CAPITULO 3
ALGORITMOS Y PROGRAMAS

Este capítulo trata de ser una introducción a la metodología y tecnología de la programación, con el objetivo de proporcionar al lector los procedimientos y técnicas para el desarrollo de programas.

No por obvio, hay que olvidar que los programas se escriben con el ánimo de resolver problemas, con ayuda de las computadoras y que la primera medida a considerar, es el análisis del problema en cuestión y la obtención, en su caso, de un algoritmo adecuado. Por este punto empezaremos nuestra exposición, hasta llegar a los métodos y etapas a seguir para obtener una aplicación informática.

Si bien los conceptos que aquí se introducen son fundamentales para la realización de programas, este capítulo no debe leerse como si se tratara de un manual de programación, sino como una fundamentación de lo que llamamos programación estructurada, mas allá de la sintaxis y de la semántica de un lenguaje de programación concreto.

3.1. CONCEPTO DE ALGORITMO

Sabemos que para que un ordenador pueda llevar adelante una tarea cualquiera, se tiene que contar con un algoritmo que le indique, a través de un programa, que es lo que debe hacer con la mayor precisión posible. Quizás esta afirmación debería ser revisada desde la óptica de la Inteligencia Artificial, pero por el momento la mantendremos como válida dentro del carácter introductorio de este curso. Consecuencia de lo anterior es la importancia del estudio de los algoritmos dentro de las Ciencias de la Computación. Recordemos que un **algoritmo** es "una sucesión finita de pasos no ambiguos que se pueden ejecutar en un tiempo finito", cuya razón de ser es la de resolver problemas; por tanto **"problema"** para nosotros, serán aquellas cuestiones, conceptuales o prácticas , cuya solución es expresable mediante un algoritmo. Afortunadamente, son muchos los problemas cuya solución puede describirse por medio de un algoritmo y ésta es

una de las razones subyacentes a la necesidad de que aprendamos a programar y a manejar un ordenador.

Nótese que no es redundante el hecho de exigir que un conjunto finito de pasos o instrucciones acaben en un tiempo finito, pues una sola instrucción del tipo: "hacer acción A1 hasta que se cumpla la condición C1", acaba dando lugar a un proceso infinito, si no llega a darse nunca la condición C1. El término 'no ambiguo' significa que la acción, a desarrollar en cada paso de la secuencia, viene unívocamente determinada, tanto por la instrucción como por los datos disponibles en este momento, de forma que en cada momento se sepa qué acción única, se tiene que llevar a cabo.

3.2. LA RESOLUCIÓN DE PROBLEMAS Y EL USO DEL ORDENADOR

Antes de entrar en la codificación de la resolución de un problema, hemos de contar con una idea bastante precisa de cómo podemos llegar a esta solución. La experiencia personal de todos nosotros nos dice que la sistematización para la resolución de problemas no es fácil.

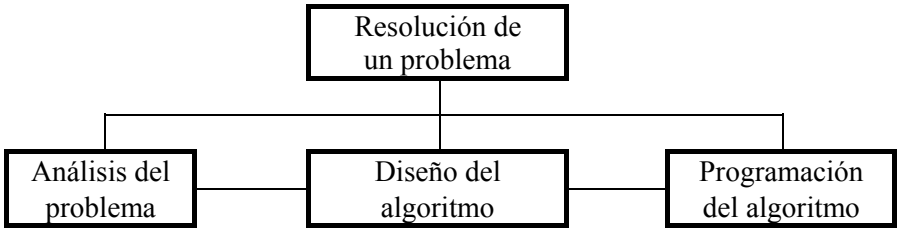


Fig. 3.1. *La resolución de un problema en Informática*

En esta línea, el matemático G. Poyla propuso, a finales de 1940, una metodología general para la resolución de problemas matemáticos, que ha sido adaptada para el caso en que se cuente con un ordenador como recurso. Esta sistemática, de forma muy esquematizada, se puede dividir en tres fases (Ver Figura 3.1):

1. Análisis del problema
2. Diseño del algoritmo
3. Programación del algoritmo

3.2.1 ANÁLISIS DEL PROBLEMA

El objetivo del análisis del problema, es ayudar al programador a llegar a una cierta comprensión de la naturaleza del mismo. Este análisis supone, en particular, la superación de una serie de pasos (Ver Figura 3.2):

- Definir el problema con total precisión.
- Especificar los datos de partida necesarios para la resolución del mismo (especificaciones de entrada).
- Especificar la información que debe proporcionarse al resolverse (especificaciones de salida).

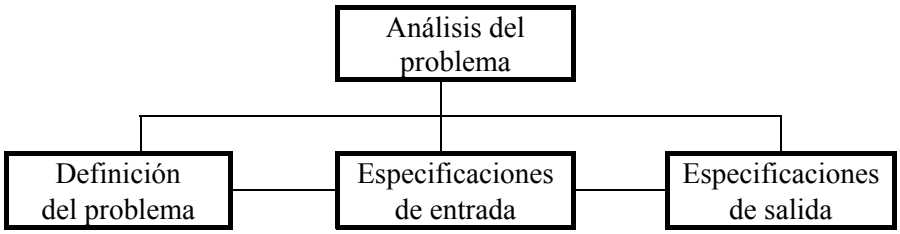


Fig. 3.2. *Análisis del problema*

Ejemplo 1:

Elaborar el análisis para obtener el área y la longitud de una circunferencia.

1.- Utilizar las fórmulas del área y la circunferencia en función del radio.
2.- Las entradas de datos se reducen al dato correspondiente al radio del círculo. Dada la naturaleza del mismo y el procesamiento al cual lo someteremos, su tipo de dato debe ser un número real.
3.- Las salidas serán dos variables también reales: área y circunferencia.

La finalización de la fase de análisis del problema nos llevaría al siguiente resultado:

Entradas: Radio del círculo (variable RADIO).
Salidas: Superficie del círculo (variable AREA).
Circunferencia del círculo (variable CIRCUNFERENCIA).
Variables: RADIO, AREA, CIRCUNFERENCIA: tipo real.

3.2.2 DISEÑO DEL ALGORITMO

Diseñar un algoritmo puede ser una tarea difícil y su aprendizaje no es inmediato, ya que requiere una buena dosis de experiencia y creatividad. Hace ya 100 años, un matemático de la talla de Henri Poincare, que no sólo trabajó en temas

relacionados con la física, el álgebra y el análisis, sino también sobre la filosofía de la ciencia, trató de explicar sus experiencias personales de cómo un problema, a cuya resolución había dedicado mucho tiempo sin éxito, podía aparecer tiempo después resuelto repentinamente en su cabeza, incluso cuando se estaba dedicando a proyectos distintos. Desgraciadamente sus resultados en este empeño, distaron mucho de la brillantez de sus logros como físico y matemático. El periodo que existe entre el análisis de un problema y el diseño de su solución recibe el nombre de **periodo de incubación** y el proceso mental, que se da durante el mismo sigue siendo un tema de investigación para los psicólogos. Estamos por tanto en el terreno de la inspiración y la madurez mental. Seamos optimistas y pensemos que vamos a tener la capacidad de tener ideas, propias o adquiridas, para desarrollar algoritmos que nos permitan actuar ante los problemas que se nos planteen.

Para diseñar algoritmos hay que tener presente los requisitos siguientes:

- indicar el orden de realización de cada paso,
- estar definido sin ambigüedad y
- ser finito

Ejemplo 2:

Averiguar si un número es primo o no, suponiendo que razonamos de la siguiente forma: "Del análisis del hecho de que un número N es primo si sólo puede dividirse por sí mismo y por la unidad, un método que nos puede dar la solución sería dividir sucesivamente el número por 2, 3, 4..., etc. y, según el resultado, podríamos resolver el problema". Un diseño del mismo sería:

1. Inicio
2. Poner X igual a 2 (X = 2, X, variable que representa a los posibles divisores de N)
3. Dividir N por X (N/X)
4. Si el resultado es entero, entonces N no es primo, y saltar al punto 9 (en caso contrario continuar el proceso en el siguiente punto, 5)
5. Incrementar X en una unidad
6. Si X es menor que N saltar al punto 3 (en caso contrario continuar el proceso en el siguiente punto, 7)
7. Declarar N es primo;
8. Saltar al Fin (punto 10)
9. Declarar N no es primo
10. Fin

Como parte del diseño de algoritmo está la selección de uno que sea razonablemente aceptable, entre todos los muchos posibles que resuelven el mismo problema (el ejemplo que acabamos de dar es claramente mejorable, pues si N no era divisible por 2 no tiene mucho sentido volverse a preguntar si lo es por 4).

Durante el diseño es posible y aconsejable, realizar comparaciones entre algoritmos que resuelven el mismo problema. La bondad de un algoritmo puede medirse por dos factores:

- El tiempo que se necesita para ejecutarlo. Para tener una idea aproximada de ello, basta con saber el número de instrucciones de cada tipo necesarias para resolver el problema.
- Los recursos que se necesitan para implantarlo.

Así, una vez diseñado un primer algoritmo, conviene realizar una evaluación del mismo, cuestión a veces nada banal y sobre la que volveremos en capítulos posteriores. Si se decide que éste no es eficiente será necesario o bien diseñar uno nuevo o bien optimizar el original. **Optimizar un algoritmo** consiste en introducir modificaciones en él, tendentes a disminuir el tiempo que necesita para resolver el problema o a reducir los recursos que utiliza. (En el ejemplo 2 el algoritmo se optimiza, si N se declara como primo cuando X supera a N/2).

3.2.2.1 Diseño Descendente o Modular

Los problemas complejos se pueden resolver más eficazmente cuando se descomponen en subproblemas que sean más fáciles resolver el original. Este método se denomina **divide y vencerás** y consiste en convertir un problema complejo en otros más simples que, una vez resueltos, en su conjunto nos solucionen el original. Al procedimiento de descomposición de un problema en subproblemas más simples, (llamados **módulos)** para, a continuación, seguir dividiendo estos subproblemas en otros más simples, se le denomina **diseño descendente**. Las ventajas más importantes de este tipo de diseño son:

1) El problema se comprende más fácilmente al dividirse en **módulos** o partes más simples. Adelantemos que cuando demos el salto a la programación, utilizaremos esta idea constantemente, de forma que hablaremos también de **procedimientos,** o subprogramas.
2) Las modificaciones en los módulos son más fáciles, pues estamos ante algoritmos más sencillos.
3) La comprobación del problema se puede realizar más fácilmente, al poder localizar los posibles fallos con mayor precisión.

3.2.2.2 Refinamiento por pasos

Durante el diseño, entenderemos por refinamiento por pasos, la metodología por la que en un primer esbozo del algoritmo nos limitamos a señalar

o describir un reducido numero de pasos, que deberán ser expresados con mayor detalle posteriormente. Tras esta primera descripción, éstos se especifican con mayor minuciosidad, de forma más extensa y con más pasos específicos. En cada nivel de refinamiento hay que considerar dos fases: ¿Qué hace el módulo? para a continuación responder a ¿Cómo lo hace?.

Como es natural, dependiendo de la complejidad del problema se necesitarán diferentes y sucesivos niveles de refinamiento antes de que pueda obtenerse un algoritmo con suficiente nivel de detalle. Así en el Ejemplo 1, del cálculo de la longitud y superficie de un círculo, a pesar de presentar un bajo nivel de complejidad, en su diseño, se puede descomponer en subproblemas más simples:
1) leer datos de entrada,
2) calcular superficie y longitud,
3) escribir resultados.

El ejemplo siguiente, nos muestra el diseño de un algoritmo para un problema de carácter no numérico

Ejemplo 3:

Diseñar un algoritmo que responda a la pregunta: ¿Qué debo hacer para ver la película XYZ?.
Un primer análisis nos conduce a un esbozo de solución, descomponiéndolo en cuatro módulos sucesivos:

1 ir al cine donde proyectan XYZ
2 comprar una entrada
3 ver la película
4 regresar a casa

Estos cuatro pasos se pueden refinar un poco más y así este problema lo podríamos descomponer de la siguiente forma:

```
inicio          {algoritmo para ver la película XYZ}
consultar la cartelera de cines
    si proyectan "XYZ" entonces
       ir al cine correspondiente
    si_no proyectan "XYZ"
      declarar el fracaso del objetivo y terminar
acudir al cine correspondiente
si hay cola entonces ponerse en ella
```

```
mientras haya personas delante en la cola hacer
    avanzar en la cola
preguntar si quedan entradas
    si hay entradas entonces
        comprar una entrada
    si_no quedan entradas
        declarar el fracaso del objetivo, regresar a casa y terminar

encontrar el asiento correspondiente
mientras proyectan la película hacer
    ver la película
abandonar el cine

regresar a casa
fin
```

Algunas de estas acciones son primitivas para nosotros, es decir, no es necesario descomponerlas más, como el abandonar el cine. Sin embargo hay otras acciones que son susceptibles de mayor descomposición. Este es el caso de la acción:

"encontrar el asiento correspondiente"

si los números de los asientos están impresos en la entrada, esta acción compuesta se resuelve con el siguiente algoritmo:

```
inicio {algoritmo para encontrar el asiento del espectador}
caminar hasta llegar a la primera fila de asientos

repetir
        comparar número de fila con número impreso en billete
        si no son iguales, entonces pasar a la siguiente fila
hasta_que se localice la fila correcta

mientras número de asiento no coincida con número de billete
          hacer avanzar a través de la fila a la siguiente butaca
sentarse en la butaca
fin
```

De esta forma, podríamos seguir hasta la descomposición de las distintas acciones en instrucciones susceptibles de ser interpretadas, directamente por el ordenador.

3.2.3 PROGRAMACIÓN DEL ALGORITMO

Una vez que el algoritmo está diseñado y representado, se debe pasar a la fase de resolución práctica del problema con el ordenador. Esta fase se descompone a su vez en las siguientes subfases: (Ver Figura 3.3)

1. *Codificación del algoritmo en un programa*.
2. *Ejecución del programa*.
3. *Comprobación del programa*.

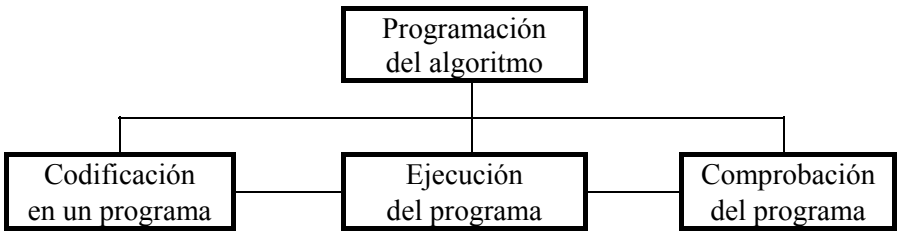


Fig. 3.3 *Programación del algoritmo*

La fase de conversión de un algoritmo en instrucciones de un lenguaje de programación, como sabemos, se denomina **codificación**. El código deberá estar escrito de acuerdo con la sintaxis del lenguaje de programación ya que solamente las instrucciones sintácticamente correctas pueden ser interpretadas por el computador.

Nótese que durante el proceso de programación, se debe separar el diseño del algoritmo de su posterior implementación en un lenguaje de programación específico. Por ello distinguimos entre el concepto más general de programación y el más particular de codificación, que depende del lenguaje de programación utilizado. Al llegar a este punto, se supone que el lector conoce al menos uno de estos lenguajes y éste es el momento en el que tiene que mostrar sus habilidades para efectuar una codificación lo más correcta y eficiente posible.

Tras la codificación del programa, éste deberá ejecutarse en un computador. El resultado de esta primera ejecución es incierto, ya que existe una alta posibilidad de que aparezcan errores, bien en la codificación bien en el propio algoritmo. Por tanto, el paso siguiente consiste en comprobar el correcto funcionamiento del programa y en asegurarse, en la medida de lo posible, de la validez de los resultados proporcionados por la máquina.

3.3. REPRESENTACIÓN DE ALGORITMOS

Un algoritmo es algo puramente conceptual que necesita una forma de representación, bien para comunicarlo a otra persona bien para ayudar a convertirlo en un programa. De hecho, la codificación en un lenguaje de programación, es una representación muy utilizada de un algoritmo, sin embargo tiene el inconveniente de que no todas las personas conocen el lenguaje que se haya elegido. Por ello,

existen diferentes métodos que permiten que se pueda independizar el algoritmo de su correspondiente codificación. Veamos dos de ellos:

3.3.1 PSEUDOCODIGO

El **pseudocódigo** es un lenguaje de especificación de algoritmos (no de programación) basado en un sistema notacional, con estructuras sintácticas y semánticas, similares a los lenguajes procedurales, aunque menos formales que las de éstos, por lo que no puede ser ejecutado directamente por un computador. El pseudocódigo utiliza para representar las sucesivas acciones, palabras reservadas -similares a sus homónimas en los lenguajes de programación-, tales como **start, end, stop, if-then-else, while-do, repeat-until, (inicio, fin, parar, si-entonces-sino, mientras-hacer, repetir-hasta)**, etc. A lo largo de este capítulo, a medida que vayamos describiendo las estructuras de control utilizadas en los programas, iremos haciendo una lista de las instrucciones más usuales del pseudocódigo. La ventajas del uso del pseudocódigo residen en:

- Su uso en la planificación de un programa; permitiendo que el programador se pueda concentrar en la lógica y en las estructuras de control y no tenga que preocuparse, por ahora de detalles acerca de las reglas sintácticas y semánticas de un lenguaje específico. Consiguientemente es más fácil de modificar, en el caso de que se descubran errores o anomalías en la lógica del algoritmo.

- Aunque el pseudocódigo es independiente del lenguaje de alto nivel que vaya a utilizarse, un algoritmo expresado en pseudocódigo puede ser traducido más fácilmente a muchos de ellos.

Ejemplo 4:

Supongamos que tenemos un algoritmo para averiguar si un número es par, que puede ser descrito narrativamente de la siguiente forma: "Si restando consecutivamente doses del número se obtiene el numero 2, es par, si se obtiene otro valor (el 1), entonces es impar". Este algoritmo escrito en pseudocódigo sería:

```
leer N
mientras N > 2 hacer
   N ← N - 2
si N = 2 entonces
        escribe "es par"
sino
        escribe "es impar"
```

fin

Nótese que en este ejemplo y en otros anteriores hemos utilizado dos estructuras que son muy usadas en programación: **mientras-hacer** y **si-entonces-si_no;** y que la escritura del pseudocódigo usa normalmente la indentación (sangría en el margen izquierdo) de diferentes líneas para ayudar a delimitar visualmente cada una de las estructuras utilizadas.

3.3.2 ORGANIGRAMAS

Para ganar claridad expositiva se han desarrollado una serie de símbolos gráficos que permiten representar los algoritmos y que son universalmente reconocidos. Veamos algunos ejemplos:

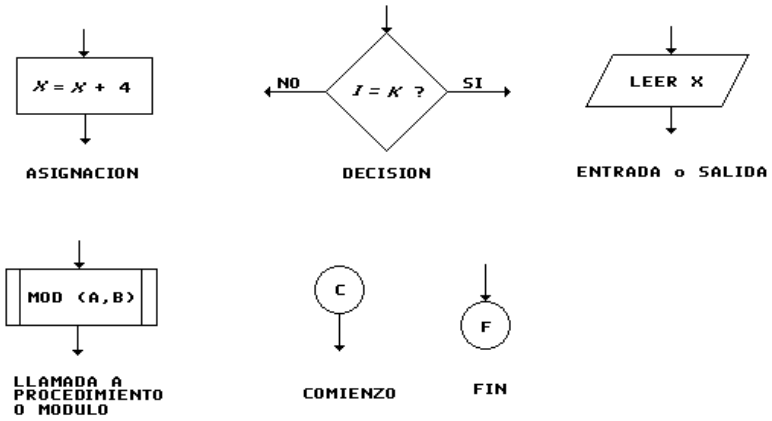


Fig. 3.4. *Símbolos usados para confeccionar organigramas*

Los **organigramas o diagramas de flujo** son herramientas gráficas utilizadas tanto para representar algoritmos, como en la ayuda en el diseño de programas. Están compuestos por una serie de símbolos, unidos con flechas, donde cada símbolo representa una acción distinta y las flechas el orden de realización de las acciones. Cada símbolo, por tanto, tendrá al menos una flecha que conduzca a él y una flecha que parta de él, exceptuando el comienzo y final del algoritmo. En la Figura 3.4, se muestran los símbolos utilizados habitualmente en la confección de organigramas, cuyo significado completaremos más adelante.

La Figura 3.5. representa, en forma de organigrama, el algoritmo del Ejemplo 4 que ha sido expresado en pseudocódigo en la sección anterior.

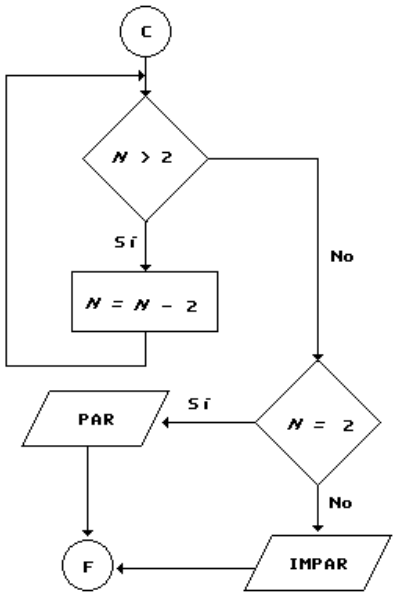


Fig. 3.5. *Organigrama del Ejemplo 4*

3.4. ESTRUCTURAS DE CONTROL

En el Capítulo 1, vimos los elementos básicos constitutivos de un programa:

- palabras reservadas (inicio, si-entonces, etc.)
- identificadores (nombres de variables, procedimientos, etc.)
- caracteres especiales (coma, punto y coma, apóstrofo, etc.)
- constantes
- variables
- expresiones
- instrucciones

Sin embargo como hemos visto al diseñar algoritmos para escribir un programa, además de estos elementos básicos, hemos de conocer determinadas estructuras, cuyo objetivo es controlar su ejecución y sin cuya comprensión es imposible programar.

Llamaremos **estructuras de control** a las acciones que tienen por objeto marcar el orden de realización de los distintos pasos de un programa ó algoritmo. Cada estructura tiene un punto de entrada y uno de salida, lo que facilita la depuración de posibles errores. Estas son de tres tipos:

- estructuras secuenciales

- estructuras selectivas
- estructuras repetitivas

y vamos a estudiarlas con un cierto detalle. El uso de las estructuras de control es una de las características de la programación estructurada que constituye la principal orientación de este texto, aunque otros lenguajes procedurales no estructurados también utilizan estas estructuras.

3.4.1 ESTRUCTURAS SECUENCIALES

Son aquéllas en las que una acción (instrucción) sigue a otra de acuerdo con su orden de escritura. Las tareas se suceden de tal modo que tras la salida (final) de una se efectúa la entrada (principio) en la siguiente y así sucesivamente hasta el fin del proceso. Su organigrama obedece al esquema de la Figura 3.6:

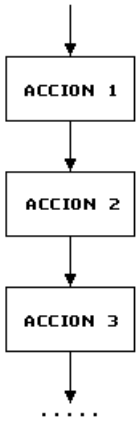


Fig. 3.6. *Esquema de una estructura secuencial*

Las estructuras secuenciales se codifican de forma directa en cualquier lenguaje de programación, pues como sabemos el orden de ejecución de todo programa es precisamente, salvo orden en sentido contrario, de arriba abajo. A pesar de su simplicidad, ya sabemos, que algunos problemas se pueden resolver con la sola utilización de esta estructura, como por ejemplo el cálculo del volumen del cilindro visto en el Capítulo 1, y codificado en los cuatro lenguajes de programación.

3.4.2 ESTRUCTURAS SELECTIVAS

Como hemos tenido ocasión de comprobar, la especificación formal de algoritmos tiene utilidad real, cuando éstos requieren una descripción más complicada que una simple secuencia de instrucciones. Uno de estos casos se

produce cuando existen varias alternativas, resultantes de la evaluación de una determinada condición, como ocurre, por ejemplo, al resolver una ecuación de segundo grado, donde el procedimiento a seguir es distinto según el discriminante sea positivo, nulo ó negativo. Las estructuras selectivas en un programa se utilizan para tomar decisiones, de ahí que se suelan denominar también *estructuras de decisión o alternativas*. En estas estructuras se evalúa una condición, especificada mediante expresiones lógicas, en función de cuyo resultado, se realiza una opción u otra. En una primera aproximación, para esta toma de decisiones, podemos pensar en una variable **interruptor o conmutador (switch)**, que representa un estado y por tanto puede cambiar de valor a lo largo de la ejecución regulando el paso a una u otra parte del programa, lo que supone una bifurcación en el flujo del programa, dependiendo del valor que tome el conmutador. Los interruptores pueden tomar dos valores diferentes, frecuentemente 1 y 0, de ahí su nombre de interruptor ("encendido"/"apagado", "abierto"/"cerrado"). En la Figura 3.7, si SW es igual a 1, se ejecuta la acción S1; y si SW es igual a 0, se ejecuta la acción S2.

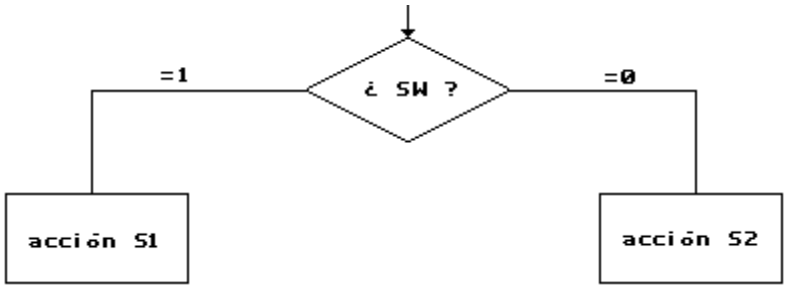


Fig. 3.7. *Funcionamiento de un interruptor*

Ejemplo 5:

Sea un archivo formado por un conjunto de registros constituidos por dos campos, M y N. Se desea listar el campo M de los registros pares y el campo N de los registros impares. Expresar el algoritmo correspondiente en forma de organigrama (Ver Figura 3.8).

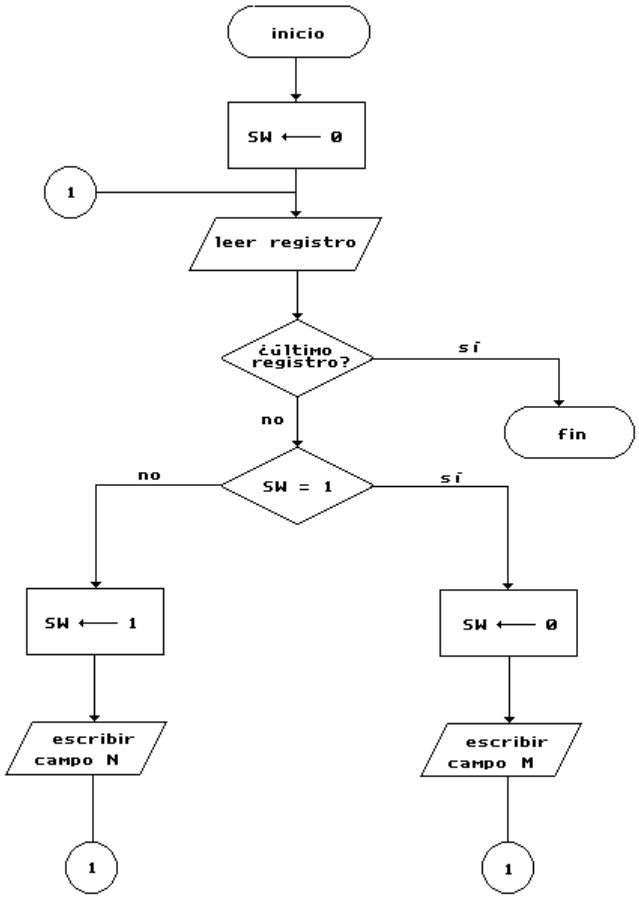


Fig. 3.8. *Organigrama del Ejemplo 3*

En el ejemplo SW es un interruptor que se inicializa con un valor determinado (0 en este caso) y luego se va modificando su valor alternativamente a medida que se leen los registros. De este modo, cuando SW = 0 se leerán las fichas impares y cuando SW = 1 se leerán las fichas pares.

3.4.2.1 Alternativas simples (si-entonces/if-then)

La estructura alternativa más sencilla, es la llamada simple y se representa por **si-entonces**. Su efecto es el de ejecutar una determinada acción cuando se cumple una cierta condición y en caso contrario seguir el orden secuencial. La selección **si-entonces** evalúa la condición y de acuerdo con su resultado:

- Si es verdadera, entonces ejecuta una o varias acciones (S1).

- Si es falsa, entonces no hace nada y sigue la ejecución normal del programa, pasando a la instrucción siguiente a la finalización de la estructura selectiva. Para ello es necesario que ésta venga claramente delimitada, cosa que se hace en pseudocódigo con fin_si.

Su expresión en *Pseudocódigo* (para S1 compuesta de varias acciones) es:

si <condición> **entonces**
<acciones>
fin_si

FORTRAN	BASIC

IF *logico*
IF (condición) S
IF (condicion) GOTO etiqueta
 	
IF condición THEN
S1
S2
.
Sn
END IF
	IF expresión THEN numero linea
	IF expresión THEN GOTO etiqueta
	
	IF condición THEN
	S1
	S2
	.
	Sn
	END IF

PASCAL	**C**

if condicion **then**
begin
S1
S2
.
Sn
....**end**
	`if (condicion)`
	`{`
	`S1`
	`S2`
	`...`
	`Sn`
	`};`

3.4.2.2 Alternativas dobles (si-entonces-si_no/if-then-else)

La estructura anterior es muy limitada y muchas veces se necesitará una estructura que permita elegir entre dos opciones o alternativas posibles en función del cumplimiento o no de una condición, que juega el papel de un interruptor. Si la condición es verdadera, se ejecuta la acción o acciones S1, y si es falsa, se ejecuta la acción ó acciones S2, pasando en cualquier caso a la instrucción siguiente a la finalización de la estructura selectiva. El pseudocódigo correspondiente es el siguiente

```
si <condición> entonces
    <acciones S1>
si_no
    <acciones S2>
fin_si
```

FORTRAN	BASIC

IF condición THEN
acciones S1
ELSE
acciones S2
END IF
	IF condición THEN
	acciones S1
	ELSE
	acciones S2
	END IF

PASCAL	**C**

if condicion **then**
begin
acciones S1
end
else
begin
acciones S2
end
	`if (condicion)`
	`{`
	`acciones S1`
	`}`
	`else`
	`{`
	`acciones S2`
	`}`

Ejemplo 6:

Informar si un estudiante ha superado o no un determinado examen consistente en 20 preguntas de igual valor y calcular su nota en caso de aprobar.

```
leer num_correctas
si num_correctas < 10 entonces
        escribir "no ha superado Vd. el examen"
        faltan ← 10 - num_correctas
        escribir "le faltaron" , faltan
si_no
        nota ← num_correctas / 2
        escribir "aprobó Vd. con un" , nota
fin_si
```

3.4.2.3 Alternativas múltiples (según_sea, en_caso_de)

Con frecuencia existen más de dos elecciones posibles (por ejemplo, en la resolución de la ecuación de segundo grado hay tres casos). La estructura de alternativa múltiple evaluará una expresión que podrá tomar *n* valores (o grupos de valores) distintos e_1, e_2,....,e_n. Según el valor que tome la condición, se realizará una de las *n* acciones posibles, o lo que es lo mismo, el flujo del algoritmo seguirá un determinado camino entre los *n* posibles (Ver Figura 3.9).

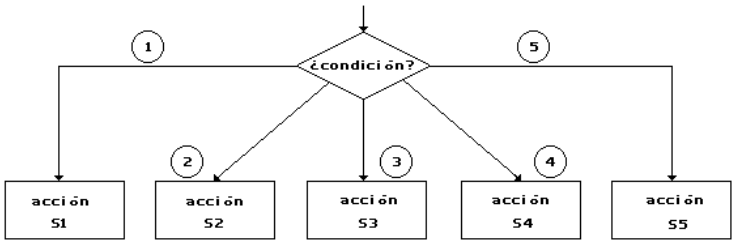


Fig. 3.9. *Esquema de una alternativa múltiple*

La estructura de decisión múltiple en pseudocódigo se puede representar de diversas formas, aunque la más simple es la siguiente (donde e_1, e_2, ..., e_n son los distintos valores (o grupos de ellos) que puede tomar la evaluación de la expresión que sirve de condición):

```
según_sea  expresión (E) hacer
      e1:  <acciones S1>
      e2:  <acciones S2>
           .
      en:  <acción Sn>
fin_según
```

Ejemplo 7:

Escribir un algoritmo que permita obtener las raíces reales de la ecuación de segundo grado, $ax^2 + bx + c$.

```
algoritmo RESOL2
inicio
    leer a,b,c
    D ← b^2-4*a*c
    segun_sea D hacer
      D<0:  escribir "raíces complejas"     {no existen raíces reales}
```

```
        D=0:  x ← -b/2a
              escribir x , "raiz doble"
        D>0:  rc← raizcua(D)
              x1← (-b-rc)/2a
              x2 ←(-b+rc)/2a
              escribir x1,x2
    fin_segun
fin
```

Para implementar esta estructura de alternativas múltiples hemos de recurrir a estructuras alternativas simples o dobles, adecuadamente enlazadas. Las estructuras de selección **si-entonces** y **si_entonces_sino** implican la selección de una de dos alternativas y utilizándolas debidamente, es posible diseñar con ellas estructuras de selección que contengan más de dos posibilidades. Una estructura **si-entonces** puede contener otra y así sucesivamente cualquier número de veces (se dice que las estructuras están *anidadas* o en *cascada*). El esquema es del tipo:

```
si (condición e1) entonces
   <acciones S1>
sino
      si (condición e2) entonces
         <acciones S2>
      sino
            si (condición e3) entonces
               <acciones S3>
            sino
               ...
               <acciones Sn>
            fin_si
      fin_si
fin_si
```

3.4.3 ESTRUCTURAS REPETITIVAS

El computador está especialmente diseñado para aplicaciones en las que una operación o un conjunto de ellas deben repetirse muchas veces. En este sentido, definiremos **bucle** o lazo (loop), como un segmento de un programa cuyas instrucciones se repiten bien un número determinado de veces o mientras se cumpla una determinada condición.

Es imprescindible que se establezcan mecanismos para controlar esta tarea repetitiva, ya que si éstos no existen, el bucle puede convertirse en un proceso

infinito. Así, en el bucle representado por el organigrama de la Figura 3.10, se observa que las instrucciones incluidas en él se repiten indefinidamente. El mecanismo de control citado se establece mediante una condición que se comprueba en cada paso o **iteración** del bucle. En la Figura 3.11, se coloca una condición tras la lectura de la variable N (comprobar si su valor es cero), de forma que tenemos la oportunidad de que el bucle deje de ser infinito, ya que podrá interrumpirse cuando la condición sea verdadera.

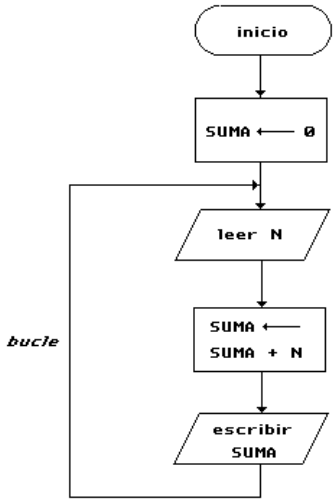


Fig. 3.10 *Bucle infinito*

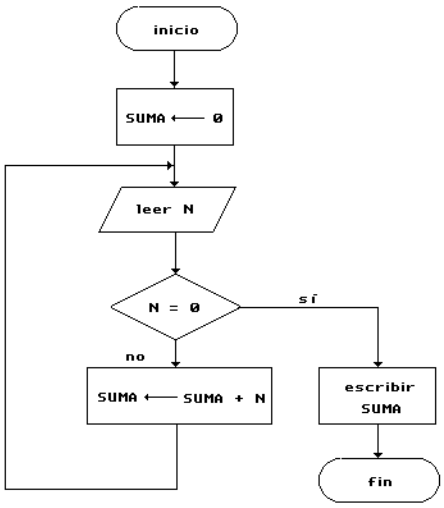


Fig. 3.11 *Bucle con condición de salida*

Los procesos que se repiten varias veces en un programa necesitan en muchas ocasiones contar el numero de repeticiones habidas. Una forma de hacerlo es utilizar una variable llamada **contador,** cuyo valor se incrementa o decrementa en una cantidad constante en cada repetición que se produzca. La Figura 3.12 presenta un diagrama de flujo para un algoritmo en el que se desea repetir 50 veces un grupo de instrucciones, que llamaremos cuerpo del bucle, donde el contador se representa con la variable CONT. La instrucción que actualiza al contador es la asignación:

CONT ← CONT+1.

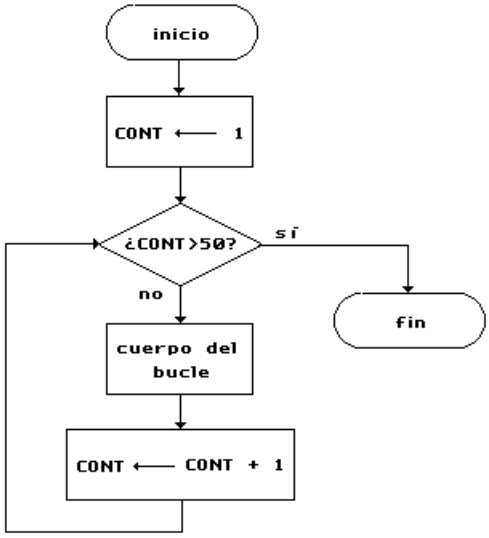


Fig. 3.12 *Ejemplo de bucle con contador positivo*

El contador puede ser positivo (incrementos de uno en uno) o negativo (decrementos de uno en uno). En la Figura 3.12, el contador cuenta desde 1 a 50 y deja de repetirse cuando la variable CONT toma el valor 51 y termina el bucle. En la Figura 3.13 se muestra un algoritmo que efectúa la operación de multiplicación n x m, sumando m un número n de veces. En él, el contador se decrementa: comienza a contar en *n* y se va decrementando hasta llegar a cero; en ese momento se termina el bucle y se realiza la acción escribir.

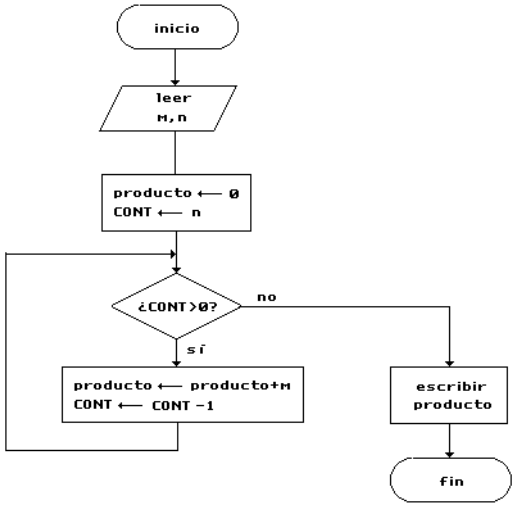


Fig. 3.13 *Ejemplo de bucle con contador negativo*

Otro tipo de variable, normalmente asociada al funcionamiento de un bucle es un **acumulador** o totalizador, cuya misión es almacenar una cantidad variable, resultante de operaciones sucesivas y repetidas. Un acumulador realiza una función parecida a la de un contador, con la diferencia de que el incremento o decremento, de cada operación es variable en lugar de constante. En la Figura 3.11 la instrucción, SUMA ← SUMA + N, añade en cada iteración el valor de la variable N, leída a través de un periférico, por lo que SUMA es un ejemplo de acumulador.

Una **estructura repetitiva** es aquella que marca la reiteración de una serie de acciones basándose en un bucle. De acuerdo con lo anterior, esta estructura debe constar de tres partes básicas:

- decisión (para finalizar la repetición)
- cuerpo del bucle (conjunto de instrucciones que se repiten)
- salida del bucle (instrucción a la que se accede una vez se decide finalizar)

Tomando el caso de la Figura 3.11, donde para obtener la suma de una serie de números, hemos utilizado la estrategia siguiente: tras leer cada número lo añadimos a una variable SUMA que contenga las sucesivas sumas parciales (SUMA se hace igual a cero al inicio). Observemos que el algoritmo correspondiente deberá utilizar sucesivamente instrucciones tales como:

```
leer número
si N < 0 entonces
  escribir SUMA
si-no
  SUMA ←SUMA+número
fin-si
```

que se pueden repetir muchas veces; éstas constituyen el cuerpo del bucle.

Una vez se ha decidido el cuerpo del bucle, se plantea la cuestión de cuántas veces se debe repetir. De hecho conocemos ya la necesidad de contar con una condición para detener el bucle. En el Ejemplo 8, se pide al usuario el número N de números que desea sumar, esto es, el número de iteraciones del bucle. Usamos un contador de iteraciones, TOTAL, que se inicializa a N y a continuación se decrementa en uno cada vez que el bucle se repite; para ello introducimos una acción más al cuerpo del bucle: TOTAL ← TOTAL -1. También podríamos inicializar la variable TOTAL en 0 o en 1, e ir incrementándolo en uno, en cada iteración, hasta llegar al número deseado N.

Ejemplo 8:

Hallar la suma de N números, a través de una estructura repetitiva

```
algoritmo suma_números
{leer número total de números a sumar en variable N}
TOTAL ← N
SUMA ← 0   { la suma parcial es 0 al inicio}
{comienzo de bucle}
mientras que TOTAL > 0 hacer
      leer número
      SUMA ← SUMA+número
      TOTAL ← TOTAL-1
fin_mientras
{fin del bucle}
escribir "la suma de los" , N , "números es " , SUMA
```

Aunque la condición de finalización puede evaluarse en distintos lugares del algoritmo, no es recomendable que ésta se pueda efectuar a mitad del cuerpo del bucle, por lo que es bueno que se produzca al principio o al final del mismo. Según donde se sitúe la condición de salida, dará lugar a distintos tipos de estructuras repetitivas que analizaremos a continuación: estructura **desde-hasta**, estructura **mientras** y estructura **repetir-hasta_que**.

3.4.3.1 ESTRUCTURA DESDE-HASTA

Esta estructura consiste en que la condición de salida se basa en un contador que cuenta el número de iteraciones. Por ejemplo, el ejemplo 8 podría hacerse de la siguiente manera:

```
desde i = 1 hasta N con_incremento 1 hacer
    leer número
    SUMA ← SUMA + número
fin_desde
```

donde i es un contador que cuenta desde un valor inicial (1) hasta el valor final (N) con los incrementos que se consideren (de uno en uno en este caso). Esta es la llamada estructura **Desde ("for"),** que es la más simple desde el punto de vista de la condición de salida, ya que viene predeterminada por el código. Su utilidad reside en el hecho de que, en muchas ocasiones, se conoce de antemano el número

de iteraciones. Esta estructura ejecuta las acciones del cuerpo del bucle, un número especificado de veces y de modo automático controla el número de iteraciones. Su formato en pseudocódigo es:

desde v=vi **hasta** vf **hacer**
  <acciones>
  .
  .
fin_desde

v: variable índice
vi, vf: valores inicial y final de la variable

La variable índice o de control normalmente será de tipo entero y es normal emplear como identificador, las letras I,J,K como herencia de los índices y subíndices utilizados en cálculo científico. El incremento de la variable índice es 1 en cada iteración si no se indica expresamente lo contrario. Si debemos expresar incrementos distintos de +1 el formato de la estructura es:

desde v = vi **hasta** vf **inc** incremento **hacer**
  <acciones>
  .
  .
fin_desde

si vi > vf entonces usar en lugar de la expresión **inc** incremento **dec** decremento

Obviamente, si el valor inicial de la variable índice es menor que el valor final, los incrementos deben ser positivos, ya que en caso contrario la secuencia de acciones no se ejecutaría. De igual modo si el valor inicial es mayor que el valor final, el incremento debe ser en este caso negativo. Así:

desde I=20 **hasta** 10 **hacer**
  <acciones>
fin_desde

no ejecutaría nunca el cuerpo del bucle.

FORTRAN	**BASIC**
DO n I =M1, M2, M3 <acciones> n CONTINUE n : etiqueta de fin de bucle M1: valor inicial del contador M2: valor final del contador M3: valor de incremento (+ ó -)	FOR V = Vi TO Vf <acciones> NEXT V FOR V = Vi TO Vf STEP X <acciones> NEXT V
PASCAL	**C**

for v:=vi to vt do
  <accion o bloque de acciones>
 	
for v:=vi downto vt do
  <accion o bloque de acciones >
	for(v=vi; v<=vf; vt+=step)
	  <accion o bloque de acciones >
	
	(más genéricamente)
	for(inicio;condición;actualización)
	  <accion o bloque de acciones >

3.4.3.2 ESTRUCTURA MIENTRAS

Cuando la condición de salida del bucle se realiza al principio del mismo, éste se ejecuta mientras se verifica una cierta condición. Es la llamada estructura repetitiva **mientras ("while")**; en ella el cuerpo del bucle se repite mientras se cumple una determinada condición. Su pseudocódigo es:

mientras condición **hacer**
  <acciones>
fin_mientras

Cuando se ejecuta la instrucción *mientras*, la primera cosa que sucede es la evaluación de la condición. Si es *falsa*, no se ejecuta ninguna acción y el programa prosigue en la siguiente instrucción a la finalización del bucle; si la condición es *verdadera*, entonces se ejecuta el cuerpo del bucle. No todos los lenguajes incluyen la estructura mientras.

PASCAL	**C**
while (condicion) do <accion o bloque de acciones>	while (condicion) <accion>

Obsérvese que en una estructura **mientras** si la primera evaluación de la condición es *falsa,* el cuerpo del bucle nunca se ejecuta. Puede parecer *inútil* ejecutar el cuerpo del bucle *cero veces*, ya que no tendrá efecto en ningún valor o salida; sin embargo, puede ser una acción deseada. Por ejemplo el siguiente bucle para procesar las notas de unos examenes contando el número de alumnos presentados dejará de ejecutarse cuando el numero leído sea negativo. Si la primera nota introducida fuera negativa, la acción deseada es, efectivamente, que no se ejecute el bucle ninguna vez.

C ← 0
leer nota
mientras nota ≥ 0 **hacer**
  {procesar nota}

```
    C ← C+1
    leer nota
fin_mientras
```

3.4.3.3 ESTRUCTURA REPETIR-HASTA_QUE

En esta estructura la condición de salida se sitúa al final del bucle; el bucle se ejecuta hasta que se verifique una cierta condición. Es la llamada estructura **Repetir-hasta ("repeat-until")**. Existen muchas situaciones en las que se desea que un bucle se ejecute al menos una vez, antes de comprobar la condición de repetición. Para ello la estructura **repetir-hasta_que** se ejecuta hasta que se cumpla una condición determinada que se comprueba al final del bucle. En pseudocódigo se escribe:

```
repetir
  <acciones>
hasta_que <condición>
```

PASCAL	C
repeat <accion> ...<accion> until (condicion);	do <accion o bloque> while (condicion); (*)

** (en C se "dice" mientras se cumpla la condición, no hasta que se cumpla)*

El bucle **repetir-hasta_que** se repite mientras la condición sea falsa, justo lo opuesto a la estructura **mientras**. Sea el siguiente algoritmo:

```
inicio
    contador ←1
    repetir
        leer número
        contador ←contador+1
    hasta_que contador > 30
    escribir "números leídos: 30"
fin
```

Este bucle se repite hasta que el valor de variable contador exceda a 30, lo que sucederá después de 30 ejecuciones del mismo. Nótese que si en vez de 30 pusiéramos 0, el cuerpo del bucle se ejecutará *siempre al menos una vez*.

Ejemplo 9:

Calcular el factorial de un número N, usando la estructura repetir.

```
inicio
   leer N
   Factorial ←1
   I ← 1
      repetir
         Factorial← Factorial * I
         I ← I+1
      hasta_que I > N
   escribir "el factorial del número", N, "es", Factorial
fin
```

Las tres estructuras repetitivas son susceptibles de intercambio entre ellas, así por ejemplo es posible, sustituir una estructura **desde,** por una **mientras**; con incrementos positivos o negativos de la variable índice. En efecto, la estructura **desde** con incremento positivo es equivalente a la estructura **mientras** marcada con la A), y la estructura **desde** con incremento negativo es equivalente a la estructura **mientras** marcada con la B).

A)
```
v ← vi
mientras v < = vf hacer
   <acciones>
   v ← v + incremento
fin_mientras
```

B)
```
v ← vi
mientras v > = vf hacer
   <acciones>
   v ← v - decremento
fin_mientras
```

3.4.3.4 Anidamiento de estructuras repetitivas

En un algoritmo pueden existir varias estructuras repetitivas, siendo sus bucles respectivos anidados o independientes. Se dice que los bucles son anidados cuando están dispuestos de tal modo que unos son interiores a otros. Nótese que, en ningún caso, se admite que sean cruzados, pues su ejecución sería ambigua (Ver Figura 3.14 donde las líneas indican el principio y el fin de cada bucle).

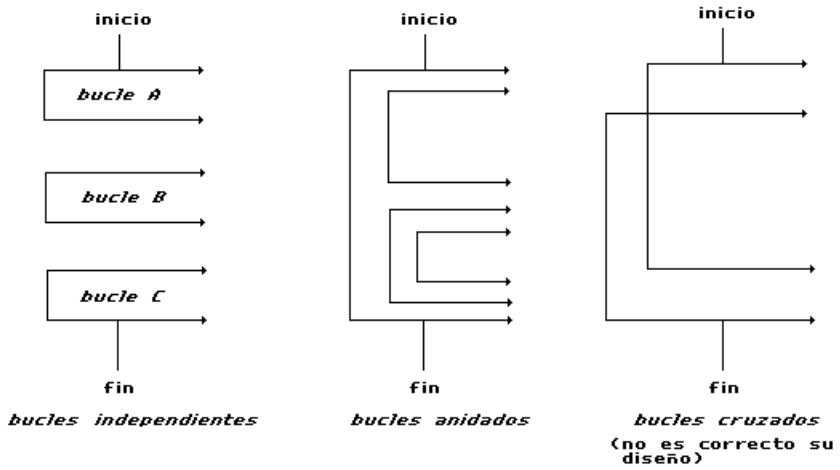


Fig. 3.14. *Posiciones relativas de los bucles*

En las estructuras repetitivas anidadas, la estructura interna debe estar incluida totalmente dentro de la externa, no pudiendo existir solapamiento entre ellas. Las variables índices o de control de los bucles toman valores tales que por cada valor de la variable índice del ciclo externo se ejecuta totalmente el bucle interno.

Ejemplo 10:

Calcular los factoriales de *n* números leídos por el teclado.

El problema consiste en realizar una primera estructura repetitiva de *n* iteraciones del algoritmo de cálculo del factorial, que a su vez se efectúa con una segunda estructura repetitiva.

```
inicio
leer n                  {lectura de la cantidad de números}
desde i = 1 hasta n hacer
    leer NUMERO
    FACTORIAL ← 1
    desde j = 1 hasta NUMERO hacer
          FACTORIAL ←FACTORIAL *j
    fin_desde
    escribir "el factorial del número", NUMERO, "es", FACTORIAL
fin_desde
fin
```

Nótese que cada valor del contador i, el bucle interno cuyo contador es j, se ejecuta totalmente por lo que las variables que sirven de contadores en ambos bucles deben ser distintas. (No es necesario que las estructuras anidadas sean iguales; podríamos haber anidado un *mientras* o un *repetir* dentro del *desde*).

3.4.3.5 Bucles infinitos

A pesar de lo dicho hasta ahora, podemos encontrarnos en la práctica con bucles que no exigen una finalización y otros que no la incluyen en su diseño. Por ejemplo, un sistema de reservas de líneas aéreas puede repetir de forma indeterminada un bucle que permita al usuario añadir o borrar reservas sin ninguna condición de finalización. El programa y el bucle se ejecutan siempre, o al menos hasta que el computador se apaga. En otras ocasiones, un bucle no se termina porque nunca se cumple la condición de salida. Un bucle de este tipo se denomina *bucle infinito o sin fin*. Los bucles infinitos no intencionados, causados por errores de programación, pueden provocar bloqueos en el programa (Ver Ejemplo 11).

Ejemplo 11:

Escribir un algoritmo que permita calcular el interés producido por un capital a las tasas de interés comprendidos en el rango desde 10 a 20 % de 2 en 2 puntos, a partir de un capital dado.

```
leer capital
tasa ←10
mientras tasa < > 20 hacer
      interés← tasa *0.01*capital { tasa*capital/100=tasa*0.01*capital}.
      escribir "interés producido", interés
      tasa ←tasa+2
fin_mientras
escribir "continuación"
```

Los sucesivos valores de la tasa serán 10, 12, 14, 16,18,20, de modo que al tomar 'tasa' el valor 20 se detendrá el bucle y se escribirá el mensaje "continuación". Supongamos que ahora nos interesa conocer este dato de 3 en 3 puntos; para ello se cambia la última línea del bucle por tasa← tasa+3. Ahora los valores que tomará tasa serán 10, 13, 16, 19 saltando a 22 y nunca será igual a 20, dando lugar a un bucle infinito y nunca escribiría "continuación". Para evitarlo deberíamos cambiar la condición de finalización por una expresión del tipo:

tasa < 20 *o bien* tasa ≤ 19

El Ejemplo 11, sugiere además que, a la hora de expresar las expresiones booleanas de la condición del bucle, se utilice mayor o menor en lugar de igualdad o desigualdad.

3.4.3.6 Terminación de bucles con datos de entrada

En el caso, necesariamente muy frecuente, de que leamos una lista de valores por medio de un bucle, se debe incluir algún tipo de mecanismo para terminar la lectura. Existen cuatro métodos para hacerlo:

1.- Simplemente preguntar con un mensaje al usuario, si existen más entradas. Así en el problema de sumar una lista de números el algoritmo sería:

```
inicio
Suma ←0
escribir "existen más números en la lista s/n"
leer Resp {variable Resp, tipo carácter}
mientras Resp = "S" o Resp ="s" hacer
      escribir "numero"
      leer N
      Suma ←Suma +N
      escribir "existen más números (s/n)"
      leer Resp
fin_mientras
fin
```

Este método a veces es aceptable e incluso útil, pero es tedioso cuando trabajamos con grandes listas de números, ya que no es muy aconsejable tener que contestar a una pregunta cada vez que introducimos un dato.

2.- Conocer desde el principio el número de iteraciones que ya ha sido visto en los ejemplos anteriores y que permite emplear a una estructura **desde-hasta**.

3.- Utilizar un valor "centinela", un valor especial usado para indicar el final de una lista de datos. En estos casos es especialmente importante, de cara al usuario, advertir la forma de terminar la entrada de datos, esto es, que valor o valores indican el fin de la lista.

Por ejemplo, supongamos que se tienen unas calificaciones, comprendidas entre 0 y 100; un valor centinela en esta lista puede ser -999, ya que nunca será una calificación válida y cuando aparezca se deberá terminar el bucle. Si la lista de datos son números positivos, un valor centinela puede ser un número negativo que

indique el final de la lista. El siguiente ejemplo realiza la suma de todos los números positivos introducidos desde el teclado:

```
suma ← 0
leer numero
mientras numero >= 0 hacer
      suma ← suma + número
          leer número
fin_mientras
```

Obsérvese la ventaja, en este caso, de utilizar una estructura "mientras", puesto que el último número leído de la lista no se añade a la suma, si es negativo, ya que se sale fuera del bucle.

4.- Por agotamiento de datos de entrada, consiste en simplemente comprobar que no existen más datos de entrada. Ello es posible cuando los datos se obtienen a partir de un fichero, donde se puede detectar el agotamiento de los datos gracias al signo de fin de fichero (EOF "**end of file"**). En este caso la lectura de un EOF supone la finalización del bucle de lectura. En el caso de estar recibiendo datos de otro computador, el cierre de la conexión también supone el agotamiento de los datos de entrada.

3.5. PROGRAMACIÓN MODULAR

Una vez estudiadas las estructuras de control, hemos de ver cómo se implementa la descomposición en módulos independientes denominados *subprogramas o subalgoritmos.* Un subprograma es una colección de instrucciones que forman una unidad de programación, escrita independientemente del programa principal, con el que se asociará a través de un proceso de transferencia, de forma que el control pasa al subprograma en el momento que se requieran sus servicios y volverá al programa principal cuando aquél se haya ejecutado. Esta descomposición nos interesa por dos razones:

1) Esta asociada al diseño descendente en la programación estructurada, ya que un subprograma, a su vez, puede llamar a sus propios subprogramas e incluso a sí mismo, recursivamente.

2) Permite reutilizar un programa dentro de la resolución de otros problemas distintos, puesto que los subprogramas se utilizan por el programa principal para ciertos propósitos específicos, aquellos pueden haber sido escritos con anterioridad para resolver otros problemas diferentes.

El subprograma recibe datos desde el programa y le devuelve resultados. Se dice que el programa principal *llama* o *invoca* al subprograma. Este, al ser llamado, ejecuta una tarea y devuelve el control al programa principal. La invocación puede suceder en diferentes lugares del programa. Cada vez que el subprograma es llamado, el control retorna al lugar desde donde fue hecha la llamada. Algunos subprogramas son tan comunes, que están incluidos en el lenguaje de programación, para ser utilizados directamente en los programas, esto incluye, los propios operadores aritméticos, operaciones sobre cadenas de caracteres, manejo del cursor, etc.

Para poder utilizar esta aproximación, hay que respetar una determinada metodología, denominada **abstración procedimental** y que consta de dos etapas:

1) Asignar un nombre que identifique e independice cada módulo.
2) Parametrizar adecuadamente la entrada y la salida, a fin de que los datos que manda al programa principal puedan ser adecuadamente interpretados por el subprograma y viceversa.

Existen dos tipos de subprogramas: *funciones y rutinas o procedimientos*, aunque no todos los lenguajes distinguen entre ellos. Aquí mantendremos esta distinción, con objeto de facilitar la exposición de la programación modular.

3.5.1 FUNCIONES

Matemáticamente una función es una operación que a partir de uno o más valores, llamados *argumentos*, produce un valor denominado resultado o valor de la función, por medio de ciertas operaciones sobre los argumentos. Consideremos la función:

$$f(x) = \frac{x}{1+x^2}$$

'f' es el nombre de la función y, 'x' es el argumento. Para evaluar 'f' debemos darle un valor a 'x'. Así con x=3 se obtiene el valor 0,3 que se expresa escribiendo: $f(3) = 0,3$.

Una función puede tener varios argumentos. La función F(x,y) = $x + \sqrt{x} - 2y - \sqrt{y}$ es una función con dos argumentos. Sin embargo, solamente un **único valor** se asocia con la función, con independencia del número de argumentos que tenga ésta.

Todos los lenguajes de programación tienen la posibilidad de manejar funciones, bien estén ya incorporadas en origen, por el propio lenguaje, bien sean definidas por el usuario. Las funciones están diseñadas para realizar tareas específicas a

partir de una lista de valores (*argumentos)* y devolver un único valor al programa principal. Se llama o evoca una función, utilizando su nombre en una expresión con los argumentos encerrados entre paréntesis, que deben coincidir en cantidad, tipo y orden con los que la función fue definida en su momento.

Supongamos que durante la ejecución del programa principal nos encontramos con una instrucción, y = raízcua (A + cos (x)). El control, entonces, pasará a evaluar la función raizcua. Al hacerlo, se pasa primero al subprograma (función) coseno y se calcula cos (x). Una vez obtenido A+cos (x) donde A es una constante, este valor se utiliza como argumento de la función **raizcua**, que evalúa el resultado final según las instrucciones que la definan. El resultado correspondiente devolverá al lugar desde donde fue llamada para ser almacenado en la variable 'y' del programa que la ha llamado. Las funciones incorporadas al lenguaje en origen se denominan *funciones internas o intrínsecas* (caso de sen, cos, etc.), mientras que las funciones definidas por el programador se deben codificar mediante *una definición de función* por parte del mismo. Se supone que la relación de funciones internas de cada lenguaje es conocida por el programador que sólo debe llamarlas cuando las necesite, pero no definirlas. El mayor interés para el programador reside por tanto en las funciones definidas por el usuario.

3.5.1.1 Definición de funciones

Una función, como tal subprograma, tiene una constitución similar a un programa. Por consiguiente, constará de una cabecera con el nombre y los argumentos de la función, seguida por el cuerpo de la función, formado por una serie de acciones o instrucciones cuya ejecución produce un valor, el resultado de la función, que se devolverá al programa principal. La expresión de una función es:

función nombre-función(par1,par2,par3...)
  <acciones>
fin función

par1,par2, ,	*lista de parámetros formales o argumentos:* son nombres de identificadores que sólo se utilizan dentro del cuerpo de la función.
nombre-función	nombre asociado con la función, que será un nombre de identificador válido sobre el que se almacenará el resultado.
<acciones>	instrucciones que constituyen la función, y que deben contener al menos una acción que asigne un valor como resultado de la función.

En pseudocódigo antes del final debe aparecer nombre-función ← expresión.

Ejemplo 12:

Escribir la función factorial

```
función factorial (X)
   F ← 1
   desde j=1 hasta X hacer
      F ← F * j
   fin-desde
   factorial ← F
fin
```

Obsérvese que se ha utilizado el identificador que devuelve el valor de la función al acabar la ejecución de la misma, en lugar de haber utilizado este identificador sustituyendo la variable F, con lo que nos ahorraríamos una línea de código. Esto es una exigencia de algunos compiladores.

3.5.1.2 Invocación a las funciones

Una función puede ser llamada sólo *mediante* una referencia directa a la misma de la forma siguiente:

nombre-función (lista de parámetros actuales)

nombre función	función que es invocada.
lista de parámetros actuales	constantes, variables, expresiones, valores de funciones, nombres de subprogramas, que se corresponden con los parámetros formales, que hemos visto durante en la declaración de funciones.

Al invocar una función, hay que pasarle una serie de parámetros, a fin de proporcionarle los argumentos de entrada necesarios para poder ejecutar sus acciones, para ello distinguiremos entre los argumentos de la definición de la función (**parámetros formales** o mudos) y los argumentos utilizados en su invocación (**parámetros actuales**). Cada vez que se llama a una función, se establece sistemáticamente una correspondencia entre parámetros formales y actuales. En consecuencia, debe haber exactamente una correspondencia, tanto en

número como en tipo, de los parámetros actuales con los formales presuponiéndose una correspondencia, uno a uno, de izquierda a derecha entre ambos. Una llamada a la función implica los siguientes pasos:

1. A cada parámetro formal se le asigna el valor real de su correspondiente parámetro actual.
2. Se ejecuta el cuerpo de acciones de la función.
3. Se devuelve el valor de la función y se retorna al punto de llamada.

Ejemplo 13:

Escribir la función: $y = x^n$ (potencia n de x, tanto si es positiva, como negativa) y utilizarla para calcular $1/(2.5)^3$

```
función potencia (x, n)
        inicio                          parámetros formales (real y entero)
  y ← 1                  función interna (valor absoluto)
  desde i = 1 hasta abs(n) hacer
     y ← y * x
  fin_desde
  si n < 0 entonces y ← 1/y
  potencia ← y
fin
```

Su utilización sería:

```
z ← potencia (2.5, -3)
                  parámetros actuales
```

Transferencia: Una vez que los parámetros formales toman los valores de sus correspondientes parámetros actuales x←2.5 , n←3, se ejecuta el cuerpo de la función, devolviendo sus resultados al programa principal, resultando: z←0.064

3.5.2 PROCEDIMIENTOS O SUBRUTINAS

Aunque las funciones son herramientas de programación muy útiles para la resolución de problemas, su alcance está limitado por el hecho de devolver un solo valor al programa principal. Con frecuencia se requieren subprogramas que intercambien un gran número de parámetros, por lo que existen *procedimientos* o *subrutinas,* subprogramas que ejecutan un módulo o proceso específico, sin que ningún valor de retorno esté asociado con su nombre. Un procedimiento se invoca normalmente, desde el programa principal mediante su nombre (identificador) y

una lista de parámetros actuales; en algunos lenguajes se tiene que hacer a través de una instrucción específica **llamar_a** (call). Las funciones devuelven un sólo valor, mientras las subrutinas pueden devolver ninguno o varios.
La forma de expresión de un procedimiento es similar a la de funciones:

procedimiento *nombre(lista de parámetros formales)*
  <acciones>
fin_procedimiento

El procedimiento se llama mediante la instrucción:

[**llamar_a**] nombre (lista de parámetros actuales)

Ejemplo 14:

Usar un procedimiento que escriba en la pantalla tantos signos # como el valor del cociente de sus dos argumentos, y tantos signos $ como el resto de la división entera, escribiendo un programa que muestre del modo gráfico indicado el cociente y el resto de: M/N y de (M+N)/2, usando el procedimiento.

Procedimiento -

```
procedimiento división(DIVIDENDO, DIVISOR)
COCIENTE ← DIVIDENDO/DIVISOR {división entera}
RESTO ← DIVIDENDO - COCIENTE * DIVISOR
desde i=1 hasta COCIENTE hacer
        escribir '#
fin-desde
desde i=1 hasta RESTO hacer
        escribir '$
fin-desde
fin
```

Algoritmo principal -

```
algoritmo aritmética
inicio
    leer M, N
    llamar_a división(M, N)
    llamar_a división(M+N, 2)
fin
```

Como ocurre con las funciones, cuando se llama a un procedimiento, cada parámetro formal se sustituye por su correspondiente parámetro actual. Así, en la primera llamada, se hace DIVIDENDO←M y DIVISOR←N.

En programación modular, es un buen consejo incluir comentarios y documentación que describan brevemente lo que hace la función, lo que representan sus parámetros o cualquier otra información que explique la definición de la función, y que facilite su utilización. Esto es útil, ya que, es normal que un programador cuente con un buen número de funciones, ya codificadas con anterioridad, a las que puede recurrir para la resolución de distintos problemas, durante su trabajo de programación.

3.5.3 Ámbito de las Variables

Para ser consecuente con la abstracción procedimental, en la que se basa la programación modular, hemos de garantizar la independencia de los módulos exigiendo dos condiciones básicas, no siempre fáciles de cumplir:

a) Cada módulo se diseña sin conocimiento del diseño de otros módulos.
b) La ejecución de un subprograma particular no tiene por qué afectar a los valores de las variables que se usan en otros subprogramas.

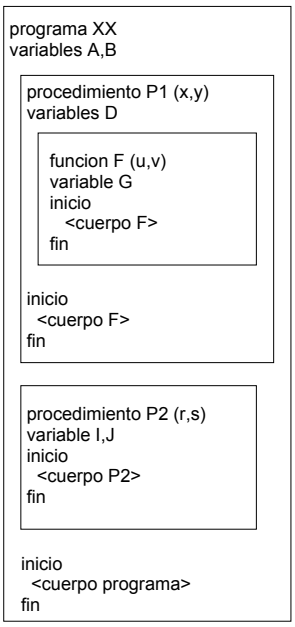
La programación modular permite, en muchos casos, el anidamiento de subprogramas, de forma que el programa principal llama a un subprograma, éste a su vez lo hace a un segundo que puede ser el mismo y así sucesivamente. En cada una de estas llamadas se establece la necesaria correspondencia entre parámetros actuales del programa llamante y los parámetros formales del programa invocado. En este proceso de ejecución de procedimientos anidados, es necesario considerar el ámbito de actuación de cada identificador, ya que finalmente en el programa ejecutable se unirán en un bloque de código único; al juntarse, programa principal y sus subprogramas, que han sido escritos de forma independiente, puede plantear problemas relacionados con los identificadores de variables utilizados en las distintas partes del código. Para resolver este problema, las variables utilizadas en los programas principales y subprogramas se clasifican en dos tipos:

Una **variable local** es aquella que está declarada y es utilizada dentro de un subprograma y es distinta de las posibles variables que con el mismo nombre, pueden estar declaradas en cualquier otra parte del programa. Por tanto su significado y uso se confina al procedimiento o función en que está declarada; esto significa que cuando otro subprograma utiliza el mismo nombre, en realidad se refiere a una posición diferente en memoria. Se dice entonces que tales variables son locales al subprograma en el que están declaradas. Si un subprograma asigna un valor a una de sus variables locales, este valor no es accesible al resto de subprogramas, a menos que demos instrucciones en sentido contrario.

Una **variable global** es aquella que está declarada para el programa o algoritmo completo, esto es, tiene validez para el programa principal y todos sus subprogramas. Las variables globales tienen la ventaja de permitir a los diferentes subprogramas compartir información, sin que tengan que figurar en las correspondientes listas de transferencia de parámetros entre programas.

El uso de variables locales tiene ventajas, siendo la más importante el hecho de facilitar el uso de subprogramas independientes, siempre que la comunicación entre el programa principal y los subprogramas funcione adecuadamente, a través de las listas de parámetros actuales y formales. Para utilizar un procedimiento, ya utilizado y comprobado, sólo necesitamos conocer que hace, sin tenernos que preocupar por los detalles de su codificación interna (¿Cómo lo hace?). Esta característica hace posible por un lado dividir grandes proyectos en módulos autónomos que pueden codificar diferentes programadores que trabajan de forma independiente y por otro facilitan la reusabilidad del software.

Llamaremos **ámbito** (scope) de un identificador, sea variable, constante o procedimiento, a la parte del programa donde se le reconoce como tal. Si un procedimiento está definido localmente respecto a otro procedimiento, tendrá significado sólo dentro del ámbito de ese procedimiento. A las variables les sucede lo mismo; si están definidas localmente dentro de un procedimiento, su significado o uso se confina a cualquier función o procedimiento que pertenezca a esa definición. Los lenguajes que admiten variables locales y globales suelen tener la posibilidad explícita de definir dichas variables como tales en el cuerpo del programa, o lo que es lo mismo, ofrecen la posibilidad de definir su ámbito de actuación. Para ello se utilizan las cabeceras de programas y subprogramas, donde figuran declaradas las variables, de donde se puede extraer el ámbito correspondiente de cada una. (Ver Figura 3.15)



Ámbito	Identificador
Cuerpo del programa	A, B, P1, P2
Cuerpo de P1	A, B, D, x, y, F, P1, P2
Cuerpo de F	A, B, D, G, x, y, u v, F, P1
Cuerpo de P2	A, B, I, J, r, s, P1, P2

Fig. 3.15 *Ejemplo de ámbito de definición de variables*

Las variables definidas en un ámbito son accesibles en él y en todos sus procedimientos interiores. Así la Figura 3.15 muestra el esquema de un ejemplo de un programa principal con diferentes subprogramas y el ámbito correspondiente de sus identificadores, señalando las partes del programa a las que tienen acceso los diferentes identificadores (variables, procedimientos y funciones). Nótese que es perfectamente posible que P2 sea accesible por P1 y viceversa, y que un subprograma sea accesible por sí mismo, (a esta posibilidad le dedicaremos la sección siguiente: recursividad).

Ejemplo 15:

Considérese el siguiente programa y ejecútese paso a paso:

```
algoritmo XXX
    {variable global A}
    {variables locales: X, Y}
inicio
    X← 5
    A← 10
    Y← F(X)
    escribir X,A,Y
fin {algoritmo}
función F(N)
```

```
{comienzo función}
{variable global A}
{variable local  X}
inicio
    A← 6
    X←12
    F←N+5+A
fin {función}
```

A la variable global A se puede acceder desde el algoritmo XXX y desde la función F. Sin embargo, X representa a dos variables distintas: una local al algoritmo que sólo se puede acceder desde él y otra local a la función. Al ejecutar el algoritmo XXX se obtendrían los siguientes resultados:

1) X←5, A←10, Y←F(5), la invocación de la función F(N) da lugar al paso del parámetro actual X al parámetro formal N.

2) Al ejecutarse la función, se hace N←5 ya que el parámetro actual en la llamada del programa principal F(X) se asigna al parámetro formal N.

   A←6, cosa que modifica el valor de A en el algoritmo principal por ser A global.

   X←12 sin que se modifique el valor X en el algoritmo principal, porque X es local.

   F←16 (5+5+6)

3) El resultado de la función se almacena en Y, de forma que Y←16.

4) El resultado final, es que se escriba la línea

   5 6 16

   ya que X es el valor de la variable local X en el módulo principal; A toma el valor de la última asignación (que fue dentro de la función) e Y toma el valor de la función F(X).

Del Ejemplo 15 sacamos la conclusión de que, en general, toda la información que intercambia un programa con sus subprogramas, debe vincularse a través de la lista de parámetros, y no mediante variables globales. Si se usa este último método, se corre el riesgo de obtener resultados inesperados, indeseables en muchos casos, llamados *efectos laterales*; En el Ejemplo 15, posiblemente en contra de nuestros deseos, la variable global A ha cambiado de valor al ejecutarse la función, cosa que no ha ocurrido con X.

3.5.4 Paso de parámetros

Los parámetros que intervienen en programación modular pueden jugar distintos papeles:

entrada: Proporcionan valores desde el programa que llama y que se utilizan dentro de un subprograma. Las entradas de un subprograma se dice, en general, que son sus argumentos.
salida: Corresponden a los resultados del subprograma. En el caso de una función el valor de retorno, se utiliza como salida.
entrada/salida: Un mismo parámetro se utiliza tanto para mandar argumentos, como para devolver resultados.

Como veremos a continuación, el papel que juega cada parámetro no es suficiente para caracterizar su funcionamiento ya que los distintos lenguajes de programación utilizan distintos métodos para pasar o transmitir parámetros, entre un programa y sus subprogramas. Es el correcto intercambio de información, a través de parámetros, quien hace que los módulos sean independientes, siendo preciso conocer el método utilizado por cada lenguaje particular, ya que esta elección afecta a la propia semántica del lenguaje. Dicho de otro modo, un mismo programa puede producir diferentes resultados, bajo diferentes sistemas de paso de parámetros. Veamos los principales métodos de transmisión de parámetros.

3.5.4.1 Paso por valor

El paso por valor se utiliza en muchos lenguajes de programación, la razón para ello es la analogía que presenta con los argumentos de una función, donde los valores se proporcionan para el cálculo de resultados. Los parámetros se tratan como variables locales de forma que los parámetros formales reciben como valores iniciales los valores de los parámetros actuales. A partir de ellos se ejecutan las acciones descritas en el subprograma. Otra característica es que no tiene relevancia el que los argumentos sean variables, constantes o expresiones, ya que sólo importa el valor de la expresión que constituye el argumento o parámetro formal.

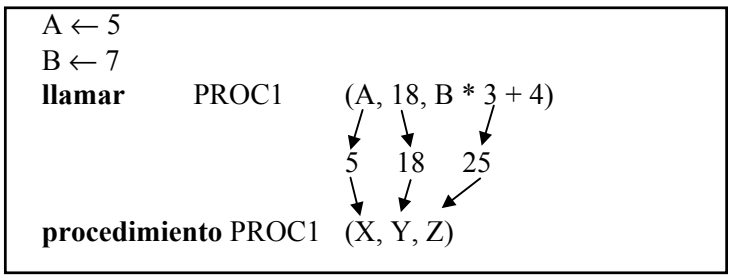


Fig. 3.16 *Ejemplo de paso por valor*

La Figura 3.16 muestra el mecanismo de paso por valor de un procedimiento con tres parámetros, que se resume así. En el momento de invocar a PROC1 la situación es ésta:

Primer parámetro: valor de la variable A = 5;
Segundo parámetro: valor constante = 18;
Tercer parámetro: valor de la expresión B* 3+4 = 25.

El paso por valor asigna estos tres, respectivamente a los parámetros formales X, Y, Z al iniciar la ejecución del procedimiento PROC1.

Aunque el paso por valor es sencillo, tiene una limitación importante: no existe ninguna posibilidad de otra conexión con los parámetros actuales, y por tanto los cambios que se produzcan por efecto del subprograma no producen cambios en los argumentos originales. Por consiguiente, no se pueden devolver valores de retorno al punto de llamada. Es decir, todos los parámetros son solo de entrada y el parámetro actual no puede modificarse por el subprograma, ya que cualquier cambio realizado en los valores de los parámetros formales durante la ejecución del subprograma se destruye cuando finaliza éste. Así, en el ejemplo de la figura 3.16, aunque X, Y, Z variasen en el interior del procedimiento, A y B seguirían valiendo 5 y 7, en el programa principal.

Esta limitación, no obstante, constituye al mismo tiempo una ventaja en determinadas circunstancias, ya que asegura que las variables de un módulo no serán modificadas al invocar a una subrutina o función, hagan éstas lo que hagan.

Ejemplo 16:

Escribir un programa y una función que por medio del paso por valor obtenga el máximo común divisor de dos números.

```
algoritmo Maximo_comun_divisor
inicio
  leer (x, y)
  m ← mcd (x, y)
  escribir (x, y, m)
fin
funcion mcd(a, b): entero        {a, b: enteros}
inicio
  mientras a <> b hacer
      si a > b
          entonces a ← a - b
          sino  b ← b - a
      fin_si
```

```
    fin_mientras
    mcd ← a
fin
```

Al ejecutarse este algoritmo se producirán los siguientes resultados:

Variable del algoritmo			*Variables de la función*		
x	y	m	a	b	mcd
10	25		10	25	
			10	15	
			10	5	
		5	5	5	5

Obsérvese que el paso de mcd desde la función al programa principal es posible gracias a que ambos comparten el identificador mcd.

3.5.4.2 Paso por referencia

En numerosas ocasiones se requiere que ciertos parámetros sirvan como parámetros de salida o de entrada/salida, es decir, se devuelvan los resultados a la unidad o programas que llama al subprograma. Este método se denomina paso por referencia. El programa que llama pasa al subprograma llamado la dirección de memoria donde se halla almacenado el parámetro actual (que está en el ámbito de la unidad llamante), es decir, se pasa una referencia a la variable, no el valor que contiene. Toda referencia al correspondiente parámetro formal se trata como una referencia a la variable, cuya dirección se ha pasado. Con ello, una variable pasada como parámetro actual es compartida, es decir, también se puede modificar directamente por el subprograma (además de utilizar su valor). Si comparamos la Figura 3.16 con 3.17, observamos que en esta última los parámetros formales se utilizan para referirse a las posiciones de memoria de los parámetros actuales. Por tanto utilizamos esta referencia para pasar información de entrada y/o salida. Nótese que un cambio en el parámetro formal durante la ejecución del subprograma, se refleja en un cambio en el correspondiente parámetro actual (ver Ejemplo 17), ya que ambos se refieren a la misma posición de memoria.

```
A ← 5
B ← 7
C ← B * 3 + 4
llamar_a PROC1(  A,      B,      C)
                 ┌─┐     ┌─┐     ┌──┐      direcciones o
                 │5│     │7│     │25│      posiciones
                 └─┘     └─┘     └──┘      de memoria
procedim PROC1(REF X,  REF Y,  REF Z)
```

Fig. 3.17. *Paso por referencia*

Cuando un parámetro se pase por valor lo denotaremos, en pseudocódigo poniendo REF delante del parámetro formal.

Ejemplo 17:

Dado el siguiente esquema de programa, analizar las dos llamadas al procedimiento, asumiendo que el paso de parámetros se efectúa por referencia.

```
programa XYZ
{parámetros actuales a, b, c y d paso por referencia}

procedimiento PP(REF x, REF y)
   inicio          {procedimiento}
      {proceso de los valores de x e y}
   fin

inicio       {programa}
   ...
(1)   prueba (a,c)
   ...
(2)   prueba (b,d)
   ..
fin
```

La primera llamada en (1) conlleva que los parámetros formales *x* e *y* se refieran a los parámetros actuales *a* y *c*. Si los valores de *x* o *y* se modifican dentro de esta llamada, los valores de *a* o *c* en el algoritmo principal también lo habrán hecho. De igual modo, cuando *x* e *y* se refieren a *b* y *d* en la llamada (2) cualquier modificación de *x* o *y* en el procedimiento afectará también a los valores de *b* o *d* en el programa principal.

Nótese, por tanto, que mientras en el paso por valor los parámetros actuales podían ser variables, constantes o expresiones (pues lo que se pasa es su valor), cuando un parámetro se pasa por referencia el parámetro actual indicado en la invocación debe ser siempre una variable que es la forma de referirse a una posición de memoria.

3.5.4.3 Comparaciones de los métodos de paso de parámetros.-

Para examinar de modo práctico los diferentes métodos, consideremos los ejemplos 18 y 19, en los que podemos observar los diferentes valores que toman los parámetros del mismo programa según sea el método de paso que utilicemos

Ejemplo 18:

Consideremos los siguientes algoritmos y el valor que toman sus variables:

```
algoritmo ABC
inicio
  A ←3
  B ← 5
  C ←17
  llamar_a SUB(A, B, C)
  escribir A,B,C
fin

procedimiento SUB(x,y,z)

  inicio
    x ← x+1
    z ← x+y
  fin
```

(a)

```
algoritmo ABC
inicio
  A ←3
  B ← 5
  C ←17
  llamar_a SUB(A, B, C)
  escribir A,B,C
fin

procedimiento SUB(REF x,REF y,REF z)
  inicio
    x ← x+1
    z ← x+y
  fin
```

(b)

a*)* Paso por valor:
No se transmite ningún resultado desde SUBR, por consiguiente ni A, ni B, se modifican y se escribirá 3, 5, 17, al ejecutarse el programa ABC.
b) Paso por referencia:
Al ejecutar el procedimiento ejecutará:

x=x+1=3+1=4 (almacenado en la posición de A)
z=x+y=4+5=9 (almacenado en la posición de C)

Por lo tanto, el programa escribirá 4, 5, 9.

Es importante señalar que un mismo subprograma puede tener simultáneamente parámetros pasados por valor y parámetros pasados por referencia, como podemos ver en el Ejemplo 19.

Ejemplo 19:

Consideremos el siguiente programa M con un subprograma N que tiene dos parámetros formales: *i* pasado por valor, y *j*, por referencia.

```
programa M
inicio
  A ← 2
  B ← 3
  llamar_a N(A,B)
  escribir A,B
fin     {programa M}

procedimiento N(i, REF j)
{ i por valor, j por referencia}
inicio
  i ← i+10
  j ← j+10
  escribir i , j
fin     {procedimiento N}.
```

Al ejecutar el programa M, se escribirán: A y B en el programa principal e i y j en el procedimiento N. Como i es pasado por valor, se transmite el valor de A a i, es decir, i=A=2. Cuando i se modifica (por efecto de la instrucción, i ← i+10) a 12 , A no cambia, y por consiguiente a la terminación de N, A sigue valiendo 2. El parámetro B se transmite por referencia. Al comenzar la ejecución de N, el parámetro j se refiere a la variable B, y cuando se suma 10 al valor de j, el valor de B se cambia a 13. Cuando los valores i, j se escriben en N, los resultados son: 12 y 13.

Cuando retornamos al programa M y se imprimen los valores de A y B, sólo ha cambiado el valor B. El valor de i=12 se pierde en N cuando éste termina. El valor de j también se pierde, pero éste es una dirección, no el valor 13. Se escribirá como resultado final (de la instrucción **escribir** A,B) 2, 13.
Señalemos finalmente que muchos lenguajes de programación permiten pasar parámetros por valor y por referencia, especificando cada modalidad. La elección entre un método u otro viene determinado por cada programa en particular y por el objetivo de evitar efectos laterales no deseados.

3.6. CONCEPTO DE PROGRAMACIÓN ESTRUCTURADA

Cuando introdujimos las estructuras de control indicamos que la orientación no excluyente de este texto iba hacia la programación estructurada, sin detallar en qué consistía este enfoque. Ahora estamos en condiciones de hacerlo.

Llamaremos programa **propio** a un programa que cumpla las siguientes características:

1.- Posee un solo punto de entrada y uno de salida.
2.- Existen caminos que van desde la entrada hasta la salida, que pasan por todas las partes e instrucciones del programa.
3.- Todas sus instrucciones son ejecutables y no presenta bucles infinitos.

Desde 1966, gracias a diferentes trabajos teóricos, sabemos que un programa propio puede ser escrito utilizando solamente los tres tipos de estructura de control que hemos analizado (secuenciales, selectivas, repetitivas). Llamaremos **programación estructurada** a un conjunto de técnicas de programación que incluye :

- Uso del diseño descendente.
- Descomposición modular, con independencia de los módulos.
- Utilización de las tres estructuras de control citadas.
-

La programación estructurada, gracias a la utilización exclusiva de las tres estructuras de control, permite realizar fácilmente programas legibles.

Sin embargo, en determinadas circunstancias, parece necesario realizar bifurcaciones incondicionales, no incluidas en las estructuras de control estudiadas, para lo que hay que recurrir a la instrucción **ir_a** (*goto*).

FORTRAN	BASIC
GO TO etiqueta	GOTO nº de linea
PASCAL	C
goto etiqueta	GOTO etiqueta

Esta instrucción siempre ha sido problemática para los programadores y se recomienda evitar su utilización. Aunque **ir_a** es una instrucción incorporada por todos los lenguajes de programación, existen algunos lenguajes como FORTRAN y BASIC que dependen más de ella que otros. En general, no existe ninguna necesidad de utilizarla, ya que cualquier algoritmo o programa escrito con

instrucciones **ir_a** se puede reescribir de forma que lleve a cabo las mismas tareas prescindiendo de bifurcaciones incondicionales. Sin embargo, esta instrucción es útil en algunas situaciones excepcionales, como en ciertas salidas de bucles o cuando se encuentra un error u otra condición brusca de terminación. La instrucción **ir_a** puede ser utilizada para saltar directamente al final de un bucle, subprograma o procedimiento; la instrucción a la cual se salte debe tener una etiqueta o numero de referencia. Por ejemplo, un programa puede diseñarse de forma que finalice cuando se detecte un determinado error, de la forma:

```
si <condición error>
        entonces ir a  100
          .
100 fin
```

En cualquier caso y para preservar las ventajas de la programación estructurada, conviene saber que en los lenguajes más desarrollados existen instrucciones específicas de **ir_a_fin_de_bucle**, **continuar_bucle** e **ir_a_fin_de_rutina** para evitar usos indiscriminados de los saltos incondicionales.

3.7. RECURSIVIDAD

Entenderemos por recursividad la propiedad que tienen muchos lenguajes de programación de que los subprogramas puedan llamarse a sí mismos. Esta idea es interesante pues es una alternativa a la repetición, si nos enfrentamos a problemas con "estructura de solución recursiva". Esta estructura recursiva se presenta en situaciones como la siguiente: Sea un problema A; al diseñar su algoritmo conseguimos dividirlo en dos partes B y C. Si una de estas partes, por ejemplo C, es formalmente idéntica a A, el problema es recursivo ya que la resolución de A se basa en C, que a su vez se descompone de la misma forma y así sucesivamente. En términos de programación esto supone utilizar un procedimiento para computar el problema, que va a llamarse a sí mismo para resolver parte de ese problema. La potencia de la recursión reside en la posibilidad de definir un número potencialmente infinito de operaciones de cálculo mediante un subprograma recursivo finito. Para que esta solución sea aceptable debe haber un momento en que ya no sea necesario que el procedimiento se llame a sí mismo ya que si la recursión continuara por siempre, no hablaríamos de un genuino algoritmo.

Insistamos que la solución recursiva no es nunca única, ya que siempre es posible utilizar una estructura repetitiva, en lugar del planteamiento recursivo. No obstante, en algunos casos la solución recursiva es la solución natural del problema, y por tanto la más clara. Veamos algunos ejemplos:

Ejemplo 20:

Obtener una función recursiva que calcule el factorial de un entero no negativo. Este problema ya ha sido resuelto de forma no recursiva, ahora lo haremos utilizando la condición recursiva: n! = n * (n-1) ! si n = > 0 y n! = 1 si n = 0

```
función Factorial (n)
inicio
   si n = 0
      entonces Factorial ←1
      sino Factorial ← n * Factorial (n-1)
   fin_si
fin
```

Para demostrar cómo esta versión recursiva de FACTORIAL calcula el valor de n!, consideremos el caso de n = 3. Un proceso gráfico se representa en la Figura 2.18.

Los pasos sucesivos extraídos de la Figura 2.18 son:

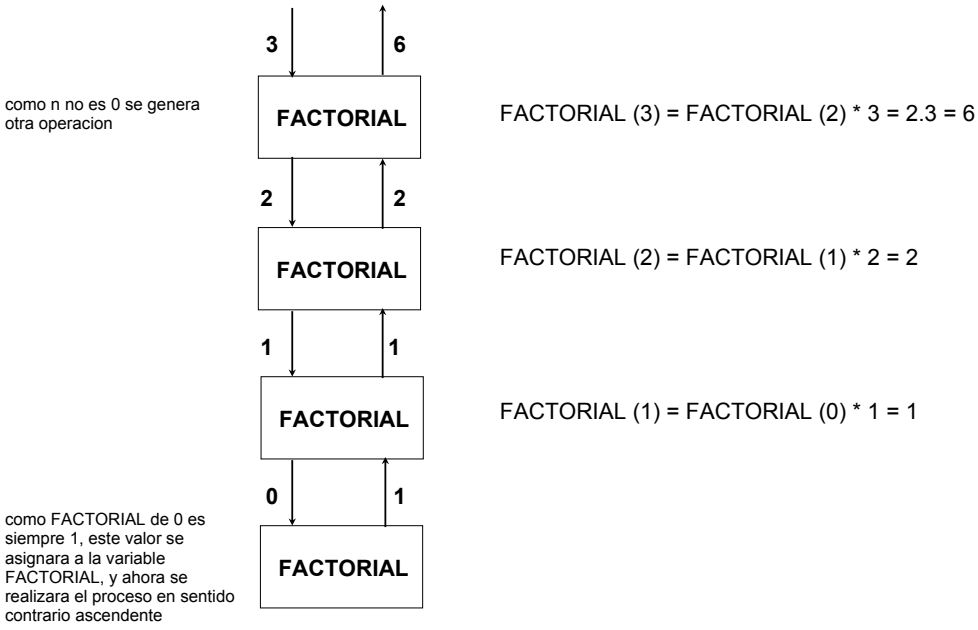


Fig. 2.18. *Secuencia para el cálculo recursivo de factorial de n*

Ejemplo 21:

Leer una lista de números de un fichero y escribirlos en orden inverso.

Para resolver este problema se podrían leer todos los números, invertir su orden y escribirlos. No obstante, se podría pensar en una solución más sencilla si se hace una descomposición recursiva del problema. El problema se podría resolver con el siguiente algoritmo:

```
algoritmo escribir_en_orden_inverso
leer N                  { Primer número }
leer el resto de los números y escribir_ en_orden_inverso
escribir N {El primer número se escribe el último }
```

El algoritmo consta de tres acciones, una de las cuales -la segunda- es semejante al problema inicial. La necesidad de llamar al procedimiento desaparece cuando no quedan más números por leer, esto es, cuando se ha llegado al fin de fichero (FF). El algoritmo quedará, por tanto, como:

```
algoritmo INVERTIR
inicio
  leer N
  si (N <> FF) entonces          { hay más números }
     INVERTIR
     escribir N
fin
```

Observar que el primer número que se escribirá es el último leído, después de que el número de llamadas a INVERTIR, sea igual a la cantidad de números que contiene el archivo. Consecuentemente, el primer número leído será escrito en último lugar, después de que la llamada recursiva a INVERTIR haya leído y escrito el resto de los números. Esto supone que el programa, al ejecutarse, deberá guardar tantas copias del subprograma y sus datos como sean necesarios.

La forma como hace esto el lenguaje de programación no debe importarnos, salvo que este proceso sature la memoria y aparezca el correspondiente error.

Ejemplo 22:

Escribir una función que calcule recursivamente el término n-esimo de la serie de Fibonacci, dada por la expresión:

fib(1) = 1
fib(2) = 1

fib(n) = fib(n-1) + fib(n-2) para n>2

Esta serie fue concebida originalmente como modelo para el crecimiento de una granja de conejos (proceso claramente recursivo) por el matemático italiano del siglo XVI, Fibonacci. Como no podía ser menos, esta serie crece muy rápidamente. Como ejemplo, el término 15 es 610.

```
función FIBONACCI(n)
inicio
   si (n=1) o (n=2)
      entonces
         FIBONACCI ← 1
      sino
         FIBONACCI ← FIBONACCI (n-2) + FIBONACCI (N-1)
   fin_si
fin_función
```

La recursión es una herramienta muy potente y debe ser utilizada como una alternativa a la estructura repetitiva. Su uso es particularmente idóneo en aquellos problemas que pueden definirse de modo natural en términos recursivos. El caso de la función factorial es un ejemplo claro.

Insistamos en que, para evitar que la recursión continúe indefinidamente, es preciso incluir una condición de terminación. Como veremos en el capítulo siguiente, la posibilidad de implementar la recursividad se debe a la existencia de estructuras de datos específicos del tipo pila.

3.8. DESARROLLO Y GENERACIÓN DEL SOFTWARE

Una vez hemos visto cómo se podían implementar las estructuras modulares y de control sobre las que descansa la programación estructurada, volvamos al problema de la resolución de un problema por medio de un ordenador. De acuerdo con lo que allí afirmamos, cuando se ha comprobado el programa, éste se podría dar, en principio, por terminado y el problema abordado como resuelto. Sin embargo, un programa no puede darse como totalmente terminado hasta que no ha sido depurado y contrastado con toda una batería de datos de entrada distintos. Además, una vez se está razonablemente seguro de su funcionamiento, debe documentarse para que pueda ser utilizado por cualquier usuario, al tiempo que hay que tomar toda una serie de medidas que faciliten su actualización y

mantenimiento, ya que un programa, siempre debe estar en condiciones de mejorar sus prestaciones y de adaptarse a pequeños cambios, sin tener que proceder a su reescritura completa. Aunque estas últimas fases están fuera de la óptica de este capítulo, conviene retener la totalidad de fases que constituyen el proceso completo de generación del software y que se esquematiza en la Figura 3.18.

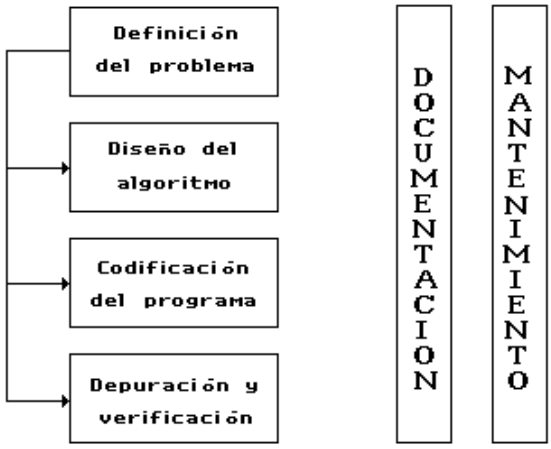


Fig. 3.18 *El proceso completo de programación*

Para completar el capítulo, demos algunas ideas relacionadas con la programación de aplicaciones complejas, que dan lugar a la generación de una gran cantidad de líneas de código.

3.8.1 INGENIERÍA DEL SOFTWARE

Conviene saber que la tarea de diseño y construcción de un programa puede ser, en algunos casos, bastante compleja. En las aplicaciones reales, no es usual que la resolución de un problema implique el desarrollo de un solo programa. Normalmente cualquier aplicación práctica requiere de más de un programa individual y conviene considerar alguna de las características que hay que tener en cuenta al enfrentarse a ellas desde un punto de vista profesional:

A) Volumen: un proyecto, en la práctica, suele ser grande (decenas de miles de líneas de código) lo que presenta problemas de:

- *Complejidad*: El software es complejo y resulta difícil que una única persona conozca todos los detalles de una aplicación.
- *Coordinación*: En el desarrollo de una aplicación intervienen muchas personas. Hay que coordinar el trabajo de todos, de forma que al final los distintos componentes y módulos encajen.

B) Evolución: El software no es estático. Evoluciona con las necesidades del usuario, los cambios del entorno (nuevo hardware, nueva legislación, nuevas necesidades, ampliaciones, etc.) deben ser tenidos en cuenta.

C) Comunicación: Cuando se desarrolla una aplicación es porque, de una forma u otra, hay alguien interesado en usarla. Antes de comenzar el desarrollo habrá que concretar con el futuro usuario las características de la aplicación. Esta comunicación conlleva problemas, pues normalmente el informático no conoce en profundidad las necesidades del cliente y éste no sabe discernir la información que es útil para el informático de la que no lo es.

La ingeniería del software se ocupa del estudio de estos problemas y de sus soluciones. Se puede definir la **ingeniería del software** como la disciplina que se ocupa del establecimiento y uso de principios de ingeniería para obtener software económico que sea fiable y funcione eficientemente en las máquinas que estén disponibles en cada momento. La ingeniería del software se ocupa de: la planificación y estimación de proyectos, análisis de requisitos, diseño del software, codificación, prueba y mantenimiento. Para realizar estas tareas propone una serie de métodos, como cualquier otra ingeniería. No obstante, existen grandes diferencias con éstas, la principal de las cuales procede de la ausencia del proceso de fabricación que se da en el caso del desarrollo de software. Cuando se construyen automóviles, una vez diseñados se fabrican. La ausencia de proceso de fabricación en la ingeniería software hace que el coste de la producción sea fundamentalmente el coste del diseño.

Por otra parte, el mantenimiento del software es muy diferente al del hardware (o cualquier otro sistema físico en ingeniería), ya que el software no se desgasta con el tiempo. El mantenimiento del software tiene que ver con la detección de fallos o con la necesidad de adaptarlo a unas nuevas circunstancias. En ambos casos el proceso de mantenimiento no consiste en la sustitución de un componente por otro, sino en la repetición de parte del proceso de desarrollo.

3.8.2 CICLO DE VIDA DEL SOFTWARE

La creación de cualquier sistema software implica la realización de tres pasos genéricos: definición, desarrollo y mantenimiento.

En la **fase de definición** se intenta caracterizar el sistema que se ha de construir. Esto es, se trata de determinar qué información ha de usar el sistema, qué funciones ha de realizar, qué condiciones existen, cuáles han de ser las interfaces del sistema

y qué criterios de validación se usarán. En resumen; se debe contestar detalladamente a la pregunta: ¿qué hay que desarrollar?.

En la **fase de desarrollo** se diseñan las estructuras de datos (bases de datos o archivos) y de los programas; se escriben y documentan éstos y se prueba el software.

La **fase de mantenimiento** comienza una vez construido el sistema, coincidiendo con su vida útil. Durante ella, el software es sometido a una serie de modificaciones y reparaciones.

La detección de errores durante la definición y el desarrollo se realiza mediante revisiones de la documentación generada en cada fase. En estas revisiones se pueden detectar fallos o inconsistencias que obliguen a repetir parte de la fase o una fase anterior. Este esquema general se puede detallar más, dando lugar a modelos concretos del ciclo de vida del software. Dos son los paradigmas usados más frecuentemente: el *ciclo de vida clásico* y el de *prototipos*.

3.8.2.1 Ciclo de vida clásico

El ciclo de vida clásico consta de fases, que siguen un esquema en cascada, análogo al esquema general que acabamos de ver. Este paradigma contempla las siguientes fases:

Análisis del sistema: El software suele ser parte de un sistema mayor formado por hardware, software, bases de datos y personas. Por él, se debe comenzar estableciendo los requisitos del sistema, asignando funciones a los distintos componentes y definiendo las interfaces entre componentes.

Análisis de los requisitos del software: Antes de comenzar a diseñar el software se deben especificar las funciones a realizar, las interfaces que deben presentarse y todos los condicionantes, tales como rendimiento, utilización de recursos, etc. Como fruto de este análisis se genera un documento conocido como **especificación de requisitos del software.**

Diseño: El diseño del software consiste en construir una estructura para el software que permita cumplir los requisitos con la calidad necesaria. El diseño concluye con la confección de un documento de diseño. A partir de él, se debe responder a la pregunta ¿cómo se ha de construir el sistema?.

Codificación: Consiste en plasmar el diseño en programas, escritos en un lenguaje de programación adecuado.

Prueba: Cuando se han escrito los programas es necesario probarlos. En las pruebas se debe comprobar que los programas se corresponden con el diseño, realizan correctamente sus funciones y que el sistema satisface los requisitos planteados. Es decir, se ha de contestar a la pregunta ¿se ha construido el sistema que se deseaba?.

Mantenimiento: Esta fase se corresponde con la fase de mantenimiento del esquema general. Dependiendo de la naturaleza y motivación de cada operación de mantenimiento concreta, será necesario revisar desde la codificación, desde el diseño o desde la fase de análisis.

El conjunto de la documentación generada en todas las fases constituye la configuración del software. Un buen sistema no es solamente un conjunto de programas que funcionan, sino también una buena documentación. La documentación es fundamental para un buen desarrollo y es esencial en el proceso de mantenimiento del software.

3.8.2.2 Ciclo de vida de prototipos

Hay situaciones en que no es posible usar el ciclo de vida clásico, fundamentalmente debido a la dificultad de establecer los requisitos del software a priori. En estas situaciones es posible seguir el modelo de **ciclo de vida de prototipos**. En esencia, este paradigma se basa en la construcción de un prototipo durante las primeras etapas del ciclo de vida. Un prototipo es un modelo "a escala reducida" del sistema que se va a construir. El prototipo incorpora sólo los aspectos relevantes del sistema y se utiliza como ayuda en la especificación del software, sirviendo como base tangible sobre la que discutir con el usuario final.

El ciclo de vida comienza con la realización de un breve análisis de los requisitos, tras el cual se diseña y codifica el prototipo. Sobre el prototipo se discuten y detallan las especificaciones, modificando el prototipo de acuerdo con éstas, siguiendo un proceso cíclico. El resultado de este proceso es el documento de especificación de requisitos del software. Normalmente se desecha posteriormente el prototipo, diseñándose el sistema definitivo. A partir de este punto se puede seguir el mismo esquema que un ciclo clásico.

Es posible que el lector quede abrumado por toda esta metodología cuando se está introduciendo en las técnicas de programación. Sin embargo, es posible que estas reflexiones le puedan ser de utilidad cuando se enfrente a la tarea de programar algún problema no trivial y se encuentre con que algo no funciona como debe. Construir software es una ingeniería y hemos tratado de esbozar sus normas.