrucciones dan el significado de ellas. Cada segmento consta de una sola instrucción. Por lo general varias instrucciones determinan una función específica y el comentario que describe la función de tal segmento debe anteceder a estas instrucciones.

PROBLEMAS PROPUESTOS

1. Traducir al lenguaje Pascal las siguientes expresiones aritméticas:

   a) AB

   b) $\frac{AB + 60}{D^2}$

   c) $AX^2 + BX + C$

   d) $AM + \frac{X + \frac{Y}{DR}}{X^2 - \frac{2}{MR}}$

   e) $\frac{P}{4A - \frac{5D}{8M} + A^2}$

2. Identificar los errores de las siguientes proposiciones de asignación que no son válidas, debido a que cada una de ellas tiene por lo menos un error.

   a) `A:=AB;`

   b) `R:=R**4;`

   c) `X+Y:=R+4;`

   d) `-X:=A*B:=A*D+2`

   e) `SALIDA:=R*-X;`

3. Decir de qué tipo debe ser la declaración de la variable R en las siguientes proposiciones:

   a) `R:=3.2*X+8.01*R;`

   b) `R:=4;`

   c) `R:=2*M;`

   d) `R:=FALSE;`

4. Escribir un programa que lea una cantidad dada en libras y escribir su equivalente en kilogramos.

5. Escribir un programa que evalúe la siguiente expresión matemática:

$$\text{SALIDA} = \frac{P}{4A - \frac{5D}{M} + A^2}$$

dando como datos de entrada los siguientes valores:

A	7.8
D	0.02
M	-4.85
P	5

6. Escribir un programa que calcule la aceleración de un automóvil después de 10 segundos, si parte del reposo con una velocidad constante 60 km/h.

Velocidad=Aceleración*Tiempo

7. Escribir un programa que calcule la calificación promedio de cinco alumnos, si las calificaciones son: 10, 2.8, 9.5, 8.3, 5.99.

8. Escribir un programa que lea un número, lo eleve al cuadrado y le sume dos veces el valor leído.

CAPITULO 3 LAS ESTRUCTURAS LOGICAS DE CONTROL

GENERALIDADES

Las estructuras de control, conocidas como estructuras iterativas, permiten que un segmento de programa se ejecute n veces hasta que se cumpla una condición dada. Esta clase de estructuras de control explotan la velocidad de la computadora.

Las estructuras de control que permiten tomar un curso de acción determinado por una condición dada, se conocen como estructuras de decisión.

3.1 LA ESTRUCTURA DE CONTROL FOR-DO

En algunas ocasiones será necesario que un segmento de un programa se ejecute varias veces, tal es el caso si en el último ejemplo del capítulo 2 se corriera el programa con diferentes valores de radio, obviamente para cada valor diferente de radio se tendría que correr cada vez el programa.

La estructura de control FOR-DO permite que un segmento determinado de un programa se ejecute n veces en una sola corrida. La sintaxis o construcción de la estructura es como sigue:

```
FOR variable de control:=expa1 TO expa2 DO
proposicion simple o compuesta;
```

donde la variable de control se inicia con el valor de la expresión aritmética y la proposición simple o compuesta se ejecuta tantas veces como el valor expa2-expa1+1.

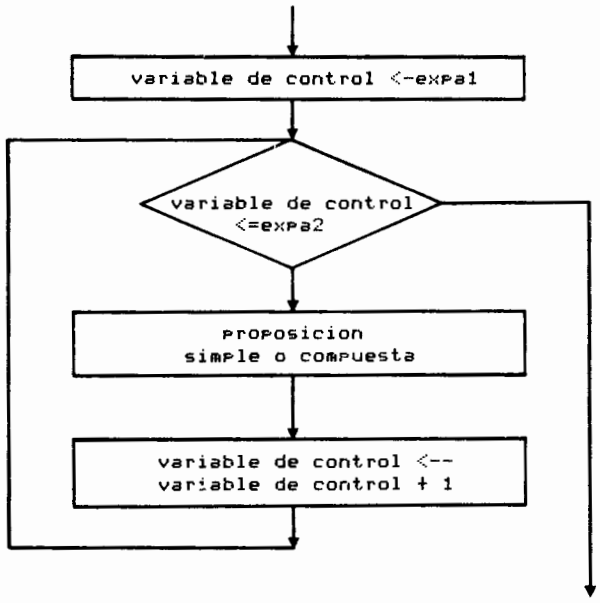
Debe tenerse presente que:

1) Los incrementos de la variable de control son de uno en uno.

2) Los valores de las expresiones aritméticas 1 y 2 no pueden alterarse dentro del alcance de la estructura de control.

3) Los valores de las expresiones aritméticas 1 y 2 deben ser enteros.

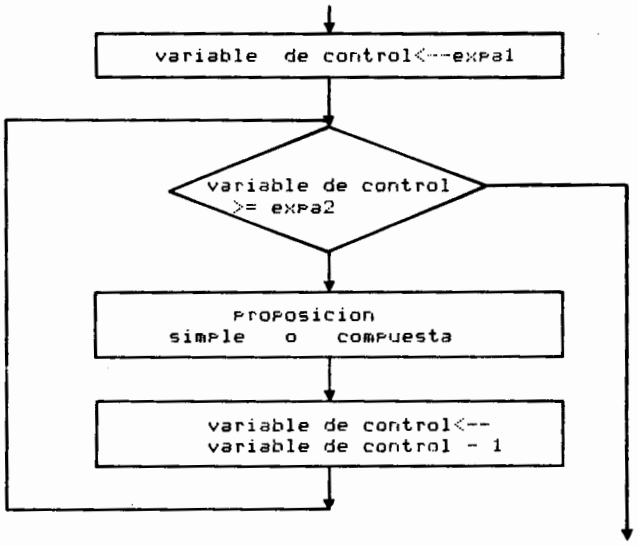
El diagrama correspondiente a esta estructura da una idea más precisa del flujo de control de esta estructura.



La siguiente estructura del FOR-DO permite decrementar la variable en vez de incrementarla.

```
FOR variable de control:=expa1 DOWNTO expa2 DO
proposicion simple o compuesta;
```

Se observa que en este caso se usa la palabra reservada DOWNTO en vez de TO. También que expa1>=expa2 para que el FOR-DO se ejecute una o más veces. El diagrama de flujo correspondiente se muestra a continuación:



Una sola proposición es una proposición simple y una secuencia de proposiciones entre las palabras reservadas BEGIN y END es una proposición compuesta.

El siguiente programa es un ejemplo ilustrativo del uso del FOR-DO con una proposición compuesta dentro de su alcance.

```
PROGRAM calculo;

CONST
     pi=3.1416;
VAR
     longitud,area,
     volumen,radio   : REAL;
     i               : INTEGER;

BEGIN
FOR i:=1 TO 10 DO
  BEGIN
  READ(radio);
  longitud:=pi*radio;
  area:=pi*radio*radio;
  volumen:=pi*radio*radio*radio;
  WRITELN('radio = ',radio,'longitud',
          ' de circunferencia = ',longitud,
          'area del circulo = ',area,
          'volumen de esfera = ',volumen);
  END;
END.
```

Se observa que todo el bloque ejecutable del último ejemplo del capítulo 2 se encuentra dentro del alcance de la estructura FOR-DO, y puesto que este bloque consta de más de una proposición y está limitado por un BEGIN y END, se trata por consiguiente de una proposición compuesta que está sangrada con respecto a la estructura FOR-DO con el propósito de delimitar visualmente el alcance de esta estructura.

3.2 ARREGLOS

Un arreglo consta de un conjunto ordenado de componentes. Se puede hacer referencia (tener acceso) a un componente de un arreglo mediante uno o más índices, dependiendo de la dimensión del arreglo que haya sido declarada.

Los siguientes son ejemplos de declaraciones de arreglos:

```
VAR
    x,y:ARRAY [1..10] OF INTEGER;
    m,w,t:ARRAY [-1..2,0..3] OF REAL;
```

x y y son arreglos de una dimensión (vectores) de tipo entero y tienen diez componentes cada uno. Cada componente puede tener un valor entero. El siguiente diagrama muestra la representación gráfica del vector x:

x[1]	8
x[2]	17
x[3]	15
x[4]	303
x[5]	102
x[6]	37
x[7]	25
x[8]	73
x[9]	101
x[10]	803

A la izquierda se tiene la forma de hacer referencia a un componente particular del vector. Por ejemplo x[7] significa que se hace referencia al componente 7 del vector x. A la derecha del diagrama se tiene el contenido de cada componente; así, el contenido del componente 7 del vector x es 25.

x y y son arreglos de una dimensión y sus respectivos índices toman valores que van de 1 a 10 tal y como está indicado en la declaración.

Los arreglos m, w y t son arreglos de dos dimensiones (matrices). Para hacer referencia a un componente de cualquiera de los tres arreglos, es necesario hacerlo con dos índices.

El siguiente diagrama muestra la representación gráfica del arreglo m:

renglones \ columnas	0	1	2	3
-1	1.5	2.37	4.8	3.5
0	48.0	301.05	607.1	0.37
1	0.48	37.1	3.7	4.5
2	12.0	43.3	15.0	1.1

Por ejemplo m[1,2] significa que se hace referencia al componente que se encuentra en la hilera 1 y la columna 2, es decir, en esa posición se encuentra el valor real 3.7. Obsérvese que el primer índice se refiere a hileras y el segundo a columnas.

De acuerdo a la declaración de m, w y t el primer índice (renglones) toma valores que van de -1 a 2, y el segundo índice (columnas) toma valores que van de 0 a 3. La coma es un separador entre los rangos de valores que toma cada índice.

El ejemplo siguiente muestra un segmento de programa que lee un vector (arreglo de una dimensión).

```
FOR i:=1 TO 10 DO
    READ(x[i]);
```

En este ejemplo se leen 10 valores que se encuentran en una tarjeta o línea si se está en una terminal. Suponiendo que los valores son los del ejemplo anterior para el vector x, los datos deben estar como sigue:

Tarjeta

```
          1         2         3         4
1234567890123456789012345678901234567890 - - -
8 17 15 303 102 37 25 73 101 803
```

Debe haber un blanco al menos entre cada valor. Estos valores se asignarán de acuerdo al ejemplo mencionado; es decir, x[1] tendría el valor 8, x[2] tendría el valor 17, etcétera.

El ejemplo que sigue es también un segmento de programa que suma los vectores x y y y el resultado queda en el vector x:

```
FOR i:=1 TO 10 DO
    x[i]:=x[i]+y[i];
```

Obsérvese que en este ejemplo y en el de la lectura del vector x, hay una sola proposición dentro del alcance del FOR-DO, es decir: una proposición simple.

El siguiente ejemplo imprime un vector:

```
FOR i:=1 TO 10 DO
   WRITE(x[i]);
```

La impresión de los datos será sobre una línea, cada valor se imprimirá en un campo de 10 columnas justificado a la derecha.

El siguiente ejemplo habla por sí sólo:

```
PROGRAM SUMA;

VAR
   x,y,z : ARRAY [1..10] OF INTEGER;
   i     : INTEGER;
BEGIN
(* LECTURA DEL VECTOR 'X' *)
FOR i:=1 TO 10 DO
    READ(x[i]);
READLN;

(* LECTURA DEL VECTOR 'Y' *)
FOR i:=1 TO 10 DO
    READ (y[i]);

(* SUMA LOS VECTORES 'X' Y 'Y' *)
FOR i:=1 TO 10 DO
    z[i]:=x[i]+y[i];

(* IMPRESION DEL VECTOR 'Z' *)
FOR i:=1 TO 10 DO
    WRITE(z[i]);
END.
```


El siguiente segmento de programa es un ejemplo de lectura de una matriz:

```
FOR i:=1 TO 4 DO
    BEGIN
    FOR k:=1 TO 4 DO
        READ(s[i,k]);
    READLN;
    END;
```

s es una matriz de cuatro renglones y cuatro columnas, tanto el índice (i) para los renglones como (k) para las columnas van de 1 a 4. La lectura se hace por renglón.

Por ejemplo, si s es una matriz entera con valores:

$$\begin{bmatrix} 7 & 8 & 7 & 4 \\ 5 & 16 & 17 & 3 \\ 10 & 15 & 1 & 5 \\ 8 & 4 & 3 & 2 \end{bmatrix}$$

primero se lee el primer renglón, luego el segundo, etc. Se sugiere al lector que haga una prueba de escritorio (seguir paso a paso) con el segmento de programa y los datos dados.

3.3 LA ESTRUCTURA DE CONTROL IF-THEN-ELSE

A menudo es necesario tomar un curso de acción (decisión) de acuerdo a una condición dada.

Por ejemplo, si se quiere detectar en el ejemplo anterior que el radio sea negativo o cero, valores para los cuales no tiene sentido hacer los cálculos, el programa, en este caso, podría tomar uno de los dos cursos de acción siguientes: si el radio es mayor que cero se realizan los cálculos correspondientes, en caso contrario se imprimiría un mensaje indicando que el radio es negativo o cero. En el lenguaje Pascal se cuenta con una estructura de control semejante.

La sintaxis de esta estructura es como sigue:

```
IF (condicion) THEN
   proposicion simple o compuesta-1
ELSE
   proposicion simple o compuesta-2
```

La condición sólo puede asumir dos valores: falso o verdadero. Si es verdadero se realiza la proposicion simple o compuesta-1, si es falsa se realiza la proposicion simple o compuesta-2 En una condición pueden aparecer operadores de relación así como operadores lógicos.

La tabla siguiente muestra los operadores de relación en Pascal y su significado correspondiente:

OPERADOR	SIGNIFICADO
=	IGUAL A
>	MAYOR QUE
>=	MAYOR O IGUAL QUE
<	MENOR QUE
<=	MENOR O IGUAL QUE
<>	DIFERENTE A

Son ejemplos de expresiones de relación:

```
a>b
0.5*x-d/e<>0.7
a*c<=x+5.06
```

Tomando en cuenta ahora la posibilidad de leer valores menores o iguales a cero para el radio en el ejemplo anterior, se tendrá:

```
PROGRAM calculo;

CONST
     pi=3.1416;
VAR
     longitud,area,
     volumen,radio   : REAL;
     i               : INTEGER;

BEGIN
FOR i:=1 TO 10 DO
  BEGIN
  READ(radio);
  IF (radio>0) THEN
     BEGIN
     longitud:=pi*radio;
     area:=pi*radio*radio;
     volumen:=pi*radio*radio*radio;
     WRITELN('radio = ',radio,'longitud'
             'de circunferencia = ',longitud,
             'area del circulo = ',area,
             'volumen de esfera = ',volumen);
     END
  ELSE { radio<=0 }
     WRITELN('El radio es invalido');
  END;
END.
```

Obsérvese que el END que se encuentra antes del ELSE no lleva punto y coma (;) y que después del ELSE hay un comentario que explica la condición que se tiene que dar para que se ejecute ese segmento de programa. Esta práctica es recomendable ya que hace más leíble un programa.

Un caso particular de la estructura IF-THEN-ELSE es el IF-THEN cuya sintaxis es como sigue:

```
IF (condicion) THEN
   proposicion simple o compuesta
```

donde condición es una expresión lógica o booleana.

Si en el ejemplo anterior el END que se encuentra antes del ELSE tuviera punto y coma (;) el compilador interpretaría a la estructura IF-THEN-ELSE como el caso particular IF-THEN, razón por la cual el END antes del ELSE no lleva punto y coma (;).

La tabla siguiente muestra los operadores lógicos en Pascal y su significado.

operador logico	significado
AND	y
OR	o
NOT	no

Son ejemplos de expresiones lógicas o booleanas los siguientes:

```
a<>0.5 AND b>=c*d
a<=a OR b>0.5*x
alfa OR(beta AND gamma)
```

El valor falso o verdadero de una expresión de relación o lógica (booleana) se puede asignar a una variable de tipo BOOLEAN.

El siguiente es un ejemplo de una declaración de variables de tipo BOOLEAN y proposiciones de asignación booleana:

```
VAR
    a,b,c,d,radio:REAL;
    x,w,t:BOOLEAN;

          *
          *
          *

  x:=a<>0.5 AND b>=c*d;
  w:=radio>=0.0;

          *
          *
          *
```

Obsérvese que una variable de tipo lógico o booleano únicamente puede asumir dos valores: falso o verdadero. Es posible asignarle a una variable booleana los valores constantes TRUE (verdadero) y FALSE (falso). TRUE y FALSE son palabras reservadas.

```
x:=TRUE;
w:=FALSE;
```

o bien estos valores constantes pueden aparecer en una expresión booleana. Por ejemplo:

```
(alfa<>0.5*beta) OR TRUE
(FALSE AND gamma) OR (NOT(omega))
```

El ejemplo siguiente calcula e imprime todas las combinaciones posibles de valores booleanos de la expresión booleana a OR(b AND c) así como el valor booleano resultante de la expresión.

```
PROGRAM  tabla;

VAR
   a,b,c,d : BOOLEAN;
   i,J,k,l : INTEGER;

BEGIN
FOR i:=0 TO 1 DO
   FOR J:=0 TO 1 DO
        FOR K:=0 TO 1 DO
           BEGIN
           IF (i=0) THEN
              a:=FALSE
           ELSE
              a:=TRUE;
           IF (J=0) THEN
              b:=FALSE
           ELSE
              b:=TRUE;
          IF (k=0) THEN
             c:=FALSE
          ELSE
             c:=TRUE;
             d:=a OR (b AND c);
          IF (d) THEN
             l:=1
           ELSE
             l:=0;
          WRITELN(i,J,k,l);
          END;
  END.
```

Obsérvese que el anidamiento de los tres FOR-DO es el responsable de la generación de todas las combinaciones posibles de la expresión booleana. Un FOR-DO por cada variable diferente en la expresión. El programa imprime un 1 para el valor TRUE (verdadero) y un 0 para FALSE (falso).

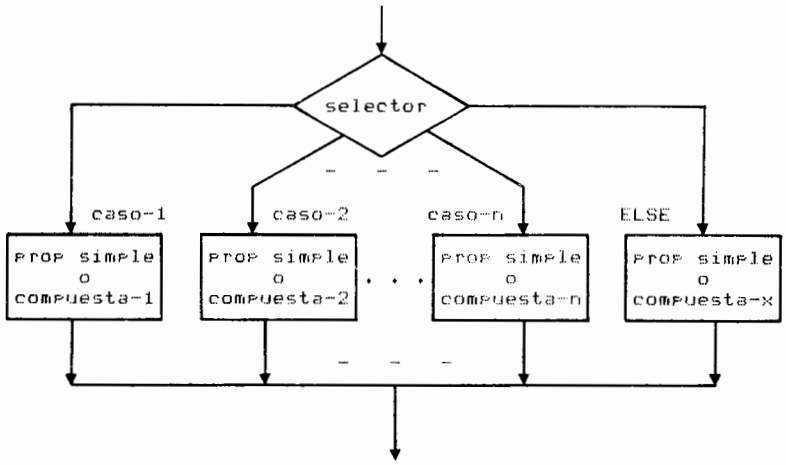
3.4 LA ESTRUCTURA DE CONTROL CASE-OF

Esta estructura permite elegir un curso de acción de acuerdo al valor de un selector. Esta estructura no es más que una estructura de control de decisión múltipl

Su sintaxis es como sigue:

```
CASE selector OF
   caso 1 :prop simple o compuesta-1
   caso 2 : prop simple o compuesta-2
                    *
                    *
                    *
   caso n :prop simple o compuesta-n
   ELSE   :prop simple o compuesta-x
   END;
```

El diagrama de flujo de control correspondiente es:



Esta estructura de control se ejecuta de la siguiente manera: si el valor del selector es igual a uno de los casos (etiquetas) se ejecuta únicamente la proposición simple o compuesta con esa etiqueta. Si ninguno de los casos concuerda con el valor del selector, se ejecuta la proposición simple o compuesta que tiene la etiqueta ELSE. Obsérvese que esta estructura debe terminar con un END y que este END no tiene un correspondiente BEGIN. La etiqueta ELSE y su correspondiente proposición son optativas.

En general caso 1,---,caso n son constantes escalares tales como números, caracteres, etc., posteriormente se definirán los escalares.

El siguiente programa calcula cuántos alumnos de un grupo de 45 obtuvieron:

R reprobado

S suficiente

B bueno

E excelente

```
PROGRAM calcula;
VAR
   notatr,i,contr,conts,contb,conte : INTEGER;
   nota : REAL;
BEGIN
contr:=0;
conts:=0;
contb:=0;
conte:=0;
FOR i:=1 TO 45 DO
   BEGIN
   READ(nota);
   CASE (TRUNC(nota/10)) OF
        0,1,2,3,4,5:  contr:=contr+1;
      6,7:            conts:=conts+1;
      8,9:            contb:=contb+1;
      10:             conte:=conte+conte+1;
   END;
   END;
   WRITELN('R':5,contr:3,'S':5,conts:3,'B':5,contb:3,
           'E':5,conte:3);
END.
```

Obsérvese que una proposición (simple en este caso) que se encuentra dentro del alcance del CASE-OF puede tener más de una etiqueta, por ejemplo la línea:

```
0,1,2,3,4,5:   contr:=contr+1;
```

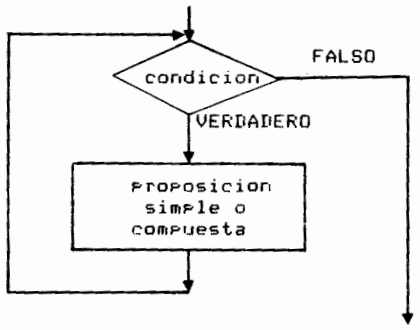
tiene 6 etiquetas. Si el selector, (una expresión aritmética en este caso) toma cualquiera de estos valores, se ejecutará contr:=contr+1.

3.5 LA ESTRUCTURA DE CONTROL WHILE-DO

Es una estructura de control repetitiva. Su sintaxis es como sigue:

```
WHILE condicion DO
      proposicion simple o compuesta
```

Su correspondiente diagrama de flujo de control es el siguiente:



Obsérvese que esta estructura primero verifica el valor verdadero o falso de la condición: si es verdad se ejecuta la proposición que se encuentra dentro del alcance del WHILE-DO y luego se vuelve a verificar el valor de la condición. En cuanto ésta tenga el valor falso el flujo de control pasa fuera del alcance del WHILE-DO.

Evidentemente el valor de la condición debe alterarse dentro del alcance del WHILE-DO, de lo contrario la proposición se ejecutará indefinidamente.

El siguiente programa calcula el promedio de n (n>0) números que se leen. En cuanto se lee un número igual a cero y el WHILE-DO lo detecta, esta estructura deja de ejecutarse.

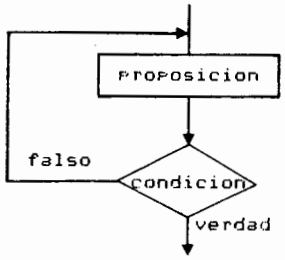
```
PROGRAM ejemplo;
VAR
  promedio,nro,temp : REAL;
  contador : INTEGER;
BEGIN
temp:=0.0;
contador:=1;
READ(nro);
WHILE nro<>0 DO
  BEGIN
  temp:=temp + nro;
  contador:=contador + 1;
  READ(nro);
  END;
promedio:=temp/contador;
WRITELN('PROMEDIO = ',promedio);
END.
```

3.6 LA ESTRUCTURA DE CONTROL REPEAT-UNTIL

La estructura de control REPEAT-UNTIL es también una estructura de control repetitiva. Su sintaxis es como sigue:

```
REPEAT
   proposicion simple o compuesta
UNTIL condicion
```

Su diagrama de flujo de control correspondiente es:



Contrariamente al WHILE-DO esta estructura verifica la condición después de ejecutarse la proposición simple o compuesta dentro del alcance del REPEAT-UNTIL. De modo que independientemente del valor de la condición, la proposición dentro de su alcance se ejecuta al menos una vez.

Usando un REPEAT-UNTIL, en lugar de un WHILE-DO, en el ejemplo anterior del cálculo del promedio de n valores, quedaría como sigue (sólo se muestra el segmento ejecutable del programa).

```
BEGIN
temp:=0.0;
contador:=1;
REPEAT
  READ(nro);
  temp:=temp+nro;
  contador:=contador+1;
UNTIL nro=0;
promedio:=temp/contador;
WRITELN('PROMEDIO = ',promedio);
END.
```

PROBLEMAS PROPUESTOS

1. Hacer un programa que calcule el área de un triángulo dando como datos de entrada los valores de los lados.

$$A = \sqrt{S(s - a)\ (s - b)\ (s - c)}$$

$$S = \frac{a + b + c}{2}$$

NOTA: No se puede obtener el área cuando la suma de las longitudes de dos lados cualesquiera es igual a la longitud del lado restante.

2. Un trabajador paga el 5% de impuesto si su sueldo anual es mayor a $ 100,000.00 y el 10% si su sueldo es mayor a $ 150,000.00. Si su sueldo anual es menor o igual a $ 100,000.00 no paga impuestos.

   Escribir un programa que lea el sueldo anual y calcule el impuesto a pagar y la cantidad neta que debe recibir el trabajador.

3. Escribir el programa anterior para 15 trabajadores.

4. Escribir un programa que obtenga cuántos alumnos y cuántas alumnas de un grupo de 50 aprobaron el primer examen de la materia Programación Estructurada. Los datos de entrada deben tener el siguiente formato:

NUMERO DE ALUMNO	SEXO	CALIFICACION

   Ambos campos son numéricos y los valores que pueden tener son los siguientes:

NUMERO DE ALUMNO.- 1,...,50

SEXO

1 HOMBRE

2 MUJER

CALIFICACION.- 0 < = CALIFICACION < = 100

La calificación aprobatoria debe ser mayor o igual a 60.

5. Escribir un programa que lea 50 datos con el formato del problema 4 e imprima como salida los datos de aquellas personas que aprobaron el primer examen, traduciendo el segundo campo numérico a letras. Por ejemplo, si los datos del primer alumno son:

   1 2 60.00

   la salida deberá ser:

   1 MUJER 60.00

6. Escribir un programa que evalúe la expresión:

$$\sum_{i=0}^{n} \sum_{j=-p}^{r} X_i^2 \, Y_j$$

   dando como datos de entrada los valores de X_i, Y_j, n, p y r.

7. Escribir un programa que lea e imprima un valor entero y obtenga e imprima el número de dígitos del dato leído.

8. Escribir un programa que lea un número entero y lo invierta; que imprima el dato leído y el dato resultante.

CAPITULO 4 SUBPROGRAMAS

GENERALIDADES

En un proceso de manufactura, tal como la fabricación de un automóvil, no se comienza por cómo obtener sus componentes básicos, por ejemplo, extraer el hierro de la mina, la gasolina y los plásticos del petróleo, etc., sino que se procede a armar el automóvil con partes previamente manufacturadas como la batería, llantas, motor, radiadores, etcétera.

Todas estas partes han sido hechas por compañías diferentes y por personas distintas. Cada una de las partes del automóvil ha sido diseñada y manufacturada bajo especificaciones rigurosas, de tal modo que durante el ensamblado todas las partes se correspondan. Además, una vez armado, el automóvil debe satisfacer los requerimientos previamente especificados durante el diseño, por ejemplo: confort, economía de combustible, etcétera.

De manera semejante, las diferentes partes de un programa que ha sido modularizado (dividido en partes, donde cada una de ellas tiene una función específica), pueden manufacturarse por personas diferentes bajo rigurosas especificaciones. El ensamblado de estas partes o módulos da origen a un programa.

4.1 FUNCIONES

Se define una función usando la declaración FUNCTION. Como ejemplo, a continuación se define la función maximo, que busca el elemento máximo en un vector de 10 enteros.

```
PROGRAM maxi;
VAR
        x:ARRAY [1..10] OF INTEGER;
    max,i:INTEGER;
(*--------------------------------------*)

FUNCTION maximo:INTEGER;
  VAR
       J:INTEGER;
   BEGIN
   maximo:=x[1];
   FOR J:=2 TO 10 DO
        IF(maximo<x[J])THEN
             maximo:=x[J];
   END;
(*--------------------------------------*)

BEGIN
FOR i:=1 TO 10 DO
   READ(x[i]);
max:=maximo*1;
WRITELN('maximo=',max);
END.
```

FUNCTION es la palabra reservada, máximo es el nombre de la función (de 1 a 6 caracteres, siendo el primero una letra y los restantes letras y/o dígitos).

INTEGER es el tipo de la función y significa que el nombre de la función tiene asociado un valor de tipo entero.

La función está declarada en el programa (programa principal) que hace uso de ella o la llama.

La variable i y el arreglo x declarados en el programa principal se dice que son globales a la función. Esto significa que estas variables pueden usarse dentro del alcance de la función, como es el caso del arreglo x .

La variable j declarada en la función se dice que es local a ella. Esto significa que esta variable sólo puede usarse dentro de la función y no fuera de ella.

Hacer uso de una variable local en un contexto global, ocasiona que el programa aborte. En el caso de que una variable global y una local tengan el mismo nombre, la global no es válida dentro del alcance de la local.

Nótese que una función debe aparecer en una expresión. En el ejemplo anterior la función no tiene argumentos y en el ejemplo que sigue la función tiene un argumento.

```
PROGRAM maxi;
VAR
        x:ARRAY [1..10] OF INTEGER;
    max,m,i:INTEGER;
(*--------------------------------------*)

FUNCTION maximo(n:INTEGER):INTEGER;
  VAR
        J:INTEGER;
   BEGIN
   maximo:=x[1];
   FOR J:=2 TO  n DO
        IF(maximo<x[J])THEN
             maximo:=x[J];
   END;
(*--------------------------------------*)

BEGIN
READ(m);
FOR i:=1 TO  m DO
   READ(x[i]);
max:=maximo(m)*1;
WRITELN('maximo=',max);
END.
```

El nombre de la función dentro del alcance de la misma, debe aparecer a la izquierda de una instrucción de asignación al menos una vez, es decir, en ella se calcula algún valor y se asigna al nombre de la función. Este debe ser del mismo tipo que el de la función.

En la llamada de la función:

```
max:=maximo(m)*1;
```

aparece como argumento o parámetro la variable m , declarada en el programa principal. A este parámetro se le llama parámetro actual . La variable m debe tener un valor entre 1 y 10 inclusive.

En la definición de la función:

```
FUNCTION maximo(n:INTEGER):INTEGER;
```

aparece como argumento o parámetro la variable n , ésta no aparece declarada en el programa principal ni en la función máximo. A este parámetro se le conoce como parám

metro formal y es un parámetro ficticio. Esto significa que su nombre puede ser cualquiera, incluso m , lo que importa en la definición y la llamada de la función es que los argumentos o parámetros se correspondan en número y en tipo.

Por ejemplo, en la definición de la función máximo se tiene un parámetro de tipo entero, por consiguiente en la llamada debe aparecer como argumento un parámetro de tipo entero como es el caso. Una función puede tener más de un argumento, en este ejemplo sólo tiene uno.

Como un ejemplo adicional se tiene el siguiente programa: Se lee un número real, el cual se pasa como argumento a dos funciones; la primera obtiene la parte entera del número y la segunda la fraccionaria. Las funciones hacen uso de la función de biblioteca TRUNC.

```
PROGRAM ejemplo;
VAR
     nro,partefra:REAL;
     parteen:INTEGER;
(*-----------------------------------*)

FUNCTION entero(x:REAL):INTEGER;
   BEGIN
   entero:=TRUNC(x);
   END;

FUNCTION fraccio(y:REAL):REAL;
   BEGIN
   fraccio:=y-TRUNC(y);
   END;
(*-----------------------------------*)

BEGIN
READ(nro);
parteent:=entero(nro);
partefra:=fraccio(nro);
WRITELN('parte entera=',parteent,
        'parte fraccionaria=',partefra);
END.
```

4.2 PROCEDIMIENTOS

Un procedimiento y una función tienen la misma estructura que un programa. La diferencia consiste en que el procedimiento no tiene asociado un valor tipo con su nombre y los argumentos los pasa por valor y/o variable. Una función sólo los pasa por valor.

Además, al hacer uso o llamar a un procedimiento, éste no debe aparecer en una expresión. Para hacer uso de él basta con escribir el nombre con sus argumentos (si los hay). Se define un procedimiento usando la declaración PROCEDURE.

El siguiente programa hace uso de un procedimiento que busca el valor máximo y mínimo de un vector de 10 enteros.

```
PROGRAM maxmin;
VAR
     x:ARRAY [1..10] OF INTEGER;
     max,min,i:INTEGER;
(*-------------------------------------*)

PROCEDURE busca(VAR x:ARRAY [1..10] OF INTEGER;
                VAR maximo,minimo:INTEGER);

   VAR
        k:INTEGER;
   BEGIN
   maximo:=x[i];
   minimo:=maximo;
   FOR k:=1 TO 10 DO
        BEGIN
        IF(maximo < x[i]) THEN
              maximo:=x[i];
        IF(minimo > x[i]) THEN
              minimo:=x[i];
        END;
   END;
(*-------------------------------------*)

BEGIN
FOR i:=1 TO 10 DO
   READ(x[i]);
busca(x,max,min);
WRITELN('maximo=',max,'minimo=',min);
END.
```

Obsérvese que en:

```
PROCEDURE busca(VAR x:ARRAY [1..10] OF INTEGER;
                VAR maximo,minimo:INTEGER);
```

los nombres de variables x, maximo y minimo están precedidos por la palabra reservada VAR. Esto significa que los argumentos se pasan por variable. La distinción entre parámetros por valor y referencia variable se explica a continuación.

Considérese el ejemplo siguiente:

```
PROGRAM cambia;
VAR
     temp,a,b:INTEGER;
(*-------------------------------------*)
PROCEDURE intercam(u,v:INTEGER);
   BEGIN
   temp:=u;
   u:=v;
   v:=temp;
   END;
(*-------------------------------------*)
BEGIN
a:=5;
b:=11;
intercam(a,b);
WRITELN('a=',a:5,'b=',b:5);
END.
```

Claramente la función realizada consiste en iniciar a las variables a y b, para después intercambiar sus valores y finalmente imprimirlos.

Es de suponerse que se imprimiría:

a= 11 b= 5

en realidad se tiene:

a= 5 b= 11

es decir no habrá ocurrido intercambio alguno entre los valores de a y b. La razón de esto consiste en que los argumentos formales se han definido por valor:

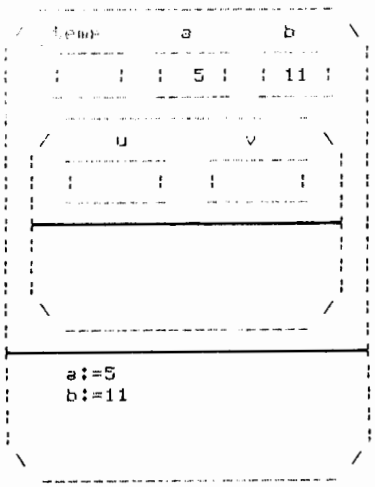
```
PROCEDURE intercam(u,v:INTEGER);
                   ↑
              aquí se han definido
              por valor
```

Esto significa que los valores de los argumentos que aparecen en la llamada del procedimiento intercambia, se copian a los argumentos formales u y v cuando se llama al procedimiento.

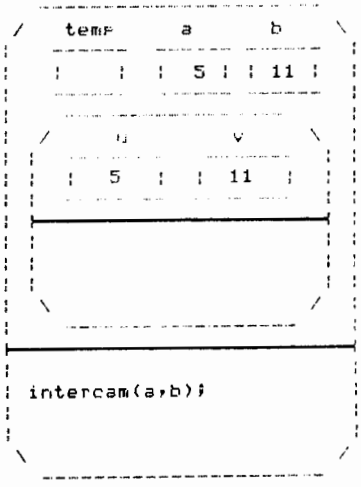
Durante la ejecución de éste se utilizan las variables u y v con sus valores asociados, no las variables a y b, por consiguiente los valores de a y b no se modifican.

La siguiente secuencia de diagramas muestra la serie de acciones que ocurre durante la ejecución del programa.

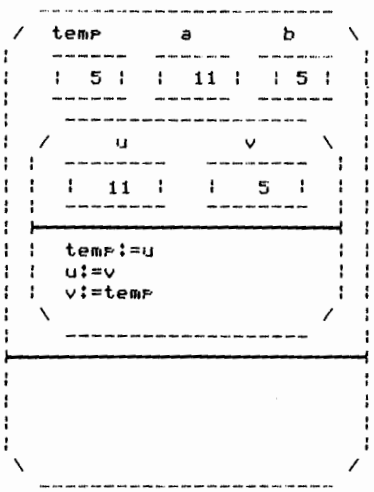
Los rectángulos con bordes redondeados representan el programa principal (rectángulo externo) y el procedimiento (rectángulo interno). Los rectángulos pequeños con bordes no redondeados representan las variables utilizadas. Debajo de la línea continua van las acciones realizadas de acuerdo al pie explicativo del diagrama. En la parte superior de ésta, en el procedimiento, van los parámetros formales y en la inferior las variables declaradas en el procedimiento, en este caso no las hay.



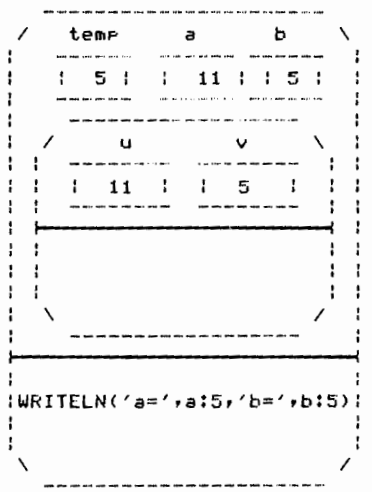
Se inician las variables a y b



Se llama al procedimiento intercambia.



Se intercambian los valores de a y b.



Se imprimen los valores de a y b.

4.3 SUBPROGRAMAS COMO ARGUMENTOS EN SUBPROGRAMAS

En ocasiones es conveniente que un subprograma aparezca como un argumento de otro, tal es el caso del ejemplo siguiente en el que se imprimen las tablas de valores lógicos (0 para falso y 1 para verdadero) de dos ecuaciones booleanas (lógicas).

```
PROGRAM ejemplo;
(*-------------------------------------*)

FUNCTION fn1(a,b,c:BOOLEAN):BOOLEAN;
   BEGIN
   fn1:=(a OR b) AND c;
   END;

FUNCTION fn2(a,b,c:BOOLEAN):BOOLEAN;
   BEGIN
   fn2:=(a AND c) OR (NOT b);
   END;

PROCEDURE tabla(FUNCTION fn(a,b,c: BOOLEAN):BOOLEAN);
   VAR
        x,y,z:BOOLEAN;
        i,j,k,l:INTEGER;
   BEGIN
   FOR i:=0 TO 1 DO
        FOR j:=0 TO 1 DO
             FOR k:=0 TO 1 DO
```

```
                BEGIN
                x:=(i=0);
                y:=(j=0);
                z:=(k=0);
                IF(fn(x,y,z)) THEN
                    l:=1
                ELSE
                    l:=0;
                WRITELN(i:4,j:4,k:4,l:4);
        END;
(*---------------------------------------*)

BEGIN
tabla(fn1);
tabla(fn2);
END.
```

Este programa consta de dos funciones que evalúan las ecuaciones booleanas y de un procedimiento que imprime una tabla con todas las combinaciones posibles de los argumentos (a, b y c) de una función, así como el valor resultante de ésta para cada caso.

En general una función y un procedimiento pueden tener como argumento a funciones y procedimientos.

4.4 PROGRAMAS MODULARES

Anteriormente se estudió cómo declarar subprogramas en "Pascal". Ahora se verá la característica de Pascal para producir programas modulares.

Se puede decir que un programa modular es aquel que ha sido dividido en varias partes y cada una realiza una función específica. La integración de estas partes realiza una tarea: lo que se quiere que haga el programa.

El ejemplo siguiente ayudará a fijar las ideas. Supónga se que se pide diseñar un programa que lea dos matrices y verifique si se pueden sumar y multiplicar; si es el caso, que se sumen y multipliquen y se imprima la matriz resultante. En caso contrario, que se imprima un mensaje diciendo que las matrices no son compatibles para suma o producto según el caso.

Como una primera aproximación a la solución del problema considérese el siguiente seudocódigo:

```
Se leen dimensiones de A y B
Se lee matriz A
Se lee matriz B
Se imprime matriz A
Se imprime matriz B
Si (la suma es compatible) luego
     Sumar A y B
     Imprimir C
o bien
       mensaje('A y B no son compatibles
                para la suma')
Si (el producto es compatible) luego
      Multiplicar A y B
      Imprimir C
o bien
      mensaje('A y B no son compatibles
               para el producto')
```

El seudocódigo anterior está a un nivel de abstracción muy alto, es decir, nos dice qué hacer mas no nos dice cómo hacerlo. La línea que dice:

```
Se lee matriz A
```

Se puede expander como sigue:

```
Desde i<-1 hasta ia repetir
      Desde k<-1 hasta ka repetir
          Leer X(i,k)
      fin
fin
```

Obsérvese que esta aproximación a la solución del subproblema 'Se lee matriz A' tiene un nivel de detalle lo suficientemente alto como para pasarlo directamente a Pascal:

```
FOR i:=1 TO ia DO
    BEGIN
    FOR k:=1 TO ka DO
        READ(x[i,k]);
    READLN;
    END;
```

Por otra parte, obsérvese que leer una matriz es una función específica, por consiguiente el segmento de código anterior puede ser un subprograma.

```
PROCEDURE leemat(VAR x:ARRAY [1..10] OF REAL;
                     ix,kx:INTEGER);
    VAR
        i,k:INTEGER;

    BEGIN
    FOR i:=1 TO ix DO
        BEGIN
        FOR k:=1 TO kx DO
            READ(x[i,k]);
        READLN;
        END;
    END;
```

Una llamada a este subprograma desde el programa principal para leer la matriz a sería como sigue:

```
leemat(a,ia,ka);
```

Es evidente que se puede proceder de manera semejante para cada línea del seudocódigo que especifique una función definida. A continuación se da el listado completo del programa. La correspondencia entre el seudocódigo para cada línea con una función específica y los subprogramas es directa.

```
PROGRAM sumproma;
VAR
     a,b,c:ARRAY [1..6,1..6] OF REAL;
     ia,ka,ib,kb:INTEGER;

PROCEDURE leemat(VAR x:ARRAY [1..6,1..6] OF REAL;
                      ix,kx:INTEGER);
     VAR
         i,k:INTEGER;

     BEGIN
     FOR i:=1 TO ix DO
         BEGIN
         FOR k:=1 TO kx DO
             READ(x[i,k]);
         READLN;
         END;
     END;


PROCEDURE impmat(VAR x:ARRAY [1..6,1..6] OF REAL;
                      ix,kx:INTEGER);
     VAR
         i,k:INTEGER;

     BEGIN
     FOR i:=1 TO ix DO
         BEGIN
         FOR k:=1 TO kx DO
             WRITE(x[i,k]);
         WRITELN;
         END;
     END;


PROCEDURE sumamat(VAR x,y,z:ARRAY [1..6,1..6] OF REAL;
                      ix,kx:INTEGER);
     VAR
         i,k:INTEGER;

     BEGIN
     FOR i:=1 TO ix DO
         FOR k:=1 TO kx DO
             z[i,k]:=x[i,k]+y[i,k];
     END;

PROCEDURE promat(VAR x,y,z:ARRAY [1..6,1..6] OF REAL;
                      ix,kx,ky:INTEGER);
     VAR
         i,k,J:INTEGER;
         sigma:REAL;

     BEGIN
     FOR i:=1 TO ix DO
         BEGIN
         sigma:=0.0;
         FOR k:=1 TO kx DO
             BEGIN
             FOR J:=1 TO kx DO
             sigma:=sigma+x[i,J]*y[J,k];
             z[i,k]:=sigma;
             END;
```

```
        END;
    END;

BEGIN
(* SE LEEN DIMENSIONES DE a Y b                *)
READLN(ia,ka,ib,kb);
(* SE LEE MATRIZ a                             *)
leemat(a,ia,ka);
(* SE LEE MATRIZ b                             *)
leemat(b,ib,kb);
(* SE IMPRIME MATRIZ a                         *)
impmat(a,ia,ka);
(* SE IMPRIME MATRIZ b                         *)
impmat(b,ib,kb);
IF ((ia=ib) AND (ka=kb)) THEN
   BEGIN
   (* SE SUMAN a Y b                                 *)
   sumamat(a,b,c,ia,ka);
   (* SE IMPRIME MATRIZ c                            *)
   impmat(c,ia,ka);
   END
ELSE
   WRITELN('LA SUMA DE a Y b NO ES COMPATIBLE');
IF (ka=ib) THEN
   BEGIN
   (* SE MULTIPLICAN  a Y b                          *)
   promat(a,b,c,ia,ka,kb);
   (* SE IMPRIME MATRIZ c                            *)
   impmat(c,ia,kb);
   END
ELSE
   WRITELN('EL PRODUCTO DE a Y b NO ES COMPATIBLE');
END.
```

PROBLEMAS PROPUESTOS

1. Un vendedor gana el 10% de sus ventas, si éstas son menores o iguales a $ 30,000.00; el 15%, si son mayores a $ 30,000.00 y menores o iguales a $ 50,000.00; y si son mayores a $ 50,000.00 el 30% del total de sus ventas. Escribir una función que calcule cuánto gana un vendedor pasando como parámetro el total de las ventas.

   Escribir un programa principal que lea el total de ventas y que imprima el valor obtenido por la función y el total de ventas.

2. Escribir una función que obtenga el elemento mayor de un arreglo de 50 números.

3. A continuación se da la tabla de calificaciones de un grupo de cinco alumnos de Programación Estructurada.

ALUMNO	CALIFICACIONES
1	20,100,80,90,10
2	100,60,70,90,80
3	30,90,100,10,5
4	80,40,90,100,80
5	90,95,75,100,70

   Se desea saber, qué alumnos aprobaron la materia.

   Escribir una función que obtenga el valor de TRUE cuando el promedio de calificaciones para un alumno sea mayor o igual a 60 y el valor de FALSE cuando el promedio sea menor de 60.

   Escribir un programa principal que imprima el siguiente formato:

ALUMNO	CALIF 1	CALIF 2	CALIF 3	CALIF 4	CALIF 5	APROBADO
1	20	100	80	90	10	
2	100	60	70	90	80	
3	30	90	100	10	5	
4	80	40	90	100	70	

y que en la columna de aprobado imprima el valor de la función obtenida para cada alumno.

4. Escribir una función que obtenga la suma de los valores de un arreglo de 4 x 5.

5. Escribir un *procedure* que tenga como parámetro un arreglo de 50 elementos enteros y verifique que todos sean positivos. En caso de que exista algún elemento negativo, identificar en qué posición se encuentra y cuál es su valor.

   Escribir un programa principal que imprima la posición en la que se encuentra dentro del arreglo y el valor del número negativo.

6. Escribir un programa principal que tenga lo siguiente:

   a) Un *procedure* que lea un arreglo de 50 números enteros ordenados de mayor a menor.

   b) Un *procedure* que imprima los datos leídos.

   c) Un *procedure* que, dado un número, busque en el arreglo leído y si ya se encuentra en él que imprima en qué posición se encuentra. En caso contrario, que obtenga y escriba entre qué números debería de estar y cuáles son las posiciones de éstos dentro del arreglo.

   El programa principal debe llamar a los *procedure's* a) y b), en ese orden. Leer un número y pasarlo como parámetro en la llamada al tercer *procedure*.

7. Escribir un programa principal que tenga los siguientes *procedure's* y las llamadas a ellos en el orden en que aparecen:

   a) Un *procedure* que lea un arreglo de 50 números enteros ordenados de mayor a menor.

   b) Un *procedure* que imprima los datos leídos.

   c) Un *procedure* que lea cinco datos, los imprima y los inserte en el arreglo en el lugar que les corresponda para que los datos sigan ordenados de mayor a menor.

   Escribir un programa utilizando *procedure's*, que lea un arreglo de 50 enteros ordenados de menor a mayor, que lo imprima, que lo ordene de mayor a menor y que imprima el arreglo resultante.

   NOTA: Se deben declarar como máximo tres *procedure's*.

CAPITULO 5 CARACTERES Y CADENAS DE CARACTERES

GENERALIDADES

Una de las aplicaciones de los lenguajes de programación que ha tomado incremento recientemente, es el proceso de cadenas de caracteres. Algunas de las aplicaciones donde este tipo de proceso toma lugar son por ejemplo: editores, procesadores de texto, etc. Pascal es uno de los lenguajes de programación que facilita el proceso de cadenas de caracteres debido a que ha implementado funciones y procedimientos primitivos para realizar esta tarea.

5.1 CARACTERES

La palabra reservada CHAR en Pascal se utiliza para declarar variables de tipo caracter, es decir, variables que contienen un caracter.

El siguiente ejemplo es un programa en el que aparecen variables y constantes de tipo caracter:

```
PROGRAM ejemplo;
CONST
      w='w';

VAR
      k,x,y,z : CHAR;

BEGIN
x:='x'; { se asigna a la variable x el caracter 'x' }
y:=x;   { se asigna a la variable y el valor de la variable x}
READ(z,k);
WRITELN(k:5,x:5,y:5,z:5);
END.
```

En la sección CONST se ha definido al identificador w con el valor inicial 'w'. Obsérvese que w es el identificador y w entre apóstrofes es el valor que se le asigna. Por otra parte se han declarado a k,x,y y z como

variables de tipo caracter en la sección VAR. Las dos primeras líneas de las instrucciones se explican por sí mismas.

Los datos que se asignan a las variables z y k deben darse de la siguiente manera:

```
ab
```

El caracter a se asigna a la variable z y el caracter b a la variable k.

Obsérvese que:

1) En la lectura los valores que se leen no están entre apóstrofes.

2) Los caracteres a y b no están separados por blancos.

Si por ejemplo los datos se hubieran dado como sigue:

```
a b
```

se asignará el caracter a a la variable z y blanco a la variable k, debido a que blanco también es un caracter.

Obsérvese también que los valores de las variables k,x,y y z se imprimen en campo de 5 columnas o espacios. Estos valores aparecerían como sigue:

```
b    x    x    a
```

Si en la impresión se omite el ancho del campo, esto es:

```
WRITELN(k,x,y,z);
```

la impresión de los valores queda:

```
bxxa
```

Es decir, el ancho de campo para la impresión de caracteres es por omisión de una columna o espacio.

Cada caracter tiene asociado un valor entero, valor que depende del código que utilice el sistema, por ejemplo ASCII,EBCDIC, etc. Estos valores son ordenados, es decir:

a<b<c<d< . . . <z

Debido a esto se pueden tener expresiones de relación y booleanas que involucren constantes y variables de tipo caracter. Por ejemplo:

b<'b' AND x<=omega

siendo b,x y omega variables de tipo caracter y 'b' la constante b.

5.2 LA FUNCION ORD

La función de biblioteca ORD hace posible que el programador conozca el valor entero asociado con cada caracter. La función ORD tiene como argumento una constante o variable de tipo caracter, y regresa el valor entero asociado con el caracter.

Por ejemplo:

```
*
*
*
x:='x'
I:=ORD(x);
*
*
*
```

En este ejemplo a la variable I, tipo entero, se le asigna el valor asociado con el caracter x que previamente se asignó a la variable x.

En el siguiente ejemplo:

J:=ORD('T');

se asigna a J el valor entero asociado, en el código utilizado por la computadora, con la constante T que aparece como argumento en la función ORD.

5.3 LA FUNCION CHR

La inversa de la función ORD es la función CHR. Esta función tiene como argumento una variable o constante de tipo entero, y regresa el caracter asociado con el entero, de acuerdo al código utilizado por la computadora, así por ejemplo:

```
   *
   *
   *
i:=101;
a:=CHR(i);
   *
   *
   *
```

a la variable a de tipo caracter se le asigna (según el código de símbolos utilizados por la computadora) el caracter asociado con el valor 101 que previamente se le asignó a la variable i de tipo entero.

```
alfa:=CHR(105);
```

A la variable alfa de tipo caracter se le asigna el caracter asociado con el valor 105.

5.4 CADENAS DE CARACTERES

La palabra reservada STRING en Pascal se utiliza para declarar variables que pueden contener hasta 256 caracteres, dependiendo de la computadora. No todo compilador tiene implementado este tipo de variables.

El ejemplo siguiente ilustra la declaración y uso de las variables de tipo STRING:

```
PROGRAM ejemplo;
CONST
    alfa='C12345';

VAR
   x,beta,r,t : STRING;
BEGIN
x:='abcdefg';
beta:=x;
READLN(r);
READLN(t);
WRITELN(x:5,alfa:8,beta:4);
END.
```

En la sección CONST se ha definido al identificador alfa con el valor inicial 012345.

Obsérvese que el valor que se asigna a alfa va entre apóstrofes por tratarse de una cadena de caracteres.

En la sección VAR las variables x,beta,r y t se han declarado de tipo STRING.

En el bloque de instrucciones a la variable x se le asigna el valor abcdefg y a beta el valor de x.

En la lectura, se tratará de asignar hasta 256 caracteres a la variable k, aun cuando la cadena dada tenga menos de 256 caracteres.

Por esta razón es conveniente hacer la lectura de una sola variable de tipo STRING con un READLN.

Si por ejemplo se quisieran leer dos cadenas de caracteres en las variables r y t, serían necesarias dos lecturas, una para cada variable.

```
READLN(r);
READLN(t);
```

Los datos de lectura correspondiente tendrían la siguiente presentación:

```
XABCDEFGHIJKL
ABC012345678901234567
```

una cadena de caracteres por línea. La primera cadena se asigna a r y la segunda a t.

Algunos compiladores permiten que se limite a menos de 256 caracteres a las variables de tipo STRING.

Por ejemplo en la declaración:

```
VAR
   x,beta,r,t:STRING[5];
```

a las variables x,beta,r y t sólo se les puede asignar como máximo 5 caracteres. Ahora es posible hacer una lectura de más de una variable de tipo STRING en un READ o READLN.

Por ejemplo:

```
READ(x,beta);
READLN(r,t);
```

Los datos de lectura tendrán la siguiente presentación:

```
abcde01234
56789xywst
```

Se asignará la cadena abcde a x, la cadena 01234 a beta, la cadena 56789 a r y la cadena xywst a t. Obsérvese que las diferentes cadenas no deben estar separadas por blancos ya que un blanco es un caracter y por consiguiente puede formar parte de una cadena de caracteres.

En la línea de impresión:

```
WRITELN(x:5,alfa:8,beta:4);
```

el valor de la variable x se imprime en un ancho de campo de 5 columnas. Puesto que a x se le han asignado 5 caracteres, éstos se imprimirán.

El valor de la variable alfa se imprimirá en un campo de 8 columnas. Como a alfa se le ha asignado una cadena de 6 caracteres, se imprimirán en el campo de 8 columnas justificados a la derecha.

El valor de la variable beta se imprime en un campo de 4 columnas, ya que a beta se le asignó una cadena de 5 caracteres, únicamente los cuatro primeros se imprimirán.

La impresión quedaría como sigue:

```
abcde  0123450123
   --
-------------------
campo  campo   campo
de 5   de 8    de 4
colum- colum-  co-.
nas    nas     lum-
               nas
```

5.5 OPERACIONES CON CADENAS DE CARACTERES

Las funciones y procedimientos de biblioteca que se verán a continuación, no en todos los compiladores se tienen implementados y en caso de tenerlos, pueden diferir ligeramente de los aquí mostrados; el lector debe verificar esto para el compilador que use. Las funciones y procedimientos dados aquí corresponden a los del sistema B6800.

5.5.1 LA FUNCION LENGTH (LONGITUD)

Esta función LENGTH tiene un argumento, variable o constante, de tipo STRING y regresa el número de caracteres de que consta el argumento.

Por ejemplo:

```
x:='0123456789';
k:=LENGTH(x);
```

A la variable de tipo entero k se le asigna el valor 10 que es el número de caracteres de la cadena que previamente se ha asignado a la variable x de tipo STRING.

```
J:=LENGTH('JUAN PEREZ');
```

Aquí se le asigna el valor 10 ya que la cadena JUAN PEREZ tiene 10 caracteres incluyendo el blanco.

5.5.2 LA FUNCION CONCAT (CONCATENAR)

Esta función CONCAT puede tener más de dos argumentos, variables o constantes de tipo STRING, y regresa concatenados sus argumentos.

Por ejemplo:

```
x:='0123456789';
w:=CONCAT(x,'abcdefgh');
```

A la variable w se le asigna:

```
01234567890abcdefgh
```

Otro ejemplo:

```
a:='abc';
b:='012';
c:='axab';
d:=CONCAT(a,b,c);
```

A la variable d se le asigna:

```
abc012axab
```

5.5.3 LA FUNCION COPY (COPIAR)

Esta función COPY tiene dos argumentos. El primero es una variable de tipo STRING con un subíndice, el segundo es una variable o constante entera.

Por ejemplo:

```
a:='0123456789';
b:=COPY(a[2],3);
```

De la cadena a se copian tres caracteres en la cadena b a partir del segundo caracter de la cadena a, es decir a b se asigna:

5.5.4 LA FUNCION POS (POSICION)

Esta función POS tiene dos argumentos de tipo STRING. Esta función busca la posición del primer argumento en el segundo.

Por ejemplo:

```
a:='012';
b:='abcde012fgh';
k:=POS(a,b);
```

A la variable entera k se le asigna el valor 6 ya que la posición del primer caracter de la cadena 012 en la cadena abcde012fgh es esa posición.

5.5.5 EL PROCEDIMIENTO INSERT (INSERTAR)

Este procedimiento tiene dos argumentos de tipo STRING. Esta función inserta el primer argumento en el segundo.

Por ejemplo:

```
x:='abc';
w:='0123456789';
INSERT(x,w[5]);
```

Inserta la cadena x en la cadena w a partir de la posición 5. La cadena w queda como sigue:

0123abc456789

5.5.6 EL PROCEDIMIENTO DELETE (SUPRIMIR)

Este procedimiento tiene dos argumentos, el primero es de tipo STRING y el segundo de tipo entero.

Por ejemplo:

```
w:='abcdefghijkl';
DELETE(w[7],4);
```

Suprime de la cadena w, 4 caracteres a partir de la posición 7. La cadena w queda como sigue:

abcdefkl

PROBLEMAS PROPUESTOS

1. Escribir un programa que lea 20 caracteres y los ordene en forma alfabética.

2. Escribir un programa que lea 20 caracteres y obtenga e imprima el número que le corresponda a cada uno de ellos dentro del conjunto de caracteres válidos.

3. Escribir un programa que lea 10 números y obtenga el caracter a que corresponde cada uno de los números leídos, dentro de los caracteres válidos.

4. Escribir un programa que lea los nombres de 10 personas y los imprima.

5. Escribir un programa que lea los nombres de 20 personas, imprima los nombres leídos, los ordene en forma alfabética e imprima la lista resultante.

6. Escribir un programa que lea un nombre y obtenga e imprima el número de vocales y el número de consonantes que tiene el nombre leído.

CAPITULO 6 LA SECCION TYPE

GENERALIDADES

En la sección TYPE de Pascal el programador puede definir nuevos valores (un nuevo tipo de variable) y luego declarar variables, en la sección VAR, que pueden tomar esos valores. Por ejemplo, a una variable así definida se le puede asignar el nombre de un día de la semana, el nombre de un mes, el nombre de un color, etc., esta facilidad junto con la de subrangos y conjuntos contribuyen a hacer los programas en Pascal más legibles y más confiables.

6.1 RANGOS

En la sección TYPE el programador puede definir un nuevo tipo de variable al listar el rango de valores posibles que asume. Tanto el nombre para el nuevo tipo de variable así como los nombres (rango de valores) que la definen son de la elección del programador.

Por ejemplo:

```
TYPE
    color=(ROJO,AZUL,VERDE,AMARILLO);
VAR
    x,z,w:color;
```

define el nuevo tipo de variable (color) y los valores que puede asumir cualquier variable de tipo color en la sección VAR. Cada valor que está en la lista entre paréntesis tiene asociado un valor entero de la forma siguiente:

```
0  para ROJO
1  para AZUL
2  para VERDE
3  para AMARILLO
```

Por consiguiente se puede hacer uso de la función ORD con las constantes y variables de tipo color.

Por ejemplo:

```
   *
   *
   *
x:=AZUL;
i:=ORD(x);
k:=ORD(AMARILLO);
   *
   *
   *
```

siendo i y k variables de tipo entero.

En este ejemplo a i se le asigna 1 y a k se le asigna 3. También es posible usar los operadores de relación y los operadores booleanos en expresiones en las que intervengan constantes y variables de este tipo (rango), así por ejemplo son válidas las siguientes expresiones:

```
x>=z
x<>AMARILLO AND x<=z
x<z OR w=AZUL AND NOT (w<>x)
```

A toda variable a la que se le ha definido el nombre de su tipo y se ha listado el rango de valores que asume, no se le puede asignar un valor durante una lectura.

Por ejemplo los siguientes datos:

```
AZUL VERDE
```

si se intenta asignarlos a x y w respectivamente en la lectura:

```
READ(x,w);
```

no es válido.

Tampoco es posible imprimir los valores de estas variables.

Por ejemplo:

```
        *
        *
        *
x:=AMARILLO;
w:=ROJO;
WRITELN(x,w);
        *
        *
        *
```

La impresión de los valores de x y w es inválida.

6.1.1 SUBRANGOS

En la siguiente definición:

```
TYPE
    semana=(LUN,MAR,MIER,JUE,VIE,SAB,DOM);
```

el rango de valores que puede asumir una variable de tipo semana va de LUN a DOM. Es posible definir un nuevo tipo de variable, por ejemplo dialab (día laborable) tal que su rango de valores sea un subrango de la definición anterior.

Esto se logra de la siguiente manera:

```
TYPE
    semana=(LUN,MAR,MIER,JUE,VIE,SAB,DOM);
    dialab=LUN..VIE;
```

Obsérvese que en la definición del tipo dialab únicamente se indican el primer y último valor (separados por dos puntos seguidos). Los valores intermedios se sobreentiende que quedan incluidos. En la definición de subrangos no hay paréntesis.

Ejemplo:

```
                    *
                    *
                    *
TYPE
     semana=(LUN,MAR,MIER,JUE,VIE,SAB,DOM);
     dialab=LUN..VIE;
VAR
     x,w:semana;
     a,b:dialab;
BEGIN
x:=LUN;
w:=DOM;
a:=x;
b:=w;
                    *
                    *
                    *
```

Obsérvese que x y w son de tipo semana y las variables a y b son de tipo dialab. Aun cuando estas variables (x,w y a,b) son de diferente tipo es válido que se asignen valores entre sí.

Por ejemplo en la penúltima línea (a:=x;) a la variable a de tipo dialab se le asigna el valor de la variable x de tipo dialab a la cual previamente se le asignó LUN. Esta asignación es válida porque el valor de x está dentro del rango de valores que puede asumir a. No es válida la asignación b:=w; ya que el valor de w (DOM) no está dentro del rango de valores que puede asumir b.

6.2 CONJUNTOS

Es posible definir en Pascal variables de tipo conjunto y realizar las operaciones con ellos que son comunes en matemáticas. Una definición de un conjunto debe estar después de la definición de un rango.

Por ejemplo:

```
                    *
                    *
                    *
TYPE
     semana=(LUN,MAR,MIER,JUE,VIE,SAB,DOM);
     dias= SET OF semana;
VAR
     w,r,s:dias;
                    *
                    *
                    *
```

siendo SET OF palabras reservadas en Pascal.

Obsérvese que es necesario definir previamente un rango (semana) para poder definir un tipo de conjunto (dias).

A continuación se ilustra la forma en que se asignan elementos a un conjunto.

La siguiente asignación:

```
w:=[];
```

significa que al conjunto w se le asigna el conjunto vacío. Entre los corchetes va el elemento o lista de elementos que se asignan al conjunto, puesto que en este caso no hay ninguno, se asigna el conjunto vacío a w.

La asignación:

```
r:=[LUN];
```

significa que se asigna al conjunto r el elemento LUN.

La asignación:

```
s:=[LUN,MIER,JUE,VIE,SAB];
```

significa que se asigna al conjunto s los elementos listados entre corchetes.

La asignación:

```
s:=[LUN..VIE];
```

significa que se asignan al conjunto s los siguientes elementos: LUN,MAR,MIER,JUE,VIE.

6.2.1 PERTENENCIA A UN CONJUNTO

La palabra reservada IN se usa para indicar 'pertenece a' o más brevemente 'esta en'.

Por ejemplo:

```
LUN IN r
```

significa 'LUN está en el conjunto r'. Esta afirmación puede ser verdadera o falsa, dependiendo de si el elemento LUN está o no en el conjunto r. Esto quiere decir que el valor de la expresión anterior es de tipo booleano.

Ejemplo:

```
          *
          *
          *
IF (LUN IN r) THEN
     WRITELN('LUNES ESTA EN r')
ELSE
     WRITELN('LUNES NO ESTA EN r');
          *
          *
          *
```

6.2.2 OPERACIONES CON CONJUNTOS

Las operaciones de unión, intersección y diferencia para conjuntos se realizan en Pascal utilizando los operadores aritméticos de suma, producto y resta respectivamente.

Para ilustrar esto supóngase que se han realizado las siguientes asignaciones:

```
w:=[LUN..VIE];
r:=[MIER..DOM];
```

La unión o suma de los conjuntos w y r es como sigue:

```
s:=w+r;
```

La suma de los elementos de w y r se asigna al conjunto s. Una vez realizada esta operación los elementos de s son [LUN,MAR,MIER,JUE,VIE,SAB,DOM].

La intersección o producto de los conjuntos w y r es de la siguiente manera:

```
s:=w*r;
```

Ahora el conjunto s tiene como elementos {MIER,JUE,VIE}.

La diferencia o resta de los conjuntos w y r es como sigue:

```
s:=w-r;
```

El conjunto s tiene como elementos {LUN,MAR}.

PROBLEMAS PROPUESTOS

1. Escribir un programa en Pascal que realice la unión de los siguientes conjuntos:

   A=[0,1,2,3,4,5,6,7,8,9]

   B=[ROJO,AZUL,VERDE,AMARILLO,GRIS]

   C=[A..Z]

   y que imprima los conjuntos originales y el conjunto resultante.

2. Escribir un programa en Pascal que realice la intersección de los siguientes conjuntos:

   E=[0,1,2,3,4,5,6,7,LUN,MAR,MIER,
   JUE,VIE,SAB,DOM]

   F=[1,3,5,ENE,MAR,MAY]

3. Escribir un programa en Pascal que compare los siguientes conjuntos:

   A=[ENE,FEB,MAR]

   B=[ENE,MAR]

   C=[ENE,FEB,MAR,ABR]

   D=[ENE..ABR]

   y obtenga cuáles son iguales, cuáles son distintos y qué conjunto pertenece a otro.

4. Escribir un programa en Pascal que a partir de los conjuntos:

CJTO1=[A,B,C,D]

CJTO2=[A,B,E,F,G]

CJTO3=[B,C,E,H]

y el conjunto universo:

UNIVERSO=[A,B,C,D,E,F,G,H]

obtenga e imprima los conjuntos resultantes de las siguientes operaciones:

a) CJTO1 ∩ CJTO3

b) CJTO1 ∪ CJTO3

c) (CJTO1 ∩ CJTO2)- CJTO2

d) CJTO2' - CJTO1'

e) CJTO2' ∩ CJTO3

f) (CJTO3 - CJTO2) - CJTO1

5. Escribir un programa en Pascal que obtenga e imprima, a partir de los conjuntos:

A={X | 0 <= X <= 3}

B={X | 1 < X < 4}

C={X | 2 <= X < 5}

y el conjunto universo:

U={X | 0 <= X <= 6}

los conjuntos resultantes de las siguientes operaciones:

a) $A \cap B$

b) $(B \cap C) - A$

c) $A' - B$

d) $A' - C'$

e) $(A - B')'$

f) $(A \cup B) - C'$

g) $(A' \cap B')'$

6. Al elegir un identificador (nombre de variable) en Pascal se debe cumplir, para que sea válido, que el primer caracter del identificador sea una letra.

   Escribir un programa en Pascal que verifique, utilizando conjuntos, la condición de que el primer caracter de una cadena de caracteres dada la cumpla, que imprima la cadena leída y diga si cumple o no con la condición.

   Probar el programa con 25 cadenas de caracteres distintas.

7. Un identificador en Pascal debe tener como primer caracter una letra y los siguientes caracteres pueden ser letras o dígitos, pero no caracteres especiales ($, %, #, /, ?, &, +, -, etc.).

   Escribir un programa en Pascal que verifique, utilizando conjuntos, si una cadena de caracteres dada puede ser utilizada como un identificador, que imprima la cadena leída y diga si puede utilizarse como identificador. En caso de que no cumpla con las condiciones de los identificadores, que imprima el (los) caracter (es) no válido (s) y la (s) posición (es) en que se encuentran dentro de la cadena leída.

   Probarlo con 25 cadenas distintas.

CAPITULO 7 REGISTROS Y APUNTADORES

GENERALIDADES

Un registro, al igual que un arreglo, tiene varios componentes (campos). Mientras en un arreglo los componentes son del mismo tipo y se hace referencia a cada uno mediante un índice, en un registro, contrariamente a un arreglo, sus componentes pueden ser de diferente tipo y se hace referencia a cada uno por su nombre.

En este capítulo se introducen también los conceptos de variable dinámica y apuntador. Estos conceptos suelen ocasionar alguna dificultad inicial para entenderlos, debido a que no se toma en cuenta el hecho de que una variable dinámica no tiene un nombre mediante el cual hacer referencia a ella. La forma de hacer referencia a una variable dinámica es indirecta, mediante un apuntador.

7.1 REGISTROS

El siguiente ejemplo ilustra la definición de un registro:

```
TYPE
        datos=RECORD
                nombre:STRING[20];
                sueldo:REAL;
              END;
```

siendo RECORD una palabra reservada.

Obsérvese que para la palabra reservada END no existe su correspondiente BEGIN, debido a que los componentes (campos) del registro se encuentran entre las palabras reservadas RECORD y END.

En este ejemplo el registro tiene dos campos. El primer campo se llama nombre y es de tipo cadena (20 caracteres como máximo). El segundo componente se llama sueldo y es de tipo REAL.

Obsérvese que el registro como un todo tiene el nombre datos, siendo este nombre el de un tipo de variable. Este nuevo tipo de variable que se acaba de definir puede utilizarse en la sección VAR para declarar variables de tipo datos como sigue:

```
TYPE
        datos=RECORD
                nombre:STRING[20];
                sueldo:REAL;
              END;
VAR
        x,w,alfa:datos;
```

Para asignar información a cada campo de la variable x, por ejemplo, se procede de la siguiente manera:

```
x.nombre:='JUAN PEREZ';
x.sueldo:=8234.25;
```

Obsérvese que se hace referencia a la variable (x) y al campo (nombre y sueldo) de la variable al que se asigna la información.

La asignación:

```
w:=x;
```

asigna toda la información de x a w, es decir, el valor del campo nombre de w es JUAN PEREZ y el valor del campo sueldo de w es 8234.25.

La asignación de información a los campos de un registro durante una lectura es como sigue:

```
READ(alfa.nombre,alfa.sueldo);
```

La impresión del valor de los campos de un registro es:

```
WRITELN(alfa.nombre:25,alfa.sueldo:15);
```

donde el campo nombre se imprime en un campo de 25 columnas y el campo sueldo se imprime en un campo de 15 columnas.

7.1.1 NOTACION SIMPLIFICADA DE REGISTROS

La instrucción WITH-DO se utiliza para simplificar la notación de registros, por ejemplo la impresión:

```
WRITELN(alfa.nombre:25,alfa.sueldo:15);
```

puede simplificarse:

```
WITH alfa DO
   WRITELN(nombre:25,sueldo:15);
```

Obsérvese que entre las palabras reservadas WITH y DO va el nombre del registro al cual se hace referencia y dentro del alcance de la instrucción WITH-DO sólo se mencionan los nombres de los campos del registro alfa.

Otros ejemplos con el WITH-DO son los siguientes:

Ejemplo 1

```
WITH w DO
    READ(nombre,sueldo);
```

Ejemplo 2

```
WITH x DO
    BEGIN
    nombre:='CHUCHO SALINAS';
    sueldo:=9343.85;
    END;
```

Supóngase que se tiene la siguiente definición:

```
datos=RECORD
       nombre:STRING[20];
       sueldo:REAL;
       fechain:fecha;
       END;
```

donde fechain (fecha de ingreso) es de tipo fecha; siendo fecha a su vez otro registro, es decir:

```
                    *
                    *
                    *
TYPE
      fecha=RECORD
             dia:1..31;
             mes:1..12;
             ano:INTEGER;
            END;
      datos=RECORD
             nombre:STRING[20];
             sueldo:REAL;
             fechain:fecha;
            END;
VAR
      alfa,omega,w,x:datos;
                    *
                    *
                    *
```

La asignación del día en el registro omega es como sigue:

```
omega.fechain.dia:=27;
```

Obsérvese que la parte:

```
omega.fechain
```

hace referencia al campo fechain del registro alfa y la porción:

```
dia
```

hace referencia al campo dia del campo fechain, que a su vez es otro registro.

El ejemplo siguiente muestra la asignación de la fecha de ingreso en la lectura del registro alfa.

```
READLN(alfa.fechain.dia,alfa.fechain.mes,
       alfa.fechain.ano);
```

La representación de la lectura anterior puede hacerse más clara utilizando el WITH-DO.

```
WITH alfa DO
    WITH fechain DO
        READLN(dia,mes,ano);
```

Obsérvese que se utilizaron dos WITH-DO's debido a que alfa y fechain son registros.

7.2 APUNTADORES

Las variables que se han tratado hasta ahora son variables estáticas; es decir, estas variables existen durante toda la ejecución del programa.

El valor de una variable estática puede cambiar durante el tiempo de ejecución; pero estas variables no pueden ser creadas o destruidas durante este tiempo. Una variable estática se crea en tiempo de compilación y se destruye al finalizar la ejecución del programa.

Una variable dinámica en Pascal, contrariamente a una estática, se crea y se destruye en tiempo de ejecución. Debido a esto no existe un identificador mediante el cual hacer referencia directa a las variables dinámicas; en otras palabras, una variable dinámica no se declara.

Sin embargo, puede hacerse referencia, en forma indirecta, a una variable dinámica mediante las variables estáticas conocidas como apuntadores.

El apuntador se utilizará para hacer referencia indirecta a una variable dinámica de tipo entero. La siguiente declaración es un ejemplo de cómo declarar un apuntador:

```
VAR
    p:@INTEGER;
```

Obsérvese que se antepone el caracter @ al tipo, en este caso entero. La declaración quiere decir que p es un apuntador que va a 'apuntar', 'señalar' o 'hacer referencia' a una variable dinámica de tipo entera.

La notación:

```
p@
```

denota a la variable dinámica 'señalada' por p.

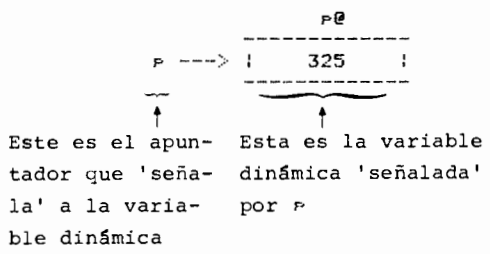
Se puede suponer que p@ es el nombre de la variable dinámica en un tiempo determinado, es decir, en el tiempo en que es 'señalada' por p.

La asignación:

```
p@:=325;
```

asigna el valor entero 325 a la variable dinámica entera 'señalada' por el apuntador p.

Por conveniencia se designa con una flecha a un apuntador y a la variable dinámica señalada por p con un rectángulo, por ejemplo:



Supóngase que se tiene la siguiente declaración:

```
p,q,r,s:@INTEGER;
```

La creación en el tiempo de ejecución de una variable dinámica entera, utilizando cualquiera de los cuatro apuntadores arriba declarados, se hace mediante el siguiente procedimiento predefinido en Pascal.

El diagrama de la izquierda muestra el efecto del procedimiento NEW al ejecutarse, es decir crea una variable dinámica 'apuntada' por P.

El efecto inverso de NEW, es decir destruir, lo hace el procedimiento predefinido DISPOSE.

Por ejemplo:

```
(* SE CREA UNA VARIABLE DINAMICA *)
```


```
(* SE DESTRUYE LA VARIABLE DINAMICA *)

   DISPOSE(P)      P
```

El procedimiento DISPOSE destruye a la variable dinámica no así a P por ser una variable estática. Cuando DISPOSE destruye a la variable dinámica 'apuntada' por P, le asigna a P el valor nulo, es decir, P no apunta a variable dinámica alguna.

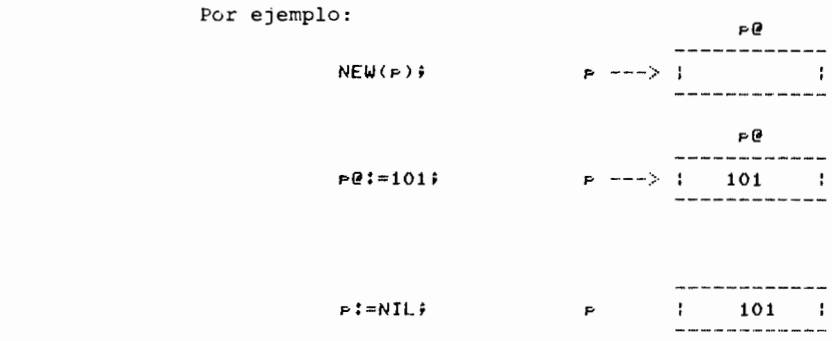
La asignación:

```
P:=NIL;
```

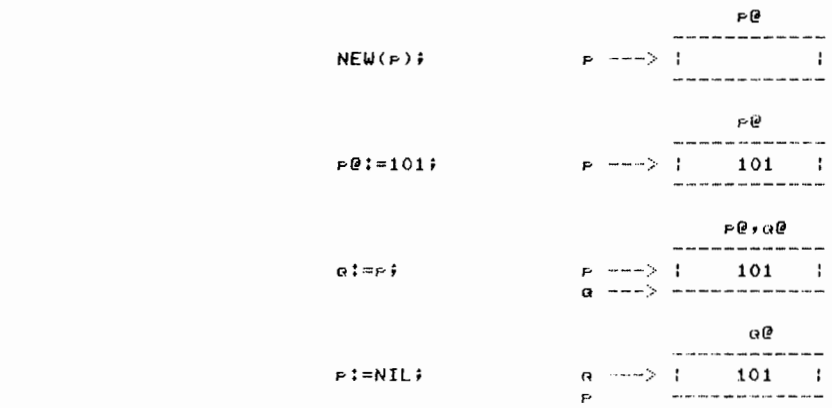
asigna el valor nulo a P, siendo NIL una palabra reservada en Pascal.

NIL debe usarse con cuidado ya que si se asigna su valor a un apuntador que 'señala' a una variable dinámica, ésta se pierde y ya no se puede hacer referencia a ella.

Por ejemplo:



Para evitar esto se procede como sigue:



Obsérvese que en la asignación Q:=P tanto P y Q señalan a la variable dinámica entera y se puede hacer referencia a ésta mediante P o Q.

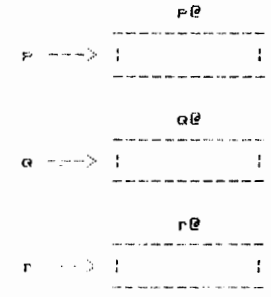
La siguiente secuencia de instrucciones corresponde a un segmento de programa que suma dos valores utilizando variables dinámicas. Cada instrucción va acompañada de un diagrama que ilustra el efecto de cada instrucción, así como de un comentario.

```
(* SE LEEN DOS VALORES ENTEROS EN LAS VARIABLES
   ESTATICAS ENTERAS a Y b                          *)

        READLN(a,b);
```

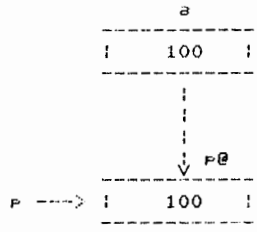

```
(* SE CREAN 3 VARIABLES DINAMICAS QUE SON
   APUNTADAS POR p,q Y r                            *)

        NEW(p);
        NEW(q);
        NEW(r);
```



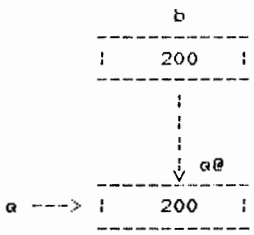

```
(* SE ASIGNA EL VALOR DE a A LA VARIABLE DINAMICA
   'SEÑALADA' POR p                                 *)

        p@:=a;
```



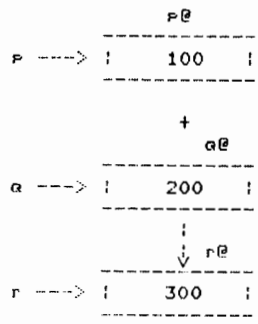

```
(* SE ASIGNA EL VALOR DE b A LA VARIABLE DINAMICA
   'SEÑALADA' POR q                                 *)

        q@:=b;
```



```
(* SE SUMAN LOS VALORES DE LAS VARIABLES
   DINAMICAS SEÑALADAS POR P Y Q. EL RESULTADO
   SE ASIGNA A LA VARIABLE DINAMICA SEÑALADA
   POR R                                          *)

    R@:=P@+Q@;
```




```
(* SE IMPRIMEN LOS VALORES DE LAS VARIABLES
   DINAMICAS APUNTADAS POR P,Q Y R          *)

    WRITELN(P@,'+',Q@,'=',R@);
```

```
100 + 200 = 300
```

```
(* SE DESTRUYEN LAS VARIABLES DINAMICAS APUNTADAS POR
   P,Q Y R                                        *)

        DISPOSE(P);          P
        DISPOSE(Q);          Q
        DISPOSE(R);          R
END.
```

En este ejemplo simple se tienen cinco variables estáticas: los apuntadores P,Q y R y los enteros a y b. Estas variables se crean en el tiempo de compilación y se destruyen al terminar la ejecución del programa. Las variables dinámicas que se crearon y destruyeron durante la ejecución del programa son P@, Q@ y R@, es decir, las señaladas por P, Q y R respectivamente.

Lo siguiente es un listado completo del ejemplo anterior:

```
PROGRAM apunta;

VAR
   p,q,r:@INTEGER;
   a,b  :INTEGER;
BEGIN
READ(a,b);
NEW(p);
NEW(q);
NEW(r);
p@:=a;
q@:=b;
r@:=p@+q@;
WRITELN(p@,'+',q@,'=',r@);
DISPOSE(q);
DISPOSE(P);
DISPOSE(r);
END.
```

Usualmente los apuntadores se utilizan para 'apuntar' o 'señalar' a variables dinámicas que son registros, en vez de a variables dinámicas simples como en los ejemplos anteriores.

Por ejemplo, si se define el registro estudiante en la sección TYPE y en la sección VAR se definen los apuntadores que 'apunten' a variables dinámicas de tipo estudiante:

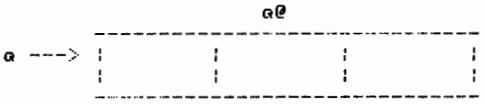
```
                    *
                    *
                    *
TYPE
    estudiante:=RECORD
                    nombre:STRING[20];
                    semestre:1..10;
                    siguiente:@estudiante;
                  END;
VAR
    p,q,r:@estudiante;
                    *
                    *
                    *
```

se puede generar una lista dinámica de estudiantes ya que el campo siguiente del registro es un apuntador que 'señala' a la siguiente variable dinámica (registro) que tiene los datos del estudiante que sigue en la lista.

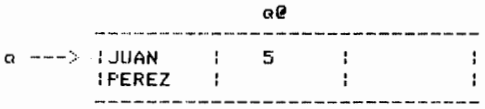
A continuación se detallan, mediante segmentos de código y sus diagramas correspondientes, operaciones que son comunes con registros dinámicos.

a) Asignación de datos a un registro dinámico durante una lectura:

```
1.- NEW(a);
```



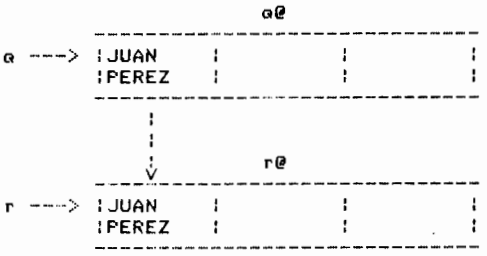

```
2.- READ(a@.nombre,a@.semestre);
```



b) Copiar el contenido del campo de un registro en el campo de otro registro:

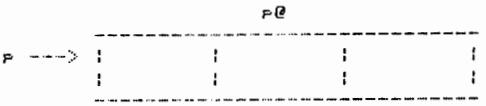
```
1.- NEW (r);
```


```
2.- r@.nombre:=a@.nombre;
```

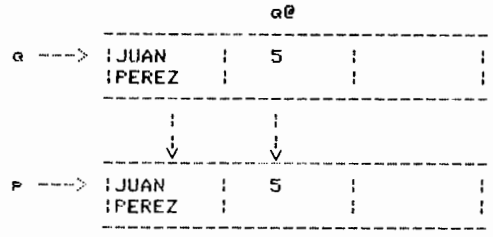


c) Copiar el contenido de un registro en otro:

```
1.- NEW(p);
```

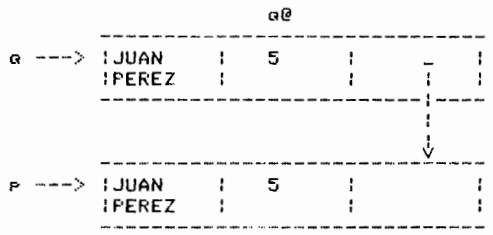


2.- p@:=q@;



d) 'Ligar' dos registros:

1.- q@.siguiente:=p;



e) Hacer nulo el campo siguiente de un registro:

1.- r@.siguiente:=NIL;

NOTA: La diagonal en el campo siguiente significa nulo.

El ejemplo siguiente genera una lista dinámica de estudiantes. Se sugiere al lector seguir paso a paso (prueba de escritorio) el código del programa ayudándose con los diagramas correspondientes a cada instrucción.

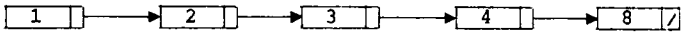
110

```
PROGRAM ejemplo;

TYPE
     estudiante;=RECORD;
                  nombre:STRING[20];
                  semestre:INTEGER;
                  siguiente:@estudiante;
                END;
VAR
     p,q,r:@estudiante;
     lonlista:INTEGER;
BEGIN
READLN(lonlista);
NEW(q);
READ(q@.nombre,q@.semestre);
FOR i:=2 TO lonlista DO
    BEGIN
    NEW(p);
    q@.siguiente:=p;
    READ(q@.nombre,q@.semestre);
    p:=q;
    END;
END.
```

PROBLEMAS PROPUESTOS

1. Escribir un programa que cree la siguiente lista:



2. Escribir un programa que ordene 25 dígitos de menor a mayor, utilizando registros y apuntadores.

3. Escribir un programa que ordene alfabéticamente 25 nombres distintos, utilizando registros y apuntadores.

4. Escribir un programa que suprima un registro, que contenga un número cualquiera de la lista creada en el problema 2, que imprima la lista original y la lista resultante después de la supresión del registro.

5. Escribir un programa que utilizando listas ligadas, reste dos números enteros de N dígitos, que imprima los números que se desea restar y el resultado. Probarlo con dos números enteros de 25 dígitos cada uno.

6. Escribir un programa que utilizando listas ligadas, sume dos números enteros de N dígitos cada uno, que imprima los números que se desea sumar y el resultado. Probarlo con dos números enteros de 25 dígitos cada uno.

CAPITULO 8 ARCHIVOS

GENERALIDADES

Los archivos permiten almacenar grandes cantidades de datos en disco, evitando con esto volver a dar datos a un programa cada vez que se ejecute. Los archivos pueden ser manipulados desde diferentes programas con diferentes propósitos.

Pascal sólo permite archivos de acceso secuencial. Algunos compiladores en Pascal permiten además el uso de archivos de acceso directo, esto tiene que verificarlo el lector interesado con el sistema que esté a su disposición.

La estructura de un archivo es semejante a la de un arreglo unidimensional. En un arreglo el índice de un elemento puede ser un entero negativo, cero o un entero positivo. En un archivo el valor del índice del primer elemento es siempre cero y el valor del último es N-1, siendo N el número de elementos o registros de que consta el archivo. El elemento o registro de un archivo puede ser una variable simple (entera, real, etc.) o una variable estructurada (registro).

La dimensión de un arreglo permanece sin cambio a lo largo de la ejecución del programa, mientras que un archivo puede sólo crecer (agregar registros) a lo largo de la ejecución del programa.

Las palabras reservadas FILE OF son usadas para declarar un archivo.

Por ejemplo:

```
VAR
    datos:FILE OF INTEGER;
```

siendo datos de un archivo cuyos elementos son enteros.

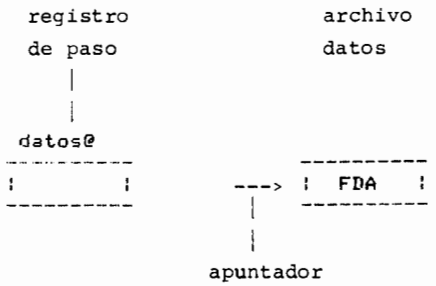
8.1 INICIACION DE UN ARCHIVO

Para iniciar un archivo se hace uso de la instrucción REWRITE como sigue:

```
REWRITE(datos);
```

Todo archivo en Pascal tiene asociado un apuntador, que es el equivalente al índice en un arreglo, y un registro de paso (BUFFER).

La ejecución de la instrucción anterior queda representada diagramáticamente como sigue:



Las siglas FDA significan fin de archivo. El apuntador en este momento apunta al fin de archivo (no hay datos).

Obsérvese que datos@ es el nombre del registro de paso asociado con el archivo de datos.

8.2 ESCRITURA EN UN ARCHIVO

Supóngase que alfa es una variable entera, a la cual se le asigna un valor entero.

Por ejemplo:

```
alfa:=101;
```

Las siguientes dos instrucciones:

```
datos@:=alfa;
PUT(datos);
```

son equivalentes a la sola instrucción:

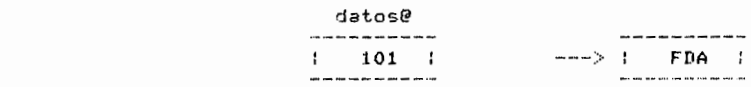
```
WRITE(datos,alfa);
```

es decir en el archivo datos se escribe el valor de la variable alfa.

La ejecución de la instrucción:

```
datos@:=alfa;
```

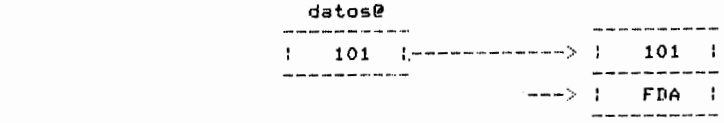
ocasiona que el programa asigne el contenido de la variable alfa al registro de paso datos@ asociado con el archivo datos, como lo muestra el diagrama siguiente:



La ejecución de la instrucción:

```
PUT(datos);
```

siendo PUT (poner) una palabra reservada, que ocasiona que el contenido del registro de paso se asigne al archivo:



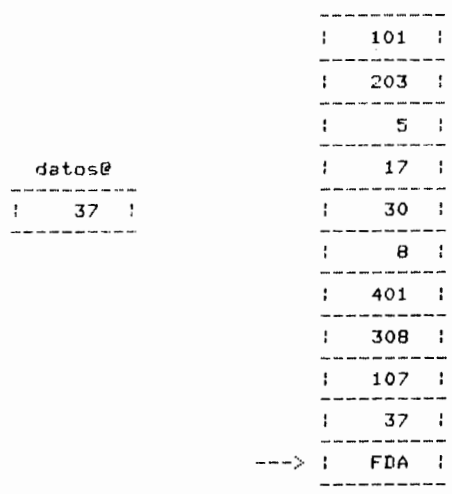
Si se asigna otro entero al archivo, éste quedará al final del archivo, es decir, después del 101.

Por ejemplo:

```
alfa:=281;
WRITE(datos,alfa);
```

El siguiente segmento de programa inicia y escribe 10 enteros en el archivo datos:

```
REWRITE(datos);
FOR i:=1 TO 10 DO
    BEGIN
    READ(alfa);
    WRITE(datos,alfa);
    END;
```



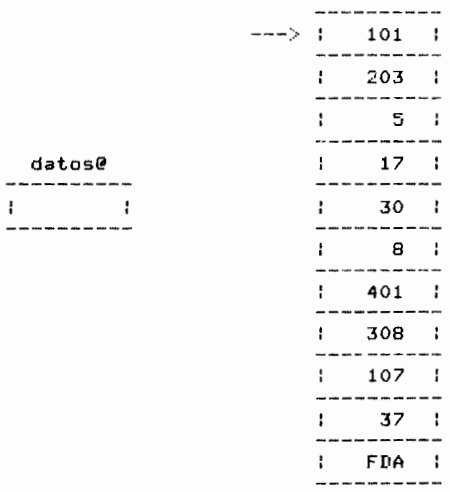
8.3 LECTURA DE UN ARCHIVO

Para iniciar la lectura de un archivo, el apuntador debe ubicarse en el primer elemento o registro del archivo. Esto se hace con la instrucción RESET.

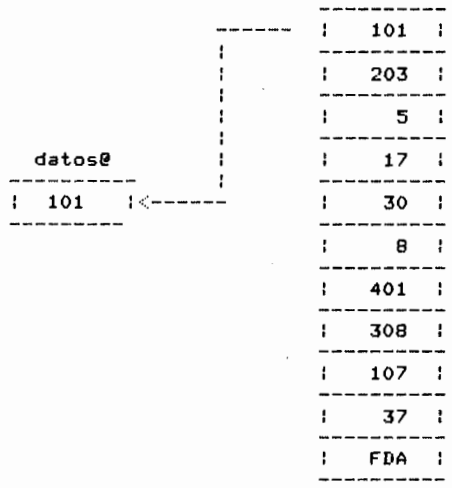
Por ejemplo:

```
RESET(datos);
```

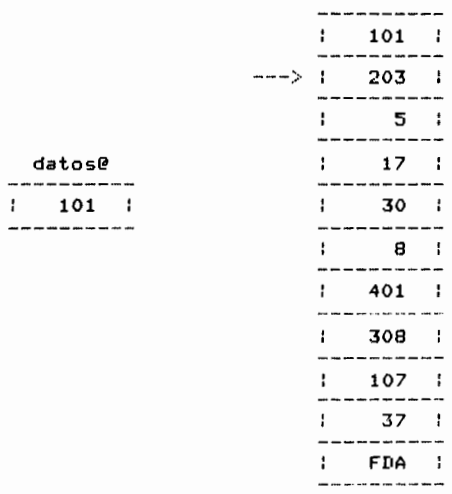
Esta instrucción tiene los siguientes efectos: primero el apuntador se ubica al inicio del archivo, es decir, en el primer registro:



Luego, el contenido del primer registro del archivo se asigna al registro de paso:



Finalmente el apuntador se ubica en el segundo registro del archivo:



La siguiente secuencia de dos instrucciones:

```
alfa:=datos@;
GET(datos);
```

es equivalente a la sola instrucción:

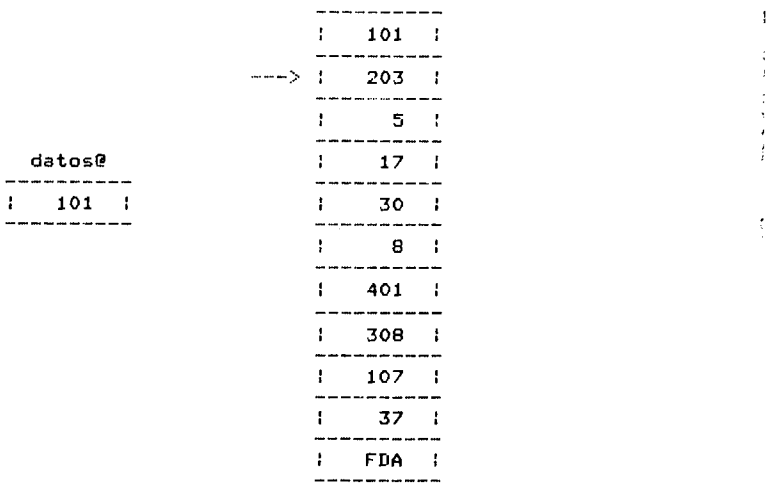
```
READ(datos,alfa);
```

Es decir, se lee del archivo datos un entero y se asigna a la variable alfa.

La ejecución de la instrucción:

```
alfa:=datos@;
```

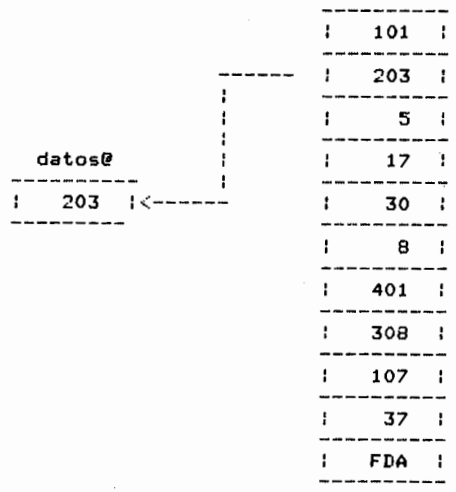
ocasiona que el contenido del registro de paso se asigne a la variable alfa del programa, como lo muestra el siguiente diagrama:



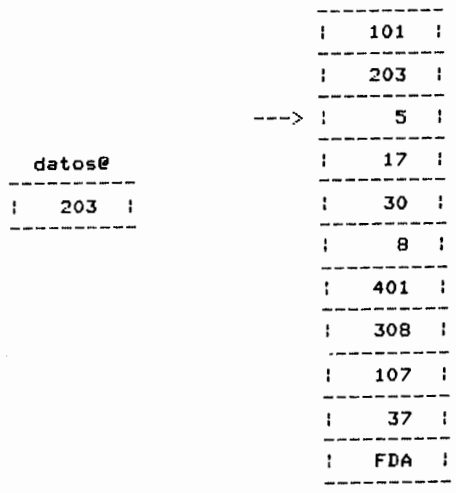
La ejecución de la siguiente instrucción:

```
GET(datos);
```

ocasiona que el contenido del elemento o registro del archivo, que es señalado por el apuntador, se asigne al registro de paso; siendo GET (obtener) una palabra reservada.



Luego el apuntador del archivo se ubica en el siguiente registro:



El siguiente segmento de programa lee del archivo datos e imprime los valores leídos:

```
RESET(datos);
WHILE(NOT(EOF(datos))) DO
    BEGIN
    READ(datos,alfa);
    WRITELN(alfa);
    END;
```

La palabra reservada EOF (End Of File) en el predicado del WHILE-DO significa fin de archivo, EOF es una bandera booleana que es falsa en tanto no se alcance el fin de archivo (FDA). Obsérvese que EOF tiene como argumento el nombre del archivo de datos.

El siguiente ejemplo es un listado de un programa que lee y escribe en un archivo, (el archivo fue previamente creado al programa). Se imprimen aquellos registros cuyo sueldo sea menor o igual al salario mínimo (saminimo).

Obsérvese que nomina es un archivo cuyos registros son de tipo empleado e item es un registro de tipo empleado.

```
PROGRAM ejemplo;
CONST
      saminimo=1000;

TYPE
      empleado= RECORD
                 nombre:STRING[20];
                 sueldo:REAL;
                END;

VAR
      nomina:FILE OF empleado;
      item:empleado;


BEGIN
RESET(nomina);
WHILE (NOT(EOF(nomina))) DO
    BEGIN
    READ(nomina,item);
    IF (item.sueldo<=saminimo) THEN
        WRITELN(item.nombre:20,item.sueldo:15);
    END;
END.
```

PROBLEMAS PROPUESTOS

1. Escribir un programa en Pascal que obtenga los primeros diez números de la serie de Fibonacci y los almacene en un archivo y que lea e imprima el archivo resultante.

2. Escribir un programa en Pascal que inicie un archivo en el que cada registro debe tener el nombre de una persona, su dirección y su teléfono. Leer y escribir el contenido del archivo. Probarlo con 25 registros distintos.

3. Un alumno quiere llevar un registro personal de las materias del plan de estudios de su carrera que ha cursado, las acreditadas y la calificación obtenida de cada una de las acreditadas.

   Escribir un programa que cree un archivo de las materias del plan de estudios de la carrera de Ingeniería en Computación y que obtenga el promedio global de un alumno en particular, cada registro debe tener el siguiente formato:

MATERIA	CURSADA (SI/NO)	APROBADA (SI/NO)	CALIFICACION

   Los datos de CURSADA, APROBADA y CALIFICACION son de un alumno en particular.

4. Una persona desea tener un registro de los libros que tiene en su casa. De cada libro le interesa saber cuál es el autor, el título, la materia y la fecha de edición.

   Hacer un programa que cree y escriba un archivo para guardar la información de los libros, cada registro del archivo debe tener los siguientes campos:

AUTOR	TITULO	MATERIA	EDICION

Probar el programa con 25 conjuntos de datos distintos.

5. Se desea llevar un registro de las calificaciones y el promedio de los alumnos de un grupo de Programación Estructurada en el que se aplicaron tres exámenes.

   Escribir un programa que registre el nombre y las calificaciones de cada uno de los alumnos. Originalmente el registro de cada uno de ellos contendrá únicamente el nombre, y después de aplicar un examen se registrará la calificación obtenida y se obtendrá el promedio. El registro de cada alumno debe tener el siguiente formato:

NOMBRE	CALIF 1	CALIF 2	CALIF 3	PROMEDIO

   Probar el programa con una lista de 15 alumnos.

PARTE III ELEMENTOS DEL ANALISIS Y DISEÑO ESTRUCTURADO

CAPITULO 9 ANALISIS ESTRUCTURADO

GENERALIDADES

El análisis estructurado tiene como propósito fundamental especificar, en la forma más precisa posible, los requerimientos del usuario para un programa o conjunto de programas.

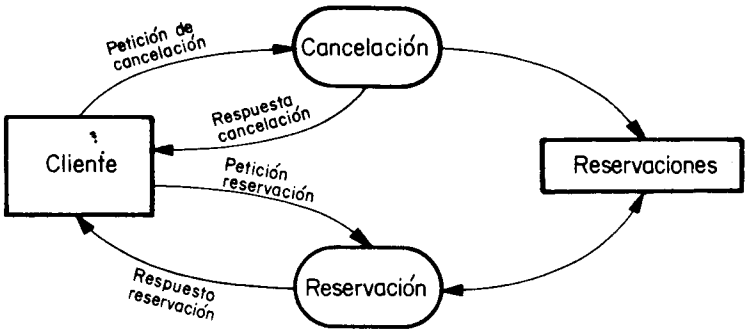
El diagrama de flujo de datos (DFD) es la principal herramienta gráfica del análisis estructurado y tiene como objeto mostrar las transformaciones de los datos a medida que éstos fluyen a través de los procesos del programa; es decir, ayuda a analizar los cambios que ocurren a los datos de entrada a fin de lograr la salida deseada.

El diagrama de flujo de datos, debido a su sencillez y a que es una herramienta gráfica, resulta comprensible tanto para el usuario como para el analista del sistema.

9.1 EL DIAGRAMA DE FLUJO DE DATOS

Un diagrama de flujo de datos es un instrumento de modelación que permite mostrar a un sistema como una red de subsistemas conectados unos a otros mediante flujos de datos que muestren las relaciones entre subsistemas.

Un ejemplo simple de un diagrama de flujo de datos es el siguiente:



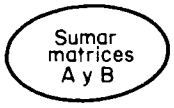
Este diagrama muestra un sistema de reservación y cancelación de una posible aerolínea o un hotel, y además muestra los cuatro elementos que comúnmente aparecen en un diagrama de este tipo.

9.2 LOS ELEMENTOS DE UN DIAGRAMA DE FLUJO DE DATOS

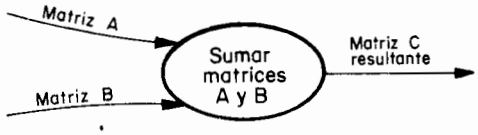
Un diagrama de flujo de datos consta de los siguientes elementos:

1. Un círculo, con un nombre inscrito, para indicar un proceso. El nombre indica la función del proceso, el cual actúa sobre los datos para transformarlos o generar nueva información.

   Un ejemplo de esto último consiste en la validación de algún dato; este proceso no modifica el dato pero genera nueva información sobre el mismo: si es válido o inválido.



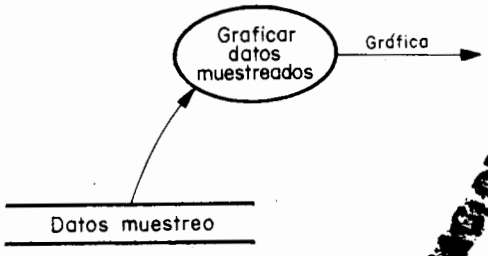
2. Una flecha, con un nombre asociado, para indicar la entrada y salida de datos a un proceso. La dirección de la flecha indica la dirección del flujo de datos.



3. Dos líneas paralelas, con un nombre entre ellas para indicar un contenedor de datos; es decir, un archivo en disco o en cinta, o un archivo de tarjetas, etcétera.

   Se muestra un archivo cuando se tiene acceso más de una vez a una pieza de información (mismo dato) y cuando la información se utiliza en un orden diferente al que fue registrada.

   Un archivo con una flecha saliente indica que se extrae información de él. Con una flecha entrante se indica escritura de información.



4. Un rectángulo que indica dónde se origina o se destina la información (sentido de la flecha). Un mismo rectángulo puede ser fuente o destino. El rectángulo tiene un nombre que identifica la fuente o el destino.

Obsérvese que con el diagrama de flujo de datos se obtiene una estructura del tipo entrada-proceso-salida; es decir, datos de entrada a un proceso que actúa sobre los mismos y datos de salida transformados.

9.3 CARACTERISTICAS DEL DIAGRAMA DE FLUJO DE DATOS

Las características del diagrama de flujo de datos son las siguientes:

1. *Es gráfico.*

   La ventaja de una herramienta gráfica consiste en su impacto visual; es decir, que de un vistazo se perciben rápidamente las funciones principales del sistema.

   El DFD debe usarse de una manera concisa para evitar que el usuario pierda interés en él..

2. *Es modular.*

   Esto significa que el DFD muestra la partición de un sistema en funciones tan independientes entre sí como sea posible, lo cual permite tanto al usuario como al diseñador revisar cada función del sistema de una manera aislada.

3. *Enfatiza el flujo de datos.*

   El DFD muestra solamente el flujo de datos que se transforman a medida que pasan a los procesos (funciones) desde la entrada a la salida.

4. *Desenfatiza el flujo de control.*

   El DFD no muestra información de control (banderas), ni secuencia de acciones en el tiempo.

5. *Es modificable.*

   Esto significa que se pueden reconsiderar algunas partes del DFD con las cuales no se haya quedado satisfecho y volver a trabajarlas.

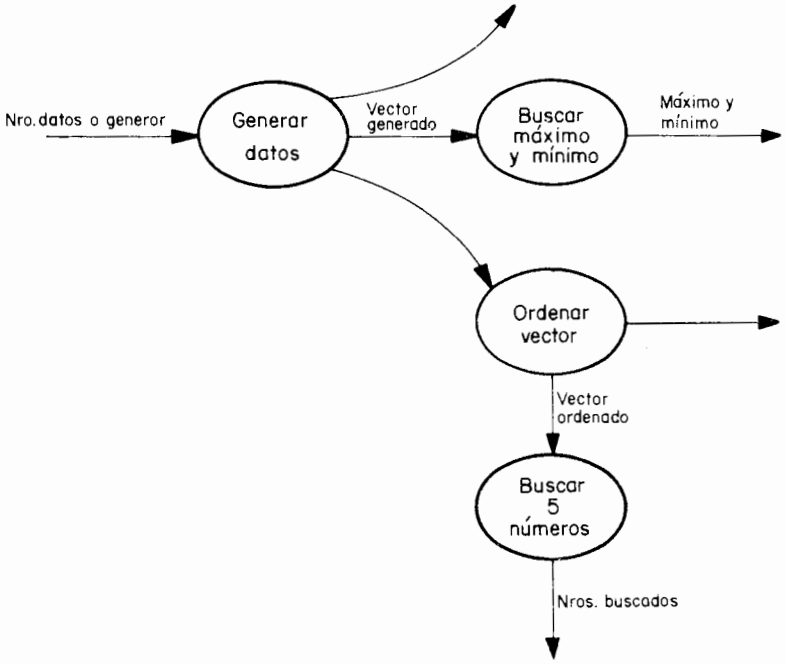
6. *No es redundante.*

Esto quiere decir que una función debe registrarse sólo una vez para que el sistema, al cual dará origen el DFD, sea consistente y de fácil actualización.

9.4 EJEMPLOS DE DIAGRAMAS DE FLUJO DE DATOS

La construcción del diagrama de flujo de datos se hace teniendo en cuenta el aspecto funcional del problema, así como la secuencia que se da entre estas funciones.

Considérese el siguiente DFD:

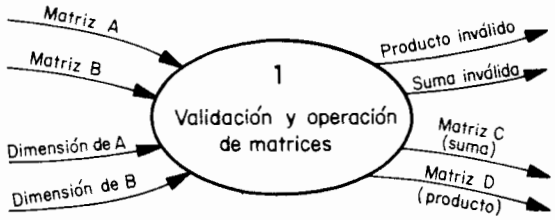


El propósito del ejemplo se ve claro en el diagrama.

Considérese ahora la validación de la suma y producto de dos matrices. Si las matrices A y B son compatibles para la suma, se efectúa esta operación; si no, se emite el mensaje: suma inválida.

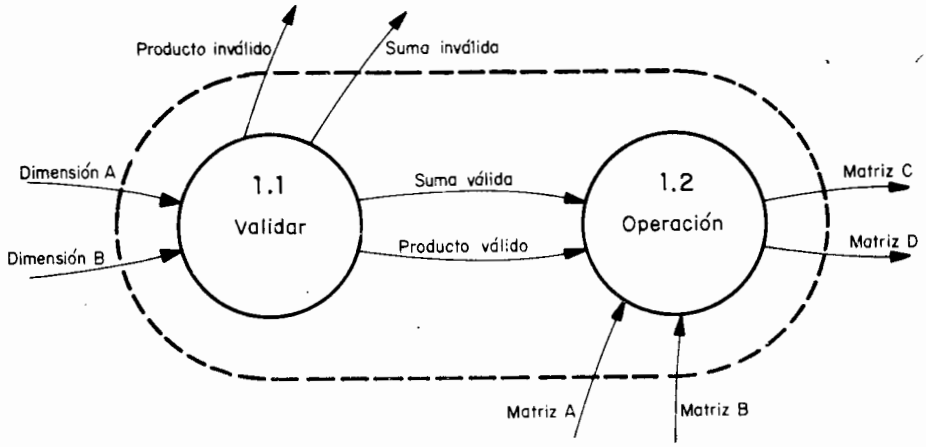
Si son compatibles para el producto, se multiplican; si no, se emite el mensaje: producto inválido.

El problema puede establecerse inicialmente como sigue:



Obsérvese que este DFD inicial sólo muestra entradas, proceso y salidas de acuerdo a la especificación del problema.

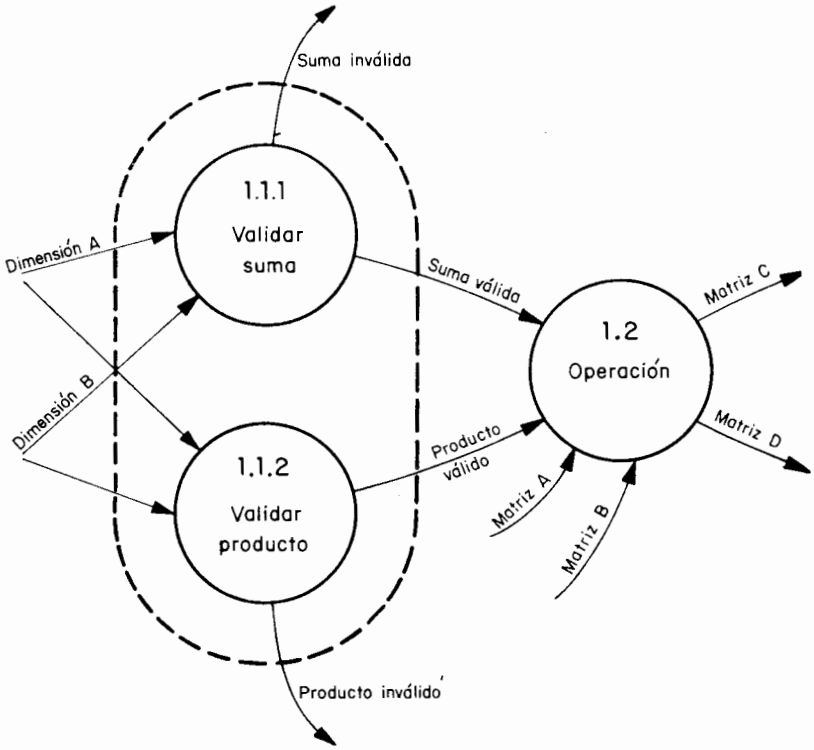
Se puede derivar un segundo DFD que muestre la división del proceso como sigue:



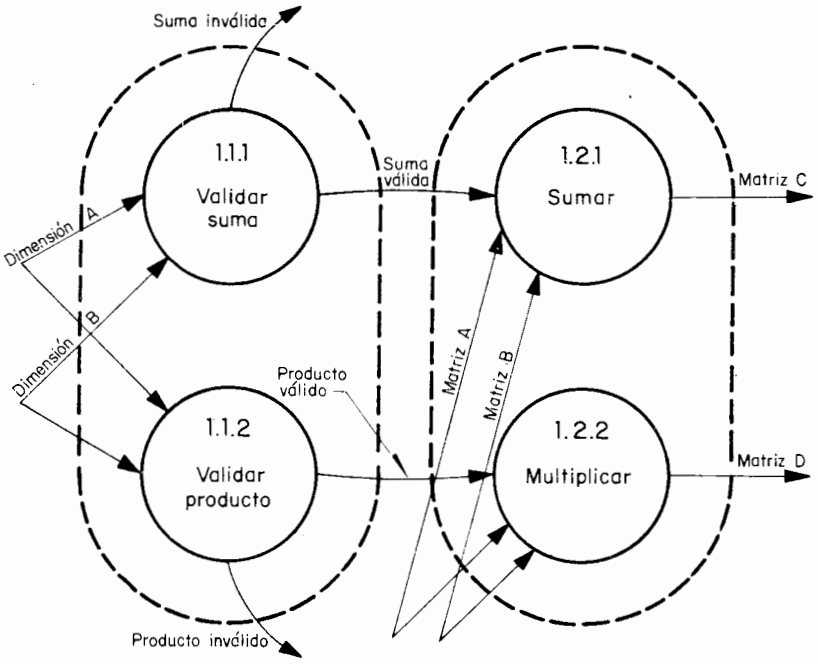
Nótese que se ha subdividido el proceso inicial en dos subprocesos, donde cada uno tiene una función muy específica. Además obsérvese que los subprocesos se han numerado para indicar de qué proceso se han obtenido.

Por otra parte, nótese que los flujos de datos de entrada y salida son los mismos (a través de la línea punteada) que en el DFD anterior, es decir, hay consistencia en los flujos de datos.

Considérese ahora el subproceso validar con numeración 1.1 que a su vez puede subdividirse como sigue:

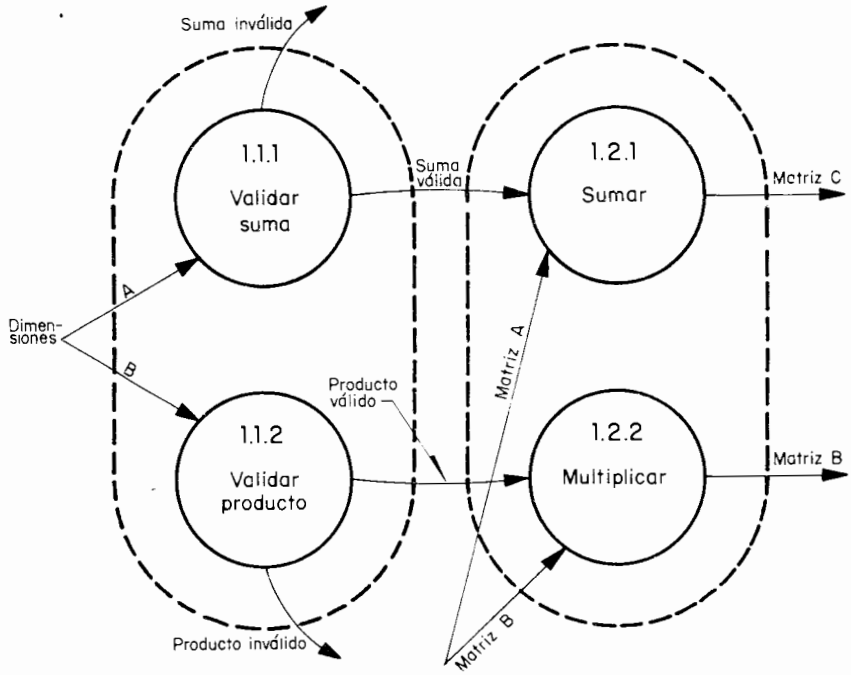


Por último, considérese el subproceso con numeración 1.2 que también puede subdividirse de la siguiente manera:



Esta forma de proceder de las funciones mayores a las funciones más básicas en el DFD es una característica que lo convierte en un instrumento de gran flexibilidad en el modelado de sistemas más complejos.

Una forma más simplificada en cuanto a los flujos de datos es la siguiente:


Suma inválida
1.1.1
Validar
suma
Suma
válida
1.2.1
Sumar
Matriz C
Dimen-
siones
A
B
Producto
válido
Matriz A
1.1.2
Validar
producto
1.2.2
Multiplicar
Matriz B
Producto inválido
Matriz B

1. Describir el proceso que se realiza para consultar la colocación de un libro de cualquier materia, autor y título, utilizando un DFD.

2. Un médico desea hacer un sistema manual o automatizado, que le permita programar las consultas; hacer una evaluación semanal de sus ingresos y calcular los impuestos sobre ellos; formar un directorio de pacientes donde tenga el nombre cada uno de ellos, dirección, teléfono, edad, la fecha de la última consulta, el número de veces que ha estado en consulta y una descripción breve de su enfermedad (no más de 100 letras).

   Las consultas se concertan por teléfono y en ellas el médico quiere tener toda la información respecto al paciente.

   Dibujar el DFD del sistema que necesita el médico.

3. Una empresa desea tener un sistema de inventario manual o automatizado, que le permita programar sus compras con base en la utilización de un producto durante los últimos seis meses. Cada ficha de artículo debe tener: descripción, clave, costo, clave de proveedor y existencia actual. Además, se debe tener un registro de cuáles son los proveedores y qué artículos proporcionan para poder realizar las compras de determinado artículo.

   Dibujar el DFD del sistema anterior.

CAPITULO 10 DISEÑO ESTRUCTURADO

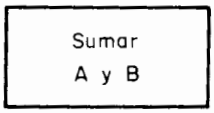
GENERALIDADES

La herramienta principal del diseño estructurado es la carta de estructura, la cual muestra la partición del sistema en módulos y la relación jerárquica entre éstos. Además muestra los flujos de datos y control entre los módulos.

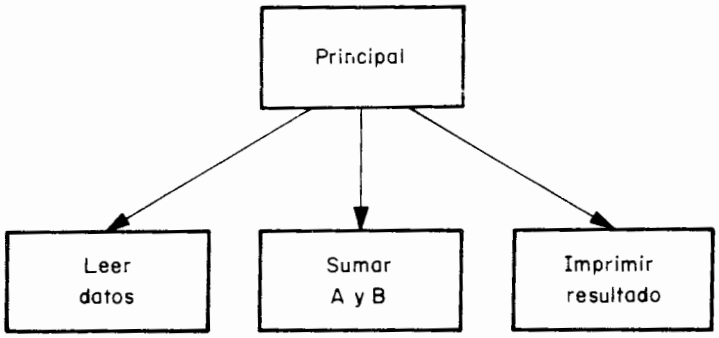
10.1 ELEMENTOS DE UNA CARTA DE ESTRUCTURA

Una carta de estructura cuenta con los siguientes elementos:

1. Un rectángulo con un nombre inscrito para indicar un módulo. El nombre indica la función del mismo.

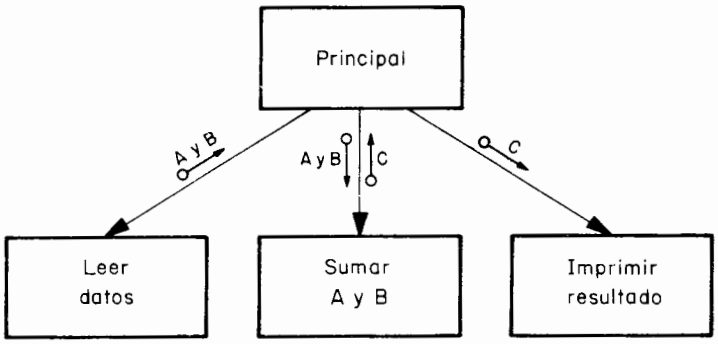


2. Líneas que indican la liga entre módulos (llamadas a módulos).

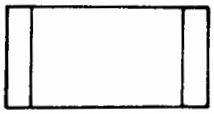


3. Flechas que indican el flujo de datos y de control respectivamente (comunicación entre módulos).

   Un ejemplo de flujo de control lo constituye una bandera.



4. Un módulo representado como:



   significa que es un módulo predefinido.

   Un ejemplo de un módulo predefinido lo constituyen los subprogramas de biblioteca.

5. El nombre del módulo debe resumir los nombres de sus subordinados inmediatos o resumir su función y las funciones de sus subordinados inmediatos.

10.2 ATRIBUTOS BASICOS DE LOS MODULOS

Un módulo tiene cuatro atributos básicos:

1. Entrada: Los datos que le pasa quien lo invoca.

Salida: Los datos que regresa a quien lo invoca.

2. Función: Lo que hace a sus datos de entrada para producir sus datos de salida.

3. Mecánica: Cómo realiza su función, es decir, su lógica.

4. Datos internos: Su propio espacio de trabajo, es decir, las variables locales.

Son ejemplos de módulos en PASCAL:

PROCEDURE, FUNCTION

En FORTRAN:

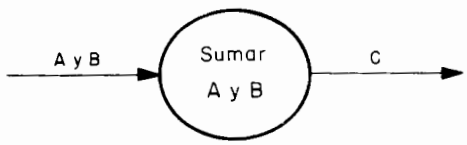
SUBROUTINE, FUNCTION

En COBOL:

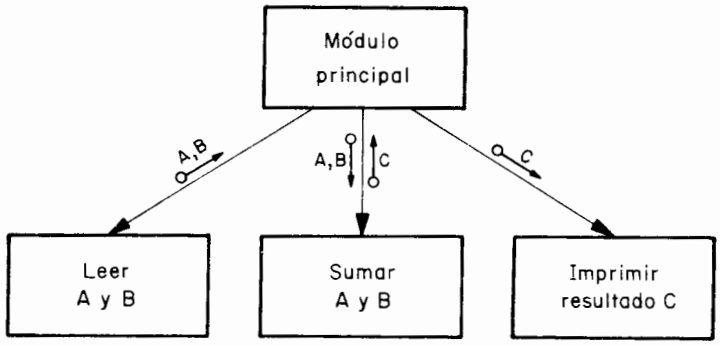
PROGRAM, SUBPROGRAM, SECTION, PARAGRAPH

10.3 EJEMPLO DE UN DISEÑO

La carta de estructura se deriva del diagrama de flujo de datos. Por ejemplo, considérese el siguiente diagrama de flujo de datos:



Su correspondiente carta de estructura es la siguiente:



Como regla general, una carta de estructura muestra a su izquierda los módulos de entrada, al centro los módulos que procesan la información y al lado derecho los módulos de salida.

A la entrada (A y B) en el diagrama de flujo de datos, le corresponde el módulo "LEER DATOS" en la carta de estructura. Obsérvese que este módulo tiene como salida A y B.

Al proceso "sumar A y B" en el diagrama de flujo de datos, le corresponde el módulo "SUMAR A y B" en la carta de estructura. Nótese que este módulo tiene como entrada A y B y como salida C.

A la salida en el diagrama de flujo de datos (C) le corresponde el módulo "IMPRIMIR RESULTADO". Obsérvese que este módulo tiene como entrada a C.

10.4 CARACTERISTICAS DE LA CARTA DE ESTRUCTURA

Una carta de estructura muestra:

1. La partición del programa, es decir, los módulos de que consta.

2. La estructura jerárquica, es decir, la relación entre módulos.

3. Los nombres de módulos y por consiguiente su función.

4. El grado de acoplamiento entre módulos.

5. Flujo de datos entre módulos.

6. Las decisiones e iteraciones que involucran la llamada a un módulo.

Una carta de estructura no muestra:

1. El número de veces que se llama un módulo.

2. La secuencia en que se llaman los módulos.

3. Cómo realiza su función.

4. Datos internos al módulo.

10.5 ACOPLAMIENTO Y COHESION

Acoplamiento es la medida del grado de la interdependencia entre dos módulos.

A continuación se describen tres tipos de acoplamiento. El orden en que aparecen corresponde al grado de calidad del tipo de acoplamiento, es decir se presentan del mejor al peor.

Acoplamiento por datos.- Dos módulos están acoplados por datos si se comunican por datos que no sean banderas, ni arreglos ni registros.

Acoplamiento por estampilla.- Dos módulos están acoplados por estampilla si se comunican mediante registros o arreglos.

Acoplamiento por control.- Dos módulos están acoplados por control si se comunican al menos por bandera.

Obsérvese la carta de estructura del punto tres del inciso 10.1, en la cual existe acoplamiento por estampilla.

Cohesión es la medida del grado de asociación de los elementos dentro de un módulo.

Un elemento puede ser:

a) Una instrucción.

Por ejemplo:

```
a:=b*c+0.5/d;
```

b) Un grupo de instrucciones.

Por ejemplo:

```
FOR  i:=1 TO n DO

   FOR  k:=1 TO n DO

        a[i,k] := b[i,k] +c[i,k];
```

c) Una llamada a otro módulo.

Por ejemplo:

```
sumar (a,b,c,ib,kb);
```

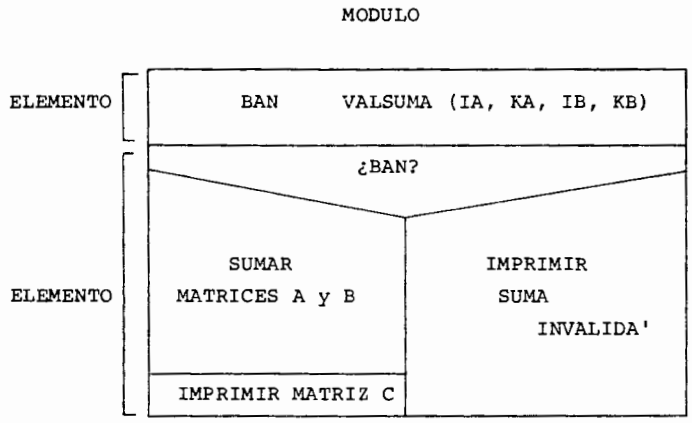
Se describen tres tipos de cohesión partiendo del mejor al peor.

Cohesión funcional.- Un módulo con cohesión funcional es aquél en que todos los elementos contribuyen a una y sólo a una tarea.

Por ejemplo:
- Imprimir una matriz,
- leer una matriz,
- calcular raíz cuadrada,
- sumar dos matrices.

Cohesión secuencial.- Un módulo con cohesión secuencial es aquél en que los datos de salida de un elemento sirven como datos de entrada a otro elemento.

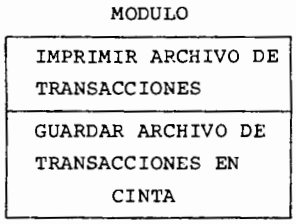
Por ejemplo:



Obsérvese que el módulo lleva a cabo varias funciones. Nótese que cada elemento de este ejemplo tiene una función muy específica y, consecuentemente, cada elemento tiene cohesión funcional.

Cohesión comunicacional.- Un módulo con cohesión comunicacional es aquél cuyos elementos contribuyen a tareas diferentes; pero cada tarea tiene los mismos parámetros de entrada y salida.

Por ejemplo:



10.6 APLICACION DEL ANALISIS Y DISEÑO ESTRUCTURADO

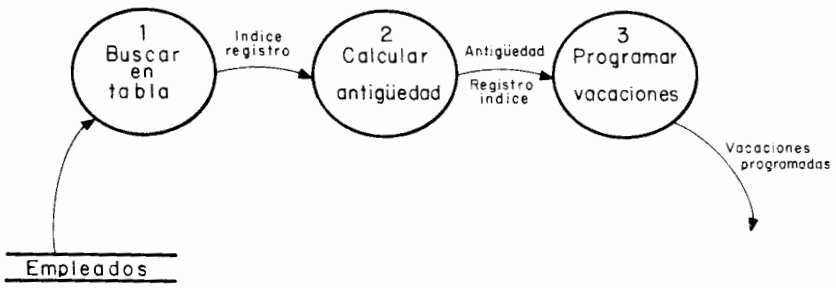
En una fábrica se desea programar las vacaciones de los trabajadores. Si su antigüedad es mayor a seis meses en el momento en que les correspondan, los trabajadores podrán salir de vacaciones.

Para saber en qué mes sale de vacaciones un trabajador, se toma en cuenta la letra con que empieza su primer apellido, de acuerdo a la siguiente tabla:

LETRA DEL APELLIDO	MES EN QUE SALEN DE VACACIONES
A-B	ENERO
C-D	FEBRERO
E-F	MARZO
G-H	ABRIL
I-J	MAYO
K-L	JUNIO
M-N	JULIO
O-P	AGOSTO
Q-R	SEPTIEMBRE
S-T	OCTUBRE
U-W	NOVIEMBRE
X-Z	DICIEMBRE

La programación de las vacaciones se hace al principio del año y en el reporte de programación se imprimirá el nombre del trabajador y el mes en que sale de vacaciones, si es que le tocan; en caso contrario se imprimirá el nombre solamente.

A continuación se muestra el DFD correspondiente a este problema:

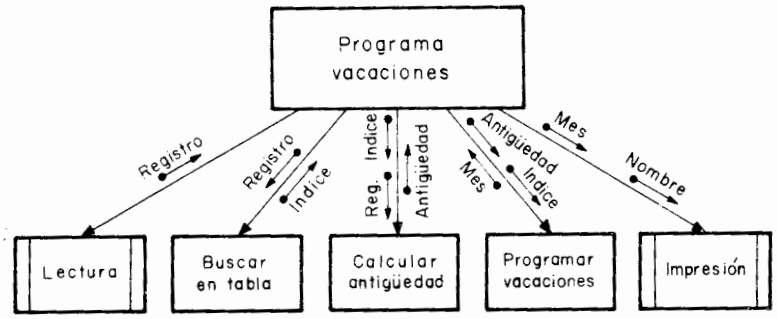


Obsérvese que la burbuja 1 obtiene el registro de un empleado, busca la primera letra de su apellido en la tabla de programación de vacaciones y pasa el índice en que se encuentra la letra del apellido en la tabla y el registro a la burbuja 2.

La burbuja 2 calcula la antigüedad total a la fecha de vacaciones sumando la antigüedad al principio del año (dada en meses) y el índice de la tabla para obtener la antigüedad total, pasando este dato a la burbuja 3.

La burbuja 3 verifica que la antigüedad sea mayor a 6 meses para imprimir el nombre y el mes en que el trabajador sale de vacaciones.

La carta de estructura correspondiente al DFD es la siguiente:



Se puede observar que en la carta de estructura anterior y en el DFD existe una correspondencia uno a uno entre las burbujas y la parte correspondiente a transformaciones en la carta de estructura.

La carta de estructura muestra que el módulo principal (programa vacaciones) sólo coordina las funciones de entrada y salida y las transformaciones.

El módulo de lectura pasa como parámetro el registro completo al módulo principal; este registro será pasado como parámetro al módulo buscar en tabla y éste, a su vez, regresará un índice al módulo principal.

El módulo calcular antigüedad necesita los parámetros registro e índice para poder obtener la antigüedad que será la información que se regresa al módulo principal.

El módulo principal llama al módulo programar vacaciones a través de los datos nombre y antigüedad y éste, a su vez, regresa los datos nombre y mes que serán utilizados para que el módulo impresión imprima el reporte de programación de vacaciones.

A continuación se muestra el seudocódigo de cada uno de los módulos de la carta de estructura.

```
MODULO LECTURA (REGISTRO)
  Leer registro
FIN
```

```
MODULO BUSCAR EN TABLA (REGISTRO, INDICE)
  1. Tomar 1era letra del apellido
  2. Buscar letra en tabla y asignar a índice el
     valor correspondiente al número de línea en
     que se encuentra esa letra en la tabla.
FIN
```

```
MODULO CALCULAR ANTIGUEDAD (REGISTRO, INDICE, ANTIGUEDAD)
  1.  Sumar el valor del campo antigüedad del re-
      gistro con índice.
FIN

MODULO PROGRAMA VACACIONES (ANTIGUEDAD, INDICE, MES)
  Sí (antigüedad ≥ 6) luego
     sí (índice = 1) luego
        mes ← enero
     o sí (índice = 2) luego
        mes ← febrero
          .
          .
          .
     o bien
        mes ← diciembre
     FIN
  O bien
     mes ← '
  FIN
FIN


MODULO IMPRESION (NOMBRE, MES)
  Imprime nombre y mes
FIN

MODULO PROGRAMA VACACIONES
  En tanto (no sea fin de archivo) repetir
     Lectura (registro)
     Buscar en tabla (registro, índice)
     Calcular antigüedad (registro, índice, antigüedad)
     Programar vacaciones (nombre, antigüedad, índice, mes)
     Impresión (nombre, mes)
  FIN
FIN
```

PROBLEMAS PROPUESTOS

1. Dibujar la carta de estructura del problema 1 del capítulo 9.

2. Dibujar la carta de estructura del problema 2 del capítulo 9.

3. Dibujar la carta de estructura del problema 3 del capítulo 9.

B I B L I O G R A F I A

Programación Estructurada

LINGER, MILLS y WITT
Structured programming
Addison Wesley. Reading, Massachusetts, 1979.

TREMBLAY y BUNT
An introduction to computer science
Mc Graw Hill. New York, 1979.

Pascal

CONWAY, GRIES y ZIMMERMAN
A primer on Pascal
Winthrop. Cambridge, Massachusetts, 1981.

CHERRY
Pascal programming structures
Reston. Virginia, 1980.

DYCK, LAWSON, SMITH y BEACH
Computing. An introduction to structured
problem solving using Pascal
Reston. Virginia, 1982.

GLINERT
Introduction to computer science using
Pascal.
Prentice Hall. Englewood Cliffs, 1983.

GROGONO, PETER
Programando en Pascal
Fondo Educativo Interamericano, México.

KELLER, ARTHUR, M.
Programación en Pascal
Mc Graw Hill. New York, 1983.

Análisis Estructurado

DE MARCO, TOM
Structured analysis and system specification
Prentice Hall. Englewood Cliffs, 1978.

GANE, CHRIS y SARSON, TRISH
Structured systems analysis, tools and techniques.
Prentice Hall. Englewood Cliffs, 1979.

WEINBERG
Structured analysis
Prentice Hall. Englewood Cliffs, 1978.

Diseño Estructurado

PAGE JONES, MELLIR
The practical guide to structured system design.
Yourdon Press. New York, 1980.

STEVENS
Using structured design
Wiley Interscience. New York, 1981.

YOURDON, EDWARD y CONSTANTINE, LARRY L.
Structured design
Prentice Hall. Englewood Cliffs, 1978.

Impreso en la
Coordinación de Servicios Generales
Unidad de Difusión
1989