

Tema II

Programación estructurada

Capítulo 6

Instrucciones estructuradas

6.1	Composición de instrucciones	86
6.2	Instrucciones de selección	88
6.3	Instrucciones de iteración	94
6.4	Diseño y desarrollo de bucles	103
6.5	Dos métodos numéricos iterativos	113
6.6	Ejercicios	117

En el capítulo anterior se ha visto una breve introducción a Pascal donde se han presentado algunos tipos de datos e instrucciones básicas. Hasta ahora todos los ejemplos estudiados han sido de estructura muy simple: cada instrucción se ejecuta una sólo vez y además en el mismo orden en el que aparecen en el listado del programa.

Para escribir programas que traten problemas más arduos es necesario combinar las acciones primitivas para producir otras acciones más complejas. Este tipo de *acciones combinadas* se componen a partir de otras, más sencillas, mediante tres métodos fundamentales: la *secuencia o composición*, la *selección* y la *repetición*. Estos tres métodos se describen informalmente como sigue:

- La forma más simple de concatenar acciones es la *composición*, en ella se describe una tarea compleja como una sucesión de tareas más elementales.
- La *selección* de una alternativa tras valorar una determinada circunstancia se refleja mediante las instrucciones **if** (en sus dos formas) y **case**.

- Finalmente, las instrucciones repetitivas (**while**, **for** y **repeat**) permiten expresar en Pascal la *repetición* de acciones, ya sea un número de veces prefijado o no.

En los siguientes apartados estudiamos cada una de las construcciones anteriores junto con métodos que permiten estudiar su corrección.

6.1 Composición de instrucciones

En bastantes ocasiones una tarea concreta se especifica como una serie de tareas que se ejecutan secuencialmente. Por ejemplo, si algún día alguien nos pregunta cómo llegar a algún sitio, la respuesta podría ser parecida a ésta:

1. *Tuerza por la segunda a la derecha.*
2. *Siga caminando hasta un quiosco.*
3. *Tome allí el autobús.*

En el capítulo anterior se usó, aún implícitamente, la composición de instrucciones simples para obtener una acción más compleja; en este apartado se presenta su estudio completo.

En Pascal la composición de instrucciones se realiza concatenando las instrucciones y separándolas con el carácter punto y coma (;). La construcción de una instrucción compleja como una sucesión de instrucciones simples se muestra en el siguiente segmento de programa, que intercambia los valores de dos variables numéricas *a* y *b* sin hacer uso de ninguna variable auxiliar:

```
begin
  a:= a + b ;
  b:= a - b ;
  a:= a - b
end
```

Una *composición* de instrucciones indica que las instrucciones citadas son ejecutadas secuencialmente siguiendo el mismo orden en el que son escritas. El diagrama sintáctico de una instrucción compuesta aparece en la figura 6.1, y su descripción usando notación EBNF (véase [PAO94], pg. 132–134) es la siguiente:

```
begin instrucción {; instrucción} end
```

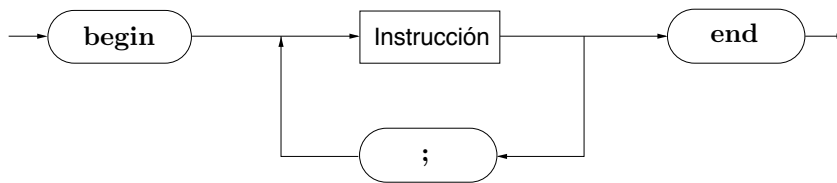


Figura 6.1. Diagrama sintáctico de una instrucción compuesta.

- ☞ Téngase en cuenta que la interpretación del punto y coma es la de nexos o separador de sentencias; por lo tanto no debe aparecer después de la última sentencia de la sucesión.

Obsérvese además que el significado de las palabras reservadas **begin** y **end** es el de principio y fin de la composición, esto es, actúan como delimitadores de la misma. Después de esta interpretación es sencillo deducir que la agrupación de una sola instrucción, por ejemplo **begin x := x + 1 end**, es redundante, y equivalente a la instrucción simple $x := x + 1$. Asimismo, resulta superfluo anidar pares **begin...end** como en el programa de la izquierda (que resulta ser equivalente al de la derecha).

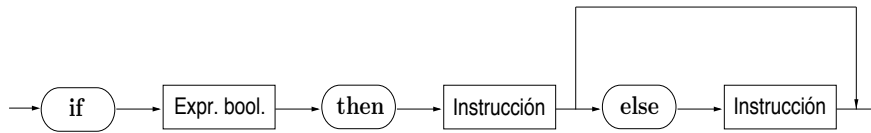
<pre> begin Read(x); Read(y); begin x := x + 1; y := y + 2; end; WriteLn(x * y) end </pre>	<pre> begin Read(x); Read(y); x := x + 1; y := y + 2; WriteLn(x * y) end </pre>
--	---

Para facilitar la legibilidad del programa es aconsejable mantener el sangrado dentro de cada par **begin-end**, así como la inclusión de comentarios que informen sobre el cometido de cada segmento de código, como se indicó en el apartado 5.2. Por ejemplo, en el programa anterior podrían haberse incluido comentarios indicando los segmentos de lectura de datos, cálculo y resultado del programa:

```

begin
  {Lectura de datos:}
  Read(x);
  Read(y);
  {Cálculos:}
  x := x + 1;
  y := y + 2;
end

```

Figura 6.2. Diagrama sintáctico de **if-then-else**.

```

{Resultados:}
WriteLn(x * y)
end

```

6.2 Instrucciones de selección

6.2.1 La instrucción *if-then-else*

Esta instrucción es el equivalente en Pascal a una expresión condicional del tipo *si apruebo entonces iré de vacaciones y si no tendré que estudiar en verano*, con la cual se indica que dependiendo del cumplimiento o no de una condición se hará una cosa u otra.

En Pascal, la instrucción **if-then-else** (en adelante, **if**) es la más importante instrucción de selección. Su diagrama sintáctico aparece en la figura 6.2.

La interpretación de la sentencia de selección genérica

if expresión booleana **then** instrucción-1 **else** instrucción-2

se puede deducir directamente de la traducción del inglés de sus términos:¹ si la expresión booleana es evaluada a **True** entonces se ejecuta la **instrucción-1** y en caso contrario (se evalúa a **False**) se ejecuta la **instrucción-2**.

Un ejemplo típico en el que aparece una instrucción de selección podría ser el siguiente segmento de programa que calcula el máximo de dos números, **x** e **y**, y lo almacena en la variable **max**:

```

if x > y then
  max:= x
else
  max:= y

```

Es importante sangrar adecuadamente el texto del programa para mantener la legibilidad del código obtenido. Por otra parte, nada nos impide que las acciones que se emprendan tras evaluar el predicado sean acciones compuestas; en tal

¹Hay que advertir que, si bien en inglés se prefiere el uso de *otherwise* al de *else*, aquél resulta ciertamente más propenso a ser escrito incorrectamente.

caso la instrucción compuesta se pondrá entre las palabras **begin** y **end** como se señaló en el apartado anterior.

Como muestra considérese el refinamiento del código anterior:

```

if x > y then begin
    max:= x;
    WriteLn('El máximo es ', x)
end
else begin
    max:= y;
    WriteLn('El máximo es ', y)
end

```

Es aconsejable evitar la introducción de código redundante dentro de las posibles alternativas, ya que se facilita en gran manera el mantenimiento del programa. El segmento anterior es un flagrante ejemplo de redundancia que puede ser evitada fácilmente colocando el `WriteLn` una vez realizada la selección:

```

if x > y then
    max:= x
else
    max:= y; {Fin del if}
WriteLn('El máximo es ', max)

```

- ☉☉ A efectos de la colocación de los puntos y comas debe tenerse en cuenta que toda la construcción **if-then-else** corresponde a *una sola instrucción*, y no es una composición de las instrucciones **if**, **then** y **else**; en particular, la aparición de un punto y coma justo antes de un **then** o de un **else** dará como resultado un error sintáctico (bastante frecuente, por cierto).

Una particularidad de esta instrucción es que la rama **else** es opcional; en caso de no ser incluida se ha de interpretar que cuando la expresión booleana resulta ser falsa entonces no se realiza ninguna acción. Por esta razón, la forma **if-then** es útil como sentencia para controlar excepciones que pudieran afectar el procesamiento posterior. Por ejemplo, en el siguiente fragmento de programa se muestra el uso de la forma **if-then** como sentencia de control.

```

ReadLn(year);
feb:= 28;
    {No siempre, ya que puede ser año bisiesto}
if year mod 4 = 0 then
    feb:= 29;
WriteLn('Este año Febrero tiene ',feb,' días')

```

El programa asigna a la variable `feb` el número de días del mes febrero, en general este número es 28 salvo para años bisiestos.²

- ☉☉ Obsérvese el uso de los puntos y comas en el ejemplo anterior: la instrucción de selección acaba tras la asignación `feb:= 29` y, al estar en una secuencia de acciones, se termina con punto y coma.

Aunque, en principio, la instrucción `if` sólo permite seleccionar entre dos alternativas, es posible usarla para realizar una selección entre más de dos opciones: la idea consiste en el *anidamiento*, esto es, el uso de una instrucción `if` dentro de una de las ramas `then` o `else` de otra instrucción `if`. Como ejemplo supóngase que se quiere desarrollar un programa que asigne a cada persona una etiqueta en función de su altura (en cm), el siguiente fragmento de código realiza la selección entre tres posibilidades: que la persona sea de estatura baja, media o alta.

```
if altura < 155 then
  WriteLn('Estatura Baja')
else if altura < 185 then
  WriteLn('Estatura Media')
else
  WriteLn('Estatura Alta')
```

En el segmento anterior se asigna la etiqueta de estatura baja a quien mida menos de 155 cm, de estatura media a quien esté entre 156 y 185 cm y de estatura alta a quien mida 186 cm o más.

El anidamiento de instrucciones `if` puede dar lugar a expresiones del tipo

$$\text{if } C1 \text{ then if } C2 \text{ then } I2 \text{ else } I3 \quad (6.1)$$

que son de interpretación ambigua en el siguiente sentido: ¿a cuál de las dos instrucciones `if` pertenece la rama `else`? En realidad, la ambigüedad sólo existe en la interpretación humana, ya que la semántica de Pascal es clara:

El convenio que se sigue para eliminar la ambigüedad consiste en emparejar cada rama `else` con el `then` “soltero” más próximo.

Siguiendo el convenio expuesto, la expresión anterior se interpreta sin ambigüedad como se indica a continuación:

$$\text{if } C1 \text{ then begin if } C2 \text{ then } I2 \text{ else } I3 \text{ end}$$

²El criterio empleado para detectar si un año es o no bisiesto ha sido comprobar si el año es múltiplo de 4; esto no es del todo correcto, ya que de los años múltiplos de 100 sólo son bisiestos los múltiplos de 400.

Si, por el contrario, se desea forzar esa construcción de modo que sea interpretada en contra del convenio, entonces se puede usar un par **begin-end** para aislar la instrucción **if** anidada del siguiente modo:

```
if C1 then begin if C2 then I2 end else I3
```

Otro modo de lograr la misma interpretación consiste en añadir la rama **else** con una instrucción vacía, esto es

```
if C1 then if C2 then I2 else else I3
```

- ☉☉ En la explicación del convenio sobre la interpretación del anidamiento de instrucciones **if** se ha escrito el código linealmente, en lugar de usar un formato vertical (con sangrado), para recordar al programador que la semántica de Pascal es independiente del formato que se dé al código. Es conveniente recordar que el sangrado sólo sirve para ayudar a alguien que vaya a leer el programa, pero no indica nada al compilador.

Por ejemplo, en relación con la observación anterior, un programador poco experimentado podría escribir la instrucción (6.1) dentro de un programa del siguiente modo

```
if C1 then
  if C2 then
    I2
  else
    I3 {¡¡Cuidado!!}
```

interpretando erróneamente que la rama **else** está ligada con el primer **if**. Como consecuencia, obtendría un programa sintácticamente correcto que arrojaría resultados imprevisibles debido a la interpretación incorrecta, por parte del programador, del anidamiento de instrucciones **if**. La solución del problema reside en “forzar” la interpretación del anidamiento para que el compilador entienda lo que el programador tenía en mente, esto sería escribir

<pre>if C1 then begin if C2 then I2 end else I3</pre>	o bien	<pre>if C1 then if C2 then I2 else else I3</pre>
---	--------	--

Con frecuencia, aunque no siempre, puede evitarse el anidamiento para elegir entre más de dos opciones, pues para ello se dispone de la instrucción de selección múltiple **case**, que permite elegir entre un número arbitrario de opciones con una sintaxis mucho más clara que la que se obtiene al anidar instrucciones **if**.

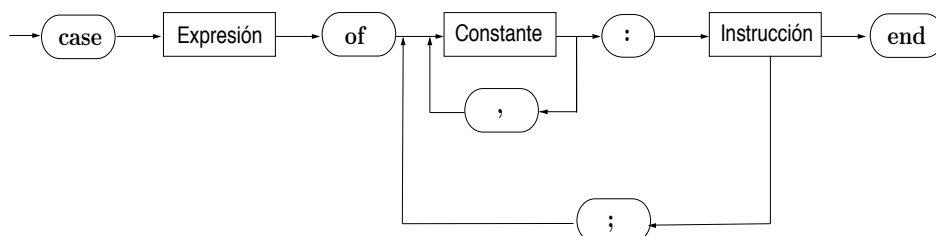


Figura 6.3. Diagrama sintáctico de la instrucción **case**.

6.2.2 La instrucción **case**

La instrucción **case** permite la selección entre una cantidad variable de posibilidades, es decir, es una sentencia de selección múltiple. Un ejemplo de esta selección en lenguaje natural podría ser el siguiente “menú semanal”: *según sea el día de la semana, hacer lo siguiente: lunes, miércoles y viernes tomar pescado, martes, jueves y sábado tomar carne, el domingo comer fuera de casa.*

Esta instrucción consta de una expresión (llamada *selector*) y una lista de sentencias etiquetadas por una o varias constantes del mismo tipo que el selector; al ejecutarse esta instrucción se evalúa el valor actual del selector y se ejecuta la instrucción que tenga esa etiqueta, si no existe ninguna instrucción con esa etiqueta se produce un error.³ El diagrama sintáctico de la instrucción **case** aparece en la figura 6.3.

- ☉ La expresión selectora de una instrucción **case** así como las etiquetas deben ser de un tipo ordinal (véase el apartado 3.6).

Como ejemplo de aplicación de la instrucción **case** considérese el siguiente segmento de código que asigna la calificación literal según el valor almacenado en la variable **nota** de tipo **integer**:

```
var
  nota: real;
...
ReadLn(nota);
case Round(nota) of
  0..4: WriteLn('SUSPENSO');
  5,6:  WriteLn('APROBADO');
```

³Esto es lo que ocurre en Pascal estándar; en Turbo Pascal no se produce ningún error, simplemente se pasa a la siguiente instrucción.

```

7,8: WriteLn('NOTABLE');
9: WriteLn('SOBRESALIENTE');
10: WriteLn('MATRICULA de HONOR')
end {case}

```

Otra situación en la que es frecuente el uso de la instrucción **case** es cuando algunos programas se controlan mediante *menús*, es decir, aparecen en pantalla las diferentes acciones que se pueden ejecutar dentro del programa y el usuario elige, mediante un número o una letra, aquélla que quiere utilizar.

Por ejemplo, supongamos un programa de gestión de una biblioteca. Tal programa proporcionaría en pantalla un menú con las siguientes acciones:

- B. Búsqueda.
- P. Petición préstamo.
- D. Devolución préstamo.
- S. Salir.

En un primer nivel de refinamiento, el programa podría escribirse de la siguiente forma:

```

var
  opcion: char;
...
Mostrar el menú
Leer opcion
case opcion of
  'B': Búsqueda.
  'P': Petición Préstamo.
  'D': Devolución Préstamo.
  'S': Salir.
end

```

Si las acciones son complejas, pueden aparecer submenús donde se seleccionan ciertas características de la acción. Por ejemplo, al elegir la opción de búsqueda puede aparecer un segundo menú con las distintas opciones disponibles:

- A. Búsqueda por Autores.
- M. Búsqueda por Materias.
- I. Búsqueda por ISBN.
- S. Salir.

Se deduce fácilmente que este fragmento de programa debe repetir las acciones de búsqueda, petición y devolución hasta que se elija la opción de salida. El cometido de este fragmento consiste en mostrar el menú al usuario y leer un valor, que se asigna a la variable `opción`. Este valor determina la opción elegida y se utiliza en una instrucción `case` para activar las acciones correspondientes. Por lo tanto, la instrucción `case` abunda en este tipo de programas al determinar las acciones que hay que ejecutar en cada opción.

6.3 Instrucciones de iteración

Las instrucciones iterativas permiten especificar que ciertas acciones sean ejecutadas repetidamente; esto es lo que se llama usualmente un *bucle*.

Se dispone en Pascal de tres construcciones iterativas (`while`, `repeat` y `for`), no obstante se puede demostrar que todas ellas pueden ser especificadas sólo con la instrucción `while` (véase el apartado 7.2). En los siguientes apartados se estudia detenidamente cada una de las instrucciones de iteración y se realiza una comparación entre las características de cada una de ellas para ayudarnos a escoger la que más se adecua al bucle que se desea desarrollar.

6.3.1 La instrucción `while`

En algunas ocasiones es necesario especificar una acción que se repite siempre que se cumpla una determinada condición; una frase en lenguaje natural tal como *mientras haga calor usar manga corta* es un ejemplo de este tipo de construcciones.

En Pascal esta construcción se hace mediante la instrucción `while`. Su diagrama sintáctico aparece en la figura 6.4, que se corresponde con el esquema

while Expresión booleana **do** Instrucción

cuya interpretación es: mientras que la expresión booleana sea cierta se ejecutará la instrucción, que se suele llamar *cuerpo del bucle*, indicada tras el `do`.

A continuación tenemos un fragmento de programa que calcula la suma de los n primeros números naturales:

```
ReadLn(n);
suma:= 0;
contador:= 1;
while contador <= n do begin
    suma:= suma + contador;
    contador:= contador + 1
end; {while}
WriteLn(suma)
```

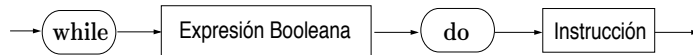


Figura 6.4. Diagrama sintáctico de la instrucción **while**.

La ejecución de una instrucción **while** comienza con la comprobación de la condición (por esto a los bucles **while** se les llama *bucles preprobados*); si ésta es falsa entonces se finaliza la ejecución, esto es, se salta la sentencia que aparece tras el **do**; si la condición es verdadera entonces se ejecuta la instrucción, se vuelve a comprobar la condición y así sucesivamente.

Para una correcta utilización de la instrucción **while** es necesario que la instrucción modifique las variables que aparecen en la condición, ya que en caso contrario, si la condición es verdadera siempre permanecerá así y el bucle no terminará nunca.

Una situación en que se puede producir este error surge cuando el cuerpo del bucle es una secuencia de instrucciones y se olvida utilizar los delimitadores **begin** y **end**. Por ejemplo, el siguiente segmento de código *no* calcula la suma de los enteros desde el 1 hasta el n :

```

ReadLn(n);
suma:= 0;
contador:= 0;
while contador <= n do
  suma:= suma + contador;   {OJO: Fin de while}
  contador:= contador + 1;
WriteLn(suma)
  
```

Al olvidar delimitar el cuerpo del bucle, la instrucción por iterar termina antes de actualizar el valor del contador, con lo cual el bucle se repite sin cesar y el programa se “cuelga”. La corrección de tal error se reduce a incluir un par **begin-end** para delimitar la sentencia interior del bucle.

- ☞ La instrucción **while** admite sólo una instrucción tras el **do**, con lo que para iterar una acción múltiple se ha de emplear la composición de instrucciones con su correspondiente par **begin-end**.

La sintaxis de Pascal permite escribir un punto y coma inmediatamente después del **do**. Sin embargo, cuando se entre en el bucle, esta construcción dará lugar a un bucle infinito, puesto que se interpreta que el interior del bucle es la instrucción vacía que, obviamente, no modifica los parámetros de la condición del bucle. Por lo tanto, a efectos prácticos *no se debe escribir un punto y coma detrás del do*.

Antes de continuar con más ejemplos de bucles **while** vamos a introducir un par de funciones booleanas que aparecen muy frecuentemente en el uso de bucles: **EoLn** y **EoF**.⁴ La función **EoLn** se hace verdadera cuando se alcanza una marca de fin de línea y falsa en otro caso, mientras que la función **EoF** se hace verdadera cuando se alcanza una marca de fin de archivo y falsa en otro caso. Así, el siguiente fragmento de programa cuenta y escribe los caracteres de una línea:

```

var
  c: char;
  numCar: integer;
...
numCar:= 0;
while not EoLn do begin
  Read(c);
  numCar:= numCar + 1
end; {while}
WriteLn(numCar)

```

y este otro fragmento cuenta el número de líneas del **input**

```

var
  numLin: integer;
...
numLin:= 0;
while not EoF do begin
  ReadLn;
  numLin:= numLin + 1
end; {while}
WriteLn(numLin)

```

- ☞☞ Obsérvese cómo se usa la característica de preprobado en los ejemplos anteriores para asegurarse de que no ha terminado la línea (resp. el archivo) antes de leer el siguiente carácter (resp. línea).⁵

Las instrucciones **while** se pueden anidar y obtener instrucciones del siguiente tipo

⁴Estas funciones serán estudiadas en mayor profundidad en el apartado 14.3.

⁵En la versión 7.0 de Turbo Pascal se puede marcar el fin de la entrada de datos con la combinación de teclas [CONTROL] + [Z].

```
while condición 1 do begin  
  Instrucciones  
  while condición 2 do  
    Instrucción;  
  Instrucciones  
end {while}
```

simplemente escribiendo el **while** interior como una instrucción más dentro del cuerpo de otro bucle **while**. Si el bucle **while** exterior no llega a ejecutarse, por ser falsa su condición, tampoco lo hará el **while** interior. Si, por el contrario, el **while** exterior se ejecutara por ser su condición verdadera, entonces se evaluará la condición del **while** interior y, si también es verdadera, se ejecutarán sus instrucciones interiores hasta que su condición se vuelva falsa, tras lo cual el control vuelve al **while** exterior.

Un ejemplo de frecuente aplicación de anidamiento de instrucciones **while** puede ser la gestión de ficheros de texto, en el siguiente fragmento de código se cuenta el número de caracteres del **input**, que está compuesto a su vez por varias líneas

```
var  
  c:char; numCar:integer;  
  ...  
numCar:= 0;  
while not EoF do begin  
  while not EoLn do begin  
    Read(c);  
    numCar:= numCar + 1  
  end; {while not EoLn}  
  ReadLn  
end; {while not EoF}  
WriteLn(numCar)
```

Las propiedades principales de la instrucción **while** que se deben recordar son las siguientes:

1. La condición se comprueba al principio del bucle, antes de ejecutar la instrucción; por eso se le llama bucle preprobado.
2. El bucle termina cuando la condición deja de cumplirse.
3. Como consecuencia de los puntos anteriores la instrucción se ejecuta cero o más veces; por lo tanto puede no ejecutarse.

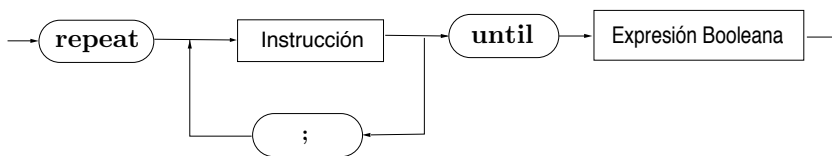


Figura 6.5. Diagrama sintáctico de la instrucción **repeat**.

6.3.2 La instrucción *repeat*

Comenzamos este apartado retomando el ejemplo en lenguaje natural con el que se presentó la instrucción **while**: *mientras haga calor usar manga corta*. La característica de *preprobado* de **while** hace que este consejo sólo sea válido para gente *previsora* que comprueba el tiempo que hace antes de salir de casa.

¿Cómo se podría modificar el ejemplo anterior para que fuera válido también para quien no sabe qué tiempo hace fuera hasta que ya es demasiado tarde? Una forma sería *llevar un jersey puesto hasta que haga calor*; de este modo se evitarán bastantes enfriamientos indeseados.

La instrucción **repeat** permite la construcción de bucles similares al de este último ejemplo, con características ligeramente distintas a la del bucle **while**. El diagrama sintáctico de la instrucción **repeat** aparece en la figura 6.5. La forma general de la instrucción **repeat** obedece al esquema

repeat Lista de instrucciones **until** Expresión booleana

donde

Lista de instrucciones := instrucción { ; instrucción }

por lo tanto, la interpretación de una instrucción **repeat** es: repetir las instrucciones indicadas en el cuerpo del bucle hasta que se verifique la condición que aparece tras **until**.

- ☞ En este tipo de bucles las palabras reservadas **repeat** y **until** funcionan como delimitadores, no siendo necesario usar **begin-end** para delimitar la lista de instrucciones.

En la ejecución de una instrucción **repeat** se comienza ejecutando la lista de instrucciones y después se comprueba si se cumple la condición (por eso el bucle es *postprobado*); si la condición aún no se cumple entonces se repite el bucle, ejecutando la lista de instrucciones y comprobando la condición. La iteración

termina cuando la condición se hace verdadera, en cuyo caso se pasa a la siguiente instrucción externa al bucle.

Como ejemplo de utilización del bucle **repeat**, se incluye otra versión de la suma de los n primeros números naturales.

```
ReadLn(n); {Supuesto que n >= 1}
suma:= 0;
contador:= 0;
repeat
  contador:= contador + 1;
  suma:= suma + contador
until contador = n
```

Obsérvese que la condición $n \geq 1$ es imprescindible para que el resultado final sea el esperado. En general, siempre es conveniente comprobar el comportamiento del bucle en valores extremos; en este ejemplo, para $n = 0$ se generaría un bucle infinito, lo cual se evitaría sustituyendo la condición **contador = n** por **contador >= n**. En este caso, dada la característica de postprobado del bucle **repeat**, las instrucciones interiores se ejecutarán al menos una vez, por lo que la suma valdrá al menos 1 y el resultado arrojado sería incorrecto para $n \leq 0$.

Un caso frecuente de utilización de **repeat** se produce en la lectura de datos:

```
{lectura de un número positivo:}
repeat
  WriteLn('Introduzca un número positivo');
  ReadLn(numero)
until numero > 0
```

donde si alguno de los datos introducidos no es positivo entonces la condición resultará ser falsa, con lo cual se repite la petición de los datos.

Podemos mejorar el ejemplo de aplicación a la gestión de una biblioteca mostrado en el apartado 6.2.2 usando la instrucción **repeat** para controlar el momento en el que se desea terminar la ejecución.

```
Mostrar el menú
  {Elegir una acción según la opción elegida:}
WriteLn('Elija su opción: ');
ReadLn(opcion);
repeat
  case opcion of
    B: Búsqueda.
    P: Petición Préstamo.
    D: Devolución Préstamo.
```

```

    S: Salir.
  end
until (opcion = 'S') or (opcion = 's')

```

El anidamiento de instrucciones **repeat** se realiza de la forma que cabe esperar. Como ejemplo se introduce un programa que determina el máximo de una secuencia de números positivos procedentes del **input** terminada con el cero.

```

Program MaximoDelInput (input, output);
  {Calcula el máximo de una secuencia de números terminada en 0}
  var
    max, n: integer;
begin
  max:= 0;
  repeat
    {Lee un número positivo, insistiendo hasta lograrlo;}
    repeat
      Write('Introduzca un número positivo: ');
      ReadLn(n)
    until n >= 0;
    if n > max then
      max:= n
    until n = 0;
  WriteLn('El máximo es: ',max)
end. {MaximoDelInput}

```

Las propiedades principales de la instrucción **repeat** son las siguientes:

1. La instrucción **repeat** admite una lista de instrucciones interiores, *no* siendo necesario utilizar los delimitadores **begin-end**.
2. Este bucle se llama postprobado; es decir, la condición se comprueba después de ejecutar la lista de instrucciones, por lo que ésta se ejecuta al menos una vez.
3. El bucle termina cuando se cumple la condición.
4. Como consecuencia de los puntos anteriores la lista de instrucciones siempre se ejecuta una o más veces.

6.3.3 La instrucción **for**

La instrucción de repetición **for** se utiliza para crear bucles con un número predeterminado de repeticiones. Un ejemplo sencillo en lenguaje natural podría ser para los bloques desde el A hasta el K hacer la inspección del ascensor, según

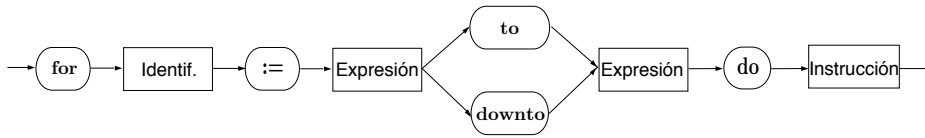


Figura 6.6. Diagrama de flujo de las instrucciones **for**.

el cual se especifica una tarea repetitiva (la inspección) que ha de realizarse exactamente en 11 ocasiones (para los bloques A, \dots, K).

La sentencia **for** admite dos variantes: la **for-to-do** (instrucción **for** ascendente) y la **for-downto-do** (instrucción **for** descendente). El diagrama sintáctico de estas sentencias aparece en la figura 6.6. De otro modo:

for variable:= expresión ordinal (to | downto) expresión ordinal **do** instrucción

donde se acostumbra a llamar *variable de control* o *índice* del bucle a la variable **variable**.

El funcionamiento del bucle **for** es el siguiente: primero se comprueba si el índice rebasa el límite final, con lo que es posible que el cuerpo del bucle no llegue a ejecutarse ninguna vez, en caso positivo se le asigna el valor inicial a la variable de control **vble**, se ejecuta la instrucción interior una vez y se incrementa (o decrementa, según se trate de **to** o **downto** respectivamente) una unidad el valor de **vble**, si este nuevo valor está comprendido entre el valor inicial y el valor final, entonces se vuelve a ejecutar la instrucción interior, y así sucesivamente hasta que **vble** alcanza el valor final.

En particular, si en una instrucción **for-to-do** el valor inicial de la variable es posterior al valor final entonces no se ejecutan las instrucciones interiores y se sale del bucle. La instrucción **for-downto-do** tiene un comportamiento análogo cuando el valor inicial de la variable es anterior al valor final.

- ☉☉ En teoría, nada impide que en el cuerpo de un bucle **for** se modifique el valor de la variable de control o las expresiones inicial y final del bucle; sin embargo, debe ponerse el mayor cuidado en evitar que esto ocurra. En particular, conviene recordar que la variable de control se actualiza automáticamente. El siguiente fragmento de código es un ejemplo sintácticamente correcto

```

for i:= 1 to 5 do begin
  Write(i);
  i:= i - 1
end {for}
  
```

pero genera un bucle infinito dando como resultado una sucesión infinita de unos.⁶

Como ejemplo de aplicación de la instrucción **for** podemos considerar, una vez más, la suma de los primeros números naturales $1, 2, \dots, n$.

```
var
  n, i, suma: integer;
...
ReadLn(n);
suma:= 0;
for i:= 1 to n do
  suma:=suma + i;
WriteLn(suma)
```

Otro ejemplo interesante es el siguiente, con el que se halla una tabulación de la función seno para los valores $0^\circ, 5^\circ, \dots, 90^\circ$.

```
const
  Pi = 3.1416;
var
  r: real;
  n: integer;
...
r:= 2 * Pi/360; {El factor r pasa de grados a radianes}
for n:= 0 to 18 do
  WriteLn(Sin(5 * n * r))
```

Es conveniente recordar que la variable de control puede ser de *cualquier* tipo ordinal; por ejemplo, la siguiente instrucción imprime, en una línea, los caracteres desde la 'A' a la 'Z':

```
for car:= 'A' to 'Z' do
  Write(car)
```

Como ejemplo de anidamiento de bucles **for** podemos considerar el siguiente fragmento que escribe en la pantalla los elementos de la matriz de tamaño $n \times m$ definida por $a_{ij} = \frac{i+j}{2}$:

```
const
  N = 3;
  M = 5;
```

⁶En realidad, ése es el comportamiento en Turbo Pascal.

```
var
  i,j: integer;
...
for i:= 1 to N do
  for j:= 1 to M do
    WriteLn('El elemento (' ,i,',',j,') es ',(i + j)/2)
```

Las siguientes características de la instrucción **for** merecen ser recordadas:

1. Las expresiones que definen los límites inicial y final se evalúan una sola vez antes de la primera iteración.
2. El bucle se repite un número predeterminado de veces (si se respeta el valor del índice en el cuerpo del bucle).
3. El valor de la variable de control se comprueba antes de ejecutar el bucle.
4. El incremento (o decremento) del índice del bucle es automático, por lo que no se debe incluir una instrucción para efectuarlo.
5. El bucle termina cuando el valor de la variable de control sale fuera del intervalo de valores establecido.

6.4 Diseño y desarrollo de bucles

6.4.1 Elección de instrucciones iterativas

Para poder elegir la instrucción iterativa que mejor se adapta a una situación particular es imprescindible conocer las características más importantes de cada instrucción iterativa, así como las similitudes y diferencias entre ellas.

El primero de todos los criterios para elegir una u otra instrucción iterativa es la *claridad*: se ha de elegir aquella instrucción que exprese las acciones por repetir con la mayor naturalidad.

Además, la elección de la instrucción adecuada depende de las características del problema. En el caso en que se conozca *previamente* el número de repeticiones que van a ser necesarias, es recomendable usar la instrucción **for**. Por ejemplo, el siguiente fragmento de código calcula la media aritmética de 5 números leídos del `input`:

```
Program Media5 (input, output);
  {Calcula la media de cinco números}
var
  entrada, total, media: real;
```

```

begin
  total:= 0;
  {Entrada de datos y cálculo de la suma total:}
  for i:= 1 to 5 do begin
    ReadLn(entrada);
    total:= total + entrada
  end; {for}
  {Cálculo de la media:}
  media:= total / 5;
  {Salida de datos:}
  WriteLn('La media es ', media:10:4)
end. {Media5}

```

Si no se conoce previamente cuántas repeticiones se necesitarán entonces se usará bien **while** o bien **repeat**; para saber cuándo conviene usar una u otra será conveniente recordar sus similitudes y diferencias.

1. Si *no* se sabe si se ha de ejecutar el cuerpo del bucle al menos una vez entonces el bucle ha de ser preprobado, con lo cual se usaría la instrucción **while**.
2. Si, por el contrario, el cuerpo del bucle se ha de ejecutar al menos una vez entonces se usaría **repeat**, pues nos basta con un bucle postprobado.

Por ejemplo, supóngase que estamos desarrollando un programa de gestión de un cajero automático, la primera tarea que se necesita es la de identificar al usuario mediante su número personal; si tenemos en cuenta la posibilidad de error al teclear el número lo mejor será colocar este fragmento de código dentro de un bucle. Puesto que, obviamente, es necesario que el usuario teclee su número de identificación al menos una vez, se usará la instrucción **repeat**.

```

var
  codigo, intentos: integer;
...
intentos:= 0;
repeat
  Read(codigo);
  intentos:= intentos + 1
until Código correcto or (intentos > 3)

```

donde se ha expresado en pseudocódigo la comprobación de la validez del número tecleado y, además, se incluye un contador para no permitir más de tres intentos fallidos.

En caso de duda, si no se sabe muy bien si el cuerpo del bucle se ha de repetir al menos una vez o no, se ha de usar **while**, pero debemos asegurarnos de que

la condición está definida en la primera comprobación. El siguiente ejemplo muestra un caso en el que esto no ocurre: supongamos que dada una línea de caracteres se desea averiguar el primer carácter que es una letra minúscula, el siguiente fragmento de programa es erróneo

```
var
  car: char;
...
while not (('a' <= car) and (car <= 'z')) do
  Read(car)
```

en el supuesto de que a `car` no se le haya dado un valor inicial, ya que entonces el valor de `car` es desconocido en la primera comprobación.

6.4.2 Terminación de un bucle

El buen diseño de un bucle debe asegurar su terminación tras un número finito de repeticiones.

Los bucles **for** no crean problemas en este aspecto siempre que se respete la variable de control dentro del cuerpo del bucle. Precisamente el uso de **for** está indicado cuando se conoce previamente el número de repeticiones necesarias; sin embargo, para los bucles condicionales (**while** y **repeat**) se ha de comprobar que en su cuerpo se modifican algunas de las variables que aparecen en su condición P de entrada (resp. salida) de manera que, en las condiciones supuestas antes del bucle, P llega a ser falsa (resp. cierta) en un número finito de iteraciones.

Considérese el siguiente fragmento de programa:

```
var
  n: integer;
...
Read(n);
while n <> 0 do begin
  WriteLn(n);
  n:= n div 2
end {while}
```

se observa que en el cuerpo del bucle se modifica el valor de la variable, `n`, que aparece en su condición. En este caso, para cualquier `n` entero se puede demostrar que el bucle termina siempre en $\lfloor \log_2 |n| + 1 \rfloor$ pasos;⁷ de hecho, estos cálculos forman parte del algoritmo para pasar a base dos (véase [PAO94], página 32).

A continuación se muestra un ejemplo de bucle que, a pesar de modificar en su cuerpo la variable de la condición del bucle, no siempre termina.

⁷La notación $\lfloor x \rfloor$ representa el mayor entero menor que x .

```
var
  n: integer;
...
ReadLn(n);
repeat
  WriteLn(n);
  n:= n - 2
until n = 0
```

Obviamente, este programa sólo termina en el caso de que el valor proporcionado para n sea par y positivo, dando $n/2$ “vueltas”, y no termina en caso contrario.

6.4.3 Uso correcto de instrucciones estructuradas

No basta con construir un programa para desempeñar una tarea determinada, hay que convencerse de que el programa que se ha escrito resuelve correctamente el problema. El análisis de la corrección de un programa puede hacerse a posteriori, como se explicó en el apartado 5.4, aplicando la llamada *verificación* de programas. Sin embargo, es mucho más recomendable usar una técnica de programación que permita asegurar que el programa construido es correcto.

En este apartado se indica cómo probar la corrección de un fragmento de código en el que aparecen instrucciones estructuradas. El estudio de la secuencia, la selección y la iteración se realiza por separado en los siguientes apartados.

En general, se expresa un fragmento de programa mediante

Precondición
Instrucciones
Postcondición

para expresar que, si la *precondición* es cierta al comienzo de las *instrucciones*, entonces a su término la *postcondición* se verifica.

La precondición (abreviadamente PreC.) representa los requisitos para que trabajen las instrucciones, y la postcondición (abreviadamente PostC.) representa los efectos producidos por las mismas. Así pues, decir que las instrucciones son correctas equivale a decir que, si en su comienzo se verifican sus requisitos, a su término se han logrado sus objetivos; en suma, cumplen correctamente con su cometido.

Uso correcto de una secuencia de instrucciones

Para estudiar la corrección de una secuencia de instrucciones se incluyen aserciones entre ellas y se analiza la corrección de cada instrucción individualmente.

Como ejemplo vamos a comprobar la corrección del fragmento de código que altera los valores de las variables **a**, **b** y **c** según se indica:

```

{PreC.: a = X, b = Y, c = Z}
a:= a + b + c;
b:= a - b;
c:= a - c;
a:= 2 * a - b - c
{PostC.: a = Y + Z, b = X + Z, c = X + Y}

```

Para la verificación de la secuencia de instrucciones hemos de ir incluyendo aserciones que indiquen el efecto de cada instrucción, tal como se presentó en el apartado 5.4:

```

{PreC.: a = X, b = Y, c = Z}
a:= a + b + c;
  {a = X+Y+Z, b = Y, c = Z}
b:= a - b;
  {a = X+Y+Z, b = (X+Y+Z) - Y = X+Z, c = Z}
c:= a - c;
  {a = X+Y+Z, b = X+Z, c = (X+Y+Z) - Z = X+Y}
a:= 2 * a - b - c
  {a = 2*(X+Y+Z) - (X+Z) - (X+Y) = Y+Z, b = X+Z, c = X+Y}
{PostC.: a = Y+Z, b = X+Z, c = X+Y}

```

En el desarrollo anterior se aprecia cómo, partiendo de la precondition, va cambiando el estado de las variables **a**, **b** y **c** hasta llegar a la postcondición.

Uso correcto de una estructura de selección

Las estructuras de selección incluyen dos tipos de instrucciones, **if** y **case**; en ambos casos el proceso de verificación es el mismo: se ha de verificar cada una de las posibles opciones individualmente, haciendo uso de la precondition y de la expresión que provoca la elección de tal opción.

Puesto que la verificación se realiza de modo similar tanto para **if** como para **case** consideraremos sólo el siguiente ejemplo, con el que se pretende calcular el máximo de las variables **a** y **b**:

```

{PreC.: a = X, b = Y}
if a >= b then
  m:= a
else
  m:= b
{PostC.: a = X, b = Y, m = max(X,Y)}

```

Para verificar este fragmento se estudiará separadamente cada una de las dos ramas (la **then** y la **else**)

```

{PreC.: a = X, b = Y}
if a >= b then
  {a = X, b = Y y X >= Y }
  m:= a
  {a = X, b = Y, X >= Y y m = X = máx(X,Y)}
else
  {a = X, b = Y y X < Y}
  m:= b
  {a = X, b = Y, X < Y y m = Y = máx(X,Y)}
{Postc.: a = X, b = Y y m = máx(X,Y)}

```

Se observa que cada una de las ramas verifica la postcondición, con lo cual la selección descrita es correcta.

Uso correcto de estructuras iterativas

Después de conocer las instrucciones de Pascal que permiten codificar bucles, en particular tras estudiar los distintos ejemplos incluidos, se puede tener la sensación de que construir bucles correctamente necesita grandes dosis de inspiración, ya que un bucle escrito a la ligera puede dar más o menos vueltas de lo necesario, no detenerse nunca o, cuando lo hace, dar un resultado incorrecto.

En este apartado se pretende demostrar que no es necesario apelar a las musas para escribir bucles correctos, ya que basta con diseñar el bucle mediante una metodología adecuada que se presenta a continuación.

Como primer ejemplo se considerará el algoritmo de la división entera mediante restas sucesivas; este algoritmo se esboza a continuación mediante un sencillo ejemplo:

Para hallar el cociente de la división de 7 entre 2 hay que ver cuántas veces “cabe” 2 dentro de 7 (en este caso $7 = 2 + 2 + 2 + 1$ con lo que el cociente será 3 y el resto 1) y, para ello, a 7 (el dividendo) se le va restando 2 (el divisor) repetidamente hasta que se obtenga un número menor que 2 (el divisor); este número será el resto de la división, y el número de repeticiones realizadas será el cociente.

No es conveniente comenzar a escribir directamente el bucle, aun habiendo comprendido perfectamente cómo trabaja el algoritmo. Antes conviene meditar qué tipo de bucle usar, especificar las variables mínimas necesarias, expresar el resultado deseado en términos de variables declaradas y especificar qué se espera que el bucle realice en cada repetición.

			ddo	dsor	coc	resto
Paso 0:	$7 = 7$	$= 2 \cdot 0 + 7$	7	2	0	7
Paso 1:	$7 = 2 + 5$	$= 2 \cdot 1 + 5$	7	2	1	5
Paso 2:	$7 = 2 + 5$	$= 2 \cdot 1 + 5$	7	2	2	3
Paso 3:	$7 = 2 + 2 + 2 + 1$	$= 2 \cdot 3 + 1$	7	2	3	1

Figura 6.7.

1. Para decidir qué tipo de bucle usar debemos observar que, en principio, no sabemos cuántas repeticiones van a ser necesarias en cada caso (no nos sirve **for**); por otro lado, no siempre va a ser necesaria al menos una repetición del cuerpo del bucle, como por ejemplo al dividir 5 entre 7 (no nos sirve **repeat**). En consecuencia tendremos que usar un bucle **while**.
2. Las variables que necesitaremos para codificar el bucle deben ser al menos cuatro: para el dividendo, el divisor, el cociente y el resto, que llamaremos respectivamente **ddo**, **dsor**, **coc** y **resto**.
3. Dados **ddo** y **dsor**, el resultado que deseamos obtener son valores para **coc** y **resto** tales que $\text{ddo} = \text{dsor} * \text{coc} + \text{resto}$ verificando que $0 \leq \text{resto} \leq \text{dsor}$.
4. Por último, ¿qué se realiza en cada iteración? Sencillamente, reducir **resto** (en **dsor** unidades) e incrementar **coc** (en 1 unidad): o sea, tantear que ha cabido una vez más el divisor en el dividendo. Si detallamos la división de 7 entre 2 (véase la figura 6.7) podemos ver cómo cambian las variables **coc** y **resto** y observar que:
 - (a) En cada iteración del bucle existe una relación que permanece constante: $\text{ddo} = \text{dsor} * \text{coc} + \text{resto}$.
 - (b) Se finaliza cuando $\text{resto} < \text{dsor}$. Al escribir el programa usaremos esta aserción para asegurar la corrección del código.

Ya podemos dar una primera aproximación al programa usando pseudocódigo y aserciones:

```

var
  ddo, dsor, coc, resto: integer;
...
{Entrada de datos:}
Leer los valores de dividendo y divisor (positivos no nulos)
{Cálculos:}
{PreC.PreC.: ddo > 0 y dsor > 0}
Dar valor inicial a las variables cociente y resto

```

```

    {La relación  $ddo = dsor * coc + resto$  debe cumplirse siempre}
  Comenzar el bucle
    {PostC.:  $ddo = dsor * coc + resto$  y  $0 \leq resto < dsor$ }
    {Salida de datos:}
  Imprimir el resultado

```

Las tareas de entrada y salida de datos no presentan demasiada dificultad, pero hay que tener cuidado al codificar los cálculos, especialmente el bucle. Con la información obtenida al estudiar el bucle se observa que éste debe repetirse mientras que la variable `resto` sea mayor que el divisor: el valor de `resto` comienza siendo el del dividendo; en cada iteración, del `resto` se sustrae el `dsor` hasta que, finalmente, se obtenga un valor para `resto` menor que el `dsor`; al mismo tiempo, la variable `coc` aumenta una unidad cada vez que se ejecuta una iteración.

- ☉☉ Un vistazo a la tabla de la figura 6.7 basta para convencerse de que hay que operar sobre la variable `resto` y no sobre la que contiene el dividendo. Por otra parte, es importante no confundir los conceptos “resto” y “cociente” con los valores de las variables `resto` y `coc`, ya que sólo al final del proceso los valores de estas variables son realmente el resto y el cociente de la división entera.

Este razonamiento nos permite escribir el siguiente programa:

```

Program Cociente (input, output);
  var
    ddo, dsor, coc, resto: integer;
begin
  {Entrada de datos:}
  repeat
    Write('Introduzca el dividendo: ');
    ReadLn(ddo);
    Write('Introduzca el divisor: ');
    ReadLn(dsor)
  until (ddo > 0) and (dsor > 0);
  {Se tiene  $ddo > 0$  y  $dsor > 0$ }
  {Cálculos:}
  coc:= 0;
  resto:= ddo;
  {Inv.:  $ddo = dsor * coc + resto$  y  $resto \geq 0$ }
  while resto >= dsor do begin
    resto:= resto - dsor;
    coc:= coc + 1
  end; {while}
  {PostC.:  $ddo = dsor * coc + resto$  y  $0 \leq resto < dsor$ }

```

```

    {Salida de datos:}
    WriteLn('El cociente es', coc,' y el resto es ', resto)
end.    {Cociente}

```

En el programa anterior se ha destacado una aserción que permanece constante antes, durante y tras la ejecución del bucle; tal aserción recibe el nombre de *invariante del bucle* (abreviadamente Inv.). En general, para la construcción de cualquier bucle conviene buscar un invariante que refleje la acción del bucle en cada iteración, pues esto facilitará la programación y la verificación posterior.

El invariante de un bucle debe verificarse en cuatro momentos:

Comienzo: El invariante debe cumplirse justo antes de ejecutar el bucle por primera vez.

Conservación: Si el invariante y la condición del bucle se cumplen antes de una iteración y se ejecuta el cuerpo del bucle, entonces el invariante seguirá siendo cierto tras su ejecución.

Salida: El invariante, junto con la falsedad de la condición (que se tiene a la salida del bucle), nos permitirá deducir el resultado, que es la postcondición del bucle.

Terminación: El cuerpo del bucle deberá avanzar hacia el cumplimiento de la condición de terminación del bucle, de forma que se garantice la finalización del mismo.

Para verificar cualquier bucle hay que comprobar estas tres etapas, donde la parte generalmente más difícil es la comprobación de conservación del invariante. Para el bucle anterior tenemos que

1. Las asignaciones a las variables `coc` y `resto` antes del bucle hacen que el invariante se cumpla antes de la primera iteración.
2. Supuesto que el invariante se cumple antes de una iteración, esto es $ddo = dsor * coc + resto$, hay que demostrar que el cuerpo del bucle conserva el invariante. En nuestro caso, debemos comprobar que los nuevos valores para `coc` y `resto` siguen cumpliendo el invariante, pero esto es trivial ya que

$$\begin{aligned}
 dsor * (coc + 1) + (resto - dsor) &= \\
 dsor * coc + dsor + resto - dsor &= \\
 dsor * coc + resto &= ddo
 \end{aligned}$$

3. La terminación del bucle la tenemos asegurada, ya que en el cuerpo del bucle se va disminuyendo el valor de `resto`, con lo que la condición de entrada `resto < dsor` siempre se va a alcanzar tras un número finito de iteraciones. La corrección del bucle se deduce del invariante y de la condición de entrada del bucle: tras la última iteración, según el invariante, tenemos `ddo = dsor * coc + resto` y además, según la condición del bucle, se tiene que `resto < dsor` con lo cual el bucle es correcto.

Dependiendo del programa en particular, aparecen distintos tipos de bucles; no todos los invariantes tienen por qué ser expresables como relaciones numéricas entre variables.

En el siguiente ejemplo tenemos que localizar la posición del primer carácter blanco (un espacio) que aparece en una frase terminada por un punto.

La idea consiste en recorrer la frase carácter por carácter teniendo en cuenta la posición del carácter rastreado. Las variables necesarias son dos: `car` y `pos`, para almacenar el carácter leído y su posición. El cuerpo del bucle, en cada iteración, debe leer el siguiente carácter y actualizar la posición; y se volverá a ejecutar a menos que se haya leído un blanco o un punto, con lo cual, el invariante ha de ser que `Pos` contiene la posición del último carácter leído.

El recorrido se va a realizar leyendo caracteres del `input`, y se supone que éste contiene algún carácter blanco o algún punto. En estas condiciones tendremos el siguiente bucle:

```

var
  car: char;
  pos: integer;
...
pos:= 0;
{Inv.: pos indica la posición del último carácter rastreado}
repeat
  Read(car);
  pos:= pos + 1
until (car = ' ') or (car = '.')
...

```

La corrección de este bucle se deja como ejercicio para el lector.

- ☞☞ Se acaba de introducir, de manera informal, el concepto de invariante de un bucle para analizar su corrección. Sin embargo, no debe pensarse que los invariantes son herramientas para verificar bucles a posteriori, esto es, después de haberlos escrito: es conveniente extraer el invariante “antes” de escribir nada, pues de esta manera la tarea de programación del bucle se facilita enormemente.

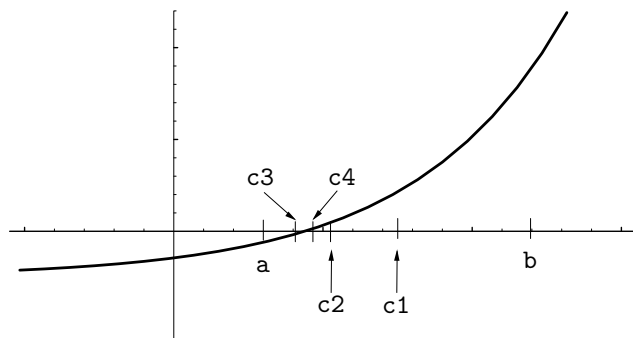


Figura 6.8. Aproximación por bipartición.

6.5 Dos métodos numéricos iterativos

Dada una función $f : \mathbb{R} \rightarrow \mathbb{R}$, se considera el problema de hallar aproximadamente un cero de la misma, esto es, un valor $x \in \mathbb{R}$ tal que $f(x) = 0$. En un computador no es posible representar todos los números reales, por lo cual será necesario conformarse con aproximaciones a un cero de f , esto es, con un x tal que $f(x) \simeq 0$. Los siguientes métodos son ampliamente conocidos por su fácil aplicación y eficiencia. El tercer apartado no es más que una aplicación directa de los mismos.

6.5.1 Método de bipartición

En este primer método, aceptaremos un valor x como aproximación aceptable de un cero x_0 de f si $|x - x_0| < \varepsilon$, para una cierta tolerancia prefijada (por ejemplo, $\varepsilon = 10^{-6}$).

Este método se basa en el teorema de Bolzano que dice que, cuando f es continua en un intervalo $[a, b]$ y los signos de $f(a)$ y de $f(b)$ son distintos, entonces existe algún cero de f en ese intervalo.

Aunque no hay modo de hallarlo en general, una posibilidad consiste en hallar el signo de $f(c)$, siendo $c = \frac{a+b}{2}$ el punto central del intervalo $[a, b]$ y, según sea igual al de $f(a)$ o al de $f(b)$, quedarnos con el intervalo $[c, b]$ o $[a, c]$, respectivamente. Iterando este proceso, tendremos un intervalo tan pequeño como deseemos, y siempre con un cero de f encerrado en él. En concreto, bastará con repetir el proceso hasta que el ancho del intervalo sea menor que 2ε para que su punto medio se pueda considerar una aproximación aceptable.

En la figura 6.8 se muestra cómo se va aproximando el cero de la función f mediante la bipartición sucesiva del intervalo.

La codificación de este algoritmo es bastante simple: En primer lugar, conocida la función a la que queremos calcular un cero, debemos solicitar el máximo error permitido, `epsilon`, y los extremos del intervalo donde buscar el cero de la función, `a` y `b`. Después habrá que reducir el intervalo hasta que sea menor que $2 * \text{epsilon}$; en la reducción del intervalo se irán cambiando los valores de los extremos del intervalo, para lo cual se usarán las variables `izda` y `dcha`. La primera aproximación en pseudocódigo es la siguiente:

```

Program Biparticion (input, output);
  var
    epsilon, a, b, izda, dcha: real;
  begin
    Leer el error permitido, epsilon;
    Leer los extremos del intervalo, a,b;
    izda:= a; dcha:= b;
    Reducir el intervalo
    Imprimir el resultado
  end.   {Biparticion}

```

La lectura y salida de datos no presenta mayor dificultad, por su parte la tarea *reducir el intervalo* requiere la utilización de una variable adicional, `c`, para almacenar el valor del punto central del intervalo. Podemos refinar esta tarea mediante un bucle **while**, ya que no sabemos si será necesario ejecutar al menos una reducción del intervalo de partida (aunque es lo previsible):

```

while (dcha - izda) > 2 * epsilon do begin
  {Inv.: signo(f(izda)) ≠ signo(f(dcha))}
  c:= (dcha + izda) / 2;
  if f(dcha) * f(c) < 0 then
    {El cero se encuentra en [c,dcha]}
    izda:= c
  else
    {El cero se encuentra en [izda,c]}
    dcha:= c;
  end {while}
  {PostC.: c es la aproximación buscada}

```

Lo único que queda por completar, dejando aparte la entrada y salida de datos, consiste en la comprobación de la condición de la instrucción **if-then-else**, es decir $f(dcha) * f(c) < 0$, cuya codificación se realizará una vez conocida la función f .

El programa final del método es el siguiente:

```

Program Biparticion (input, output);
  var
    epsilon, a, b, c, izda, dcha: real;
begin
  {Entrada de datos}
  WriteLn('¿Error permitido?');
  ReadLn(epsilon);
  WriteLn('¿Extremos del intervalo?');
  ReadLn(a,b);
  izda:= a;
  dcha:= b;
  {Reducción del intervalo}
  while (dcha - izda) > 2 * epsilon do begin
    {Inv.: signo(f(izda)) ≠ signo(f(dcha))}
    c:= (dcha + izda) / 2;
    if f(dcha) * f(c) < 0 then
      {El cero se encuentra en [c,dcha]}
      izda:= c
    else
      {El cero se encuentra en [izda,c]}
      dcha:= c;
  end; {while}
  {PostC.: c es la aproximación buscada}
  WriteLn('Un cero de la función es ',c)
end. {Biparticion}

```

6.5.2 Método de Newton-Raphson

Sea nuevamente $f : \mathbb{R} \rightarrow \mathbb{R}$ una función continua que ahora tiene, además, derivada continua y no nula en todo \mathbb{R} . Si tiene un valor c que anula a f se sabe que, para cualquier $x_0 \in \mathbb{R}$, el valor

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

es una aproximación hacia c mejor que x_0 , y que la sucesión de primer elemento x_0 y de término general

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

converge rápidamente hacia el valor c .

El método de Newton-Raphson consiste en lo siguiente: dado un x_0 , se trata de recorrer la sucesión definida anteriormente hasta que dos valores consecutivos x_k y x_{k-1} satisfagan la desigualdad $|x_k - x_{k+1}| < \varepsilon$. En este caso x_k es la aproximación buscada.

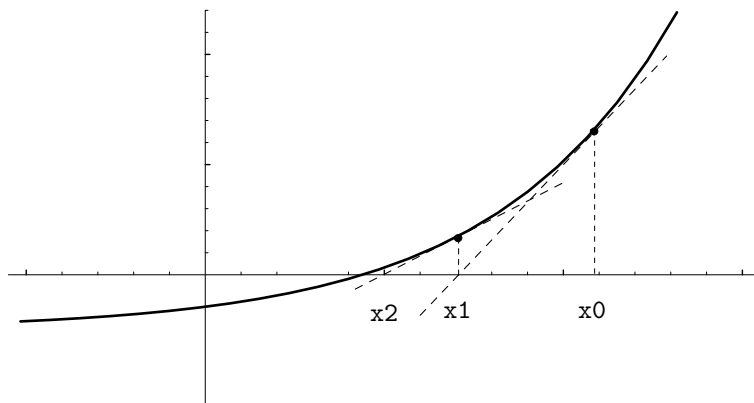


Figura 6.9. Aproximación por el método de Newton-Raphson.

En la figura 6.9 se muestra cómo se va aproximando el cero de la función f mediante la sucesión $x_0, x_1, x_2, \dots, x_n, \dots$

La codificación del método de Newton-Raphson no difiere demasiado de la de bipartición, ya que, esencialmente, ambas consisten en construir iterativamente una sucesión de aproximaciones a un cero de la función; la única diferencia estriba en la forma de construir la sucesión: en bipartición simplemente se calculaba el punto medio del intervalo de trabajo, mientras que en el método de Newton-Raphson, conocido x_n , el siguiente término viene dado por

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Para este algoritmo disponemos, al menos, de dos posibilidades que se enumeran a continuación:

1. Codificar directamente cada paso de iteración usando explícitamente la expresión de la función derivada f' (puesto que f es conocida es posible hallar f' usando las reglas de derivación), o bien
2. Usar el ejercicio 11 del capítulo 3 para aproximar el valor de f' cada vez que sea necesario.

Desde el punto de vista de la corrección del resultado obtenido resulta conveniente usar directamente la expresión de f' para evitar el posible efecto negativo causado por los errores acumulados tras cada aproximación de $f'(x_i)$. La implementación de este algoritmo se deja como ejercicio.

6.5.3 Inversión de funciones

Una aplicación del cálculo de ceros de funciones consiste en la inversión puntual de funciones, esto es, dada una función g y un punto a en su dominio calcular $g^{-1}(a)$.

Un ejemplo en el que se da esta necesidad podría ser el siguiente: Supóngase que se quiere poner en órbita un satélite de comunicaciones a una altura de 25 kilómetros, supóngase también que se conoce $h(t)$, la altura de la lanzadera espacial en metros en cada instante de tiempo t , entonces, el instante de tiempo preciso t_0 en el que se debe soltar el satélite de la lanzadera es tal que $h(t_0) = 25000$, esto es, $t_0 = h^{-1}(25000)$.

La inversión puntual de funciones consiste simplemente en un pequeño ardid que permite expresar la inversa puntual como el cero de cierta función: el método se basa en que calcular un valor⁸ de $g^{-1}(a)$ equivale a hallar un cero de la función f definida como $f(x) = g(x) - a$.

6.6 Ejercicios

1. Escriba un programa apropiado para cada una de las siguientes tareas:
 - (a) Pedir los dos términos de una fracción y dar el valor de la división correspondiente, a no ser que sea nulo el hipotético denominador, en cuyo caso se avisará del error.
 - (b) Pedir los coeficientes de una ecuación de segundo grado y dar las dos soluciones correspondientes, comprobando previamente si el discriminante es positivo o no.
 - (c) Pedir los coeficientes de la recta $ax + by + c = 0$ y dar su pendiente y su ordenada en el origen en caso de que existan, o el mensaje apropiado en otro caso.
 - (d) Pedir un número natural n y dar sus divisores.
2. (a) Escriba un programa que estudie la naturaleza de las soluciones del sistema de ecuaciones siguiente:

$$\begin{aligned} a_1x + b_1y &= c_1 \\ a_2x + b_2y &= c_2 \end{aligned}$$

y lo resuelva en el caso de ser compatible determinado.

- (b) Aplíquelo a los siguientes sistemas:

$$\left. \begin{aligned} 2x + 3y &= 5 \\ 3x - 2y &= 1 \end{aligned} \right\} \quad \left. \begin{aligned} 6x + 3y &= 12 \\ 2x + y &= 1 \end{aligned} \right\} \quad \left. \begin{aligned} 4x + 2y &= 8 \\ 6x + 3y &= 12 \end{aligned} \right\}$$

⁸Pues puede haber varios.

3. Escriba un programa que lea un carácter, correspondiente a un dígito hexadecimal:

`'0', '1', ..., '9', 'A', 'B', ..., 'F'`

y lo convierta en el valor decimal correspondiente:

`0, 1, ..., 9, 10, 11, ..., 15`

4. Para hallar en qué fecha cae el Domingo de Pascua de un **anno** cualquiera, basta con hallar las cantidades **a** y **b** siguientes:

`a := (19 * (anno mod 19) + 24) mod 30`

`b := (2 * (anno mod 4) + 4 * (anno mod 7) + 6 * a + 5) mod 7`

y entonces, ese Domingo es el *22 de marzo + a + b días*, que podría caer en abril.

Escriba un programa que realice estos cálculos, produciendo una entrada y salida claras.

5. Considere el siguiente fragmento de programa válido en Pascal:

```
...
if P then if Q then if x < 5 then WriteLn('a') else
WriteLn('b') else if x < y then WriteLn('c') else
WriteLn('d') else if x < 0 then if Q then WriteLn('e')
else WriteLn('f') else if Q then WriteLn('g')
else WriteLn('h')
...
```

siendo **P**, **Q**: **boolean** y **x**, **y**: **integer**.

- (a) Reescríbalo usando una disposición más clara.
- (b) Siendo $P \equiv x < 3$ y $Q \equiv y < x$, detecte y suprima las condiciones redundantes.
- (c) Detecte y suprima las condiciones redundantes, asumiéndose que en la entrada la siguiente sentencia es cierta:

$$(P = \neg Q) \wedge (x < y < 0)$$

6. El cuadrado de todo entero positivo impar se puede expresar como $8k + 1$, donde k es entero positivo; por ejemplo

$$\dots, \quad 3^2 = 8 * 1 + 1, \quad 5^2 = 8 * 3 + 1, \quad \dots$$

Sin embargo, algunos valores de k no producen cuadrados, como $k = 2$ que da 17. Se pide un programa que lea un número entero y lo clasifique según las posibilidades de los siguientes ejemplos,

4 es par
 5 es impar, pero no es de la forma $8k+1$
 17 es impar y de la forma $8k+1$ (para $k = 2$), pero no es un cuadrado perfecto.
 25 es impar, de la forma $8k+1$ (para $k = 3$), y cuadrado perfecto de 5

dando una salida como las indicadas.

7. ¿Qué hace el siguiente programa?

```

Program Letras (output);
  var
    fila, col: char;
begin
  for fila:= 'A' to 'Z' do begin
    for col:= 'A' to fila do
      Write(fila);
      WriteLn
    end {for}
end. {Letras}

```

8. Considérese la función

$$P(n) = \begin{cases} 3n + 1 & \text{si } n \text{ es impar} \\ n/2 & \text{si } n \text{ es par} \end{cases}$$

y sea N un número natural arbitrario. La sucesión numérica

$$\{N, P(N), P(P(N)), P(P(P(N))), \dots, P(P(\dots P(N)\dots)), \dots\}$$

son los llamados *números pedrisco* (véase el apartado 1.3.1) generados por N . Por ejemplo, para $N = 5$ su sucesión de números pedrisco es $\{5, 16, 8, 4, 2, 1, 4, 2, 1, \dots\}$ donde, a partir del sexto término, los valores 4, 2 y 1 se repiten indefinidamente.

Construya un programa que, dado un natural N , escriba su sucesión de números pedrisco y cuente las iteraciones necesarias para llegar a 1 (y entrar en el bucle 1, 4, 2, 1, ...). Aplicarlo a todos los enteros menores que 30. ¿Se observa algo especial para $N = 27$?

9. Escriba un programa cuyos datos son dos números naturales y una operación (suma, resta, multiplicación o división), y cuyo resultado es la cuenta correspondiente en el siguiente formato:

```

      1234
    *   25
    -----
      30850

```

Complete el programa anterior de manera que, antes de efectuar la operación seleccionada, verifique que va a ser calculada sin conflicto.

10. Escriba un programa para hallar $\binom{n}{k}$, donde n y k son datos enteros positivos,

(a) mediante la fórmula $\frac{n!}{(n-k)!k!}$

(b) mediante la fórmula $\frac{n(n-1)\dots(k+1)}{(n-k)!}$

¿Qué ventajas presenta la segunda con respecto a la primera?

11. **Integración definida.** Sea la función $f : \mathbb{R} \rightarrow \mathbb{R}$. Se puede dar un cálculo aproximado de su integral definida dividiendo el intervalo $[a, b]$ en n trozos, de base $\Delta = \frac{b-a}{n}$, y sumando las áreas de los n rectángulos de base Δ y alturas $f(x+i\Delta)$:

$$\int_a^b f(x)dx \simeq \sum_{i=1}^n \Delta f(x+i\Delta)$$

Escriba un programa que utilice esta expresión para calcular una aproximación de $\int_0^\pi \sin(x)dx$ y compruebe el grado de precisión del programa, comparando su resultado con el ofrecido por los métodos analíticos.

12. Los dos primeros términos de la sucesión de Fibonacci (véase el ejercicio 6 del tema 5) valen 1, y los demás se hallan sumando los dos anteriores: 1, 1, 2, 3, 5, 8, 13, 21, ... Confeccione un programa que lea una cota entera positiva, N , y genere términos de la citada sucesión hasta superar la cota fijada.

13. **Cifras de números.**

(a) Escriba un programa que averigüe cuántos dígitos tiene un número natural n . Concrete en qué condiciones debe estar ese dato para que el programa funcione correctamente.

(b) Escriba el programa “pasa pasa”, que lee dos números enteros positivos, a y b , y transfiere la última cifra de a a b :

$$(1234, 5678) \rightarrow (123, 56784)$$

(c) Escriba un programa que invierta un número natural, *dato*, transfiriendo todas sus cifras a otro, *resultado*, que inicialmente vale 0:

$$(1234, 0) \rightarrow (123, 4) \rightarrow (12, 43) \rightarrow (1, 432) \rightarrow (0, 4321)$$

(d) Escriba un programa que lea del **input** un literal entero positivo, expresado en el sistema de numeración hexadecimal, y halle la cantidad que representa expresándola en el sistema decimal usual.

14. **Primos y divisores.** Escriba un programa que realice las siguientes tareas:

(a) Que pida un número entero estrictamente mayor que uno y, una vez comprobada esa condición, busque su mayor divisor distinto de él.

(b) Que averigüe si un número n es primo, tanteando divisores a partir de 2 hasta dar con uno, o deteniendo la búsqueda al llegar al propio n .

- (c) Que averigüe si un número n es primo, pero parando la búsqueda (a lo más) al llegar a $\lfloor \frac{n}{2} \rfloor$.
- (d) Que averigüe si un número n es primo, pero parando la búsqueda (a lo más) al llegar a $\lfloor \sqrt{n} \rfloor$.

15. Número perfecto es el que es igual a la suma de sus divisores, excluido él mismo.

$$\begin{aligned} \text{Ejemplo:} & \quad 6 = 1 + 2 + 3 \\ \text{Contraejemplo:} & \quad 12 \neq 1 + 2 + 3 + 4 + 6 \end{aligned}$$

- (a) Escriba un programa que dé los números perfectos de 1 a 200.
 - (b) Escriba un programa que busque el primer número perfecto a partir de 200 (supuesto que, en efecto, hay alguno).
16. Escriba un programa que aplique el procedimiento de bipartición en el cálculo de un cero de la función f definida como:

$$f(x) = x^4 - \pi$$

en $[0, 3]$, con una tolerancia menor que una millonésima.
(Obsérvese que el resultado es una aproximación de $\sqrt[4]{\pi}$.)

17. Aplicar el método de bipartición en el cálculo aproximado de $\arcsen(\frac{1}{5})$, a través de un cero de la función f , definida así:

$$f(x) = \text{sen}(x) - \frac{1}{5}$$

con la misma tolerancia.

18. Aplicar el método de bipartición en el cálculo aproximado de $f(x) = e^x - 10$. Obsérvese que el resultado es una aproximación de $\log_e(10)$.
19. Usar el método de Newton-Raphson para cada uno de los apartados anteriores. ¿Qué método requiere menos iteraciones?

Capítulo 7

Programación estructurada

7.1	Introducción	123
7.2	Aspectos teóricos	125
7.3	Aspectos metodológicos	139
7.4	Refinamiento correcto de programas	146
7.5	Conclusión	151
7.6	Ejercicios	151
7.7	Referencias bibliográficas	153

En este capítulo se explican los aspectos necesarios para aplicar de una forma adecuada las instrucciones estructuradas presentadas en el capítulo anterior, de forma que los programas obtenidos sean claros, correctos y eficientes. En particular, se presentan los resultados teóricos que fundamentan la programación estructurada, y la metodología que permite la construcción de programas según este estilo de programación.

7.1 Introducción

Durante la corta historia de los computadores, el modo de programar ha sufrido grandes cambios. La programación era en sus comienzos todo un arte (esencialmente cuestión de inspiración); posteriormente diversas investigaciones teóricas han dado lugar a una serie de principios generales que permiten conformar el núcleo de conocimientos de una metodología de la programación. Ésta

consiste en obtener “programas de calidad”. Esto se puede valorar a través de diferentes características que se exponen a continuación, no necesariamente en orden de importancia:

- La *corrección* del programa que, obviamente, es el criterio indispensable, en el sentido de que se desean obtener programas correctos que resuelvan el(los) problema(s) para los que están diseñados.
- La *comprensibilidad*, que incluye la legibilidad y la buena documentación, características que permiten una mayor facilidad y comodidad en el mantenimiento de los programas.
- La *eficiencia*, que expresa los requerimientos de memoria y el tiempo de ejecución del programa.
- La *flexibilidad* o capacidad de adaptación del programa a variaciones del problema inicial, lo cual permite la utilización del programa durante mayor tiempo.
- La “*transportabilidad*”, que es la posibilidad de usar el mismo programa sobre distintos sistemas sin realizar cambios notables en su estructura.

Teniendo en cuenta que un programa, a lo largo de su vida, es escrito sólo una vez, pero leído, analizado y modificado muchas más, cobra una gran importancia adquirir técnicas de diseño y desarrollo adecuadas para obtener programas con las características reseñadas en la introducción. En este libro estudiamos dos técnicas, conocidas como *programación estructurada* y *programación con subprogramas*.

El objetivo que las técnicas anteriores se proponen es que los programas sean *comprensibles, correctos, flexibles* y “*transportables*”. Ambas técnicas no son excluyentes; más bien al contrario, un buen estilo de programación las integra, enfocando los problemas desde los dos puntos de vista simultáneamente. Para evitar la confusión que podría surgir entre ellas, en este capítulo nos centraremos en los principios de la programación estructurada, dejando los de la programación con subprogramas para los capítulos 8, 9 y 10.

Las ideas que dieron lugar a la programación estructurada ya fueron expuestas por E.W. Dijkstra en 1965, aunque el fundamento teórico (teoremas de la programación estructurada) está basado en los trabajos de Böhm y Jacopini publicados en 1966.

La programación estructurada es una técnica de programación cuyo objetivo es, esencialmente, la obtención de programas fiables y fácilmente mantenibles. Su estudio puede dividirse en dos partes bien diferenciadas: por un lado su estudio conceptual teórico, y por otro su aplicación práctica.

Por una parte, el estudio conceptual se centra en ver qué se entiende por programa estructurado para estudiar con detalle sus características fundamentales.

Por otra parte, dentro del enfoque práctico se presentará la metodología de refinamientos sucesivos que permite construir programas estructurados paso a paso, detallando cada vez más sus acciones componentes.

7.2 Aspectos teóricos

En este apartado se introducen los diagramas de flujo como medio para explicar lo que *no* es la programación estructurada. Estos diagramas han sido profusamente utilizados hasta hace bien poco. Un par de ejemplos nos demostrarán el caos que puede producir la falta de una organización adecuada.

Un uso más racional de los diagramas de flujo exige introducir condiciones que nos permitan hablar de programas “razonables”¹ y cuáles son los mecanismos cuya combinación permite expresar de forma ordenada cualquier programa que satisfaga estas condiciones.

Además se expone cómo se pueden reescribir en forma estructurada algunos programas razonables no estructurados. Ello permitirá deducir la gran aportación de esta metodología.

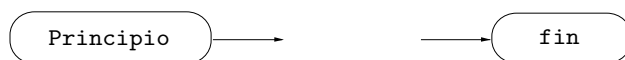
7.2.1 Programas y diagramas de flujo

Una práctica muy común de programación ha sido la utilización de *diagramas de flujo* (también llamados *organigramas*) como una descripción gráfica del algoritmo que se pretende programar. Sin embargo, esta popularidad ha ido menguando debido al débil (o nulo) soporte riguroso de su utilización; nosotros presentaremos los diagramas de flujo precisamente para mostrar lo que *no* es programación estructurada.

Para comprender mejor los problemas que surgen del uso incorrecto de los diagramas de flujo es necesario conocerlos un poco. Un diagrama de flujo se compone de bloques (que representan las acciones y las decisiones) y de líneas (que indican el encadenamiento entre los bloques).

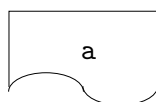
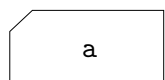
Los bloques de un diagrama de flujo pueden ser de cuatro clases distintas:

- Símbolos *terminales*, que indican el principio y el final del algoritmo. Se representan usando óvalos, como se indica a continuación:

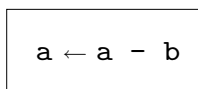


¹El sentido de este adjetivo se explicará en el apartado 7.2.2.

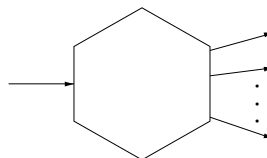
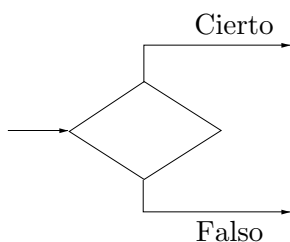
- Símbolos de *entrada* y *salida* de datos. Respectivamente, significan lectura y escritura, y se representan como se indica:



- Bloques de *procesamiento* de datos, que realizan operaciones con los datos leídos o con datos privados. Se representan mediante rectángulos que encierran la especificación del proceso, como por ejemplo:



- Nudos de *decisión*, en los que se elige entre dos o más alternativas. Según las alternativas sean dos (generalmente dependiendo de una expresión lógica) o más de dos se usa uno u otro de los siguientes símbolos:



A modo de ejemplo, en la figura 7.1 se muestra un sencillo diagrama de flujo que indica el procedimiento de multiplicar dos números enteros positivos mediante sumas sucesivas.

Sin embargo, no todos los diagramas de flujo son tan claros como el anterior. Como muestra considérese el diagrama de flujo de la figura 7.2: si un diagrama de flujo se escribe de cualquier manera, aun siendo correcto desde el punto de vista de su funcionamiento, puede resultar engorroso, críptico, ilegible y casi imposible de modificar.

Por otra parte, en la figura 7.3 se observa una disposición mucho más clara del mismo programa que favorece su comprensión y facilita su codificación.

7.2.2 Diagramas y diagramas propios

El ejemplo anterior resalta la necesidad de una metodología que sirva para evitar diagramas tan confusos como el de la figura 7.2. Para formalizar esta

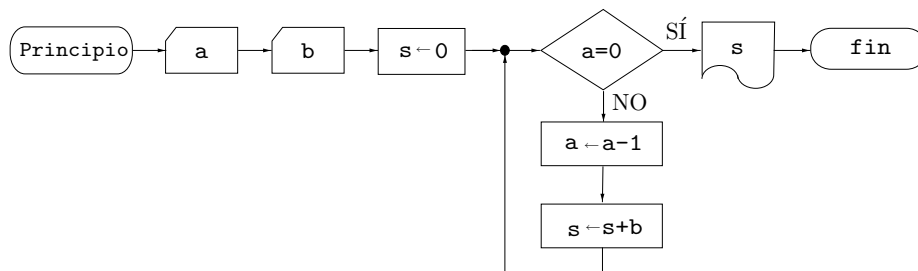


Figura 7.1. Diagrama de flujo para el producto de números naturales.

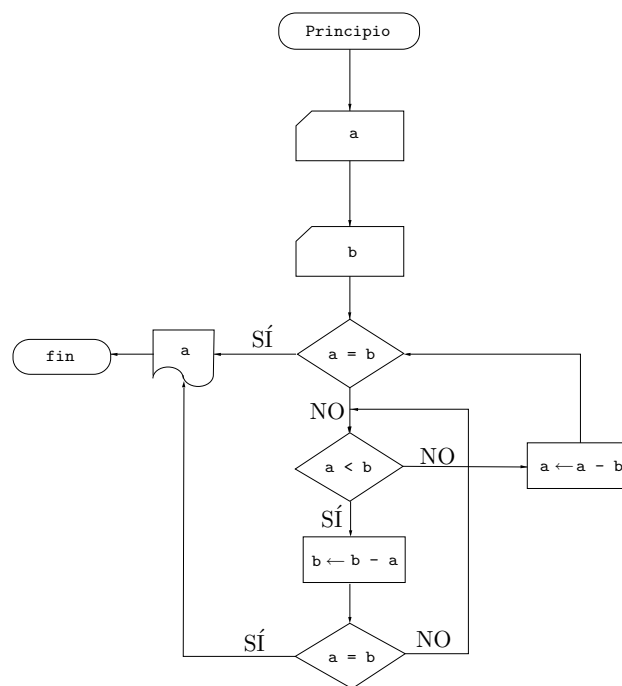


Figura 7.2. Un diagrama de flujo algo confuso.

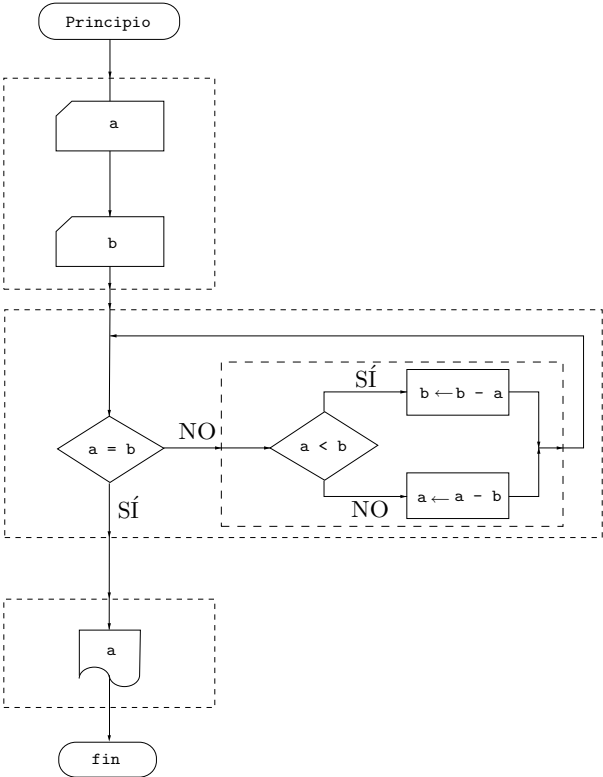
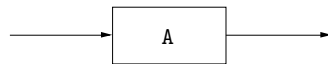


Figura 7.3. Versión bien organizada del diagrama anterior .

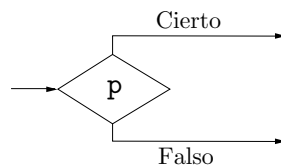
metodología será necesario disponer de una cierta clase de diagramas permitidos a partir de los cuales construir la teoría. Entre éstos se destaca la subclase de los diagramas *propios*, que representan, desde cierto punto de vista, a los programas correctamente estructurados.

En este apartado se restringe el concepto de *diagrama*, que será utilizado más adelante en la definición de programa estructurado. Consideraremos que un *diagrama* se construye usando como elementos básicos únicamente las tres siguientes piezas:

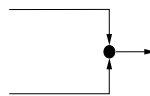
- *Acción*, que sirve para representar una instrucción (por ejemplo de lectura, escritura, asignación. . .).



- *Condición*, que sirve para bifurcar el flujo del programa dependiendo del valor (verdadero o falso) de una expresión lógica.



- *Agrupamiento*, que sirve, como su nombre indica, para agrupar líneas de flujo con distintas procedencias.



A continuación se definen y se dan ejemplos de diagramas propios, que será lo que consideraremos como programa “razonable”. El lector puede juzgar tras leer la definición y los ejemplos lo acertado del calificativo.

Definición: Se dice que un diagrama, construido con los elementos citados arriba, es un *diagrama propio* (o *limpio*) si reúne las dos condiciones siguientes:

1. Todo bloque posee un único punto de entrada y otro único punto de salida.
2. Para cualquier bloque, existe al menos un camino desde la entrada hasta él y otro camino desde él hasta la salida.

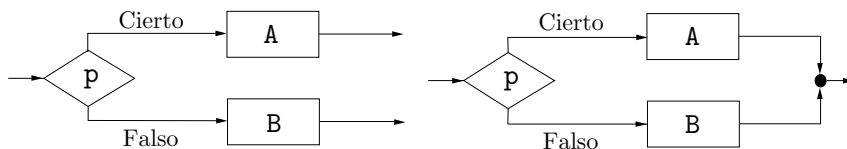


Figura 7.4. Un diagrama no propio y un diagrama propio.

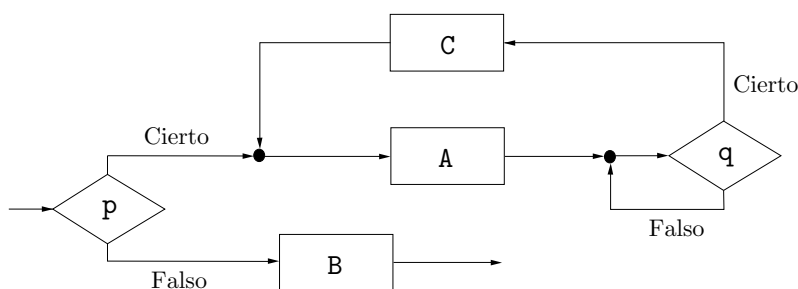


Figura 7.5. Un diagrama no propio.

Estas condiciones restringen el concepto de diagrama de modo que sólo se permite trabajar con aquéllos que están diseñados mediante el uso apropiado del agrupamiento y sin bloques superfluos o formando bucles sin salida.

En el diagrama de la izquierda de la figura 7.4 se muestra un ejemplo de un diagrama que no es propio por no tener una única salida. Agrupando las salidas se obtiene un diagrama propio (a la derecha de la figura). En la figura 7.5 se observa otro diagrama que no es propio, ya que existen bloques (los A y C y q) que no tienen un camino hasta la salida; si el programa llegara hasta esos bloques se colapsaría, pues no es posible terminar la ejecución. Finalmente, en la figura 7.6 aparece un diagrama que contiene bloques inaccesibles desde la entrada del diagrama.

Algunos autores exigen una condición adicional a un diagrama para considerarlo *propio*: no contener bucles infinitos. Sin embargo esta propiedad está más estrechamente relacionada con la verificación que con el diseño de programas.

7.2.3 Diagramas BJ (de Böhm y Jacopini)

Los diagramas BJ son tres diagramas especialmente importantes; esta importancia se debe a que pueden considerarse esquemas de acciones muy naturales y completamente expresivos, en el sentido de que cualquier programa razonable

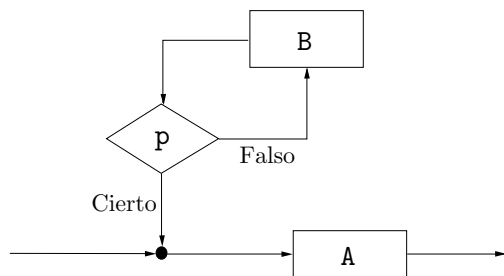
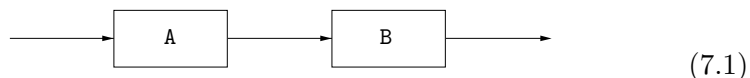


Figura 7.6. Un diagrama con elementos inaccesibles.

se puede reorganizar de forma ordenada combinando sólo estos esquemas. Esta característica tiene bastante utilidad a la hora del diseño de programas, ya que afirma que cualquier programa razonable (por ejemplo, el de la figura 7.2) puede escribirse de forma ordenada (como en la figura 7.3). Estas ideas se desarrollarán con más precisión en el apartado 7.2.4.

Definición: Un diagrama se dice que es un *diagrama BJ* (*diagrama de Böhm y Jacopini* o *diagrama privilegiado*), si está construido a partir de los siguientes esquemas:

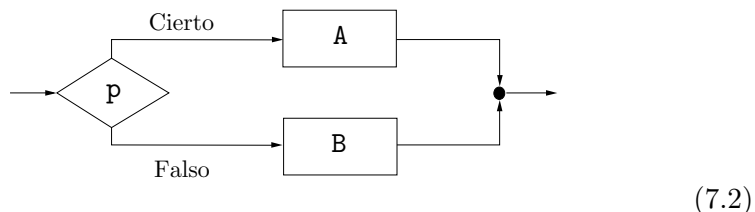
1. La *secuencia* de dos acciones A y B, ya sean simples o compuestas:



La secuencia se suele denotar como Bloque (A,B).

El equivalente en Pascal de este diagrama es la composición de instrucciones.

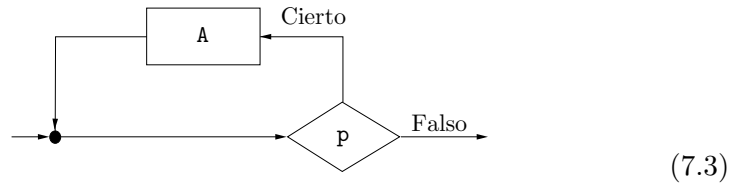
2. La *selección* entre dos acciones A y B dependiendo de un predicado p. Los subprogramas, como es obvio, pueden consistir en acciones simples o compuestas (obsérvese que el agrupamiento posterior es esencial).



El significado de esta construcción es *si p es cierto entonces se ejecuta A y si no se ejecuta B*.

En Pascal este diagrama se corresponde con la instrucción **if-then-else**.

3. La *iteración* repite una acción A dependiendo del valor de verdad de un predicado de control p.



El significado de este tipo de iteración es *mientras que p es cierto hacer A* y se denota mediante `DoWhile(p,A)`.

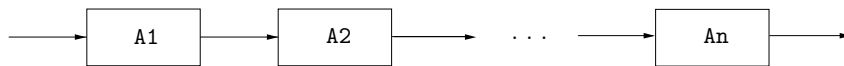
En esta construcción, el predicado p actúa como un control sobre la iteración, esto es, si se verifica p entonces se ejecuta A.

Observando el diagrama (7.3) se observa que se comprueba el valor del predicado antes de ejecutar la acción (el bucle es preprobado), con lo cual en Pascal este diagrama de iteración se corresponde con la instrucción **while**.

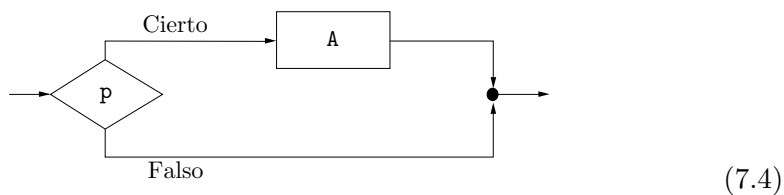
Otros diagramas de uso frecuente

Ya se ha comentado antes que es posible expresar todo diagrama propio usando solamente los tres esquemas anteriores (esto es consecuencia de los resultados matemáticos de Böhm y Jacopini); sin embargo, con vistas a obtener una representación más agradable, se pueden considerar también algunos tipos adicionales de esquemas que son versiones modificadas de la secuencia, la selección y la repetición.

Así, el esquema de la secuencia se puede generalizar para representar una secuencia de n subprogramas A_1, \dots, A_n :

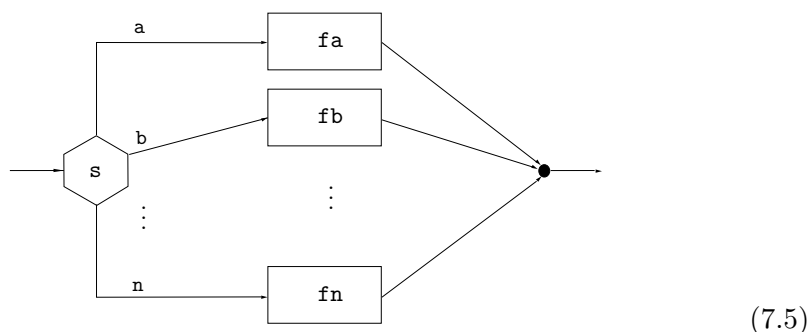


Si hacemos uso de la acción vacía (que se corresponde con una instrucción que no hace nada) se puede considerar la siguiente variante de la selección:



que se representa con la fórmula $\text{IfThen}(p, A)$, y que en Pascal se corresponde con la instrucción **if-then-else** cuando se omite la rama opcional **else**.

Una generalización interesante de la estructura de selección consiste en una ramificación en n ramas (en lugar de dos) tras evaluar una condición. Esta generalización se representa mediante el diagrama



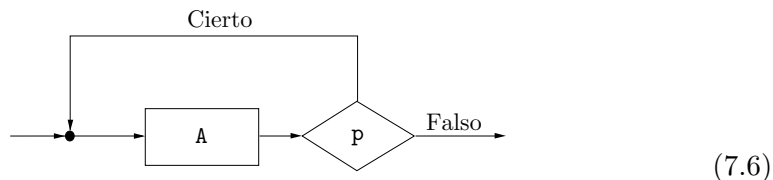
que se suele leer como:

según s valga
 $a \rightarrow$ hacer fa
 \vdots
 $n \rightarrow$ hacer fn

y se corresponde con la instrucción **case** en Pascal.

Este esquema permite en muchas ocasiones expresar situaciones que sólo podrían especificarse usando un anidamiento múltiple de instrucciones de selección. Este hecho se hace patente en la figura 7.7.

Por otra parte, a veces resulta conveniente usar un esquema de iteración alternativo al preprobado, en el que el predicado no actúe como condición de entrada a la iteración sino como una condición que tiene que ser cierta para salir de la iteración. Esta construcción puede expresarse así:



y se denota como $\text{DoUntil}(p, A)$. La interpretación de esta estructura consiste en repetir la acción A hasta que la condición p se haga falsa. En Pascal este diagrama se corresponde con los bucles **repeat**.

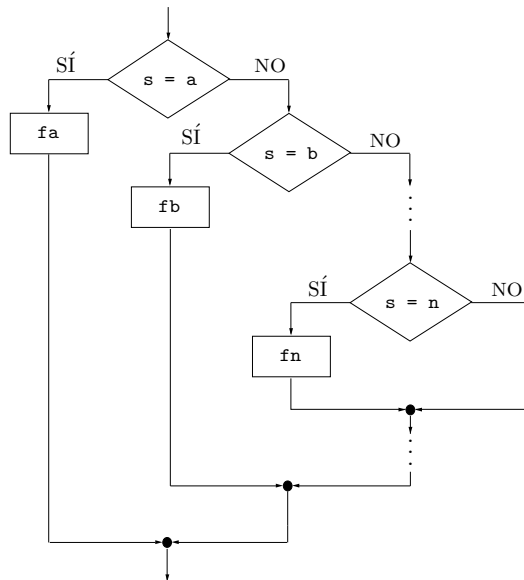


Figura 7.7. Expresión de `CaseOf` en función de `IfThenElse`.

Programas estructurados

Definición: Diremos que un diagrama representa a un *programa estructurado* si está formado combinando los diagramas privilegiados de secuencia (7.1), selección (7.2) y/o repetición (7.3).

Como consecuencia de lo expuesto en el apartado anterior, un diagrama representa un programa estructurado si se puede expresar haciendo uso de cualesquiera de los diagramas (7.1) ... (7.6); por lo tanto, todo programa estructurado presenta una descomposición arborescente en la que cada nodo se corresponde directamente con una instrucción de Pascal o con una condición.

Cualquier acción (instrucción o subprograma) de un programa estructurado puede ser sustituida por su descomposición arborescente y viceversa. Esta propiedad simplifica el razonamiento sobre el programa al hacerlo mucho más legible, además de facilitar su mantenimiento (lo más probable es que sólo haya que realizar modificaciones en subárboles de la estructura general).

Según la definición, un programa estructurado P no tiene por qué estar expresado como diagrama privilegiado, sin embargo, es obvio que precisamente esta expresión es la realmente importante. Para obtener programas estructurados se introduce la metodología de diseño descendente de programas en la cual se hace bastante uso del pseudocódigo; todo esto se verá en el apartado 7.3.

7.2.4 Equivalencia de diagramas

Tras definir lo que se entiende por programa estructurado cabe plantearse si un programa dado en forma no estructurada puede expresarse de forma “equivalente” mediante un programa estructurado. En este apartado se detalla el concepto de equivalencia de diagramas, que será usado más adelante al enunciar los teoremas de la programación estructurada.

Dos diagramas propios se dice que son *equivalentes* si designan los mismos cálculos; esto es, si para una misma entrada de datos las líneas de flujo llevan a bloques idénticos.

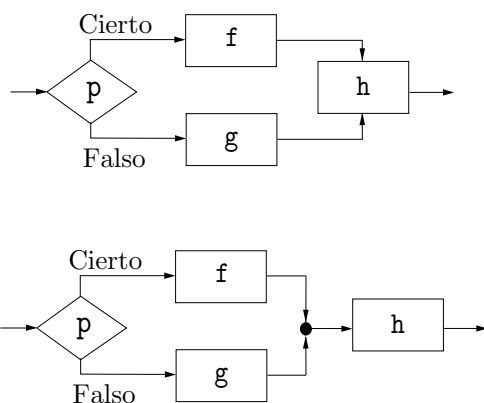
Como ejemplo, considérense los diagramas de las figuras 7.2 y 7.3: no resulta difícil comprobar que, para una misma entrada de datos, los cálculos especificados por cada diagrama son exactamente los mismos y, por lo tanto, ambos diagramas son equivalentes.

Los teoremas de la programación estructurada (véase el apartado 7.2.5) afirman que todo diagrama propio se puede expresar equivalentemente como un diagrama privilegiado. El problema es la construcción del diagrama privilegiado equivalente.

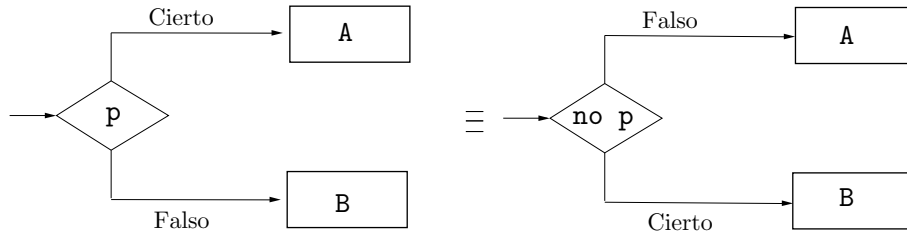
Para realizar esta transformación se pueden usar, entre otras, las operaciones de agrupamiento, inversión de predicados y desdoblamiento de bucles. A continuación se describirán estas operaciones y veremos algunos ejemplos que aclararán estos conceptos.

Mediante el *agrupamiento* podemos evitar la multiplicidad de salidas de un programa, o que un bloque tenga más de una flecha de entrada. En el ejemplo de la figura 7.4 se mostró un programa no propio porque tiene dos salidas; haciendo uso del agrupamiento se convierte en un programa estructurado. Un caso similar se muestra en el siguiente ejemplo, en el que también se hace uso de un

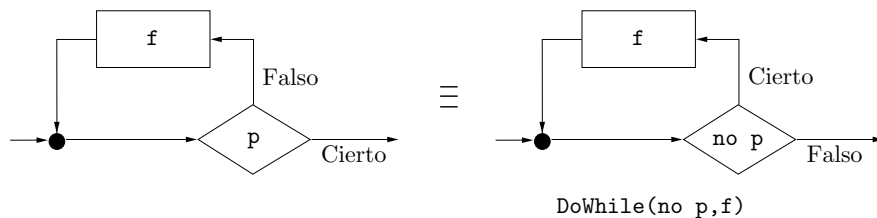
agrupamiento tras una selección.



La *inversión* de un predicado consiste en negar la condición de una selección de modo que se intercambien las etiquetas de las ramas.



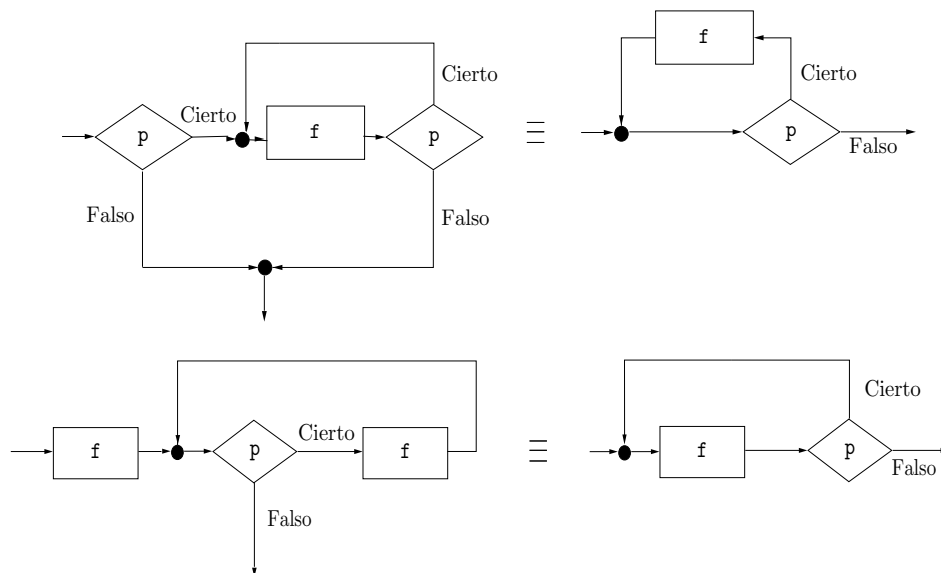
Esta transformación es especialmente útil en casos de iteración: a veces es necesario invertir el bucle para que aquella se realice sólo cuando la condición es cierta.² Por ejemplo, el diagrama de la izquierda se puede modificar mediante la inversión del predicado p y expresarlo como aparece a la derecha



que es un diagrama estructurado del tipo $\text{DoWhile}(\text{no } p, f)$.

²Véase la definición del bloque de iteración.

Finalmente, las dos equivalencias siguientes de desdoblamiento de bucles pueden hacer más compacto el diagrama con el que se esté trabajando:



7.2.5 Teoremas de la programación estructurada

En este apartado se enuncian los resultados más importantes de la programación estructurada y se comentan sus consecuencias. El primero de todos ellos es el teorema de estructura, que dice que todo programa propio admite una expresión estructurada. Más formalmente, en términos de diagramas, se enuncia así:

Teorema 7.1 (de estructura) *Todo diagrama propio es equivalente a un diagrama privilegiado.*

Puesto que todos los diagramas privilegiados admiten una expresión arborescente, como consecuencia de este teorema se obtiene que todo programa propio es equivalente a un programa que tiene alguna de las siguientes formas:

- Bloque(A,B),
- IfThenElse(p,A,B),
- DoWhile(p,A),

donde p es un predicado del programa original y las acciones A y B son bien instrucciones o bien (sub)programas privilegiados.

- ☉☉ Teniendo en cuenta que todo diagrama propio se puede codificar mediante instrucciones estructuradas, del enunciado del teorema se deduce que `IfThen`, `DoUntil`, `CaseOf` y `DoFor` se pueden expresar en términos de las construcciones `Bloque`, `IfThenElse` y `DoWhile`.

El segundo teorema de la programación estructurada es el teorema de corrección (o validación).

Teorema 7.2 (de corrección) *La corrección de un programa estructurado se puede estudiar mediante pasos sucesivos, examinando cada esquema (nodo) de su estructura arborescente y validando localmente la descomposición realizada en ese nodo.*

La importancia de este teorema reside en que permite, al menos teóricamente, validar (o comprobar la corrección de) un programa a la vez que éste se está construyendo. La técnica de diseño descendente facilita la verificación, ya que basta con validar cada uno de los refinamientos realizados; esta técnica se muestra en el apartado 7.3.2.

7.2.6 Recapitulación

Una vez presentados los aspectos teóricos de la programación estructurada, merece la pena extraer algunas consecuencias de utilidad práctica de lo visto hasta ahora, en especial de los diagramas privilegiados.

En el primer párrafo del apartado 7.2.3 se adelantaban aproximadamente las siguientes ideas:

- Los diagramas BJ representan acciones muy naturales; tanto, que se reflejan en frases corrientes de cualquier lenguaje natural, como:
 - Hacer primero esto, luego eso y luego aquello (secuencia).
 - Si llueve iré en coche, si no caminando (selección).
 - Mientras tenga fuerzas seguiré luchando (iteración).

Esta naturalidad permite construir diagramas con *organización clara y sencilla que facilita el estudio de la corrección de los programas*.

- Los diagramas BJ son suficientemente expresivos como para, combinándose entre sí, expresar cualquier programa razonable.
- Por lo tanto, resulta ser altamente recomendable habituarse a desarrollar programas estructurados.

Precisamente, en el siguiente apartado se comienzan a estudiar las repercusiones de la programación estructurada en la metodología de la programación.

7.3 Aspectos metodológicos

Hasta ahora se ha estudiado la programación estructurada, pero apenas se han incluido las implicaciones de esta teoría en el proceso de programación. En este apartado comienza la exposición de la técnica de diseño descendente, que permite aplicar la teoría introducida para construir programas estructurados.

La técnica de *diseño descendente* (en inglés, *top-down*) está basada en un proceso de aproximación sucesiva a la solución del problema planteado. El apelativo de diseño descendente surge del hecho de que se parte de una especificación abstracta del problema por resolver para, mediante refinamientos sucesivos, ir “descendiendo” hasta cubrir todos los detalles y describir el programa en un lenguaje de programación.

También existe la técnica contraria, llamada *diseño ascendente* (en inglés, *bottom-up*), que parte de soluciones concretas disponibles para diferentes partes del problema y las integra para generar la solución total.

En los capítulos anteriores ya se ha usado el seudocódigo en varias ocasiones; este apartado comienza con un repaso de sus cualidades y su papel dentro del diseño descendente de programas.

7.3.1 Seudocódigo

El *seudocódigo* es un lenguaje intermedio que sirve de puente entre un lenguaje natural (como el español, por ejemplo) y ciertos lenguajes de programación (como Pascal). Es útil como primera aproximación, y es muy apropiado para describir refinamientos progresivos de un programa, permitiendo al programador prescindir de los detalles y limitaciones que tienen los lenguajes específicos y así poder concentrarse sólo en la lógica del programa.

Puesto que el seudocódigo *no es* un lenguaje formal, existe una gran libertad en el uso de recursos, lo cual permite muchas veces dejar sin detallar algunos fragmentos de programa. Esta característica hace que el seudocódigo facilite el abordar un problema mediante el diseño descendente (véase el apartado 7.3.2).

A continuación se describe un fragmento de seudocódigo con la potencia expresiva suficiente para expresar las instrucciones estructuradas principales: la secuencia, la selección y la iteración, con sus principales variantes.

La secuencia de bloques de programa se denota mediante una lista de acciones consecutivas. Por ejemplo, la secuencia de tareas que hay que seguir para realizar una llamada desde un teléfono público se puede expresar en seudocódigo del modo siguiente:

Buscar una cabina libre

Insertar monedas
Marcar el número deseado

Para expresar una selección del tipo `IfThenElse(p,A,B)` o `IfThen(p,A)` se usará su equivalente en español. En el ejemplo anterior pudiera ocurrir que no recordemos el número al que deseamos marcar, en ese caso se presenta una selección: si se recuerda el número se marca, y si no se pide información.

Buscar una cabina libre
Insertar monedas
si se recuerda el número entonces
 Marcar el número deseado
si no
 Pedir información y marcar

En este caso, la tarea de pedir información se descompone como una secuencia: marcar el 003 y pedir el número deseado. Esta secuencia de tareas puede interferir en la secuencia primitiva (la de realizar la llamada); para evitar este posible conflicto se hace uso de un nivel más de anidamiento, con su correspondiente sangrado, como se muestra a continuación:

Buscar una cabina libre
Insertar monedas
si se recuerda el número entonces
 Marcar el número deseado
si no
 Marcar el 003
 Pedir el número deseado
 Marcar el número obtenido

Las iteraciones del tipo `DoUntil(p,A)` se ilustran a continuación con el método de estudio más perfecto que existe para aprender una lección de forma autodidacta.

repetir
 Estudiar detenidamente la lección
 Intentar todos los ejercicios
hasta que las técnicas se dominen perfectamente

El seudocódigo relativo a las iteraciones del tipo `DoWhile(p,f)` se muestra a continuación; en este caso se presenta un método alternativo al ejemplo anterior.

mientras no se dominen las técnicas hacer
 Estudiar detenidamente la lección
 Intentar todos los ejercicios

Obsérvese el distinto matiz de cada ejemplo. Este matiz refleja la distinción, resaltada anteriormente, entre los bucles `DoWhile` y `DoUntil`: para el estudio autodidacta es necesario estudiar al menos una vez la lección, mientras que si se atiende en clase³ este paso puede no ser necesario.

7.3.2 Diseño descendente

La técnica de programación de diseño descendente que se comenta en este apartado está basada en el empleo de refinamientos sucesivos para la obtención de programas que resuelvan un cierto problema.

En cierto modo, la técnica de refinamiento progresivo se puede comparar con la actitud de un escultor ante un bloque de mármol con el objetivo de obtener un desnudo humano: para empezar, usando el cincel y el martillo grandes, procederá a esbozar sin mucho miramiento una figura humanoide con cabeza, tronco y extremidades; posteriormente, y ya con útiles de precisión, comenzará la labor de refinamiento y la obtención de detalles específicos.

En el diseño descendente, se parte de una especificación en lenguaje natural del problema que se quiere resolver y, sucesivamente, se va “depurando” poco a poco, perfilándose mejor las distintas partes del programa, apareciendo unos detalles acabados y otros a medio camino. El proceso de refinamiento continúa hasta que finalmente se obtiene una versión en la que el nivel de detalle de los objetos y las acciones descritos puede expresarse de forma comprensible por el computador.

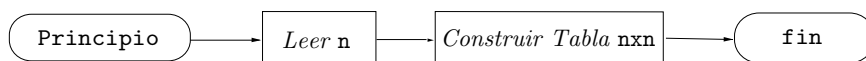
A continuación se muestran algunos ejemplos de refinamiento progresivo:

Un ejemplo numérico: tablas de multiplicar

Apliquemos la técnica de diseño descendente al problema de escribir una tabla de multiplicar de tamaño $n \times n$. Por ejemplo, para $n = 10$ tendríamos

1	2	3	...	10
2	4	6	...	20
3	6	9	...	30
⋮	⋮	⋮	⋮	⋮
10	20	30	...	100

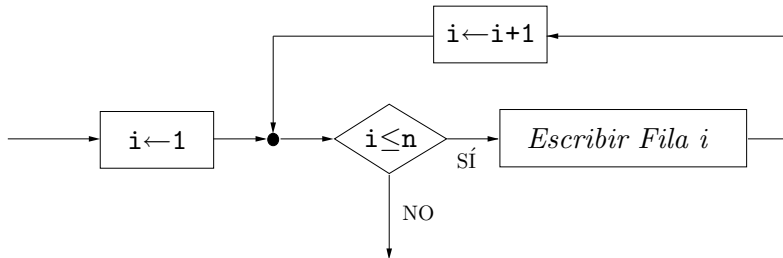
Una primera versión (burda) del programa podría ser



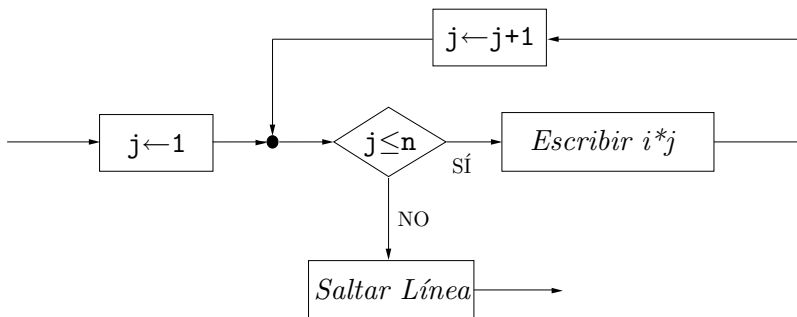
³... y se dispone de un buen profesor...

donde el programa se plantea como la secuencia de dos acciones: la primera consiste en conocer el tamaño de la tabla deseada, *Leer n*, y la segunda en construir tal tabla. La primera de las acciones está suficientemente refinada (se puede traducir directamente a un lenguaje de programación) pero no así la segunda.

La construcción de la tabla se puede realizar fácilmente escribiendo en una fila los múltiplos de 1, en la fila inferior los múltiplos de 2... hasta que llegemos a los múltiplos de n . Ésta es la idea subyacente al siguiente refinamiento de la función que construye la tabla de tamaño $n \times n$



donde aparece la acción *Escribir Fila i*, que escribe cada una de las filas de la tabla (no se debe olvidar añadir un salto de línea detrás del último número de la fila). Esta función aparece especificada con mayor detalle a continuación



en esta función todos los bloques aparecen completamente refinados, con lo cual se habría terminado.

El desarrollo descendente de programas no suele hacerse mediante diagramas como hemos hecho en este ejemplo, aunque se ha presentado así para ilustrar de forma gráfica los conceptos estudiados en este capítulo. En la práctica, lo que se hace es refinar progresivamente usando pseudocódigo, de modo que al llegar al último refinamiento la traducción a un lenguaje de programación sea prácticamente inmediata.

La primera aproximación en pseudocódigo al programa que calcula la tabla de $n \times n$ podría ser la siguiente:

Leer n
Construir la tabla n

A continuación se muestra el primer refinamiento, en el que se especifica la función que construye la tabla como la aplicación sucesiva de la acción que construye cada una de las filas de la tabla:

Leer n
 {Construir la tabla:}
 para $i \leftarrow 1$ hasta n hacer
 Escribir la línea i -ésima

El paso siguiente consiste en depurar la especificación de la función que construye las filas.

Leer n
 para $i \leftarrow 1$ hasta n hacer
 {Escribir la línea i -ésima:}
 para $j \leftarrow 1$ hasta n hacer
 Escribir $i*j$
 Salto de línea

Y esta versión admite una traducción directa a Pascal como la siguiente:

```

Program Tabla (input, output);
  var
    n, i, j: integer;
begin
  {Petición de datos:}
  Write('Escriba el valor de n y pulse intro: ');
  ReadLn(n);
  {Construcción de la tabla:}
  for i:= 1 to n do begin
    {Escribir la línea i-ésima:}
    for j:= 1 to n do
      Write(i * j:6); {Elemento i-j-ésimo}
    WriteLn {Salto de línea}
  end {for i:= 1}
end. {Tabla}

```

Es de resaltar que la idea del refinamiento progresivo consiste simplemente en caminar desde el seudocódigo hasta, por ejemplo, el PASCAL; por esta razón, durante el refinamiento se puede escribir directamente en Pascal lo que se traduce trivialmente, por ejemplo, escribir $a := b$ en lugar de $a \leftarrow b$. En el siguiente ejemplo utilizaremos esta observación.

Otro ejemplo numérico: suma parcial de una serie

En este caso pretendemos aplicar las técnicas de diseño descendente para obtener un programa que realice el cálculo de

$$\sum_{i=1}^n \frac{i+1}{i!}$$

La primera versión de este programa podría ser la siguiente:

Leer n
Calcular la suma
Escribir la suma

En este fragmento de pseudocódigo simplemente se descompone la tarea encomendada en tres subtarefas: la entrada de información, su manipulación y la salida de información buscada. El siguiente refinamiento ha de ser realizado sobre la manipulación de la información, esto es, habrá que especificar cómo se calcula la suma buscada.

La suma se puede calcular mediante una variable acumulador `suma` y un bucle del tipo `DoFor` que en la iteración `i`-ésima calcule el término `i`-ésimo y lo añada a la suma parcial `suma`. Este refinamiento aparece reflejado en el siguiente fragmento:

```
{Calcular la suma:}
suma:= 0
para i:= 1 hasta n hacer
    Hallar término i-ésimo, t
    Añadir t a suma
```

Sólo queda por refinar la acción de calcular el término `i`-ésimo, ya que, conocido éste, añadirlo a `suma` no plantea mayores problemas.

Conocido `i`, para calcular el término $t_i = \frac{i+1}{i!}$ bastará con calcular (iterativamente) el denominador y realizar la asignación correspondiente

```
{Hallar término i-ésimo:}
Hallar denominador, denom = i!
term:= (i + 1)/denom
```

El refinamiento del cálculo del denominador vuelve a ser un bucle del tipo `DoFor`, que usando pseudocódigo se puede escribir así:

```

    {Hallar denominador, denom = i!}
denom:= 1;
para j:= 1 hasta i hacer
    denom:= denom * j

```

Agrupando todos los refinamientos finalmente tendríamos el siguiente programa en Pascal:

```

Program Sumatorio (input, output);
var
    denom, n, i, j: integer;
    suma, term: real;
begin
    {Entrada de datos:}
    Write('Escriba el valor de n y pulse intro: ');
    ReadLn(n);
    {Cálculo de la suma:}
    suma:= 0;
    for i:= 1 to n do begin
        {Cálculo del término i-ésimo: (i + 1)/i!}
        denom:= 1;
        for j:= 1 to i do
            denom:= denom * j;
        term:= (i + 1)/denom;
        {Acumularlo:}
        suma:= suma + term
    end; {for i:= 1}
    {Salida de resultados:}
    WriteLn('Suma = ',suma:12:10)
end. {Sumatorio}

```

Mejora de repeticiones innecesarias

El ejemplo anterior es correcto, aunque sensiblemente mejorable. En efecto, se observa que los factoriales hallados en cada vuelta,

$$1!, 2!, \dots, (i-1)!, i!, \dots, n!$$

pueden hallarse más rápidamente simplemente actualizando el precedente. Así pues, en vez de

```

    {Hallar denom = i!}
denom:= 1;
for j:= 1 to i do
    denom:= denom * j

```

se puede hacer, simplemente,

```
{Actualizar denom = i! (desde denom=(i-1)!}
denom:= denom * i
```

con el mismo efecto e indudablemente más rápido. En este caso se requiere establecer el denominador inicialmente a uno antes de entrar en el bucle. En resumen,

```
suma:= 0;
denom:= 1;
for i:= 1 to n do begin
  {Cálculo del término i-ésimo: (i + 1)/i!}
  denom:= denom * i;
  term:= (i + 1)/denom;
  {Acumular al término:}
  suma:= suma + term
end {for}
```

Este método se basa en la posibilidad de calcular cierta(s) (sub)expresión(es) reciclando otras anteriores, economizando así los cálculos. Precisamente, el nombre de *diferenciación finita* procede de la actualización de las expresiones necesarias hallando sólo la “diferencia” con respecto a las anteriores.⁴

En el ejercicio 5f del capítulo 18 se comprueba cómo esta segunda versión constituye una mejora importante en cuanto al tiempo de ejecución del programa.

7.4 Refinamiento correcto de programas con instrucciones estructuradas

La idea de este apartado es aplicar la metodología anterior para resolver, con garantías, un problema mediante un programa descomponiendo el problema en subproblemas y refinando (resolviendo) éstos posteriormente. Para que el refinamiento sea correcto habrá que definir con precisión el cometido de cada subproblema, garantizar su corrección y organizar (estructurar) bien entre sí los (sub)algoritmos respectivos.

Dicho de otro modo, para lograr programas correctos mediante el método de los refinamientos progresivos, debe asegurarse la corrección en cada paso del

⁴El esquema de mejora ejemplificado tiene su origen en la tabulación de polinomios de forma eficiente por el método de las diferencias (Briggs, s. XVI).

desarrollo:⁵ en la definición rigurosa de los subalgoritmos (encargados de resolver problemas parciales) y en el correcto ensamblaje (estructurado) de los mismos.

Concretaremos estas ideas con un ejemplo detallado. Después de estudiarlo, recomendamos releer los dos párrafos anteriores.

7.4.1 Un ejemplo detallado

Se plantea resolver el siguiente problema: dado el natural N , deseamos averiguar si es un cuadrado perfecto o no; esto es, si existe un natural k tal que $k^2 = N$.

Dejando de lado la solución trivial,

$$\text{Sqr}(\text{Round}(\text{Sqr}(\text{N}))) = \text{N}$$

operaremos *buscando* el número natural k (si existe) tal que $k^2 = N$, o parando la búsqueda cuando sepamos con seguridad que no existe tal número k .

Las dos soluciones que explicamos a continuación se basan en buscar el mínimo $k \in \mathbb{N}$ tal que $k^2 \geq N$. Llamemos m a esa cantidad,

$$m = \min \{k \in \mathbb{N} \text{ tal que } k^2 \geq N\}$$

que existe con seguridad para cualquier $N \in \mathbb{N}$ y es único. Entonces, se puede asegurar que

- Todo natural $k < m$ verifica que $k^2 < N$
- Todo natural $k > m$ verifica que $k^2 > N$

Por lo tanto, hay dos posibilidades:

- Que $m^2 = N$, en cuyo caso N sí es un cuadrado perfecto.
- Que $m^2 > N$, en cuyo caso N no es un cuadrado perfecto.

En otros términos, tenemos en un nivel de refinamiento intermedio:

```

var
  n,      {Dato}
  m : integer;  {Número buscado}
...
ReadLn(n); {Sup. n>=0}
  Buscar el número m descrito

```

⁵En vez de razonar (verificar), *a posteriori*, la corrección de un programa ya desarrollado.

```

    {m = mín k ∈ IN tal que k2 ≥ n}
  if Sqr(m) = n then
    WriteLn(n, 'sí es cuadrado perfecto')
  else
    WriteLn(n, 'no es cuadrado perfecto')

```

La garantía de que el mensaje emitido es correcto se tiene porque:

- La rama **then** se ejecuta cuando resulta cierta la condición ($m^2 = N$), lo que basta para que N sea un cuadrado perfecto.
- La rama **else** se ejecuta cuando la condición $m^2 = N$ es falsa. Como, además,

$$m = \text{mín } \{k \in \mathbb{IN} \text{ tal que } k^2 \geq N\}$$

es seguro que ningún natural k elevado al cuadrado da N . Por consiguiente, N no es un cuadrado perfecto.

El desarrollo de este procedimiento nos lleva a refinar la acción

Buscar el número m descrito

de manera tal que, a su término, se verifique la (post)condición

$$m = \text{mín } k \in \mathbb{IN} \text{ tal que } k^2 \geq N$$

La búsqueda de m descrita puede llevarse a cabo de diversas maneras. En los subapartados siguientes desarrollamos con detalle dos posibilidades.

Búsqueda secuencial

Una primera forma de buscar el mínimo $k \in \mathbb{IN}$ tal que $k^2 \geq N$ consiste en tantear esa condición sucesivamente para los valores de $i = 0, 1, \dots$ hasta que uno la verifique. Como el tanteo se realiza ascendentemente, el primer i encontrado que verifique el test será, evidentemente, el mínimo m buscado.

El método de búsqueda descrito se puede expresar directamente en PASCAL con un bucle:

```

  i := 0;
  while Sqr(i) < N do
    i := i+1;
  {i = m}

```

(7.7)

Es importante resaltar una propiedad (invariante) de ese bucle: ningún natural $k < i$ verifica la propiedad $k^2 \geq N$, de m ; esto es, $m \geq i$. Por otra parte,

como los bucles **while** terminan cuando su condición de entrada (en nuestro caso, $\text{Sqr}(i) < N$) es falsa, a la salida del mismo se verifica que $i \geq m$. Esto, junto con el invariante, nos garantiza que, a la salida del mismo el valor de i es el mínimo en esas condiciones, esto es, m . Por consiguiente, bastará con hacer

$$m := i$$

para tener en m el valor descrito.

Búsqueda dicotómica

Partiendo del intervalo $\{0, \dots, N\}$ en que está m , se trata de reducir ese intervalo (conservando m dentro) cuantas veces sea preciso hasta lograr que sea unitario, con lo que su único valor final será m .

En otras palabras, siendo

```
var
  izda, dcha: integer;
```

los extremos del intervalo, y estableciendo

```
izda := 0;
dcha := N
```

se logra que $izda \leq m \leq dcha$, y ahora se trata de hacer

```
while not izda = dcha do
  Reducir {izda, ..., dcha}, manteniendo izda ≤ m ≤ dcha
```

A la salida de este bucle será $izda \leq m \leq dcha$ y $izda = dcha$, con lo que, efectivamente, $izda = m = dcha$. Y entonces bastará con añadir $m := izda$. Debe señalarse además que, como el tamaño del intervalo disminuye en cada vuelta, la terminación del bucle es segura.

El siguiente paso es refinar la reducción del intervalo de la forma descrita, esto es: manteniendo m en su interior y de forma que la reducción sea efectiva. Siendo c el valor central entero del intervalo,

$$c = \left\lfloor \frac{izda + dcha}{2} \right\rfloor$$

el método⁶ que escogemos para reducirlo consiste en comparar c^2 con N , para ver si m está a la izquierda de c , a su derecha o es el propio c :

⁶Y de aquí procede el nombre de búsqueda *dicotómica*, compárese con el método de bipartición para aproximar una raíz real de una ecuación.

```

var
  c: integer; {el valor central}
  ...
c := (a + b) div 2
Según  $c^2$  sea:
= N, hacer izda := c; dcha := c
< N, hacer izda := c + 1
> N, hacer dcha := c

```

Veamos ahora si esta reducción es válida, es decir, si, sea cual sea el valor de c^2 en comparación con N , las asignaciones consecuentes mantienen m entre $izda$ y $dcha$ y además la reducción es efectiva.

El primer caso cumple trivialmente esas exigencias, al ser $c^2 = N$, y además produce la última reducción del intervalo.

En los otros dos casos, debe tenerse en cuenta que $izda \leq m \leq dcha$ (invariante) y que el valor de c cumple que $izda \leq c < dcha$, por ser $a \neq b$. En el segundo caso además, al ser $c^2 < N$, es seguro que $m \geq c + 1$. Por consiguiente, tras la asignación $izda := c + 1$, se puede asegurar que $izda \leq m$ y que $izda \leq dcha$:

$$\begin{aligned}
 & \{izda \leq m \leq dcha \text{ y } izda \leq c < dcha \text{ y } c^2 < N\} \\
 & \quad izda := c + 1 \\
 & \{izda \leq m \leq dcha\}
 \end{aligned}$$

El tercer caso se deja como ejercicio.

Trivialmente se observa que, al ser $izda \leq c < dcha$, cualquiera de los intervalos producidos por las tres ramas (c, c), ($c+1, dcha$) y ($izda, c$) es estrictamente más pequeño que ($izda, dcha$), lo que asegura la terminación del bucle **while**.

7.4.2 Recapitulación

A tenor de lo expuesto en este capítulo podría parecer que el uso refinamiento progresivo y el uso de aserciones para comprobar la corrección de un programa es algo excesivo porque, en general, se suelen escribir más líneas de pseudocódigo y aserciones que de codificación propiamente dicha. La ventaja es que se propicia el diseño correcto de algoritmos complejos.

Por otro lado, es cierto que no todo programa se puede verificar, con lo cual ¿de qué nos sirve todo esto? Nuestra propuesta consiste en adoptar un punto intermedio entre la formalización absoluta que requiere la verificación rigurosa, y la ausencia total de análisis sobre la corrección, esto es:

1. Durante el proceso de aprendizaje, examinar con detalle cada constructor que se aprenda, para habituarse a desarrollar algoritmos con garantías suficientes de que son correctos.

2. En la práctica, se deben examinar aquellas fases del desarrollo que, por ser novedosas o complicadas, no presenten todas las garantías de funcionar correctamente.

En resumen, se trata de dedicar, en cada fase del desarrollo, el grado de atención que se requiera y con el rigor necesario para convencernos de que el algoritmo desarrollado es correcto.

7.5 Conclusión

La programación estructurada es una disciplina de programación desarrollada sobre los siguientes principios básicos:

1. La percepción de una estructura lógica en el problema. Esta estructura debe reflejarse en las acciones y datos involucrados en el algoritmo diseñado para la solución.
2. La realización de esa estructura mediante un proceso de refinamiento progresivo, abordando en cada momento únicamente un aspecto del problema.
3. El uso de una notación que asista al refinamiento progresivo de la estructura requerida.

Los beneficios de la programación estructurada se orientan hacia la limitación de la complejidad en el diseño, en la validación y en el mantenimiento de los programas.

Asimismo, la metodología de diseño descendente facilita la tarea de programación en equipo, puesto que en las distintas etapas de refinamiento se pueden emplear distintas personas para que se dediquen al refinamiento de distintos bloques del programa. Esta metodología de trabajo, llevada hasta las últimas consecuencias, nos dirige hacia el concepto de programación con subprogramas, que se estudiará en los capítulos siguientes.

7.6 Ejercicios

1. Construya el diagrama final resultante para el problema de las tablas de multiplicar estudiado en este capítulo.
2. Se llaman números triangulares a los obtenidos como suma de los n primeros números naturales, esto es $1, 1 + 2, 1 + 2 + 3, \dots$. Use pseudocódigo y la técnica de diseño descendente para desarrollar programas que:
 - (a) calculen el n -ésimo número triangular,

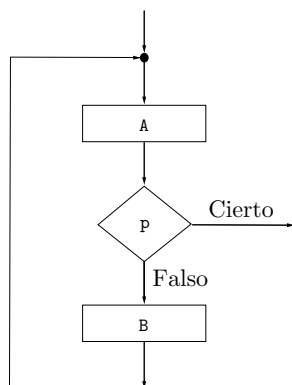


Figura 7.8.

- (b) dado un número natural decir si es triangular, y si no lo fuera decir entre qué dos números triangulares se encuentra.
3. (a) Los bucles **repeat** se pueden simular haciendo uso de bucles **while**, ¿cómo?
 - (b) Por el contrario, los bucles **while** no pueden expresarse haciendo uso únicamente de bucles **repeat**. ¿Puede encontrar una razón simple que justifique esta afirmación?
 - (c) En cambio, **while** puede simularse combinando **repeat** con la instrucción condicional **if**. ¿Cómo?
 - (d) Simúlese la instrucción **if C then I1 else I2** mediante instrucciones **for**.
4. Una estructura de repetición que aparece en algunos textos tiene la condición en el interior del bucle. Su diagrama aparece en la figura 7.8.
 - (a) ¿Es propio? Justificar la respuesta.
 - (b) ¿Es BJ? Justificar la respuesta.
 - (c) ¿Se puede convertir a BJ? Hágase si es posible considerando que la salida del bucle tiene la etiqueta Cierto (resp. Falso).
 5. Considérese de nuevo el bucle (7.7) de la página 148 para la búsqueda secuencial:
 - (a) ¿Por qué se ha escogido la instrucción **while** para codificar el bucle?
 - (b) En las condiciones de entrada de éste, ¿se puede afirmar que termina para cualquier valor de N?
 6. Halle una aproximación de π
 - (a) mediante la fórmula de Wallis

$$\frac{\pi}{2} = \frac{2}{1} \times \frac{2}{3} \times \frac{4}{3} \times \frac{4}{5} \times \frac{6}{5} \times \frac{6}{7} \times \frac{8}{7} \times \frac{8}{9} \times \dots$$

multiplicando los 50 primeros factores.

(b) mediante la fórmula de Leibnitz

$$\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$$

hasta incluir un término menor que $\varepsilon = 10^{-4}$ en valor absoluto.

(c) mediante la fórmula de Vieta

$$\frac{\pi}{2} = \frac{\sqrt{2}}{2} * \frac{\sqrt{2 + \sqrt{2}}}{2} * \frac{\sqrt{2 + \sqrt{2 + \sqrt{2}}}}{2} * \dots$$

sabiendo que, para obtener un error menor que ε , debemos iterar hasta incluir un factor mayor que $1 - \frac{\varepsilon}{2}$.

Nota: obsérvese que cada término se puede hallar rápidamente a partir del anterior siguiendo el método de la diferenciación finita.

7. Halle una aproximación de $\sin(\frac{\pi}{6})$, mediante su desarrollo de Taylor

$$x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$$

sumando los diez primeros términos.

(Aplíquese la diferenciación finita para evitar la repetición innecesaria de cálculos.)

8. En el ejemplo de búsqueda dicotómica, razonar por qué en la tercera posibilidad ($c^2 > N$), después de la correspondiente instrucción (`dcha:= c`) se tiene que $izda \leq m \leq dcha$.
9. Siguiendo los pasos de los ejemplos explicados, desarrolle el siguiente ejercicio: dado el entero positivo N , se trata de hallar su raíz cuadrada entera, es decir, el entero $\lfloor \sqrt{N} \rfloor$. Naturalmente, no se trata de escribir la expresión

`Trunc(SqRt(N))`

sino de seguir los pasos de los ejemplos explicados. Por lo tanto, se puede desarrollar con dos tipos de búsqueda: secuencial y dicotómica.

7.7 Referencias bibliográficas

En ocasiones, la programación estructurada ha sido considerada como *programación sin goto*, en alusión directa al artículo de Dijkstra [Dij68] en el que hizo la primera advertencia al mundo de la computación sobre el peligro potencial que para la programación suponía el uso irreflexivo de órdenes de bifurcación incondicional del tipo `goto`.

Cerca del origen de la programación estructurada puede ser situada la referencia [DDH72], que contiene tres artículos sobre la programación estructurada, la estructuración de datos y las estructuras jerárquicas de programas. El primero, que es el más interesante para los contenidos de este capítulo, explica los diagramas de secuencia, selección e iteración y desarrolla varios ejemplos con refinamientos progresivos.

Un número especial de la revista [dat73] se dedicó sólo a la programación estructurada. En él podemos encontrar artículos que resaltan distintos aspectos de la programación estructurada, desde el diseño descendente hasta la programación sin `goto`. Como muestra de la importancia suscitada por aquellas fechas citamos textualmente unas palabras de McCracken [McC73]: “¡Este número de *Datamation* es importante! Lean atentamente cada artículo, porque se describe un movimiento que va a cambiar su futuro.”

Un texto reciente de programación estructurada en Pascal es [CGL⁺94], que contiene un acercamiento gradual a las técnicas de programación estructurada, con numerosos ejercicios y ejemplos con el compilador Turbo Pascal de Borland. A un nivel más elevado que el presente texto y con un gran énfasis en la corrección y verificación de programas podemos citar [AA78].

Por lo que respecta a la verificación de programas cabe destacar el capítulo 4 de [Ben86] en el que, mediante un sencillo ejemplo, se muestra que la escritura del código final resulta una tarea fácil después de una correcta definición del problema, diseño del algoritmo y elección de las estructuras de datos.

Se han escrito muchos artículos en los que se reconoce la importancia de la introducción de los invariantes de bucle desde los primeros contactos con las instrucciones iterativas, como ejemplo podemos destacar [Arn94, Tam92, Col88].

Para profundizar en los métodos numéricos presentados, bipartición y Newton-Raphson, así como disponer de un gran surtido de métodos iterativos para resolver problemas matemáticos, se recomienda leer [DM84].

El problema de los números pedrisco se comenta en [Hay84]. En este artículo se presenta el problema, su historia y experimentos con computador llevados a cabo. En particular, se señala que se han ensayado todos los valores de N hasta 2^{40} (aproximadamente un billón y cien mil millones) y en todos los casos el resultado ha sido el mismo: finalmente se cae en el bucle 1, 4, 2, 1. A pesar de todo, aún no se ha encontrado una demostración general de que ocurre lo mismo para cada N .

La versión dicotómica del problema propuesto acerca de la raíz cuadrada entera se puede leer en [Hig93].

Tema III

Subprogramas

Capítulo 8

Procedimientos y funciones

8.1	Introducción	158
8.2	Subprogramas con parámetros	162
8.3	Estructura sintáctica de un subprograma	169
8.4	Funcionamiento de una llamada	170
8.5	Ámbito y visibilidad de los identificadores	174
8.6	Otras recomendaciones sobre el uso de parámetros	183
8.7	Desarrollo correcto de subprogramas	184
8.8	Ejercicios	186

Para la construcción de programas de tamaño medio o grande es necesario disponer de herramientas que permitan organizar el código. Por una parte, las técnicas de la programación estructurada hacen posible relacionar las acciones por realizar mediante constructores de secuencia, selección e iteración, tal y como se vio en los capítulos anteriores. Por otra parte, la programación con subprogramas permite al programador separar partes de código con un cometido bien determinado, los subprogramas, que pueden ser invocados desde diferentes puntos del programa principal. Así se extiende el juego de instrucciones básicas con otras nuevas a la medida del problema que se está resolviendo. Una elección adecuada de subprogramas, entre otras ventajas, hace que los programas sean más legibles y que su código sea más fácilmente reutilizable. De esta forma se facilita en gran medida el paso de los algoritmos a los programas, especialmente cuando se sigue un método de diseño descendente.

8.1 Introducción

La presentación de los principales contenidos de este capítulo se hará en base al ejemplo que se describe a continuación. Supongamos que queremos escribir un programa que pida al usuario el valor de un cierto ángulo en grados sexagesimales, calcule su tangente y escriba su valor con dos decimales. En una primera aproximación podríamos escribir:

Sean $a, t \in \mathbb{R}$
Leer el valor del ángulo a (en grados)
Calcular la tangente, t , *de* a
Escribir el valor de t *con dos decimales*

Este nivel de refinamiento¹ se puede escribir en Pascal, dejando sin definir la expresión *tangente de a* (dado en grados) y la acción *Escribir un valor dado, con dos decimales*:

```
Program CalculoTangente (input, output);
  {Se halla la tangente de un ángulo, dado en grados}
  var
    a, {ángulo}
    t: real; {su tangente}
  begin
    Leer el valor del ángulo a (en grados)
    t:= tangente de a;
    Escribir el valor de t, con 2 decimales
  end. {CalculoTangente}
```

Desde el punto de vista del seudoprograma principal, tanto la lectura del ángulo a , como la expresión *tangente de a* y la acción *Escribir el valor de t, con dos decimales* son abstractas: se ignora su particular modo de operar. Por otra parte, al no estar predefinidas, es necesario concretarlas, usando recursos del lenguaje (predefinidos o añadidos por el programador), para que puedan ser ejecutadas.

Como se puede ver, resulta útil empezar utilizando acciones o expresiones abstractas, aun sin estar definidas todavía. En principio, basta con saber qué tiene que hacer (o calcular) cada acción (o expresión) abstracta e introducir un nombre adecuado para ellas, por ejemplo

```
LeerGrados(a);
t:= TanGrados(a);
EscrDosDec(t)
```

¹En la presentación de estos primeros ejemplos seguiremos un proceso de refinamiento, pero sin detallar las especificaciones, para no entorpecer la exposición de los contenidos del capítulo.

De esta forma, se puede proceder al diseño general del algoritmo en ese nivel posponiendo el desarrollo de cada acción abstracta. Cuando, más tarde, se concreten sus detalles, el lenguaje de programación se habrá ampliado (en el ámbito de nuestro programa, véase el apartado 8.5) con esta acción o expresión, legitimando entonces su uso.

En Pascal, una acción se introduce mediante un *procedimiento*. Por ejemplo, la lectura del ángulo se puede hacer así:

```

procedure LeerGrados(var angulo: real);
begin
  Write('¿ángulo en grados?: ');
  ReadLn(angulo);
end; {LeerGrados}

```

En Pascal, una *expresión abstracta* se introduce mediante una *función*. En nuestro ejemplo, vamos a crear una función $\mathbb{R} \rightarrow \mathbb{R}$, a la que llamaremos **TanGrados**, que recibe un argumento real, lo pasa a radianes y devuelve el valor de la tangente, calculado a partir de las funciones predefinidas **Sin** y **Cos**:²

```

function TanGrados(angSexa: real): real;
  {Dev. la tangente de angSexa, en grados}
  const
    Pi = 3.141592;
  var
    angRad: real;
begin
  {Conversión de grados en radianes:}
  angRad:= angSexa * Pi/180;
  {Cálculo de la tangente:}
  TanGrados:= Sin(angRad)/Cos(angRad)
end; {TanGrados}

```

Ahora, la asignación de **t** en el programa principal definitivo:

```

t:= TanGrados(a)

```

es válida.

Finalmente, en nuestro ejemplo utilizaremos otro procedimiento al que llamaremos **EscrDosDec**, que recibe un valor de tipo real y lo muestra con dos decimales.

²En adelante emplearemos **Dev.** como abreviatura de **Devuelve** en las especificaciones de las funciones.

```

procedure EscrDosDec(valor: real);
  {Efecto:  escribe valor, con dos decimales}
begin
  WriteLn('El valor es: ', valor:14:2)
end; {EscrDosDec}

```

Una vez definido el procedimiento `EscrDosDec`, es posible sustituir la acción abstracta *Escribir el valor de t con dos decimales* por la siguiente llamada:

```
EscrDosDec(t)
```

y, de esta forma, si ensamblamos todos estos trozos obtenemos el programa completo:

```

Program CalculoTangente (input, output);
  var
    a, {ángulo}
    t: real; {su tangente}

  procedure LeerGrados(var angulo: real);
  begin
    Write('¿ángulo en grados?: ');
    ReadLn(angulo);
  end; {LeerGrados}

  function TanGrados(angSexa: real): real;
  {Dev. la tangente de angSexa, en grados}
  const
    Pi = 3.141592;
  var
    angRad: real;
  begin
    {Conversión de grados en radianes:}
    angRad:= angSexa * Pi/180;
    {Cálculo de la tangente:}
    TanGrados:= Sin(angRad)/Cos(angRad)
  end; {TanGrados}

  procedure EscrDosDec(valor: real);
  {Efecto:  escribe valor, con dos decimales}
  begin
    WriteLn('El valor es: ', valor:14:2)
  end; {EscrDosDec}

```

```

begin
  LeerGrados(a);
  t:= TanGrados(a);
  EscrDosDec(t)
end. {CalculoTangente}

```

Concretando algunas de las ideas introducidas en el ejemplo, se observa lo siguiente:

- Pascal proporciona mecanismos para ampliar los procedimientos y funciones predefinidos (tales como `WriteLn` y `Sin`), definiendo otros nuevos (como `EscrDosDec` y `TanGrados`) a la medida de las necesidades del programador.
- Cada procedimiento o función es, en sí mismo, un pequeño programa,³ tanto por su estructura sintáctica (con encabezamiento, declaraciones y cuerpo) como por su cometido (resolver un problema concreto con los datos recibidos y ofrecer los resultados obtenidos).

En nuestro ejemplo hemos tenido que incluir como declaraciones propias (*locales*, véase la sección 8.5) la constante `Pi` y la variable `angRad`.

- Existen dos puntos de consideración de estos subprogramas: su *definición*, donde se introducen, y su *llamada*, donde se utilizan.
- En su definición, ambas clases de subprogramas operan sobre datos genéricos, sus *parámetros*, que tomarán valores en cada llamada, esto es, cuando se hace uso de los subprogramas para unos datos particulares. En nuestro ejemplo el valor de la variable `a` pasa a la función `TanGrados` a través del parámetro `angSexa`, y el valor de `t` pasa al procedimiento `EscrDosDec` por medio del parámetro `valor`.
- Un procedimiento es un subprograma que desempeña el papel de una *instrucción*, mientras que una función es un subprograma que desempeña el de una *expresión*, puesto que calcula un *valor*, que se reemplaza por la llamada a la función.

Este distinto cometido se refleja en su llamada: los procedimientos se usan como las demás instrucciones,

```

WriteLn(...);
EscrDosDec(t);
a:= a + 1

```

³Por eso se conocen como *subprogramas*, o también *subrutinas*.

mientras que las funciones representan un valor, por lo que tienen sentido como expresiones:

```
t:= TanGrados(a);
WriteLn ('La medida buscada es: ', radio * TanGrados(a) - 1);
x:= 2 * TanGrados(a)/(y - 1)
```

Por el contrario, no está permitido ni tiene sentido llamar a una función como un procedimiento:

```
WriteLn(...);
TanGrados(a);
a:= a + 1
```

ni tampoco llamar a un procedimiento como una función:

```
x:= 4 * EscrDosDec(t)
```

- Una vez definido un subprograma, queda incorporado al lenguaje para ese programa, siendo posible usarlo en el mismo tantas veces como sea necesario.

8.2 Subprogramas con parámetros

Los parámetros permiten que el programa y los procedimientos y funciones puedan comunicarse entre sí intercambiando información. De esta forma las instrucciones y expresiones componentes de los subprogramas se aplican sobre los datos enviados en cada llamada ofreciendo una flexibilidad superior a los subprogramas sin parámetros. Al mismo tiempo, si la ejecución de los subprogramas produce resultados necesarios en el punto de la llamada, los parámetros pueden actuar como el medio de transmisión de esos resultados.

8.2.1 Descripción de un subprograma con parámetros

Veamos en primer lugar un ejemplo sencillo de un procedimiento sin parámetros:

```
procedure TrazarLinea;
{Efecto: traza una línea de 10 guiones}
var
  i: integer;
```

```
begin
  for i:= 1 to 10 do
    Write ('-');
  WriteLn
end; {TrazarLinea}
```

La llamada al procedimiento sería:

```
TrazarLinea
```

El procedimiento anterior realiza siempre una acción fija y totalmente determinada; traza una línea formada por diez guiones. La única relación existente entre el programa y el procedimiento es la llamada.

Si se quisiera trazar una línea de 15 guiones habría que escribir un nuevo procedimiento; en cambio, si añadimos un parámetro para determinar la longitud en caracteres de la línea por trazar:

```
procedure TrazarLineaLong(longitud: integer);
  {Efecto: traza una línea de guiones, con la longitud indicada}
  var
    i: integer;
begin
  for i:=1 to longitud do
    Write('-');
  WriteLn
end; {TrazarLineaLong}
```

Al efectuar la llamada hay que indicar la longitud de la línea por trazar, por ejemplo:

```
TrazarLineaLong(15)
```

que trazaría una línea formada por quince guiones. Otra posible llamada sería:

```
largo:= 10;
TrazarLineaLong(largo + 5)
```

que trazaría una línea idéntica a la anterior.

En resumen, mediante la inclusión de un parámetro, se ha pasado de un procedimiento que traza una línea de longitud fija y determinada a otro que puede trazar una línea de cualquier longitud aumentando la flexibilidad y el grado de abstracción del procedimiento.

En Pascal, es obligatorio indicar el tipo de los parámetros que pasan como argumentos a los subprogramas. En el caso de una función, se debe indicar además el tipo del resultado que se devuelve al programa principal o subprograma que efectuó la llamada.⁴

Hemos visto que la función `TanGrados`, que es una aplicación de \mathbb{R} en \mathbb{R} , recibe un argumento real y devuelve un resultado también real.

Veamos otro ejemplo de una función para calcular el factorial de un número entero positivo:

```
function Fac(n: integer): integer;
  {Dev. n!}
  var
    i, prodAcum: integer;
begin
  prodAcum:= 1;
  for i:= 2 to n do
    prodAcum:= prodAcum * i;
  Fac:= prodAcum
end; {Fac}
```

Como podemos ver, la función `Fac` tiene un argumento entero y devuelve un resultado también entero. Los sucesivos productos $2 * 3 * \dots * n$ se van almacenando en la variable `prodAcum` tantas veces como indica el bucle `for`. Una vez terminado, el valor del factorial presente en `prodAcum` se asigna al nombre de la función reemplazando su llamada. Por ejemplo, la instrucción

```
WriteLn(Fac(4))
```

escribe el valor 24.

No es obligado que el(los) argumento(s) de una función sea(n) del mismo tipo que su resultado. El siguiente ejemplo de función que determina si un número es o no es primo recibe un argumento entero positivo y devuelve un valor booleano: `True` si el número es primo y `False` en caso contrario.

```
function EsPrimo(n: integer): boolean;
  {Dev. True si n es primo y False en caso contrario}
  var
    divisor: integer;
    conDivisores: boolean;
```

⁴En Pascal, el resultado de una función sólo puede ser un objeto simple. Sin embargo, esta limitación se supera fácilmente (véase el apartado 8.6.3).

```

begin
  divisor:= 2;
  conDivisores:= False;
  repeat
    if n mod divisor = 0 then
      conDivisores:= True;
      divisor:= divisor + 1
    until conDivisores or (divisor > n - 1);
  EsPrimo:= not conDivisores
end; {EsPrimo}

```

En su funcionamiento se supone inicialmente que el número es primo, ya que aún no se ha encontrado divisor alguno del mismo; a continuación, se avanza desde 2, tanteando posibles divisores, hasta dar con uno (en cuyo caso la condición `conDivisores` se hace cierta), o llegar al propio n , sin haber hallado divisor alguno del mismo (en cuyo caso el número es primo).⁵

8.2.2 Parámetros formales y reales

Recordando los dos aspectos de definición y llamada que encontramos en los subprogramas, tenemos que distinguir dos tipos de parámetros.

Cuando se define un subprograma es necesario dar nombres a los parámetros para poder mencionarlos. A los parámetros utilizados en la definición de procedimientos y funciones se les denomina *parámetros formales*. A veces se llaman también *ficticios*, pues se utilizan solamente a efectos de la definición pero no con valores reales. En nuestro ejemplo de referencia, `angSexa` y `valor` son parámetros formales de la función `TanGrados` y del procedimiento `EscrDosDec` respectivamente.

En cambio, a los argumentos concretos utilizados en la llamada de un subprograma se les llama *parámetros reales*.⁶ Por ejemplo, `a` y `t` son los parámetros reales de la función `TanGrados` y del procedimiento `EscrDosDec`, respectivamente, en las llamadas que se hacen en el ejemplo anterior.

8.2.3 Mecanismos de paso de parámetros

Antes de entrar en materia conviene que nos fijemos en los procedimientos `Read` y `Write` que vamos a aplicar a una cierta variable entera a la que llamaremos `a`.

⁵En realidad, no es necesario comprobar todos los divisores desde 2 hasta $n - 1$, sino que bastará con comprobar hasta la raíz cuadrada de n , como puede confirmar fácilmente el lector.

⁶En inglés, *actual parameters*, lo que ha dado lugar en ocasiones a la traducción errónea “parámetros actuales” en castellano.

Supongamos, en primer lugar, que esta variable tiene un valor que le ha sido asignado previamente en el programa, por ejemplo 10, y a continuación esta variable es pasada como parámetro al procedimiento `Write`. Este procedimiento recibe el valor de `a` y lo escribe en la pantalla. La acción de `Write` no modifica el valor de `a`, que sigue siendo 10.

El proceso seguido es el siguiente:

```
a:= 10;
  {a = 10}
Write(a) {aparece el valor de a en la pantalla}
  {a = 10}
```

En cambio, supongamos ahora que utilizamos el procedimiento `Read` con la misma variable `a`, y que el usuario escribe por el teclado un valor distinto al que tenía `a`, por ejemplo 20. Como consecuencia de la llamada, el valor de la variable `a` es modificado, de 10 a 20.

Esquemáticamente tenemos que:

```
a:= 10;
  {a = 10}
Read(a) {el usuario da el valor 20 por el teclado}
  {a = 20}
```

Estas diferencias se deben a que en Pascal existen dos formas de pasar parámetros que se diferencian en la forma en que se sustituyen los parámetros formales por los reales al efectuarse la llamada. Estos mecanismos se conocen como:

- *Parámetros por valor:*

En este caso, se calcula el valor de los parámetros reales y después se copia su valor en los formales, por lo tanto los parámetros reales deben ser expresiones cuyo valor pueda ser calculado. Este mecanismo se llama *paso de parámetros por valor* y tiene como consecuencia que, si se modifican los parámetros formales en el cuerpo del subprograma, los parámetros reales no se ven afectados.

Dicho de otra forma, no hay transferencia de información desde el subprograma al programa en el punto de su llamada. Por lo tanto, los parámetros por valor actúan sólo como datos de entrada al subprograma.

- *Parámetros por referencia (o por dirección o por variable):*

En este otro caso, se hacen coincidir en el mismo espacio de memoria los parámetros reales y los formales, luego los parámetros reales han de ser

variables. Este segundo mecanismo se denomina *paso de parámetros por referencia* (también *por dirección* o *por variable*), y tiene como consecuencia que toda modificación de los parámetros formales se efectúa directamente sobre los parámetros reales, y esos cambios permanecen al finalizar la llamada. Es decir, que se puede producir una transferencia de información desde el subprograma al programa, o dicho de otro modo, que los parámetros por referencia no sólo actúan como datos de entrada, sino que también pueden representar resultados de salida del procedimiento.

Para distinguir los parámetros pasados por valor de los pasados por variable, éstos últimos van precedidos de la palabra reservada **var** en la definición del subprograma.

Veamos las diferencias entre parámetros por valor y referencia mediante un ejemplo consistente en un procedimiento que incrementa el valor de una variable en una unidad. En el caso de parámetros por valor, el incremento tiene efectos únicamente dentro del procedimiento, mientras que en el caso de parámetros por referencia los efectos se extienden también al programa principal.

En el paso de parámetros por valor,

```

procedure EscribirSiguiete (v: integer);
  {Efecto:  escribe en la pantalla v + 1}
begin
  v:= v + 1;
  WriteLn(v)
end; {EscribirSiguiete}

```

la siguiente secuencia de instrucciones produce la salida que se muestra a la derecha:

```

w:= 5;
WriteLn(w);
EscribirSiguiete(w);
WriteLn(w)

```

5
6
5

En este ejemplo, la variable **w** que hace de parámetro real tiene inicialmente el valor 5, como puede verse en la salida. Este valor se copia en el parámetro formal **v** y dentro del procedimiento **v** se incrementa en una unidad. Sin embargo, por tratarse de parámetros por valor, este cambio en **v** no tiene efecto sobre el parámetro real **w**, lo que comprobamos al volver al programa principal y escribir su valor que sigue siendo 5.

En el paso de parámetros por referencia,

```

procedure IncrementarYescribir (var v: integer);
begin
  v := v + 1;
  WriteLn(v)
end; {IncrementarYescribir}

```

la siguiente llamada produce esta salida:

```

w := 5
WriteLn(w);
IncrementarYescribir(w);
WriteLn(w)

```

5	5
6	6
6	6

En este segundo caso, al tratarse de parámetros por referencia, el espacio en memoria de `w` coincide durante la llamada con el de `v`; por ello, el incremento de `v` se efectúa también sobre `w`. Al terminar el procedimiento, `w` tiene el valor 6.

8.2.4 Consistencia entre definición y llamada

Es imprescindible que la definición y la llamada a un subprograma encajen: para ello, la llamada debe efectuarse utilizando el mismo identificador definido para el subprograma, seguido entre paréntesis de los parámetros, separados por comas. Estos argumentos reales deberán coincidir con los parámetros formales en número y ser respectivamente del mismo tipo. Como dijimos antes, los argumentos reales correspondientes a parámetros formales por valor podrán ser expresiones cualesquiera (con el requisito, ya mencionado, de tener el mismo tipo):

```
WriteLn(n + 2)
```

En cambio, los argumentos correspondientes a parámetros formales por referencia deberán ser necesariamente variables, para que las modificaciones efectuadas en el subprograma repercutan en el espacio de memoria asociado a las variables argumentos. Como contraejemplo, obsérvese la siguiente llamada imposible:

```
ReadLn(n + 2)
```

En el caso de las funciones, el tipo del resultado devuelto debe, además, encajar en la llamada efectuada.

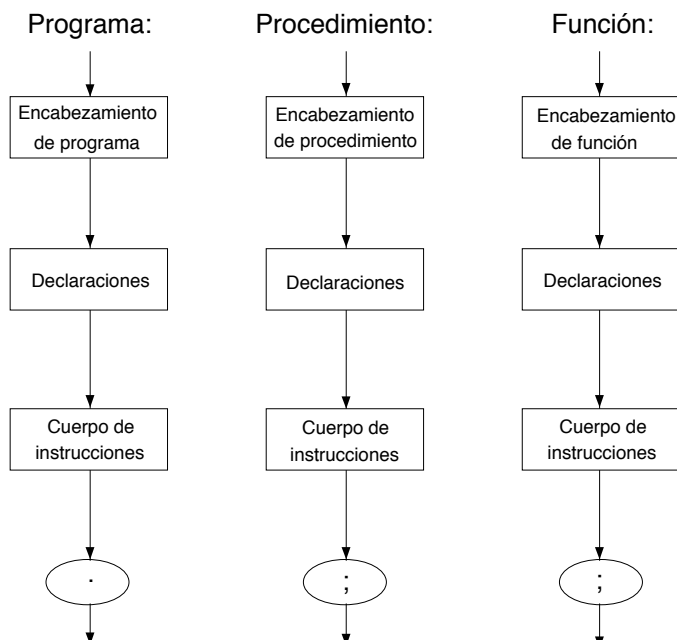


Figura 8.1. Diagramas sintácticos generales de programa, procedimiento y función.

8.3 Estructura sintáctica de un subprograma

Como es norma en Pascal, es necesario definir cualquier componente del programa, en particular los diferentes subprogramas, antes de poder utilizarlos. La estructura de un subprograma es semejante a la del programa principal: tanto los procedimientos como las funciones constan de un encabezamiento, una parte de declaraciones y definiciones y una parte de instrucciones, como se muestra en la figura 8.1.

En el encabezamiento del programa y los subprogramas, se da su identificador y la lista de sus parámetros. Recordemos que para un programa los parámetros representan los archivos, externos al mismo, con los que intercambia información con el exterior: `input` como archivo de datos y `output` para los resultados. En el caso de los subprogramas, debe especificarse el tipo de los parámetros y su mecanismo de paso, y para las funciones se debe incluir además el tipo del resultado. Las figuras 8.2 y 8.3 muestran los diagramas sintácticos correspondientes a los encabezamientos de procedimiento y función, respectivamente.

La parte de declaraciones y definiciones de un subprograma (véase la figura 8.4) es idéntica a la de un programa. En ella se pueden declarar y definir constantes, variables e incluso procedimientos y funciones, propios de cada sub-

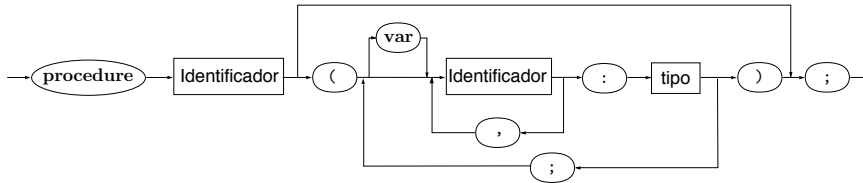


Figura 8.2. Diagrama sintáctico del encabezamiento de procedimiento.

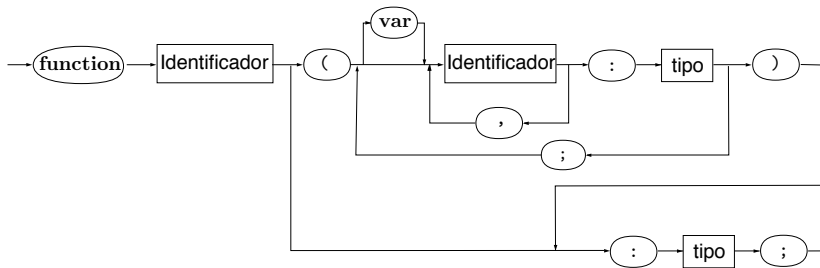


Figura 8.3. Diagrama sintáctico del encabezamiento de función.

programa y a los que sólo desde él se tiene acceso (véase el apartado 8.5).

Estos objetos y los propios parámetros son elementos locales del subprograma, se crean al producirse la llamada al subprograma y permanecen solamente mientras se ejecuta ésta. De ello tratamos en el siguiente apartado.

En la parte de instrucciones se concreta la acción o expresión que desempeña el programa o el subprograma mediante una instrucción compuesta. Estas instrucciones se ejecutan cuando se llama al subprograma.

8.4 Funcionamiento de una llamada

Veamos cómo se realiza la llamada a un subprograma y cómo se produce el paso de parámetros utilizando un ejemplo con un procedimiento para la lectura de números enteros y con la conocida función Fac dentro de un programa completo:

```

Program DemoParametros (input,output);
  var
    numero: integer;

```

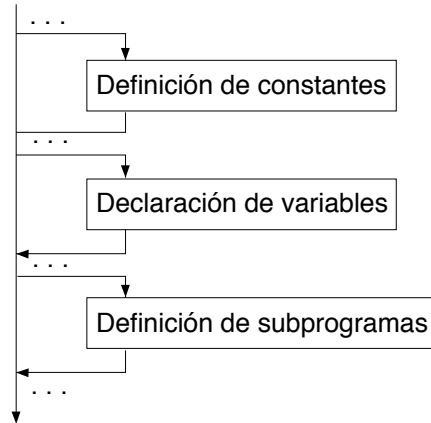


Figura 8.4. Diagrama sintáctico de las declaraciones y definiciones.

```

procedure LeerNumPos(var n: integer);
  {Efecto: solicita un entero hasta obtener uno positivo}
begin
  {2A}
  repeat
    Write('Escriba un entero positivo: ');
    ReadLn(n)
  until n >= 0
  {2B}
end; {LeerNumPos}

```

```

function Fac(num: integer): integer;
  {Dev. num!}
  var
    i, prodAcum: integer;
begin
  {4A}
  prodAcum:= 1;
  for i:= 2 to num do
    prodAcum:= prodAcum * i;
  Fac:= prodAcum
  {4B}
end; {Fac}

```

```

begin
  {1}
  LeerNumPos(numero); {num >= 0}

```

```

    {3}
    WriteLn('El factorial de ', numero, ' es ', Fac(numero))
    {5}
end. {DemoParametros}

```

Al comienzo del programa sólo se dispone de la variable `numero` que está indefinida en el punto {1} del programa (y en su correspondiente estado de memoria).

El programa llama al procedimiento `LeerNumPos` y le pasa por referencia el parámetro real `numero`. Al producirse la llamada, la ejecución del programa principal queda suspendida y se pasan a ejecutar las instrucciones del procedimiento `LeerNumPos`. Como `numero` se ha pasado por referencia, en la llamada, `numero` y `n`, que es el parámetro formal de `LeerNumPos`, coinciden en memoria. Dado que inicialmente `numero` está indefinido también lo estará `n` en el estado {2A}, al principio de `LeerNumPos`.

Una vez activado `LeerNumPos`, éste pide al usuario un número entero positivo, que queda asignado a `n`. Supongamos que el valor introducido ha sido, por ejemplo, 5. En el punto {2B}, este valor es el que queda asignado a `n` y a `numero` al coincidir ambos.

Al terminar el procedimiento `LeerNumPos`, se reanuda la ejecución del programa principal, con lo cual `n` desaparece de la memoria (estado {3}).

Le llega el turno a la instrucción de escritura, que hace una llamada a `Fac` pasándole por valor el contenido de `numero`. De nuevo, al producirse la llamada, la ejecución del programa principal queda suspendida, hasta que `Fac` termine y devuelva el resultado.

La función dispone del parámetro formal `num`, que recibe el contenido de `numero`, y de dos variables propias `i` y `prodAcum`, que al comenzar la función (estado {4A}) están indefinidas.

Al terminar el bucle `for`, se ha acumulado en `prodAcum` el producto $2 * 3 * 4 * 5$ sucesivamente, por lo que su valor es 120. Dicho valor, que corresponde al del factorial pedido, es asignado al nombre de la función (estado {4B}), quien lo devuelve al programa principal.

- ☉ El nombre de la función se utiliza como un almacenamiento temporal del resultado obtenido para transferirlo al programa principal. Aunque puede ser asignado como una variable, el parecido entre ambas termina aquí. El nombre de la función no es una variable y no puede ser utilizado como tal (es decir sin parámetros) a la derecha de la instrucción de asignación.

Al terminar la función, su valor se devuelve al programa principal, termina la escritura y finalmente termina el programa (estado {5}).

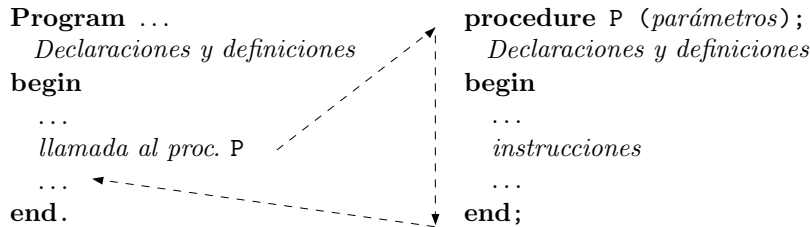
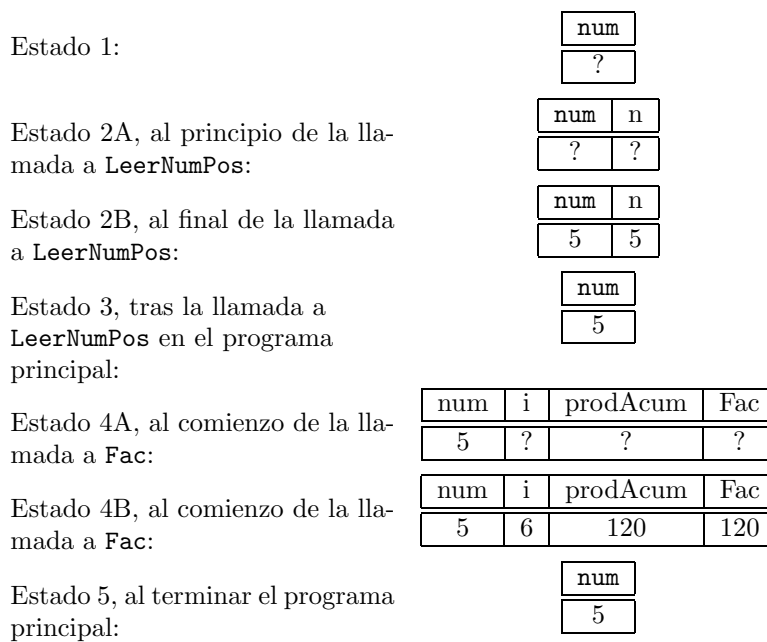


Figura 8.5. Llamada a un subprograma.

Con la entrada de datos 5 por ejemplo, la evolución de la memoria en los distintos estados sería la siguiente:



En la figura 8.5 se esquematiza el orden de ejecución de las distintas instrucciones.

- En Pascal el funcionamiento de los parámetros es el mismo tanto para procedimientos como para funciones. Sin embargo, el cometido de las funciones es calcular un valor, por lo que no tiene sentido que éstas utilicen parámetros por referencia.

8.5 Ámbito y visibilidad de los identificadores

Como sabemos, en un programa en Pascal hay que declarar los identificadores que nombran los diferentes objetos utilizados en el programa. De esta forma se declaran, entre otros, las constantes, tipos, variables y los propios identificadores de procedimientos y funciones.

A su vez, dentro de un procedimiento o función se pueden declarar sus propios identificadores, de forma similar al programa principal, e incluso pueden declararse otros procedimientos o funciones que contengan asimismo sus propios identificadores, y así sucesivamente, sin más limitaciones que las propias de la memoria disponible.

En esta sección vamos a estudiar cómo se denomina a los diferentes identificadores según el lugar en que se hayan declarado y cuál es la parte del programa en que tienen vigencia.

8.5.1 Tipos de identificadores según su ámbito

Recordemos el programa ejemplo que desarrollamos al principio del capítulo:

```

Program CalculoTangente (input, output);
  var
    a, {ángulo}
    t: real; {su tangente}

  procedure LeerGrados(var angulo: real);
  begin
    Write('¿ángulo en grados?: ');
    ReadLn(angulo);
  end; {LeerGrados}

  function TanGrados(angSexa: real): real;
  {Dev. la tangente de angSexa, en grados}
  const
    Pi = 3.141592;
  var
    angRad: real;
  begin
    {Conversión de grados en radianes:}
    angRad:= angSexa * Pi/180;
    {Cálculo de la tangente:}
    TanGrados:= Sin(angRad)/Cos(angRad)
  end; {TanGrados}

```

```
procedure EscrDosDec(valor: real);
  {Efecto: escribe valor, con dos decimales}
begin
  WriteLn('El valor es: ', valor:14:2)
end; {EscrDosDec}

begin
  LeerGrados(a);
  t:= TanGrados(a);
  EscrDosDec(t)
end. {CalculoTangente}
```

Los identificadores declarados o definidos en el programa principal, como *a* y *t*, se denominan *globales*, y su ámbito es (son *visibles* en) todo el programa, incluso dentro de los subprogramas (excepto si en éstos se declara una variable con el mismo identificador, la cual *ocultaría* a la variable global homónima, como se detalla en el apartado 8.5.2).

Los identificadores declarados o definidos dentro de subprogramas, como *Pi* y *angRad* y el identificador de la función *TanGrados*, y sus propios parámetros formales, como *angSexa* de *TanGrados* y *valor* de *EscrDosDec*, se denominan *locales*, sólo son válidos dentro de los subprogramas a los que pertenecen y, por tanto, no son reconocidos fuera de ellos (es decir, quedan *ocultos* al resto del programa).

Si dentro de un subprograma se define otro, se dice que los parámetros locales del subprograma superior se denominan *no locales* con respecto al subprograma subordinado y son visibles dentro de ambos subprogramas.

Los objetos globales se crean al ejecutarse el programa y permanecen definidos hasta que éste termina. En cambio, los objetos locales se crean en el momento de producirse la llamada al subprograma al que pertenecen y se destruyen al terminar éste. La gestión de los objetos locales suele efectuarse con una estructura de tipo pila (véase el capítulo 17 y el apartado 3.4 de [PAO94]), donde se introducen los identificadores de los objetos y el espacio necesario para almacenar sus valores cuando los haya, y de donde se extraen una vez que éste termina. El proceso de reserva y liberación de memoria para los objetos se prepara de forma automática por el compilador del lenguaje.

8.5.2 Estructura de bloques

De las definiciones anteriores se deduce que el programa principal, los procedimientos y funciones en él declarados, y los que a su vez pudieran declararse dentro de ellos, constituyen un conjunto de *bloques* anidados que determinan el ámbito de validez de los identificadores.

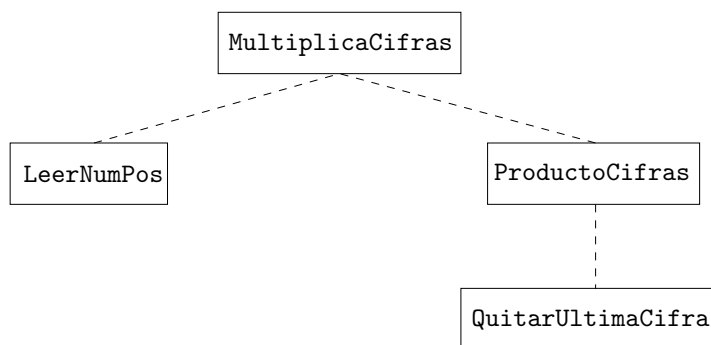


Figura 8.6.

Veamos un ejemplo algo más complejo que el anterior para explicar esta estructura de bloques. El problema que se plantea consiste en multiplicar las cifras de un número positivo. Para ello, se hace la siguiente descomposición:

Obtener un número entero positivo numPosit;
Calcular el producto p de las cifras de numPosit;
Mostrar p .

A su vez, Calcular el producto p de las cifras de numPosit podría descomponerse así:

Repetir
Tomar una cifra c de numPosit;
Multiplicar por c el producto acumulado de las restantes cifras
hasta que no queden más cifras

El programa consta de un procedimiento LeerNumPos, que lee un número positivo distinto de cero y de una función ProductoCifras que calcula el producto de sus cifras. Para ello, la función dispone de un procedimiento local QuitaUltimaCifra que, dado un número, elimina su última cifra y la devuelve junto con el número modificado. ProductoCifras llama repetidamente a QuitaUltimaCifra obteniendo las sucesivas cifras del número y calculando su producto. El número se va reduciendo hasta que no quedan más cifras, con lo que finalizan las repeticiones. En la figura 8.6 puede verse la estructura jerárquica del programa.

Para diferenciar los distintos ámbitos de visibilidad de los identificadores hemos rodeado con líneas los distintos bloques del programa en la figura 8.7.

El programa utiliza una variable global numPosit, que es pasada como parámetro a los dos subprogramas, a LeerNumPos por referencia y a ProductoCifras

```

Program MultiplicaCifras (input, output);
  var
    numPosit: integer;
  procedure LeerNumPos (var n: integer);
    {Efecto: Solicita un numero hasta obtener uno positivo}
  begin
    repeat
      Write('Escriba un entero mayor que cero: '); ReadLn(n)
    until n > 0
  end; {LeerNumPos}

  function ProductoCifras (numero: integer): integer;
    var
      acumProd, cifrUnidades: integer;
    procedure QuitarUltimaCifra (var n, ultima: integer);
      {Efecto: Elimina la ultima cifra de numero y
      la almacena en ultima}
    begin
      ultima := n mod 10;
      n := n div 10
    end;
  begin; {ProductoCifras}
    acumProd := 1;
    repeat
      QuitarUltimaCifra(numero, cifrUnidades);
      acumProd := acumProd * cifrUnidades
    until numero = 0;
    ProductoCifras := acumProd
  end; {ProductoCifras}

begin
  LeerNumPos (numPosit);
  WriteLn ('El producto de las cifras de ', numPosit,
    ' vale ', ProductoCifras (numPosit))
end.

```

Figura 8.7. El programa MultiplicaCifras.

por valor. El procedimiento `LeerNumPos` le da su valor inicial, que después se pasa a `ProductoCifras` para su procesamiento.

La función `ProductoCifras` tiene dos variables locales `acumProd` (en la que se acumula el producto de las cifras del número) y `cifrUnidades` (donde se anotan los valores de dichas cifras). Estas variables se utilizan solamente dentro de `ProductoCifras`, no teniendo ninguna utilidad fuera de la función, por lo que se han declarado como locales. Quedan ocultas al programa principal y al procedimiento `LeerNumPos`, pero son visibles desde el procedimiento `QuitaUltimaCifra` para el que son no locales.

La función tiene también un parámetro por valor llamado `numero`, a través del cual se recibe el dato inicial, y que actúa además como variable, que se va modificando al ir quitándole cifras.

El procedimiento `QuitaUltimaCifra` se ha definido dentro del procedimiento `ProductoCifras`, por lo que es local a esta función y tiene sentido sólo dentro de la misma, que es donde se necesita, quedando oculto al resto del programa. No tiene variables locales, pero utiliza los parámetros por referencia `n` y `ultima`. En el primero se recibe el número sobre el que operar y devuelve el número sin la última cifra, actuando como dato y como resultado. La cifra de las unidades se devuelve en el parámetro `ultima`, que representa sólo este resultado. Ambos son locales por ser parámetros, y se pasan por referencia para enviar los resultados a `ProductoCifras`.

Hay que observar que el identificador `n` se ha utilizado como parámetro de `LeerNumPos` y de `QuitaUltimaCifra`, sin provocar ningún conflicto.

Cada uno de estos bloques trazados en el programa representa el ámbito en el cual están definidos los identificadores del bloque. Existe un bloque exterior en el que son reconocidos todos los identificadores predefinidos de Pascal. Dentro de este bloque *universal* se encuentra el bloque del programa, correspondiente a los identificadores globales. Si dentro del bloque del programa se definen subprogramas, cada uno constituye un bloque local, si bien su nombre es global, lo que permite que sea llamado desde el programa. Sin embargo, si se definen subprogramas dentro de otros subprogramas, los primeros constituyen bloques locales, si bien los identificadores del bloque exterior son no locales al bloque interior, mientras que sus parámetros formales son locales y no tienen vigencia en el bloque del programa principal.

Los identificadores globales y los no locales son reconocidos en la totalidad de su bloque incluso dentro de los bloques interiores. Solamente hay una excepción: cuando dentro del bloque local existe un identificador con el mismo nombre. En este caso, el identificador global queda oculto por el local, y toda mención a ese identificador en el ámbito más interno corresponde al objeto más local.

Además, el orden en que se definen los subprogramas es relevante, ya que los definidos en primer lugar pueden ser usados por los siguientes.

Como ejemplo, vamos a modificar los identificadores de nuestro programa `MultiplicaCifras` de forma que coincidan sus nombres en los diferentes bloques. Llamaremos `numero` a `numPosit` del programa principal, a `n` de `LeerNumPos` y a `n` de `QuitaUltimaCifra`. El programa quedaría de la siguiente forma:

```
Program MultiplicaCifras (input, output);
  var
    numero: integer;

  procedure LeerNumPos(var numero: integer);
    {Efecto: solicita un numero hasta obtener uno positivo}
  begin
    repeat
      Write('Escriba un entero mayor que cero: ');
      ReadLn(numero)
    until numero > 0
  end; {LeerNumPos}

  function ProductoCifras(numero: integer): integer;
    {Dev. el producto de las cifras de numero}
  var
    acumProd, cifrUnidades: integer;

  procedure QuitaUltimaCifra(var numero, ultima: integer);
    {Efecto: elimina la última cifra de numero y la almacena
    en ultima}
  begin
    ultima:= numero mod 10;
    numero:= numero div 10
  end; {QuitaUltimaCifra}

  begin
    acumProd:= 1;
    repeat
      QuitaUltimaCifra(numero, cifrUnidades);
      acumProd:= acumProd * cifrUnidades
    until numero = 0;
    ProductoCifras:= acumProd
  end; {ProductoCifras}
```

```

begin
  LeerNumPos(numero);
  WriteLn ('El producto de las cifras de ', numero, ' vale ',
          ProductoCifras(numero))
end.   {MultiplicaCifras}

```

Si nos situamos dentro de `LeerNumPos`, la variable global `numero`, que en principio está definida en todo el programa, no es accesible, porque es ocultada por el parámetro formal `numero` de `LeerNumPos` que es local.

Por el mismo motivo, si nos situamos en `QuitaUltimaCifra`, el parámetro formal `numero` de `ProductoCifras`, que en principio estaría definido dentro de `QuitaUltimaCifra` por ser no local a dicho procedimiento, no es accesible, al ser ocultado por su parámetro formal `numero`, que es local.

A veces se diferencia entre los bloques en que un identificador podría ser válido si no hubiera otros identificadores con el mismo nombre, lo que se conoce como *alcance del identificador*, de los bloques en que verdaderamente el identificador es accesible, al existir otros con el mismo nombre, lo que se denomina *visibilidad del identificador*.

En el ejemplo anterior, los subprogramas `LeerNumPos`, `ProductoCifras` y `QuitaUltimaCifra` están dentro del alcance de la variable global `numero` y sin embargo no pertenecen a su visibilidad.

En resumen, para saber a qué identificador nos referimos en cada caso y si su utilización es correcta, enunciamos las siguientes *reglas de ámbito*:

1. No se puede declarar un identificador más de una vez en el mismo bloque, pero sí en bloques diferentes aunque uno esté anidado en otro. Ambos identificadores representan dos objetos distintos.
2. Para saber a qué objeto se refiere un cierto identificador, hay que buscar el bloque más interior que contenga su declaración.
3. Un identificador sólo se puede utilizar en el bloque en que se ha declarado y en los que están contenidos en éste.⁷

⁷Es conveniente destacar que, como consecuencia inmediata de esta regla, el identificador de un subprograma es visible en su propio cuerpo de instrucciones. Por consiguiente, en cualesquiera de sus instrucciones pueden estar contenidas llamadas del subprograma a sí mismo. Si esto ocurre, el subprograma se llama *recursivo*. Este tipo de subprogramas se estudia en profundidad en el capítulo 10.

8.5.3 Criterios de localidad

Los diferentes ámbitos de validez de los identificadores, correctamente utilizados, permiten alcanzar una gran independencia entre el programa principal y sus subprogramas, y entre éstos y los subprogramas en ellos contenidos. De esta forma se puede modificar un subprograma sin tener que cambiar los demás, facilitando tanto el diseño del programa como posteriormente su depuración y mantenimiento. Además, facilitan la utilización de subprogramas ya creados (*bibliotecas de subprogramas*) dentro de nuevos programas, eliminando las posibles interferencias entre los objetos del programa y los de los subprogramas.

Para lograr estos efectos es necesario comprender primero con claridad cuál es el ámbito de los identificadores y seguir en lo posible unos sencillos criterios de localidad.

Los identificadores locales se deben utilizar para nombrar objetos utilizados dentro de un subprograma, incluyendo sus parámetros formales. Para conseguir el máximo grado de independencia es recomendable que se cumplan las siguientes condiciones:

- *Principio de máxima localidad*

Todos los objetos particulares de un subprograma, necesarios para que desempeñe su cometido, deben ser locales al mismo.

- *Principio de autonomía de los subprogramas*

La comunicación con el exterior debe realizarse exclusivamente mediante parámetros, evitándose dentro de los subprogramas toda referencia a objetos globales.

Si se cumplen ambas condiciones, en el punto de la llamada el subprograma se compara con una *caja negra* de paredes opacas cuyo contenido no puede verse desde fuera del mismo.

Obsérvese que ambos principios están relacionados, pues una mayor localidad implica una mayor ocultación de la información al quedar más objetos invisibles al resto del programa. De este modo, la independencia del subprograma con respecto al programa que lo invoca es máxima.

8.5.4 Efectos laterales

Hemos visto distintos mecanismos por los cuales un procedimiento o función pueden devolver o enviar resultados al programa principal (o a otro procedimiento o función). En el caso de las funciones existe un mecanismo específico de transmisión a través del propio nombre de la función, aunque limitado a tipos

simples. Tanto para los procedimientos como para las funciones, dichos valores pueden enviarse mediante parámetros por referencia.

Una tercera vía consiste en utilizar las variables globales (o las no locales), porque dichas variables son reconocidas en cualquier lugar del bloque. En consecuencia, si dentro de un procedimiento o función se hace referencia a una variable global (o no local), asignándole un nuevo valor, dicha asignación es correcta, al menos desde el punto de vista sintáctico.

Sin embargo, esta última posibilidad merma la autonomía de los subprogramas, y es perjudicial porque puede introducir cambios en variables globales y errores difíciles de detectar. Asimismo, resta independencia a los subprogramas, reduciendo la posibilidad de reutilizarlos en otros programas.

- ☉ Si se evita sistemáticamente el uso de los objetos globales en los subprogramas, los cambios que efectúa un subprograma se identifican inspeccionando la lista de parámetros por referencia. Por ello, se recomienda adquirir esta costumbre desde el principio.

Si por el contrario se suelen escribir subprogramas que emplean objetos globales, para conocer los efectos de un subprograma se tendrá que repasar cuidadosamente la totalidad del procedimiento o función. Por ello, esta práctica es desaconsejable y debe evitarse siempre.

Como norma general, debe evitarse toda alusión a las variables globales dentro de los subprogramas. No obstante, se incluirán como parámetros cuando sea preciso. Es importante que la comunicación se realice exclusivamente a través de los parámetros para garantizar la independencia de los subprogramas.

A los cambios en variables globales producidos por subprogramas se les denomina *efectos laterales o secundarios*. Veamos un ejemplo de una función cuya ejecución modifica una variable global de la que depende el propio resultado de la función.

```

Program ConDefectos (output);
  var
    estado: boolean;

  function Fea (n: integer): integer;
  begin
    if estado then
      Fea:= n
    else
      Fea:= 2 * n + 1;
      estado:= not estado
    end; {Fea}

```

```
begin
  estado:= True;
  WriteLn(Fea(1), ' ', Fea(1));
  WriteLn(Fea(2), ' ', Fea(2));
  WriteLn(Fea(Fea(5)))
end. {ConDefectos}
```

La salida obtenida al ejecutar el programa es la siguiente:

```
1 3
2 5
11
```

Como puede apreciarse, sucesivas llamadas con los mismos parámetros devuelven resultados diferentes al estar ligados al valor de variables externas.

Una buena costumbre (posible en Turbo Pascal) es definir las variables después de los subprogramas. Así se evita el peligro de producir efectos laterales.

8.6 Otras recomendaciones sobre el uso de parámetros

8.6.1 Parámetros por valor y por referencia

Se recomienda emplear parámetros por valor siempre que sea posible (asegurando que los argumentos no se alteran) y reservar los parámetros por referencia para aquellos casos en que sea necesario por utilizarse como parámetros de salida. Cuando se trabaja sobre datos estructurados grandes, como pueden ser vectores o matrices (véase el capítulo 12), puede estar justificado pasar dichas estructuras por referencia, aunque solamente se utilicen como parámetros de entrada, porque de esta forma no hay que duplicar el espacio en la memoria para copiar la estructura local, sino que ambas comparten la misma posición de memoria. También se ahorra el tiempo necesario para copiar de la estructura global a la local. Algunos compiladores modernos disponen de mecanismos de optimización que detectan los parámetros por valor no modificados en los subprogramas, evitando el gasto innecesario de tiempo y memoria invertido en efectuar su copia. Ello evita al programador alterar el mecanismo de paso, manteniéndolo por valor (lo que refleja el comportamiento del programa, que deja intacto al argumento) y a la vez se lleva a cabo eficientemente, usando el mecanismo de referencia.

8.6.2 Parámetros por referencia y funciones

En Pascal, tanto los procedimientos como las funciones pueden utilizar parámetros por valor y por referencia. Sin embargo la utilización de los parámetros

por referencia no es apropiada en las funciones, porque su cometido natural consiste sólo en hallar el valor que representan.

8.6.3 Funciones con resultados múltiples

Ya se ha dicho que las funciones tienen en Pascal la limitación de devolver únicamente valores pertenecientes a tipos simples. Si queremos que un subprograma devuelva valores múltiples, se recomienda utilizar procedimientos. Por ejemplo, si quisiéramos descomponer una cantidad de dinero en monedas de 100, de 25, duros y pesetas:

```
procedure Descomponer(cantDinero: integer;
                    var mon100, mon25, mon5, mon1: integer);
```

Igualmente, en aquellos casos en que, además del valor de la función, interese hallar algún valor adicional (por ejemplo, un código que indique si la función ha podido calcularse correctamente o si se ha producido algún error), se debe usar un procedimiento en su lugar más en consonancia con ese cometido.

8.7 Desarrollo correcto de subprogramas

De la misma forma que se ha expuesto para otros mecanismos del lenguaje, en esta sección estudiamos los elementos necesarios para lograr desarrollar subprogramas correctos. Hay dos aspectos de interés, la llamada es la que efectúa un cierto “encargo”, y la definición la que debe cumplimentarlo. Por eso, estos aspectos son complementarios. El necesario acuerdo entre definición y llamada se garantiza por medio de la especificación, “más o menos” formalmente.

En lo que respecta a la definición, para asegurar la corrección de un subprograma lo consideraremos como lo que es: un “pequeño” programa. Así, la tarea esencial en cuanto al estudio de su corrección consistirá en considerar sus instrucciones componentes y garantizar que cumple con su cometido, que es el descrito en la especificación. Así, por ejemplo:

```
function Fac(n: integer): integer;
  {Dev. n!}
  var
    i, prodAcum: integer;
begin
  prodAcum:= 1;
  {Inv.: i ≤ n y prodAcum = i!}
  for i:= 2 to n do
    prodAcum:= prodAcum * i;
```

```

    { prodAcum = n! }
    Fac := prodAcum
    { Fac = n! }
end; { Fac }

```

- ☉☉ En el caso de que en el código de nuestro subprograma aparezcan llamadas a otros subprogramas, la verificación dependerá, naturalmente, de la corrección de éstos. Nuestra tarea en este caso consistirá en comprobar que esas llamadas son correctas de la forma que se detallará a continuación. En el caso particular de que las llamadas sean al mismo subprograma (subprogramas recursivos) habrá que recurrir a técnicas de verificación específicas que serán explicadas en el capítulo 10.

Además, para cada subprograma especificaremos su interfaz de una forma semi-formal. Para ello explicitaremos al principio de cada subprograma una *precondición* que describa lo que precisa el subprograma para una ejecución correcta y una *postcondición* que indique los efectos que producirá. Así, en nuestro ejemplo:⁸

```

function Fac(n: integer): integer;
  {PreC.:  $0 \leq n \leq 7$  y  $n \leq \text{MaxInt}$ }
  {Devuelve n!}
  var
    i, prodAcum: integer;
begin
  prodAcum := 1;
  {Inv.:  $i \leq n$  y  $\text{prodAcum} = i!$  }
  for i := 2 to n do
    prodAcum := prodAcum * i;
    {prodAcum = n!}
  Fac := prodAcum
  {Fac = n!}
end; { Fac }

```

Más precisamente consideraremos que la *precondición* de un subprograma es una descripción informal de los requisitos que se deben cumplir para su correcto funcionamiento. La *postcondición* es una descripción más o menos formal del resultado o del comportamiento del subprograma.

La línea que proponemos es, pues, incluir las precondiciones y postcondiciones como parte de la documentación del subprograma. Visto desde un punto más formal, consideraremos que la especificación del subprograma está formada por

⁸*n* debe ser menor que 8 para que $n! \leq \text{MaxInt}$, ya que $8! = 40320 > \text{MaxInt}$.

el encabezamiento (con el nombre del subprograma y su correspondiente lista de parámetros), y las precondiciones y postcondiciones.

Por otra parte, es necesario verificar también la corrección de las llamadas a subprogramas. Para ello nos basaremos en que estas llamadas son sólo instrucciones (en el caso de los procedimientos) o expresiones (en el caso de las funciones). Por lo tanto, la verificación de la llamada se hará estudiando la precondición y la postcondición de la instrucción en la que aparece (la misma llamada en el caso de los procedimientos). Estas precondiciones y postcondiciones se extraerán, a su vez, de las precondiciones y postcondiciones del subprograma llamado. Así, si comprobamos que las expresiones que se pasan como parámetros cumplen la precondición, podemos deducir que los parámetros que devuelven los resultados verificarán lo especificado en la postcondición del subprograma. Las propiedades heredadas por los parámetros de salida constituirán la postcondición de la llamada. Por ejemplo:

```
ReadLn(a);
  {a = a0}
f:= Fac(a);
  {f = a0!}
WriteLn('El factorial de ',a:4,' es ',f:6)
```

Con este tratamiento de la corrección de subprogramas se continúa en la línea adoptada por los autores: buscar un compromiso entre un estudio riguroso de los programas y una visión práctica de la programación.

8.8 Ejercicios

1. Escriba una lista de los identificadores que hay en el programa `MultiplicaCifras`, indicando el tipo de objetos de que se trata, su ámbito y, si son parámetros, su modo de paso.
2. Defina el subprograma `EscribirFecha` que, aplicado a los datos 'D', 18, 9 y 60, dé lo siguiente:

Domingo, 18 de septiembre de 1.960

3. Defina la función `mannana` que halle el día de la semana, siguiente a uno dado,
 - (a) Representando los días de la semana como los enteros {1, ..., 7}.
 - (b) Representando los días de la semana como los caracteres {'L', 'M', 'X', 'J', 'V', 'S', 'D'}.
4. Defina un subprograma que intercambie los valores de dos variables enteras.
5. Defina un subprograma que averigüe si un carácter es o no:

- (a) una letra minúscula
 - (b) una letra mayúscula
 - (c) una letra, haciendo uso de las funciones anteriores
6. Escriba funciones para calcular las expresiones de los apartados (a)-(g) del ejercicio 10 del capítulo 3.
7. Defina el subprograma `RepetirCaracter` que escriba un carácter dado tantas veces como se indique.
- (a) Con él, escriba un programa que dibuje las figuras del ejercicio 1 del capítulo 5.
 - (b) Escriba un programa que dibuje la siguiente figura, consistente en n filas, donde la fila j es la secuencia de 2^j grupos formados cada uno por 2^{n-j-1} blancos y el mismo número de estrellas:

```

*****
*****
*****
****  ****  ****  ****  ****  ****  ****  ****
** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** **
* * * * * * * * * * * * * * * * * * * * * * * *
    
```

8. Escriba un procedimiento `PasaPasa` que manipule dos números enteros suprimiendo la última cifra del primero y añadiéndola al final del segundo. Incluya ese procedimiento en un programa que invierta un número `num` (partiendo del propio `num` y de otro, con valor inicial cero),

$$(12345, 0) \rightarrow (1234, 5) \rightarrow (123, 54) \rightarrow (12, 543) \rightarrow (1, 5432) \rightarrow (0, 54321)$$

a base de repetir la operación `PasaPasa` cuantas veces sea preciso.

9. Desarrolle un subprograma,

```

procedure QuitarDivisor(var ddo: integer; dsor: integer);
    
```

que divida al dividendo (`ddo`) por el divisor (`dsor`) cuantas veces sea posible, dando la línea de la descomposición correspondiente de la manera usual:

$$dividendo \mid divisor$$

Usar el procedimiento descrito en un programa que realice la descomposición de un número en sus factores primos.

10. Escriba un subprograma que halle el máximo común divisor de dos números enteros. Para ello, se pueden usar los métodos de Nicómaco o de las diferencias (descrito en la figura 7.3) y el de Euclides, a base de cambiar el mayor por el resto de la división entera (véase el ejercicio 2 del apartado 1.6 de [PAO94]).
- (a) ¿Qué requisitos deben exigirse a los datos para poder garantizar que los subprogramas definidos pararán?

- (b) Pruébelos para distintos pares de enteros y compare la eficiencia de los mismos.
11. Escriba funciones para hallar las siguientes cantidades:
- (a) Las cifras que tiene un entero.
 - (b) La cifra k-ésima de un entero, siendo la de las unidades la 0-ésima.
 - (c) La suma de las cifras de un entero.⁹
12. Desarrolle un programa que busque el primer número *perfecto*¹⁰ a partir de un cierto entero dado por el usuario haciendo uso de la función lógica **EsPerfecto**, que a su vez se apoya en la función **SumCifras** definida en el ejercicio anterior.
13. Desarrolle un programa que escriba todos los primos del 1 al 1000 haciendo uso de la función lógica **EsPrimo**. Esta función se definirá como en el apartado 8.2.1.
14. Defina la función *SerieArmonica* : $\mathcal{Z} \rightarrow \mathcal{R}$ definida así:

$$SerieArmonica(n) = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$$

⁹A esta cantidad se le llama raíz digital.

¹⁰Un número es perfecto si la suma de sus divisores (excluido él mismo) es igual al propio número.

Capítulo 9

Aspectos metodológicos de la programación con subprogramas

9.1	Introducción	189
9.2	Un ejemplo de referencia	190
9.3	Metodología de la programación con subprogramas	192
9.4	Estructura jerárquica de los subprogramas	199
9.5	Ventajas de la programación con subprogramas	201
9.6	Un ejemplo detallado: representación de funciones	203
9.7	Ejercicios	207

En este capítulo se exponen los aspectos metodológicos necesarios para que el programador aplique de una forma adecuada las técnicas de la programación con subprogramas. Puesto que éstas no deben ser empleadas de forma aislada, también se explica cómo combinarlas con otras técnicas presentadas con anterioridad, como la programación estructurada o el refinamiento correcto de programas. En la parte final del capítulo se destacan las ventajas aportadas por la correcta utilización de la programación con subprogramas.

9.1 Introducción

Los primeros computadores disponían de una memoria limitada, lo que obligaba a dividir un programa extenso en partes más pequeñas llamadas *módulos*

que constituyeran unidades lógicas del programa. Una parte era cargada en memoria y ejecutada, almacenándose los resultados o soluciones parciales obtenidos. A continuación, otra parte era cargada y ejecutada, accediendo a los resultados parciales, y así sucesivamente hasta alcanzar la solución final. Esta forma de operar, conocida como segmentación o memoria virtual segmentada (véase el apartado 4.2.6 de [PAO94]), representa una primera aplicación del concepto de modularidad.

A partir de los años setenta, se habla de un nuevo concepto de modularidad que no deriva de las limitaciones de memoria, sino de la creciente extensión de los programas, lo que dificulta su desarrollo, depuración y mantenimiento. En efecto, las necesidades crecientes de los usuarios generan programas cada vez más extensos y por lo tanto más difíciles de comprender. Por ello, es aconsejable dividirlos en partes para resolverlos en vez de intentar hacerlo en su totalidad, de una forma monolítica. En este sentido se pronuncian diversos estudios empíricos realizados sobre la capacidad humana para resolver problemas.

Este nuevo concepto de programación con subprogramas, que es el vigente en nuestros días, complementa las técnicas de programación estructurada y está relacionado estrechamente con las técnicas de diseño descendente y de refinamientos sucesivos. En resumen, la idea esencial de la programación con subprogramas se puede expresar así:

Una forma de resolver algorítmicamente un problema complejo consiste en descomponerlo en problemas más sencillos, bien especificados e independientes entre sí, diseñar por separado los subalgoritmos correspondientes a estos subproblemas y enlazarlos correctamente mediante llamadas a los mismos.

La programación con subprogramas presenta dos aspectos inseparables: la descomposición de un programa en partes, formadas por acciones y datos. En este capítulo vamos a estudiar el primero de esos aspectos: la descomposición de las acciones involucradas en un programa en otras más sencillas, que en Pascal se realiza mediante los subprogramas y su estructura de bloques. El aspecto de los datos se abordará después de estudiar la definición de tipos por el programador, completándose entonces el concepto de programación con subprogramas y su aplicación a la construcción de *tipos abstractos de datos* (véase el capítulo 19).

9.2 Un ejemplo de referencia

Supongamos que tratamos de hacer un programa para sumar dos fracciones representadas cada una por su numerador y su denominador, ambos enteros. Veamos una posible descomposición:

Sean $\frac{n1}{d1}$ y $\frac{n2}{d2}$ las fracciones que deseamos sumar
 Hallar la fracción $\frac{n}{d}$ suma
 Simplificar $\frac{n}{d}$, obteniéndose $\frac{n'}{d'}$ que es el resultado.

A su vez, Hallar la fracción $\frac{n}{d}$ suma consiste en:

Calcular $d = d1*d2$
 Calcular $n = n1*d2+d1*n2$

y la acción Simplificar $\frac{n}{d}$, obteniéndose $\frac{n'}{d'}$ se puede refinar como sigue:

Calcular $mcd = \text{máximo común divisor de } n \text{ y } d$
 Calcular $n' = n \text{ div } mcd$
 Calcular $d' = d \text{ div } mcd$

Ahora solamente faltaría desarrollar *Calcular mcd = máximo común divisor de n y d*. Éste es un cálculo muy frecuente y que aparece resuelto fácilmente de distintas formas en la mayoría de los manuales de programación (véase el ejercicio 2 del primer capítulo de [PAO94], y también el ejercicio 10 del capítulo anterior):

Sean $n, d, r \in \mathbb{N}$
mientras $d \neq 0$ **hacer**
 $r \leftarrow n \bmod d$
 $n \leftarrow d$
 $d \leftarrow r$
 El máximo común divisor es n

Si decidimos escribir un programa en Pascal (al que podríamos llamar por ejemplo `SumaDeFracciones`), tenemos que tomar algunas decisiones. En primer lugar hay que observar que el algoritmo consta de dos partes claramente diferenciadas: en la primera se efectúa la suma y en la segunda se simplifica la fracción. Este reparto de tareas se puede concretar en forma de dos procedimientos, que podríamos llamar `SumarFracciones` y `SimplificarFraccion`.

El primero recibiría las dos fracciones dadas (parámetros por valor) y devolvería la fracción suma (parámetros por referencia). El segundo actuaría sobre la fracción suma simplificándola (parámetros por referencia).

A su vez, el procedimiento `SimplificarFraccion`, y solamente él, ha de disponer del máximo común divisor del numerador y denominador de la fracción, por lo que es preciso definir una función MCD local a `SimplificarFraccion`.

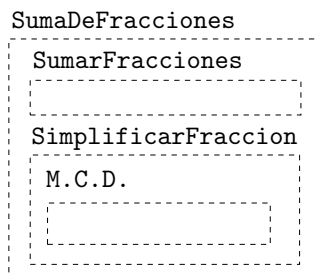


Figura 9.1. Estructura de bloques de `SumaDeFracciones`.

Por otra parte, `SumarFracciones` y `SimplificarFraccion` constituyen bloques locales e independientes entre sí. El programa principal no tiene acceso al interior de dichos bloques, sino que ha de llamarlos mediante sus nombres. En resumen, la estructura de bloques del programa sería la de la figura 9.1.

9.3 Metodología de la programación con subprogramas

La programación con subprogramas consiste en un conjunto de técnicas que permiten y facilitan la descomposición de un algoritmo en partes más simples enlazadas entre sí para su ejecución mediante llamadas realizadas por el programa principal o por otros subprogramas.

En el ejemplo de referencia, la acción correspondiente al programa principal `SumaDeFracciones` se descompone en dos más simples: `SumarFracciones` y `SimplificarFraccion`. El programa principal estará formado por una llamada a cada una de estas acciones, obteniéndose la solución buscada.

Un subprograma está formado por una agrupación de acciones y datos, de las cuales una parte (a la que llamamos *interfaz*) es visible fuera del mismo y permite su comunicación con el exterior, y la otra queda oculta al resto del programa. La interfaz está constituida por el identificador del subprograma y el tipo de sus parámetros. Esta parte tiene que ser conocida allí donde se efectúa la llamada. En cambio, el contenido de los subprogramas es privado y permanece oculto. Así, en el ejemplo, el programa principal no puede acceder a la función `MCD` porque es un objeto local del procedimiento `SimplificarFraccion`.

Cada subprograma debe desempeñar una acción específica e independiente de los demás de forma que sea posible aislar un subprograma determinado y concentrarnos en las acciones que desempeña sin preocuparnos por las posibles interferencias con los restantes. Las acciones expresadas en los subprogramas

`SumarFracciones` y `SimplificarFraccion` son totalmente independientes entre sí, de manera que ambas pueden usarse y verificarse por separado. Se facilita así la legibilidad del programa y la posibilidad de modificar alguna de sus partes sin preocuparnos por los efectos o la repercusión de éstas sobre otros subprogramas. Supongamos que en vez de sumar hubiera que multiplicar fracciones; bastaría entonces con sustituir `SumarFraccion` por un nuevo procedimiento `MultiplicarFraccion`, quedando el resto del programa inalterado.

También se facilita el mantenimiento del programa, puesto que las modificaciones necesarias afectarán solamente a algunos subprogramas.

En la programación con subprogramas debe atenderse a los siguientes aspectos:

- El cometido de cada subprograma, que se refleja en la interfaz y en la especificación.
- El desarrollo del subprograma en sí, que es un aspecto privado, oculto al exterior.
- Los objetos que surjan en este desarrollo son particulares del subprograma, por lo que deben ocultarse al exterior. Esta ocultación de la información consiste en que los objetos y acciones particulares de los subprogramas sean inaccesibles desde el exterior. Cada subprograma pasa a constituir una caja negra en la que se introducen unos datos y de la que se extraen unos resultados pero sin poder ver lo que pasa dentro.
- Como consecuencia, toda la comunicación con los subprogramas se debe realizar únicamente a través de los parámetros, alcanzándose entonces la *independencia* de los subprogramas entre sí. La independencia es deseable al facilitar el desarrollo del programa, su comprensión y verificación, su mantenimiento y reutilización posterior.
- En el desarrollo de subprogramas es corriente que surja la necesidad de crear nuevos subprogramas ..., dando lugar a una estructura jerárquica derivada de la aplicación de las técnicas de diseño descendente.

9.3.1 Diseño descendente con subprogramas

La división de un algoritmo en subprogramas requiere un proceso de abstracción por el que se usan, como si existieran, subalgoritmos sin concretar todavía. Debe entonces establecerse la interfaz y el cometido (especificación) de ese subprograma, que constituye su enunciado y que será útil para su posterior concreción y verificación según las ideas dadas en el apartado 8.7.

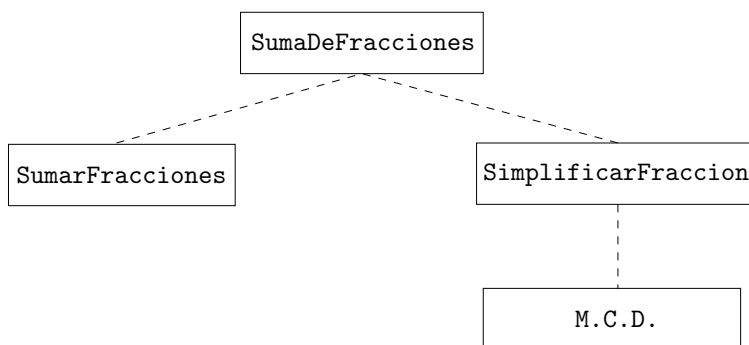


Figura 9.2. Estructura jerárquica del programa.

En fases sucesivas los subprogramas se van refinando alcanzándose un nivel inferior de abstracción. Cada una de estas fases puede determinar una nueva división en partes apareciendo nuevos subprogramas subordinados a los del nivel superior.

Al descender en la estructura de subprogramas disminuye el nivel de abstracción y, al alcanzar el nivel inferior, el algoritmo queda totalmente concretado.

Durante las etapas iniciales del proceso de diseño descendente por refinamientos sucesivos ciertas acciones quedan sin formalizar, determinadas solamente por su nombre, por los objetos sobre los que actúan y por su cometido, expresado más o menos formalmente. En la programación con subprogramas se nombran estas acciones, se establecen los parámetros necesarios para su correcto funcionamiento y se desarrollan con detalle en los siguientes niveles de refinamiento. Por este motivo, las técnicas de la programación con subprogramas son muy adecuadas para aplicar la metodología de diseño descendente.

Esta descomposición de un problema en partes suele representarse gráficamente mediante una estructura jerárquica de tipo arborescente, como la de la figura 9.2, que muestra las relaciones y dependencias entre los subprogramas.

En general, los subprogramas pertenecientes a los niveles superiores ejercen el control del flujo del programa, al efectuar llamadas a los de niveles inferiores, mientras que éstos realizan las acciones y calculan las expresiones. En otras palabras, se puede entender que los niveles superiores expresan la filosofía general del algoritmo, mientras que los inferiores o subordinados se ocupan de los detalles.

9.3.2 Programa principal y subprogramas

Al desarrollar el programa principal, lo fundamental es determinar correctamente cada una de las acciones y expresiones que lo componen y cuáles de

ellas se convertirán en subprogramas. A tal fin se definirán con precisión las especificaciones de los subprogramas (o sea, su cometido) y las condiciones que deben cumplir sus parámetros, pero no se entrará a detallar las acciones que los integran.

En consecuencia, el programa principal expresa una solución del problema con un elevado nivel de abstracción. En él se relacionan los nombres de las distintas acciones y expresiones abstractas simples en que se descompone el algoritmo, enlazándolas entre sí mediante instrucciones estructuradas. Desde él se activan dichas acciones y expresiones, y a él retorna el control de la ejecución del programa una vez que el subprograma llamado finaliza.

La descomposición de un problema en partes más sencillas para constituir el programa principal se puede hacer atendiendo a las distintas acciones necesarias para obtener la solución del problema (*descomposición por acciones*) o bien considerando cuál es la estructura de los datos, y una vez establecida, pasar a considerar las acciones que se aplicarán a dichos datos (*descomposición por datos*). En nuestro ejemplo de referencia se ha realizado una descomposición por acciones: `SumarFracciones`, `SimplificarFraccion` y `MCD` porque no se ha utilizado una estructura de datos para representar las fracciones, y por ser más natural. En el capítulo 19 estudiaremos la descomposición por datos.

¿Cuándo debe considerarse la creación de un nuevo subprograma? Si durante el desarrollo del programa principal es necesario empezar a profundizar en detalles sobre datos o instrucciones es porque en ese punto se necesita un subprograma. Por consiguiente, se dará nombre al nuevo subprograma, se definirá su cometido y se incluirá dentro del programa principal.

El programa principal depende directamente del problema por resolver, por lo tanto será diferente para cada problema, y no es reutilizable, aunque sí adaptable.

La jerarquía de la estructura del programa es, entre otros aspectos, una jerarquía de control, por lo que los efectos de un subprograma determinado deben afectar a sus subprogramas subordinados y en ningún caso a un subprograma superior. Deberá repasarse la estructura, subordinando aquellos subprogramas cuyo control sea ejercido por subprogramas inferiores.

9.3.3 Documentación de los subprogramas

Se ha dicho que, cuando surge la necesidad de un subprograma, debe definirse con precisión su cometido, incluyendo la información necesaria como documentación del subprograma. Para ello, deben tenerse en cuenta las siguientes posibilidades:

- El identificador es el primer descriptor de su cometido: suelen emplearse verbos en infinitivo para los procedimientos (`LeerDatos`, por ejemplo) y

sustantivos para las funciones (como `LetraMayuscula`), salvo las booleanas, que se indican con predicados (por ejemplo `EsDiaLaborable`).

- También en el encabezamiento, los parámetros deben nombrarse adecuadamente, y su tipo y modo ya ofrecen una información sobre los datos y resultados.
- Además del tipo, frecuentemente los datos deben acogerse a ciertos requisitos (precondición del subprograma), lo que se indicará en forma de comentario:

```
function Division (numerador, denominador: integer): integer;
  {PreC.: denominador <> 0}
```

- Cuando el identificador del subprograma deje lugar a dudas sobre su cometido, se indicará con otro comentario. En el caso de las funciones, indicando el valor que calculan:

```
function Division (numerador, denominador: integer): integer;
  {Dev. el cociente de la división entera entre numerador y
  denominador}
```

ya sea informal o formalmente. Y en el caso de los procedimientos, se indicará qué efecto tienen y qué parámetros se modifican cuando sea necesario:

```
procedure Dividir (num, den: integer; var coc, resto: integer);
  {Efecto: coc:= cociente entero de la división num/den
  resto:= resto de la división entera num/den}
```

9.3.4 Tamaño de los subprogramas

En general, el tamaño depende de lo complicado que sea el problema, siendo aconsejable descomponer un problema de complejidad considerable en subproblemas. Si los subprogramas obtenidos en una primera descomposición son excesivamente complejos, pueden descomponerse a su vez en nuevos subprogramas auxiliares que son llamados por los subprogramas de los que proceden. Sin embargo, esta división no puede proseguir indefinidamente, puesto que también aumenta el esfuerzo necesario para enlazarlas. Se debe parar la descomposición cuando el problema por resolver no presente especial dificultad o afronte una tarea de difícil descomposición en partes.¹

¹Aunque es difícil hablar de tamaño físico, rara vez se requieren subprogramas que supere una página de extensión (en Pascal), si bien éste es un valor relativo que depende además de la expresividad del lenguaje adoptado.

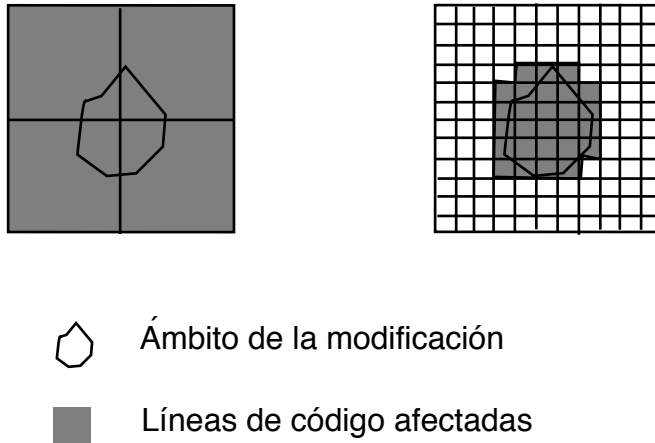


Figura 9.3.

Si la descomposición del problema es correcta, cada subprograma se tiene que corresponder con una cierta acción abstracta funcionalmente independiente de las demás que puede ser desarrollada y probada por separado. Para conseguirlo se debe analizar la estructura del programa, disminuyendo la dependencia mediante la integración de aquellos subprogramas que utilicen espacios o estructuras comunes de datos y fraccionando aquéllos que agrupen tareas diferentes.

El tamaño de los subprogramas es uno de los aspectos que más influyen en el esfuerzo requerido por las operaciones de mantenimiento de un programa. Si un programa está formado por subprogramas de tamaño reducido los efectos de una modificación afectarán a menos líneas de código, aunque probablemente aumente el número de subprogramas a los que éstas pertenecen, como se ve en la figura 9.3.

9.3.5 Refinamiento con subprogramas y con instrucciones estructuradas

Aplicando todo esto a nuestro ejemplo, y una vez que los distintos niveles han quedado refinados, pasamos a desarrollar las acciones y expresiones abstractas que componen los subprogramas utilizando las instrucciones estructuradas, como en el cálculo del máximo común divisor según Euclides:

```

function MCD(n, d: integer): integer;
  {PreC.:  $n \neq 0$  y  $d \neq 0$ }
  {Dev. el m.c.d. de n y d}

```

```

var
  r: integer;
begin
  while d <> 0 do begin
    r:= n mod d;
    n:= d;
    d:= r
  end; {while}
  MCD:= n
end; {MCD}

```

Las técnicas de programación estructurada descomponen las acciones complejas mediante instrucciones estructuradas que controlan acciones más sencillas o realizan llamadas a subprogramas. En este caso, los subprogramas realizan acciones abstractas definidas mediante sus especificaciones.

Recordemos, por ejemplo, el esquema de un programa controlado por menú:

```

repetir
  Mostrar menú
  Leer opcion
  en caso de que opcion sea
    0: Salir
    1: Entrada de datos por teclado
    2: Lectura de datos de archivo
    3: Listado de datos
    ...
    n: Ejecutar la opción n-ésima
hasta opcion = 0

```

La instrucción estructurada *Repetir... hasta* está controlando la ejecución de las acciones *Mostrar menú* y *Leer opción*, y la instrucción de selección múltiple *En caso de que... sea* controla las acciones *Entrada de datos por teclado*, *Lectura de datos de archivo*, etc. correspondientes a las sucesivas opciones del programa cuyo desarrollo está todavía por hacer. Estas acciones posiblemente pertenecerán a subprogramas con uno o más niveles inferiores cuando sean refinadas.

En consecuencia, el diseño por refinamientos sucesivos genera una estructura jerárquica de tipo arborescente en la que las llamadas a los subprogramas se controlan mediante instrucciones estructuradas (secuencia, selección y repetición) y, a su vez, los distintos subprogramas se desarrollan internamente mediante instrucciones estructuradas.

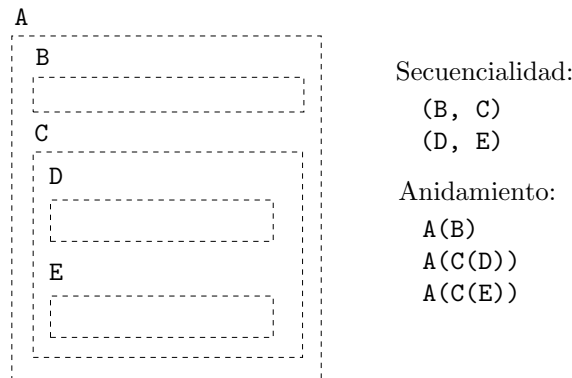


Figura 9.4. Subordinación de bloques.

9.4 Estructura jerárquica de los subprogramas

En este apartado se ofrece una visión más teórica y menos técnica de los conceptos explicados en el apartado 8.5 del capítulo anterior.

Al objeto de poder expresar la estructura arborescente que refleja la jerarquía entre subprogramas y las características deseables de ocultación de la información e independencia funcional, ciertos lenguajes de programación (como Pascal) utilizan una estructura de bloques que permite dividir el programa en partes con sus propias instrucciones y datos. La disposición de los bloques se puede hacer en forma secuencial (sin que esta secuencia tenga nada que ver con el orden de ejecución de los bloques, que vendrá dado por la disposición de las llamadas respectivas), para los bloques situados en un mismo nivel, o en forma anidada, para representar la subordinación de los bloques con distintos niveles de anidamiento, como puede verse en la figura 9.4.

Los lenguajes de programación con estructura de bloques facilitan el cumplimiento de las condiciones necesarias para alcanzar un elevado nivel de ocultación de la información:

- Cada bloque subordinado puede contar con sus propios objetos, llamados objetos locales, a los que los subprogramas superiores no tienen acceso.
- La activación de un subprograma subordinado por la llamada de otro superior o de su mismo nivel es la única forma posible para ejecutar sus instrucciones.
- La comunicación entre un bloque y su subordinado puede y debe efectuarse solamente mediante los parámetros.

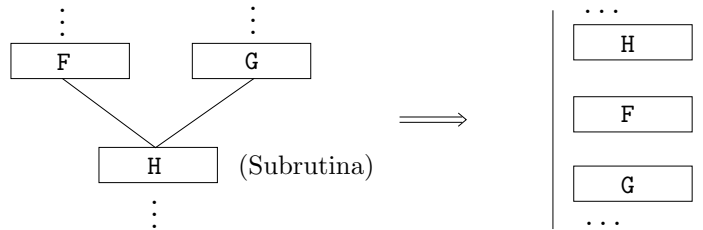


Figura 9.5.

Los objetos propios del programa principal se llaman *globales* y los objetos de un bloque que tiene otro anidado son *no locales* con respecto a este último. Desde los subprogramas subordinados de un determinado nivel se puede acceder a los objetos globales y no locales, permitiendo la utilización de espacios comunes de datos, en cuyo caso disminuiría la deseable independencia funcional de los subprogramas. En general debe evitarse este tipo de acceso, aunque en ciertos casos pueda estar justificado.

Supongamos, por ejemplo, que dos o más subprogramas situados en un mismo nivel tengan un mismo subprograma subordinado, como se muestra en la figura 9.5.

En este caso, el subprograma subordinado no puede estar anidado dentro de uno de los subprogramas superiores, pues no podría ser llamado por el otro. Tiene que estar al mismo nivel que los subprogramas que lo llaman. Algunos autores denominan *subrutinas* a este tipo de subprogramas con grado de entrada mayor que uno para diferenciarlos de los subprogramas. El uso de subrutinas puede justificar la vulneración del principio de máxima localidad (véase el apartado 8.5.3).

Los parámetros son objetos locales de los subprogramas a través de los cuáles se comunican con sus subprogramas superiores. Cuando el subprograma superior efectúa una llamada a su subordinado, además de su nombre debe incluir aquellos objetos cuyos valores van a ser utilizados por el subprograma subordinado. Este proceso se conoce como paso de parámetros y puede hacerse básicamente de dos formas:

- En la primera, el subprograma recibe únicamente el valor de los objetos, por lo que no puede modificarlos.
- En la segunda, el subprograma recibe la dirección de los objetos, por lo

que puede modificarlos.²

La primera forma es la que presenta una mayor independencia, por lo que debe utilizarse siempre que sea posible. La segunda tiene una dependencia mayor, pues el subprograma subordinado y el superior comparten el mismo espacio de datos, pero permite que el subprograma subordinado envíe resultados al superior, por lo que su uso estará justificado en dichos casos.

Cuando es necesario pasar una estructura de datos extensa desde un subprograma a otro, el paso por valor exige más tiempo y más espacio de almacenamiento que el paso por dirección, y por motivos de eficiencia, se suele hacer una excepción a esta regla.

9.5 Ventajas de la programación con subprogramas

En este apartado se van a comentar las ventajas de la programación con subprogramas, que han hecho esta metodología imprescindible para abordar cualquier problema no trivial.

Programas extensos

Las técnicas de la programación con subprogramas facilitan la construcción de programas extensos y complejos al permitir su división en otros más sencillos, formados por menos instrucciones y objetos, haciéndolos abarcables y comprensibles para el intelecto humano.

El desarrollo del programa principal de un problema extenso no es una tarea fácil, por lo que requiere programadores con gran experiencia y capacitación. Sin embargo, la creación de los restantes subprogramas es más sencilla, lo que permite la intervención de programadores noveles. En este sentido, la programación con subprogramas favorece el trabajo en grupo y permite la creación de las grandes aplicaciones tan frecuentes hoy en día, lo que sería una misión imposible para individuos aislados.

Código reutilizable

La estructura del programa principal representa la línea lógica del algoritmo, por lo que es diferente en cada caso. No sucede lo mismo con los restantes subprogramas, que pueden ser reutilizados en otros algoritmos distintos de aquél en que fue diseñado siempre que se requieran las mismas acciones simples.

²Estas dos formas de paso de parámetros se corresponden con el paso de parámetros por valor y por referencia que hemos estudiado en Pascal. (Véase el apartado 8.2.3.)

En consecuencia, el código generado aplicando los principios de la programación con subprogramas es *reutilizable*, por lo que puede ser incorporado en otros programas, lo que significa un importante ahorro de tiempo y trabajo. De hecho, es frecuente la creación de bibliotecas compuestas por subprogramas especializados para ciertas aplicaciones, como cálculo numérico, estadística, gráficos, etc. Dichas bibliotecas están disponibles en ciertas instituciones de forma gratuita o comercial; de ellas, se toman aquellos subprogramas que se precisen y se introducen dentro del programa. Las técnicas de programación con subprogramas facilitan la utilización de las bibliotecas y garantizan que no se produzcan incompatibilidades entre los subprogramas debido, esencialmente, a su independencia.

Cuando se dispone de los subprogramas más elementales, procedentes de bibliotecas o de otros programas creados con anterioridad, y se integran para realizar acciones más complejas, y éstas se integran a su vez para efectuar otras más complejas, y así sucesivamente, hasta obtener la solución de un problema, se dice que se ha seguido una metodología de *diseño ascendente* (*bottom-up*).

Depuración y verificación

Un subprograma puede comprobarse por separado, mediante un programa de prueba que efectúe la llamada al subprograma, le pase unos datos de prueba y muestre los resultados. Una vez que se hayan comprobado separadamente los subprogramas correspondientes a una sección del programa, pueden comprobarse conjuntamente, y por último probar el programa en su totalidad. La comprobación de un programa dividido en subprogramas es más fácil de realizar y por su propia estructura más exhaustiva que la de un programa monolítico.

También puede utilizarse la llamada estrategia *incremental* de pruebas, consistente en codificar en primer lugar los subprogramas de los niveles superiores, utilizando subprogramas subordinados provisionales (que realicen su tarea lo más simplificada posible). De esta forma se dispone de una versión previa del sistema funcionando continuamente durante todo el proceso de pruebas, facilitando así la intervención del usuario en éstas.

Igualmente, el proceso de verificación formal será también más llevadero sobre un programa dividido en partes que sobre la totalidad. Como se explicó en el apartado 8.7, la verificación de un programa con subprogramas consistirá en verificar cada uno de éstos, así como su correcto ensamblaje (mediante llamadas). Ninguna de estas tareas será complicada, y se simplificará notablemente la comprobación de la corrección con respecto a la de un programa de una sola pieza.

Por consiguiente, un programa construido mediante subprogramas tendrá menos errores y éstos serán más fáciles de detectar y subsanar.

Mantenimiento

Por otra parte, la programación con subprogramas sirve de gran ayuda en el mantenimiento y modificación de los programas, ya que si se ha respetado la independencia funcional entre subprogramas, introducir cambios o subsanar errores tendrá unos efectos nulos o mínimos sobre el resto del programa.

9.6 Un ejemplo detallado: representación de funciones

Se trata de representar funciones reales de una variable real en la pantalla del computador de forma aproximada. La función representada es fija para el programa; en nuestro ejemplo, se ha tomado $f(x) = \text{sen}(x)$, aunque puede cambiarse fácilmente aprovechando las ventajas de la programación con subprogramas. Los datos solicitados por el programa determinan el fragmento del plano XY que se desea representar:

$$[x_{\text{mínima}}, x_{\text{máxima}}] \times [y_{\text{mínima}}, y_{\text{máxima}}]$$

En nuestro ejemplo representaremos el fragmento

$$[0.5, 6.5] \times [-0.9, 0.9]$$

que es bastante ilustrativo acerca del comportamiento de la función seno.

Por otra parte, como el tamaño de la pantalla es fijo, la representación se efectúa sobre una cuadrícula de tamaño fijo, formada por $\text{núm}X \times \text{núm}Y$ puntos, que estará representado por sendas constantes del programa:

```
const
  NumX=15; NumY=50;
```

Por comodidad, el eje de abscisas será vertical y avanzará descendentemente, y el de ordenadas será horizontal y avanzará hacia la derecha de la pantalla, como se ve en la figura 9.6.

Como podemos ver se ha trazado una cabecera con los límites de la representación de las ordenadas (en la figura -0.90 y 0.90), el nombre de la función representada ($y = \text{sen } (x)$ en el ejemplo) y una línea horizontal de separación. Debajo, para cada línea, se ha escrito el valor de la abscisa ($0.50, 0.90, \dots$) correspondiente, una línea vertical para representar un fragmento de eje y un asterisco para representar la posición de la función. Si la función se sale fuera de la zona de representación, se ha escrito un símbolo $\langle \text{ ó } \rangle$, según caiga por la izquierda o por la derecha, respectivamente.

Así pues, el programa consta de cuatro pasos:

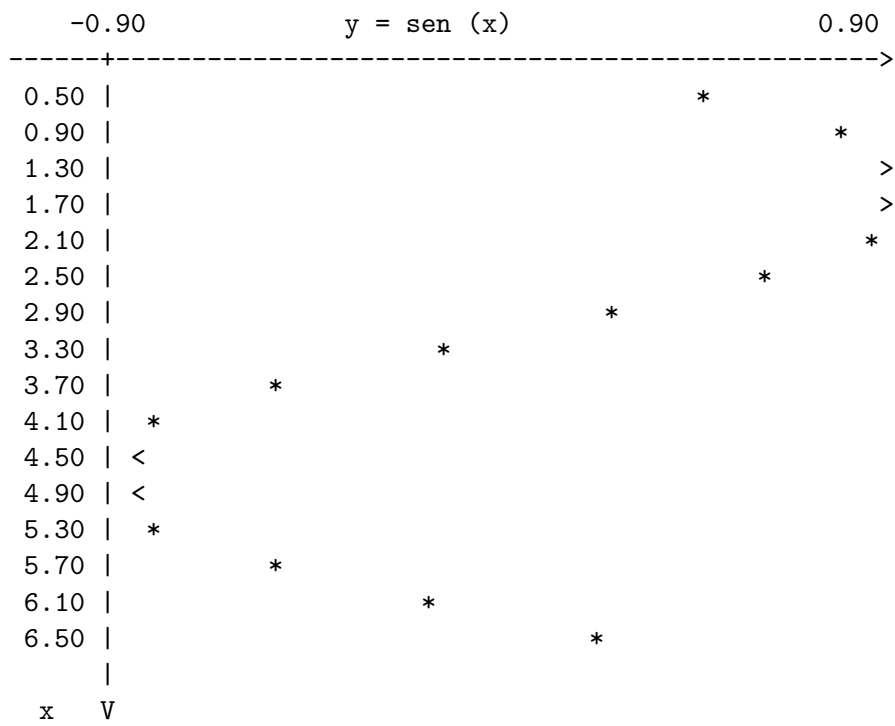


Figura 9.6.

Pedir los datos $x_{\text{mínima}}, x_{\text{máxima}}, y_{\text{mínima}}, y_{\text{máxima}}$
Trazar la cabecera de la gráfica
Trazar las líneas sucesivas
Trazar el pie de la gráfica

La lectura de los datos es trivial:

```

procedure PedirDatos(var xMin, xMax, yMin, yMax: real);
  {Efecto: lee el fragmento del plano que se desea ver}
begin
  Write('xMínimo, xMáximo: ');
  ReadLn(xMin, xMax);
  Write('yMínimo, yMáximo:');
  ReadLn(yMin, yMax)
end; {PedirDatos}

```

La cabecera de la representación gráfica debe reflejar el intervalo de las ordenadas elegido y escribir un eje del tamaño *numY*:

```

procedure TrazarCabecera(yMin, yMax: real);
  {Efecto: Traza la cabecera centrada dependiendo del tamaño de la
  pantalla}
begin
  WriteLn(yMin:9:2,                                     {a la izquierda}
          'y = sen (x)': NumY div 2-1,                 {en el centro}
          yMax:(NumY div 2-1):2);                       {a la derecha}
  Write('-----+');
  for i:= 1 to NumY do
    Write(' ');
  WriteLn('>>')
end; {TrazarCabecera}

```

siendo NumX, NumY las constantes (globales) descritas al principio. (Los parámetros de formato tienen por misión centrar el nombre de la función de manera que no haya que redefinir este procedimiento si cambia el tamaño de la pantalla.)

El trazado de cada línea consiste en lo siguiente:

Hallar la abscisa x_i
Hallar la posición (en la pantalla) de la ordenada $f(x_i)$
Escribir la línea (comprobando si cae fuera de la zona)

lo que se detalla a continuación. La abscisa x_i se halla fácilmente:

$$x_i = x_{\text{mín}} + i \frac{x_{\text{máx}} - x_{\text{mín}}}{\text{NumX}}, \quad i \in \{0, \dots, \text{NumX}\}$$

Para cada ordenada $y_i = f(x_i)$, su posición (que será un entero de $\{0, \dots, \text{NumY}\}$ cuando $y_i \in [y_{\text{mín}}, y_{\text{máx}}]$:

$$[y_{\text{mín}}, y_{\text{máx}}] \rightarrow \{0, \dots, \text{NumY}\}$$

Ello se consigue sencillamente así:

$$\text{posY}_i = \text{Round} \left(\text{NumY} \frac{y_i - y_{\text{mín}}}{y_{\text{máx}} - y_{\text{mín}}} \right)$$

Un valor de posY_i negativo o nulo indica que la función se sale por la izquierda del fragmento del plano representado, mientras que un valor mayor que NumY significa que se sale por la derecha, con lo que la línea i -ésima se traza como sigue.³

```

procedure TrazarLinea(i: integer; xMin, xMax, yMin,
    yMax: real);
    {Efecto: se imprime la línea i-ésima}
    var
        xi: real; {el valor de abscisa}
        posYi: integer; {el valor redondeado de la función en xi}
begin
    xi:= xMin + i * (xMax - xMin)/NumX;
    posYi:= Round(NumY * ((Sin(xi)-yMin)/(yMax-yMin)));
    Write(xi:5:2, '□|□');
    if posYi <= 0 then
        WriteLn('<')
    else if posYi > NumY then
        WriteLn('>':NumY)
    else {dentro de la zona}
        WriteLn('*':posYi)
end; {TrazarLinea}

```

Finalmente, el pie de la gráfica se dibuja así:

```

procedure TrazarPie;
begin
    WriteLn('□□□□□□|');
    WriteLn('□□x□□□V')
end; {TrazarPie}

```

En resumen, el programa consta de lo siguiente:

³Dadas las especiales características gráficas de este ejemplo, se indican mediante el símbolo □ los espacios en blanco en las instrucciones de escritura.

```

Program ReprGrafica (input, output);
const
  NumX = 15; NumY = 50;
var
  xMinimo, xMaximo, yMinimo, yMaximo: real;
  i: integer;

  procedure PedirDatos(...);      {... descrito antes ... }
  procedure TrazarCabecera(...); {... descrito antes ... }
  procedure TrazarLinea(...);    {... descrito antes ... }
  procedure TrazarPie;           {... descrito antes ... }

begin
  PedirDatos(xMinimo, xMaximo, yMinimo, yMaximo);
  TrazarCabecera(yMinimo, yMaximo);
  for i:= 0 to NumX do
    TrazarLinea (i, xMinimo, xMaximo, yMinimo, yMaximo);
  TrazarPie
end. {ReprGrafica}

```

9.7 Ejercicios

1. Escriba un programa en Pascal para el ejemplo de referencia del apartado 9.2.
2. Utilice la independencia de subprogramas en el programa anterior para sustituir el cálculo del máximo común divisor mediante el método de Euclides por otro que utilice las siguientes propiedades debidas a Nicómaco de Gersasa, también llamado método de las diferencias:

$$\begin{aligned}
 & \text{si } a > b, \text{ entonces } m.c.d.(a, b) = m.c.d.(a-b, b) \\
 & \text{si } a < b, \text{ entonces } m.c.d.(a, b) = m.c.d.(a, b-a) \\
 & \text{si } a = b, \text{ entonces } m.c.d.(a, b) = m.c.d.(b, a) = a = b
 \end{aligned}$$

Por ejemplo, el cálculo del m.c.d. de 126 y 56 seguiría la siguiente evolución:

$$(126, 56) \rightsquigarrow (70, 56) \rightsquigarrow (14, 56) \rightsquigarrow (14, 42) \rightsquigarrow (14, 28) \rightsquigarrow (14, 14)$$

3. Escriba un programa que pida dos fracciones, las simplifique y las sume, hallando para ello el mínimo común múltiplo de los denominadores y simplificando nuevamente el resultado. Organice los subprogramas de acuerdo con el siguiente diagrama de la figura 9.7.⁴

⁴Obsérvese que, tanto la función MCM como el procedimiento para Simplificar, se apoyan en la función MCD. Puesto que $MCM(a, b) * MCD(a, b) = a * b$, se tiene que

$$MCM(a, b) = \frac{a \cdot b}{MCD(a, b)}$$

(Utilícese el subprograma definido en el ejercicio 2 para el cálculo del MCD.)

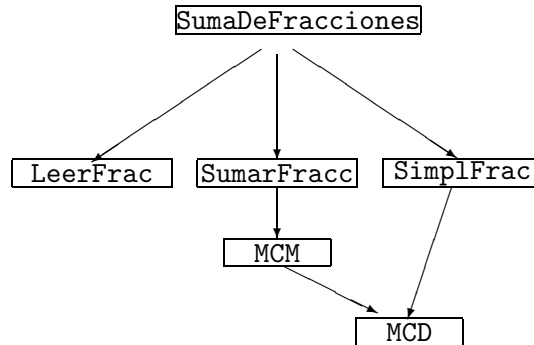


Figura 9.7.

4. Desarrolle una función **Producto**

$$\text{Producto}(a, b) = a * (a+1) * \dots * (b-1) * b$$

y, basándose en ella, escriba funciones para hallar las cantidades $n!$ y $\binom{n}{k}$.

Incluya esta última función en un programa que tabule los coeficientes binomiales $\binom{n}{k}$ de la siguiente forma:

$$\begin{array}{cccccc}
 & & & & & 1 \\
 & & & & 1 & & 1 \\
 & & & 1 & 2 & & 1 \\
 & & 1 & 3 & 3 & & 1 \\
 1 & & 4 & 6 & 4 & & 1 \\
 & & & \dots & & &
 \end{array}$$

hasta la línea `numLinea`, dato extraído del `input`.

5. Defina distintas versiones de la función *arcsen* según las siguientes descripciones:

(a) $\text{arcsen}(x) = \text{arctg} \frac{x}{\sqrt{1-x^2}}$

(b) $\text{arcsen}(x) = x + \frac{1}{2} \frac{x^3}{3} + \frac{1*3}{2*4} \frac{x^5}{5} + \frac{1*3*5}{2*4*6} \frac{x^7}{7} + \dots$

(c) Como $\text{arcsen}(a)$ es un cero de la función $f(x) = \text{sen}(x) - a$, llegar a éste

- por bipartición, es decir, siguiendo el teorema de Bolzano (véase el apartado 6.5.1)
- por el método de la tangente (véase el apartado 6.5.2)

6. Cada año, el 10% de la gente que vive en la ciudad emigra al campo huyendo de los ruidos y la contaminación. Cada año, el 20% de la población rural se traslada a la ciudad huyendo de esa vida monótona.

- Desarrolle un subprograma `EmigracionAnual` que modifique las poblaciones rural y urbana con arreglo a lo explicado.

- Desarrolle un programa que muestre la evolución de las migraciones, partiendo de unas poblaciones rural y urbana iniciales de cinco y cuatro millones de habitantes respectivamente, hasta que se estabilicen esas migraciones, esto es, cuando un año la población rural (por ejemplo) no sufra variación alguna.
7. Realice una descomposición en subprogramas y escriba el correspondiente programa para el “Juego de Nicómaco” para dos jugadores, en el que se parte de un par de números positivos, por ejemplo $(124, 7)$, y se van restando alternativamente por cada jugador múltiplos del número más pequeño al número más grande. Así, del par inicial se puede pasar al $(103, 7)$, restando $21 (=7*3)$ a 124 , o incluso al $(5, 7)$ al restar $119 (7*17)$. A continuación se restará 5 de 7 obteniéndose $(5, 2)$ y así sucesivamente hasta que un jugador consiga hacer cero uno de los números, ganando la partida.

Capítulo 10

Introducción a la recursión

10.1 Un ejemplo de referencia	212
10.2 Conceptos básicos	213
10.3 Otros ejemplos recursivos	216
10.4 Corrección de subprogramas recursivos	219
10.5 Recursión mutua	222
10.6 Recursión e iteración	226
10.7 Ejercicios	227
10.8 Referencias bibliográficas	228

En este capítulo se estudia una técnica de programación que tiene su origen en ciertos cálculos matemáticos y que consiste en describir los cálculos o las acciones de una manera autoalusiva, esto es, resolver problemas describiéndolos en términos de ejemplares más sencillos de sí mismos.

Esta técnica puede entenderse como un caso particular de la programación con subprogramas en la que se planteaba la resolución de un problema en términos de otros subproblemas más sencillos. El caso que nos ocupa en este capítulo es aquel en el que al menos uno de los subproblemas es una instancia del problema original.

10.1 Un ejemplo de referencia

Consideremos el cálculo del factorial de un entero positivo n que se define de la siguiente forma:

$$n! = n * (n - 1) * (n - 2) * \dots * 1$$

Como, a su vez,

$$(n - 1)! = (n - 1) * (n - 2) \dots * 1$$

tenemos que $n!$ se puede definir en términos de $(n - 1)!$, para $n > 0$, así:

$$n! = n * (n - 1)!$$

siendo por definición $0! = 1$, lo que permite terminar correctamente los cálculos. Por ejemplo, al calcular el factorial de 3:

$$3! = 3 * 2! = 3 * 2 * 1! = 3 * 2 * 1 * 0! = 3 * 2 * 1 * 1 = 6$$

Por lo tanto, si n es distinto de cero tendremos que calcular el factorial de $n - 1$, y si es cero el factorial es directamente 1:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n * (n - 1)! & \text{si } n \geq 1 \end{cases}$$

Observamos en este ejemplo que en la definición de factorial interviene el propio factorial. Este tipo de definiciones en las que interviene lo definido se llaman *recursivas*.

La definición anterior se escribe en Pascal directamente como sigue:¹

```
function Fac(num: integer): integer;
  {PreC.: num ≥ 0}
  {Dev. num!}
begin
  if num = 0 then
    Fac:= 1
  else
    Fac:= num * Fac(num - 1)
end; {Fac}
```

¹En estos primeros ejemplos obviaremos algunos detalles de corrección que serán explicados más adelante.

La posibilidad de que la función `Fac` se llame a sí misma existe, porque en Pascal el identificador `Fac` es válido dentro del bloque de la propia función (véase el apartado 8.5). Al ejecutarlo sobre el argumento 4, se produce la cadena de llamadas sucesivas a `Fac(4)`, `Fac(3)`, `Fac(2)`, `Fac(1)` y a `Fac(0)`, así:

```

Fac(4)  ~> 4 * Fac(3)
        ~> 4 * (3 * Fac(2))
        ~> 4 * (3 * (2 * Fac(1)))
        ~> 4 * (3 * (2 * (1 * Fac(0))))
        ~> ...

```

y, como `Fac(0) = 1`, este valor es devuelto a la llamada anterior `Fac(1)` multiplicándose `1 * Fac(0)`, que a su vez es devuelto a `Fac(2)`, donde se multiplica `2 * Fac(1)` y así sucesivamente, deshaciéndose todas las llamadas anteriores en orden inverso:²

```

...    ~> 4 * (3 * (2 * (1 * 1)))
        ~> 4 * (3 * (2 * 1))
        ~> 4 * (3 * 2)
        ~> 4 * 6
        ~> 24

```

10.2 Conceptos básicos

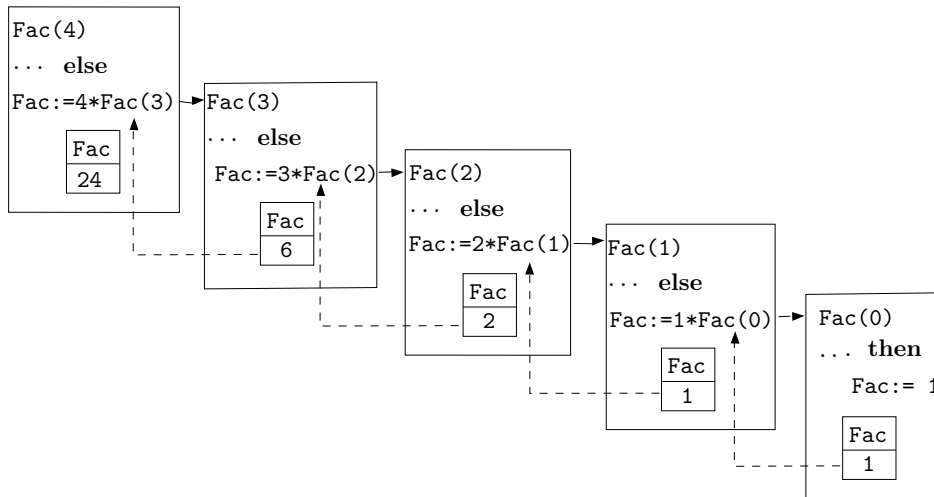
En resumen, los subprogramas recursivos se caracterizan por la posibilidad de invocarse a sí mismos.

Debe existir al menos un valor del parámetro sobre el que se hace la recursión, llamado *caso base*, que no provoca un nuevo cálculo recursivo, con lo que finaliza y puede obtenerse la solución; en el ejemplo del factorial, es el cero. Si este valor no existe, el cálculo no termina. Los restantes se llaman *casos recurrentes*, y son aquéllos para los que sí se produce un nuevo cálculo recursivo; en el ejemplo, se trata de los valores positivos 1, 2, 3...

En las sucesivas llamadas recursivas los argumentos deben aproximarse a los casos base,

$$n \rightarrow n - 1 \rightarrow \dots \rightarrow 1 \rightarrow 0$$

²La mayoría de los entornos de desarrollo (como Turbo Pascal) integran un módulo depurador que permite observar los valores adoptados por los diferentes parámetros y variables que intervienen en un programa durante su ejecución (véase el apartado C.2.6). Esto es particularmente útil para la comprensión y el desarrollo de subprogramas recursivos.

Figura 10.1. Esquema de llamadas de Fac .

para que el proceso concluya al alcanzarse éstos. De lo contrario, se produce la llamada “recursión infinita”. Por ejemplo, si se aplicase la definición de factorial a un número negativo,

$$-3 \rightarrow -4 \rightarrow -5 \rightarrow \dots$$

los cálculos sucesivos nos alejan cada vez más del valor cero, por lo que nunca dejan de generarse llamadas.

El proceso de ejecución de un subprograma recursivo consiste en una cadena de generación de llamadas (suspendiéndose los restantes cálculos) y reanudación de los mismos al término de la ejecución de las llamadas, tal como se recoge en la figura 10.1. Para comprender mejor el funcionamiento de un subprograma recursivo, recordemos el proceso de llamada a un subprograma cualquiera:

- Se reserva el espacio en memoria necesario para almacenar los parámetros y los demás objetos locales del subprograma.
- Se reciben los parámetros y se cede la ejecución de instrucciones al subprograma, que comienza a ejecutarse.
- Al terminar éste, se libera el espacio reservado, los identificadores locales dejan de tener vigencia y pasa a ejecutarse la instrucción siguiente a la de llamada.

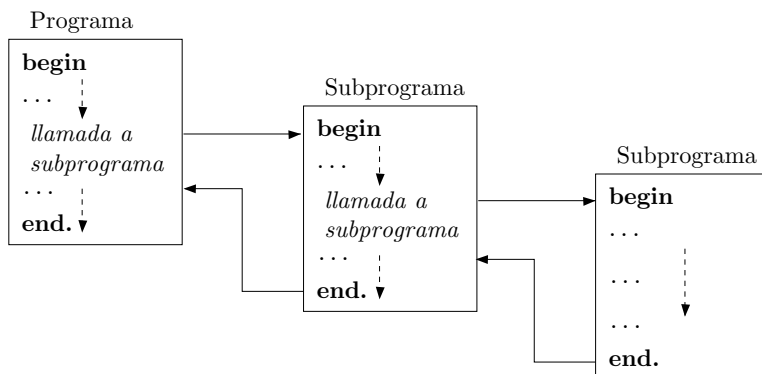


Figura 10.2. Esquema de llamadas de subprogramas.

En el caso de un subprograma recursivo, cada llamada genera un nuevo ejemplar del subprograma con sus correspondientes objetos locales. Podemos imaginar cada ejemplar como una copia del subprograma en ejecución. En este proceso (resumido en la figura 10.2) destacamos los siguientes detalles:

- El subprograma comienza a ejecutarse normalmente y, al llegar a la llamada, se reserva espacio para una nueva copia de sus objetos locales y parámetros. Estos datos particulares de cada ejemplar generado se agrupan en la llamada *tabla de activación* del subprograma.
- El nuevo ejemplar del subprograma pasa a ejecutarse sobre su tabla de activación, que se amontona sobre las de las llamadas recursivas anteriores formando la llamada *pila* recursiva (véase el apartado 17.2.3).
- Este proceso termina cuando un ejemplar no genera más llamadas recursivas por consistir sus argumentos en casos básicos.

Entonces, se libera el espacio reservado para la tabla de activación de ese ejemplar, reanudándose las instrucciones del subprograma anterior sobre la tabla penúltima.

- Este proceso de retorno finaliza con la llamada inicial.

10.3 Otros ejemplos recursivos

10.3.1 La sucesión de Fibonacci

Un cálculo con definición recursiva es el de la sucesión de números de Fibonacci:³ 1, 1, 2, 3, 5, 8, 13, 21, 34, ... (véase el ejercicio 6). Si llamamos fib_n al término n -ésimo de la secuencia de Fibonacci, la secuencia viene descrita recurrentemente así:

$$\begin{aligned} fib_0 &= 1 \\ fib_1 &= 1 \\ fib_n &= fib_{n-2} + fib_{n-1}, \text{ si } n \geq 2 \end{aligned}$$

La correspondiente función en Pascal es una transcripción trivial de esta definición:

```
function Fib(num: integer): integer;
  {PreC.: num ≥ 0}
  {Dev. fibnum}
begin
  if (num = 0) or (num = 1) then
    Fib:= 1
  else
    Fib:= Fib(num - 1) + Fib(num - 2)
end; {Fib}
```

Los casos base son los números 0 y 1, y los recurrentes los naturales siguientes: 2, 3, ...

10.3.2 Torres de Hanoi

En la exposición mundial de París de 1883 el matemático francés E. Lucas presentó un juego llamado *Torres de Hanoi*,⁴ que tiene una solución recursiva relativamente sencilla y que suele exponerse como ejemplo de la potencia de la recursión para resolver ciertos problemas cuya solución es más compleja en forma iterativa.

³Descubierta por Leonardo da Pisa (1180-1250) y publicada en su *Liber Abaci* en 1202.

⁴Según reza la leyenda, en la ciudad de Hanoi, a orillas del río Rojo, descansa una bandeja de cobre con tres agujas verticales de diamante. Al terminar la creación, Dios ensartó en la primera de ellas sesenta y cuatro discos de oro puro de tamaños decrecientes. Ésta es la torre de Brahma. Desde entonces, los monjes empeñan su sabiduría en trasladar la torre hasta la tercera aguja, moviendo los discos de uno en uno y con la condición de que ninguno de ellos se apoye en otro de menor tamaño. La leyenda afirma que el término de esta tarea coincidirá con el fin del mundo, aunque no parece que, por el momento, estén cerca de lograrlo.

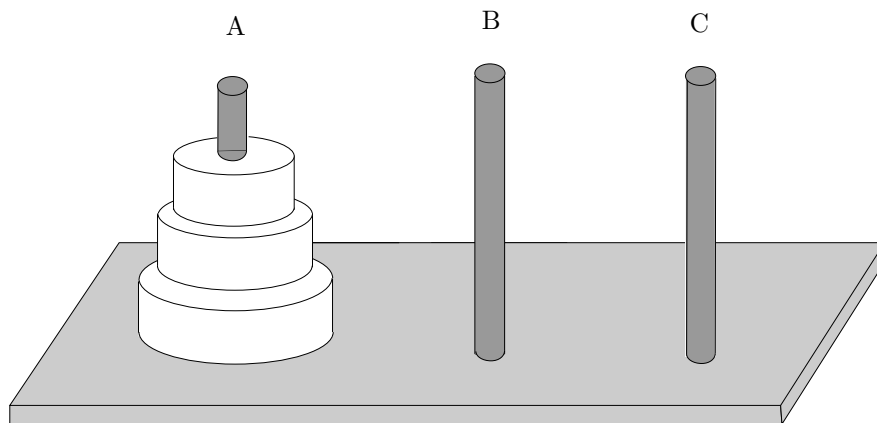


Figura 10.3. Las torres de Hanoi.

El juego estaba formado por una base con tres agujas verticales, y en una de ellas se encontraban engarzados unos discos de tamaño creciente formando una torre, según se muestra en la figura 10.3. El problema por resolver consiste en trasladar todos los discos de una aguja a otra, moviéndolos de uno en uno, pero con la condición de que un disco nunca descansa sobre otro menor. En distintas fases del traslado se deberán usar las agujas como almacén temporal de discos.

Llamaremos A, B y C a cada una de las agujas sin importar el orden siempre que se mantengan los nombres.

Consideremos inicialmente dos discos en A que queremos pasar a B utilizando C como auxiliar. Las operaciones por realizar son sencillas:

$$\text{Pasar dos discos de A a B} = \begin{cases} \text{Mover un disco de A a C} \\ \text{Mover un disco de A a B} \\ \text{Mover un disco de C a B} \end{cases}$$

Ahora supongamos que tenemos tres discos en A y queremos pasarlos a B. Haciendo algunos tanteos descubrimos que hay que pasar los dos discos superiores de A a C, mover el último disco de A a B y por último pasar los dos discos de C a B. Ya conocemos cómo pasar dos discos de A a B usando C como auxiliar, para pasarlos de A a C usaremos B como varilla auxiliar y para pasarlos de C a B usaremos A como auxiliar:

$$\text{Pasar 3 discos de A a B} = \left\{ \begin{array}{l} \text{Pasar dos de A a C} = \left\{ \begin{array}{l} \text{Mover 1 disco de A a B} \\ \text{Mover 1 disco de A a C} \\ \text{Mover 1 disco de B a C} \end{array} \right. \\ \text{Mover un disco de A a B} \\ \text{Pasar dos de C a B} = \left\{ \begin{array}{l} \text{Mover 1 disco de C a A} \\ \text{Mover 1 disco de C a B} \\ \text{Mover 1 disco de A a B} \end{array} \right. \end{array} \right.$$

En general, *Pasar n discos de A a B* (siendo $n \geq 1$), consiste en efectuar las siguientes operaciones,

$$\text{Pasar n discos de A a B} = \left\{ \begin{array}{l} \text{Pasar } n-1 \text{ discos de A a C} \\ \text{Mover 1 disco de A a B} \\ \text{Pasar } n-1 \text{ discos de C a B} \end{array} \right.$$

siendo 1 el caso base, que consiste en mover simplemente un disco sin generar llamada recursiva. Ahora apreciamos claramente la naturaleza recursiva del proceso, pues para pasar n discos es preciso pasar $n-1$ discos (dos veces), para $n-1$ habrá que pasar $n-2$ (también dos veces) y así sucesivamente.

Podemos escribir un procedimiento para desplazar n discos directamente:

```

procedure PasarDiscos(n: integer; inicial, final, auxiliar: char);
  {PreC.: n ≥ 0}
  {Efecto: se pasan n discos de la aguja inicial a la final}
begin
  if n > 0 then begin
    PasarDiscos (n - 1,inicial, auxiliar, final);
    WriteLn('mover el disco ', n:3, ' desde ', inicial, ' a ', final);
    PasarDiscos (n - 1,auxiliar, final, inicial)
  end {if}
end; {PasarDiscos}

```

Como ejemplo de funcionamiento, la llamada `PasarDiscos(4, 'A', 'B', 'C')` produce la siguiente salida:⁵

```

Cuántos discos: 4      || mover disco 4 desde A a B
mover disco 1 desde A a C || mover disco 1 desde C a B
mover disco 2 desde A a B || mover disco 2 desde C a A
mover disco 1 desde C a B || mover disco 1 desde B a A
mover disco 3 desde A a C || mover disco 3 desde C a B
mover disco 1 desde B a A || mover disco 1 desde A a C
mover disco 2 desde B a C || mover disco 2 desde A a B
mover disco 1 desde A a C || mover disco 1 desde C a B

```

⁵Como puede apreciarse los números de los discos indican su tamaño.

10.3.3 Función de Ackermann

Otro interesante ejemplo recursivo es la función de Ackermann que se define recurrentemente así:

$$\begin{aligned} \text{Ack}(0, n) &= n + 1 \\ \text{Ack}(m, 0) &= \text{Ack}(m - 1, 1), \text{ si } m > 0 \\ \text{Ack}(m, n) &= \text{Ack}(m - 1, \text{Ack}(m, n - 1)) \text{ si } m, n > 0 \end{aligned}$$

La función correspondiente en Pascal se escribe así:

```
function Ack(m, n: integer): integer;
  {PreC.: m, n ≥ 0}
  {Dev. Ack(m, n)}
begin
  if m = 0 then
    Ack := n + 1
  else if n = 0 then
    Ack := Ack(m - 1, 1)
  else
    Ack := Ack(m - 1, Ack(m, n - 1))
end; {Ack}
```

10.4 Corrección de subprogramas recursivos

En este apartado presentaremos los conceptos y técnicas necesarias para la verificación (o derivación) de subprogramas recursivos.

En este sentido, la pauta viene dada por la consideración de que un subprograma recursivo no es más que un caso particular de subprograma en el que aparecen llamadas a sí mismo. Esta peculiaridad hace que tengamos que recurrir a alguna herramienta matemática, de aplicación no demasiado complicada en la mayoría de los casos, que encontraremos en este libro.

El proceso de análisis de la corrección de subprogramas recursivos puede ser dividido, a nuestro entender, en dos partes: una primera, en la que consideraremos los pasos de la verificación comunes con los subprogramas no recursivos, y una segunda con los pasos en los que se aplican técnicas específicas de verificación de la recursión.

De acuerdo con esta división, incluiremos en primer lugar, y tal como se ha hecho hasta ahora, las precondiciones y postcondiciones de cada subprograma que, junto con el encabezamiento, formarán su especificación (semi-formal). Recordemos que las precondiciones y postcondiciones actúan como generalizaciones de las precondiciones y postcondiciones, respectivamente, de las instrucciones simples, explicitando los requisitos y los efectos del subprograma.

Asimismo, la verificación de las llamadas a subprogramas recursivos se hará igual que en el resto de los subprogramas, estableciendo las precondiciones y postcondiciones de éstas en base a las precondiciones y postcondiciones de los subprogramas llamados.

Por otra parte estudiaremos la corrección de la definición del subprograma. En esta tarea lo natural es plantearse el proceso de verificación (o corrección, según el caso) habitual, es decir, especificar las precondiciones y postcondiciones de cada una de las instrucciones implicadas, y en base a ellas y a su adecuado encadenamiento demostrar la corrección. Pero en el caso de un subprograma recursivo nos encontramos que, para al menos una de las instrucciones (aquella en la que aparece la llamada recursiva), no se tiene demostrada la corrección (de hecho es esa corrección la que intentamos demostrar).

Para salir de este ciclo recurrimos a técnicas inductivas de demostración.

10.4.1 Principios de inducción

Informalmente, podemos decir que estos principios permiten afirmar una propiedad para todo elemento de un conjunto (pre)ordenado, si se dan ciertas condiciones. Así, si suponiendo el cumplimiento de la propiedad para los elementos del conjunto menores que uno dado podemos demostrar la propiedad para el elemento en cuestión, afirmaremos que todo elemento del conjunto verifica la propiedad.

La formalización más simple y conocida del Principio de Inducción se hace sobre el conjunto de los números naturales y es la siguiente:

Si tenemos que

Hipótesis de inducción: 0 cumple la propiedad P

Paso inductivo: Para todo $x > 0$, si $x - 1$ cumple la propiedad P , entonces x cumple la propiedad P

Entonces

Para todo $y \in \mathbb{N}$, y cumple la propiedad P .

La relación entre inducción y recursión queda clara: la hipótesis de inducción se corresponde con el caso base, y el paso inductivo con el caso recurrente.

Por ejemplo, este principio se puede aplicar para demostrar que el número N de elementos del conjunto $\mathcal{P}(E)$, donde E representa un conjunto finito de n elementos y $\mathcal{P}(E)$ es el conjunto de las partes de E , es 2^n .

Como caso base tenemos que para $n = 0$, es decir, para $E = \emptyset$, se tiene que $\mathcal{P}(E) = \emptyset$, y por tanto $N = 1 = 2^0$.

Supongamos ahora que para $n - 1$, es decir, para $E = \{x_1, x_2, \dots, x_{n-1}\}$ se cumple que $N = 2^{n-1}$ y veamos si se puede demostrar que para n también se tiene que $N = 2^n$.

Distribuyamos las partes de $E = \{x_1, x_2, \dots, x_n\}$ en dos clases: una con las que no contienen al elemento x_n , y otra con las que sí lo contienen. La hipótesis de inducción expresa que la primera está constituida por 2^{n-1} subconjuntos, mientras que los subconjuntos de la segunda son los que resultan de la unión de $\{x_n\}$ con cada uno de los subconjuntos de la primera clase. Por tanto, el número total de subconjuntos es $N = 2^{n-1} + 2^{n-1} = 2^n$.

En consecuencia, aplicando el principio de inducción se puede afirmar que para todo $n \in \mathbb{N}$ se cumple que $\mathcal{P}(E)$ tiene 2^n elementos.

Aunque esta formalización del Principio de Inducción es suficiente para un gran número de casos, en otros se puede requerir otra en la que tomamos como hipótesis la verificación de la propiedad por *todos* los elementos menores que x :

Es decir,

- Si para cualquier $x \in \mathbb{N}$ se tiene que, si todo $y < x$ tiene la propiedad P , entonces x también tiene la propiedad P
- entonces todo $z \in \mathbb{N}$ tiene la propiedad P

Disponemos ya de la herramienta necesaria para retomar la verificación de nuestro subprograma recursivo. Recordemos que nos encontrábamos con el problema de comprobar la corrección de una llamada al propio subprograma que estamos verificando, lo cual nos hace entrar, aparentemente, en un ciclo sin fin. La clave para salir de este ciclo es darnos cuenta de que, si la recursión está bien definida, la llamada que intentamos verificar tiene como parámetro de llamada un valor menor (o, en otras palabras, más cercano al caso base de la recursión). Por ejemplo, en el caso de la función factorial, la estructura de selección que controla la recursión es:

```

if num = 0 then
  Fac := 1
else
  Fac := num * Fac(num - 1)

```

En este punto es donde entra en juego el Principio de Inducción, ya que, basándonos en él, si

1. el subprograma es correcto en el caso base (en nuestro caso es obvio que $\text{Fac}(0) = 1 = 0!$), y

2. demostramos que la construcción del paso recursivo es correcta, suponiendo que lo es la llamada al subprograma para valores menores del parámetro sobre el que se hace la recursión. En este caso tenemos asegurada la corrección de nuestro subprograma para cualquier valor del parámetro de entrada.

En el ejemplo, basta con demostrar que, si suponemos que

$$\text{Fac}(\text{num} - 1) = (\text{num} - 1)!$$

entonces

$$\begin{aligned} \text{Fac}(\text{num}) &= \text{num} * \text{Fac}(\text{num} - 1) \\ &= \text{num} * (\text{num} - 1)! \\ &= \text{num}! \end{aligned}$$

En resumen, para demostrar la corrección de un subprograma recursivo hemos de comprobar:

- La corrección del caso base.
- La corrección de los casos recurrentes. Para ello, se supone la de las llamadas subsidiarias, como ocurre en el paso inductivo con la hipótesis de inducción.
- Que las llamadas recursivas se hacen de manera que los parámetros se acercan al caso base; por ejemplo, en el cálculo del factorial, en las sucesivas llamadas los parámetros son $n, n - 1, \dots$, que desembocan en el caso base 0, siempre que $n > 0$, lo cual se exige en la condición previa de la función.

10.5 Recursión mutua

Cuando un subprograma llama a otro y éste a su vez al primero, se produce lo que se denomina *recursión mutua* o *cruzada*, que consiste en que un subprograma provoque una llamada a sí mismo, indirectamente, a través de otro u otros subprogramas.

En estos casos, se presenta un problema para definir los subprogramas, porque uno de ellos tendrá que ser definido antes del otro, y la llamada que haga al segundo se hace a un identificador desconocido, contraviniendo la norma de Pascal por la que un identificador tiene que ser declarado antes de usarlo.

No obstante, el mismo lenguaje nos da la solución mediante el uso de la palabra reservada **forward**. Con su uso, el identificador del subprograma definido

en segundo lugar es predeclarado, escribiendo su encabezamiento seguido por **forward**, y por lo tanto es reconocido en el subprograma definido en primer lugar. Al definir el segundo subprograma no es necesario repetir sus parámetros. El esquema de implementación en Pascal de dos procedimientos mutuamente recursivos es el siguiente:

```

procedure Segundo(parámetros); forward;
procedure Primero(parámetros);
  ...
begin {Primero}
  ...
  Segundo (...);
  ...
end; {Primero}

procedure Segundo(parámetros);
  ...
begin {Segundo}
  ...
  Primero (...);
  ...
end; {Segundo}

```

A continuación se muestra un ejemplo en el que se aplica el concepto de recursión mutua. Se trata de un programa que comprueba el correcto equilibrado de paréntesis y corchetes en expresiones introducidas por el usuario.⁶ Así, si se da como entrada la expresión $[3 * (2 + 1) - 5] + 7$, el programa debe dar un mensaje que indique que los paréntesis y los corchetes están equilibrados, y, en cambio, si la entrada proporcionada por el usuario es la expresión $(a + [b * 5) - c]$, debe dar al menos un mensaje de error en el equilibrado.

Con este objetivo, un primer paso en el diseño del programa *Equilibrado* puede ser:

```

repetir
  Leer carácter c
  en caso de que c sea
    '(' : Cerrar paréntesis
    ')' : Tratar paréntesis de cierre
    '[' : Cerrar corchete
    ']' : Tratar corchete de cierre
  hasta fin de la entrada

```

⁶Se considera que la expresión viene dada en una sola línea del input, para simplificar la codificación.

En el caso en que se encuentre en el `input` un paréntesis o un corchete abierto, hay que seguir leyendo caracteres hasta encontrar (si es que existe) el correspondiente símbolo de cierre. En caso de encontrarlo se dará un mensaje de éxito, y en caso contrario, dependiendo del símbolo encontrado, se tomará la acción correspondiente:

- si es un símbolo de cierre equivocado, se dará un mensaje de error.
- si es un paréntesis o corchete abierto, se hará una llamada recursiva.
- si es el fin del `input`, también se dará un mensaje indicándolo.

Teniendo en cuenta estas indicaciones, el siguiente nivel de refinamiento de *Cerrar paréntesis* puede ser:

```

Repetir
  Leer carácter c
  en caso de que c sea
    '(' : Cerrar paréntesis
    ')' : Dar mensaje de éxito
    '[' : Cerrar corchete
    ']' : Dar mensaje de error
  hasta c = ')' o fin de la entrada
  Si fin de la entrada entonces
    Dar mensaje de error

```

Y simplemente cambiando los corchetes por paréntesis y viceversa, puede el lector obtener el siguiente nivel de *Cerrar corchete*.

Ya en este nivel de diseño se puede observar que las tareas *Cerrar paréntesis* y *Cerrar corchete* son mutuamente recursivas, y como tales deben ser tratadas en la codificación en Pascal que se da a continuación:

```

Program Equilibrado (input, output);
  {Estudia el equilibrado de paréntesis y corchetes en secuencias de
  caracteres}
  var
    c: char;

  procedure CierraCorchete; forward;

  procedure CierraPar;
    {PreC.: se ha leído un carácter '(' y no EoLn}
    {Efecto: se ha recorrido la entrada hasta encontrar un carácter ')'
    o el fin de la entrada, dando los mensajes adecuados si se ha leído
    un símbolo inapropiado}

```

```

var
  c: char;
begin
  repeat
    Read(c);
    case c of
      '(': CierraPar;
        {Llamada recursiva para tratar una pareja de paréntesis
         anidados}
      ')': WriteLn('cuadra el paréntesis');
      '[': CierraCorchete;
        {Llamada recursiva para tratar una pareja de corchetes
         anidados}
      ']': WriteLn('error: cierra paréntesis con corchete')
    end {case}
  until (c = ')') or EoLn;
  if EoLn and (c <> ')') then
    {Se llega al fin de la entrada sin encontrar el cierre de
     paréntesis}
    WriteLn('error: se queda un paréntesis abierto')
  end; {CierraPar}

procedure CierraCorchete;
{PreC.: se ha leído un carácter '[' y no EoLn}
{Efecto: se ha recorrido la entrada hasta encontrar un carácter ']'
o el fin de la entrada, dando los mensajes adecuados si se ha leído
un símbolo inapropiado}
var
  c: char;
begin
  repeat
    Read(c);
    case c of
      '(': CierraPar;
        {Llamada recursiva para tratar una pareja de paréntesis
         anidados}
      ')': WriteLn('error: cierra corchete con paréntesis');
      '[': CierraCorchete;
        {Llamada recursiva para tratar una pareja de corchetes
         anidados}
      ']': WriteLn('cuadra el corchete')
    end {case}
  until (c = ']') or EoLn;
  if EoLn and (c <> ']') then
    {Se llega al fin de la entrada sin encontrar el cierre de
     corchete}
    WriteLn('error: se queda un corchete abierto')
  end;
end;

```

```

    end; {CierraCorchete}

begin {Equilibrado}
  repeat
    Read(c);
    case c of
      '(': if not EoLn then
            CierraPar {Se intenta equilibrar el paréntesis}
          else
            {La entrada acaba en '('}
            WriteLn('error: se queda un paréntesis abierto');
      ')'': WriteLn('error: paréntesis cerrado incorrectamente');
      '[': if not EoLn then
            CierraCorchete {Se intenta equilibrar el corchete}
          else
            {La entrada acaba en '{' }
            WriteLn('error: se queda un corchete abierto')
          end {case}
    until EoLn
end. {Equilibrado}

```

10.6 Recursión e iteración

Si un subprograma se llama a sí mismo se repite su ejecución un cierto número de veces. Por este motivo, la recursión es una forma especial de iteración y, de hecho, cualquier proceso recursivo puede expresarse de forma iterativa, con más o menos esfuerzo, y viceversa. Un ejemplo de ello es el cálculo del factorial (véanse los apartados 8.2.1 y 10.1).

Sabiendo que un determinado problema puede resolverse de las dos maneras, ¿cuándo se debe usar una u otra? Como norma general, debe adoptarse siempre (al menos en un primer momento) la solución que resulte más natural, concentrando los esfuerzos en la corrección del algoritmo desarrollado. Por ejemplo, los problemas que vienen descritos en forma recurrente se prestan más fácilmente a una solución recursiva. Un ejemplo es el problema de las torres de Hanoi, cuya versión iterativa es bastante más complicada que la recursiva.

Por otra parte el mecanismo de la recursión produce, además de la iteración, la creación automática de nuevos parámetros y objetos locales en cada llamada (apilándose éstos). Por consiguiente, se tiene un gasto adicional de memoria (el de la pila recursiva, para almacenar las sucesivas tablas de activación), además del tiempo necesario para realizar esas gestiones. Todo esto puede hacer que ciertos programas recursivos sean menos eficientes que sus equivalentes iterativos.

Por ello, cuando sean posibles soluciones de ambos tipos,⁷ es preferible la iterativa a la recursiva, por resultar más económica su ejecución en tiempo y memoria.

10.7 Ejercicios

1. Escriba una función recursiva para calcular el término n -ésimo de la secuencia de Lucas: 1, 3, 4, 7, 11, 18, 29, 47, ...
2. Dado el programa

```

Program Invertir (input, output);
  {Se lee una línea del input y se escribe invertida}

  procedure InvertirRec;
    var
      c: char;
    begin
      Read(c);
      if c <> '.' then begin
        InvertirRec;
        Write(c)
      end
    end; {InvertirRec}

  begin {Invertir}
    WriteLn('Teclee una cadena de caracteres ( "." para finalizar)');
    InvertirRec
  end. {Invertir}

```

Analice su comportamiento y estudie qué resultado dará para la secuencia de entrada "aeiou.", describiendo la evolución de la pila recursiva (véase el apartado 10.2). Obsérvese que el uso de esta pila recursiva nos permite recuperar los caracteres en orden inverso al de su lectura.

3. Escriba una función recursiva para calcular el máximo común divisor de dos números enteros dados aplicando las propiedades recurrentes del ejercicio 2 del capítulo 9.
4. Escriba una versión recursiva del cálculo del máximo común divisor de dos números enteros por el método de Euclides.
5. Defina subprogramas recursivos para los siguientes cálculos:

$$(a) \left(1 + \frac{1}{2} + \dots + \frac{1}{n}\right) = \left(1 + \frac{1}{2} + \dots + \frac{1}{n-1}\right) + \frac{1}{n}$$

⁷Como es, por ejemplo, el caso de la función factorial.

(b) La potencia de un real elevado a un entero positivo:

$$\begin{aligned} x^0 &= 1 \\ x^n &= (x * x)^{\frac{n}{2}}, & \text{si } n > 0 \text{ y es par} \\ x^n &= x * (x^{n-1}), & \text{si } n > 0 \text{ y es impar} \end{aligned}$$

(c) La cifra i -ésima de un entero n ; es decir,

- la última, si $i = 0$
- la cifra $(i-1)$ -ésima de $n \mathbf{div} 10$, en otro caso.

(d) El coeficiente binomial, definido recurrentemente:

$$\begin{aligned} \binom{n}{0} &= \binom{n}{n} = 1 \\ \binom{n}{k} &= \binom{n-1}{k} + \binom{n-1}{k-1} \end{aligned}$$

6. Sabiendo que, para $inf, sup \in \mathcal{Z}$, tales que $inf \leq sup$, se tiene

$$\sum_{i=inf}^{sup} a_i = \begin{cases} a_i, & \text{si } inf = sup \\ \sum_{i=inf}^{med} a_i + \sum_{i=med+1}^{sup} a_i & \text{si } inf < sup, \end{cases}$$

(siendo $med = (inf + sup) \mathbf{div} 2$) defina una función recursiva para $\sum_{i=inf}^{sup} \frac{1}{i^2}$, e inclúyala en un programa que halle $\sum_{i=1}^{100} \frac{1}{i^2}$,

7. Use el hecho de que

$$\int_a^b f(x)dx = \int_a^m f(x)dx + \int_m^b f(x)dx$$

(siendo $m = \frac{a+b}{2}$) para desarrollar una función recursiva que halle aproximadamente la integral definida de la función $sen(x)$ a base de dividir el intervalo $[a, b]$ en dos hasta que sea lo bastante pequeño ($|b - a| < \epsilon$), en cuyo caso aceptamos que $\int_a^b f(x)dx \simeq (b - a) * f(m)$.

8. Sabiendo que 0 es par, es decir,

$$\begin{aligned} \mathbf{EsPar}(0) &\rightsquigarrow \mathbf{true} \\ \mathbf{EsImpar}(0) &\rightsquigarrow \mathbf{false} \end{aligned}$$

y que la paridad de cualquier otro entero positivo es la opuesta que la del entero anterior, desarrolle las funciones lógicas, mutuamente recursivas, **EsPar** y **EsImpar**, que se complementen a la hora de averiguar la paridad de un entero positivo.

10.8 Referencias bibliográficas

En [Sal93] y [CCM⁺93] se ofrecen buenos enfoques de la programación con subprogramas. El primero de ellos introduce los subprogramas antes incluso que las instrucciones estructuradas. El segundo ofrece una concreción de los conceptos de programación modular explicados en los lenguajes C y Modula-2.

El libro de Alagic y Arbib [AA78] es una referencia obligada entre los libros orientados hacia la verificación con un enfoque formal.

Algunos de los conceptos contenidos en el tema provienen de la ingeniería del *software*. Para ampliar estos conceptos recomendamos un texto sobre esta disciplina, como es [Pre93]. En [PJ88] se describen con detalle las técnicas para obtener diseños de jerarquías de subprogramas de la mayor calidad apoyándose en los criterios de independencia funcional, caja negra, tamaño de los módulos y muchos otros.

La recursión es un concepto difícil de explicar y comprender, que se presenta frecuentemente relacionándolo con la iteración, a partir de ejemplos que admiten versiones iterativas y recursivas similares. En [Wie88] y [For82] se ofrece este enfoque.

En el libro [RN88] puede leerse la historia completa sobre las torres de Hanoi y los grandes logros del famoso matemático francés E. Lucas, entre otros muchos temas matemáticos, que se presentan con una pequeña introducción histórica. Existe un problema similar al de las Torres de Hanoi, llamado de los anillos chinos, cuya solución está desarrollada en [ES85] y en [Dew85b]

Tema IV

Tipos de datos definidos por el programador